Numerit

Version 1.7

User's Manual

Do your numerical computations the easy way

Go directly to the final publication-quality document

Write powerful programs with minimum effort

Use superior tools to edit and debug your programs

Numerit 1.7 User's Manual

Contents

Chapter 1 Numerit

Introduction

Numerit is an intuitive programming environment for developing numerical computation programs and producing publication quality documents. It combines the versatility of conventional programming with the powerful interface of Windows.

Numerit includes a specially designed program editor to make it easier for you to write your programs. The editor includes special editing tools for inserting and manipulating program-blocks. It highlights keywords and displays comments in a distinctive color. It lets you customize the fonts and the colors. And It has many levels of undo/redo.

Numerit has built-in debugging tools to help you while you develop your program. The program may be paused at any point during the run, and may be executed line by line. Variables may be inspected while the program is running with a simple mouse click.

Numerit includes a specially designed document editor to edit your program's output. This editor has all the tools and features you expect from an advanced scientific word processor. It has a built-in equation editor. It automatically numbers figure captions, table captions, equations, and references, and also updates all references to these numbers in the document. It lets you insert graphs and tables that are directly linked to your program's variables and automatically refreshes them each time you run the program. The program's output document may be saved as an RTF file which allows you to continue the editing in your favorite word processor (e.g., Microsoft Word).

Numerit's high-level programming language is easy-to-learn, powerful, and versatile. It supports multidimensional arrays (up to eight dimensions), complex numbers, and user-defined functions. It has many built-in functions for data analysis, including: line and polynomial fitting, linear and cubic-splines interpolation, multidimensional fast Fourier transform, numerical integration and differentiation, root finding, solution of linear algebraic equations, matrix/vector arithmetic, special functions (gamma, erf, Bessel), and more.

Numerit's Editions

Numerit is currently offered in two editions: the Standard Edition and the Professional Edition (***Numerit Pro***). ***Numerit Pro*** allows linking of several ***Numerit*** programs (modules) into a single program. In the Standard Edition a program may have only one module. ***Numerit Pro*** lets you build libraries of reusable functions; once a library is compiled and saved its functions may be called from any program.
Besides modularity ***Numerit*** and ***Numerit Pro*** are identical. So users of the Standard Edition can use this manual knowing that they are limited to single module programs.

NumRun

In addition to ***Numerit*** we also provide the program ***NumRun*** which allows execution of compiled ***Numerit*** programs outside the ***Numerit*** environment. See the section **NumRun: The Numerit Executor**.

Users of ***Numerit*** may develop programs and distribute them to customers. The customers can download ***NumRun*** from http://www.numerit.com/download.htm, for free, and use it to run the programs. No royalties are charged from users that develop ***Numerit*** programs and distribute or sell them.

System requirements

To use ***Numerit*** you need:

- A Pentium (or higher) computer.
- Windows: 98, 2000, XP, Vista, 7.

Registration

Every copy of ***Numerit*** has its own Registration Number. When you invoke the Evaluation Edition of ***Numerit*** you may see the Registration Number of your copy on the opening dialog window. In order to register your evaluation copy and upgrade it to the commercial edition visit our web site at http://www.numerit.com, fill out the order form, and submit it. After processing your order we will send you a Registration Code that matches your Registration Number. Click the button **Registration-Code** in the opening dialog and enter the Registration Code.
Please keep both your Registration Number and the Registration Code safe; you will need them to restore the registration information in case it is damaged.

Registration Number: ________________________________

Registration Code: ________________________________

You can create a Registration File by clicking the item **Create Registration File** in the **File** menu. This file together with the Registration Number and the Registration Code allows you to restore your registered copy in case of a damage to the registration information or to your disk.

Getting help

Numerit comes with a context sensitive Help. To get the Help contents, choose **Contents** from the **Help** menu, or press F1, or click the ? icon on the Program Bar (the top tool-bar).
To get help on a specific keyword, click on it with the right mouse button, or choose **Search** from the **Help** menu (Ctrl+F1) while the insertion cursor is on the keyword.
Numerit also comes with a set of sample programs that are found in the folder **Samples** which is located inside the ***Numerit*** folder (C:\Program Files\Numerit, or as otherwise specified during the installation). These samples may help you when you develop your first ***Numerit*** programs.

Getting in touch

Web Site: http://www.numerit.com/
Technical Support: support@numerit.com
General Information: info@numerit.com
Comments and Suggestions: feedback@numerit.com

Your first program

After you start ***Numerit*** you see an empty program window titled Program<1>. On the left you see the Editor pane and on the right the Document pane. The insertion cursor is inside the Editor.
To write your first program enter the following two lines:

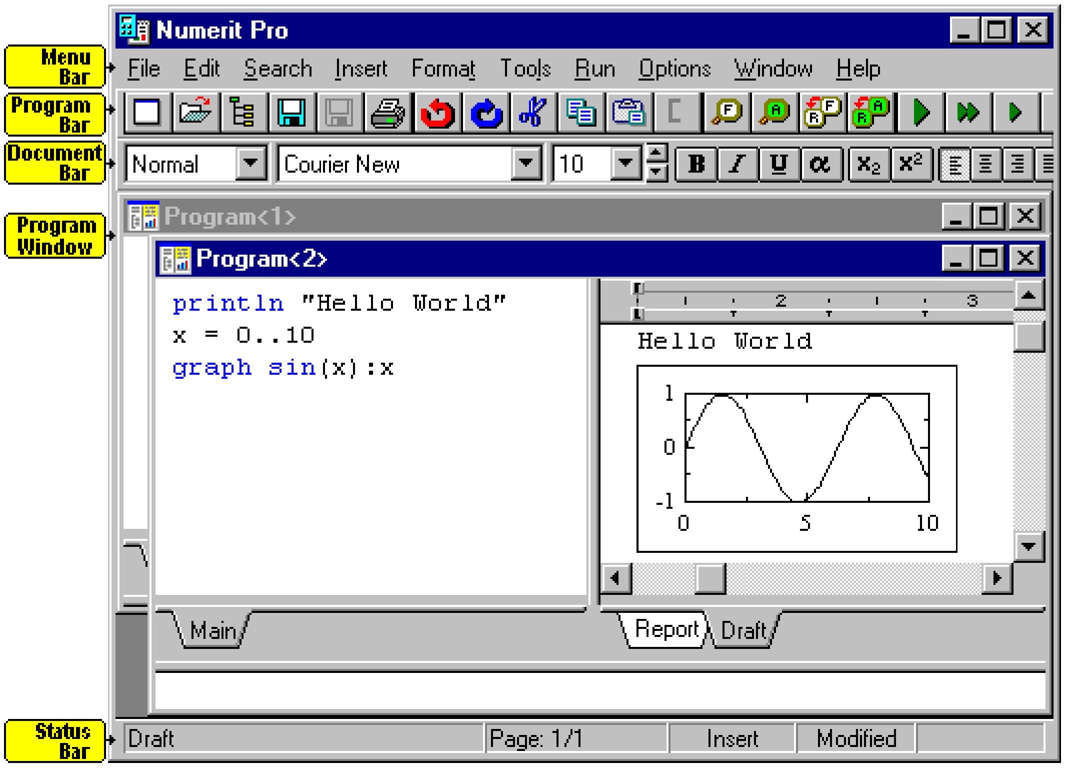
```
x = 0..10
graph sin(x):x
```

Press F9 to run. ***Numerit*** will plot the graph on the Document pane.

The Desktop

Introduction

The Desktop is ***Numerit***'s workplace, namely, the main window which consists of the Menu bar, the Tool bars, the Status bar, and the Program windows.



```
println "Hello World"
x = 0..10
graph sin(x):x
```

The Menu Bar

The menu bar is located at the top of the Desktop. It toggles between two states, one is used for program editing and the other is used for document editing.

The following menus are common to both states. Some menu items are applicable only to the program editor and are designated by (Editor), and some are only applicable to the document and are designated by (Document).

File

New (Ctrl+N)	Open a new Program
Open (Ctrl+O)	Open a Program or a Document
Close (Ctrl+F4)	Close the current Program
Close All	Close all Programs
Save (Ctrl+S)	Save the current Program
Save As	Save the current Program under a new name
Save Report As	Save the current Program's Report as a Document
Save Draft As	Save the current Program's Draft as a Document
Save All (Ctrl+Shift+S)	Save the current program and modules (Professional Edition)
Modules	Open the Modules window (Professional Edition)
Close Module	Close the current module (Professional Edition)
Save Module (Ctrl+S)	Save the current module (Professional Edition)
Save Module As	Save the current module under a new name (Professional Edition)
Open Extra Module	Open the Extra module (Professional Edition)

Close Extra Module	Close the Extra module (Professional Edition)
New Input File	Create a new input file
Open Input File	Open an input file
Close Input File	Close the current input file
Save Input File (Ctrl+S)	Save the current input file
Save Input File As	Save the current input file under a new name
Print (Ctrl+P)	Print the currently visible Editor's or Document's content
Printer Setup	Set up printer parameters
Create Executable	Create a ***Numerit*** executable for ***NumRun***
Create Library	Create a ***Numerit*** library
Create Registration File	Create a registration file for restoring the registered version in case of a damage
Restore Registration Info	Restore your registered version after a damage
Exit (Alt+F4)	End the session and exit ***Numerit***

Edit

Undo (Ctrl+Z)	Undo the last editing action
Redo (Ctrl+Shift+Z)	Redo the last undone action
Cut (Ctrl+X)	Cut the selected text (delete and copy to the clipboard)
Copy (Ctrl+C)	Copy the selected text to the clipboard
Paste (Ctrl+V)	Insert text from the clipboard to the current position
Paste Special	Insert text from the clipboard in a specific format (Document)
Delete (Del)	Delete the selected text
Delete Line (Ctrl+Y)	Delete the current line (Editor)
Select All (Ctrl+A)	Select the whole text (not including headers/footers)
Clear Draft	Clear the Draft text
Block (Ctrl+[)	Block the selected text or insert a new block (Editor)
Fold (Ctrl+Shift+F)	Fold/Unfold the current block (Editor)

Search

Find (Ctrl+F)	Find the specified text
Find (Shift+F3)	Find the specified text (same as Ctrl+F)
Find Again (F3)	Repeat the last Find operation
Replace (Ctrl+H)	Replace the specified text by another text
Replace (Shift+F2)	Replace the specified text by another text (same as Ctrl+H)
Replace Again (F2)	Repeat the last Replace operation
Go to Top (Ctrl+Home)	Move to the top of the Editor
Go to End (Ctrl+End)	Move to the bottom of the Editor
Go to Line	Move to the specified line in the Editor (Editor)
Go to Execution Point	Move to the line that will be executed next (Editor)
Go to Page	Move to the specified page in the Document (Document)

Run

Compile (Ctrl+F9)	Compile the current program (Standard Edition)
Compile Module	Compile the current module (Professional Edition)
Make (Ctrl+F9)	Compile all modules that need compiling and link external functions (Professional Edition)
Run (F9)	Run the current program (compile/make if necessary)
Step Over (F8)	Execute the current line while stepping over functions
Step Into (F7)	Execute the current line while entering functions
Pause (F6)	Pause the current program
Stop (F5)	Stop the current program
Calling Function (Ctrl+<)	Move inspection position to the calling function
Called Function (Ctrl+>)	Move inspection position to the called function
Go to Execution Point	Move to the line that will be executed next (Editor)
Inspect (F12)	Open the Inspector and show variable's content
Breakpoint (F4)	Set/Clear a breakpoint at the current line (pause before executing the line)
Clear Breakpoints	Clear all breakpoints
Previous Error (Ctrl+ Shift+<)	Move to previous (compilation or execution) error or warning
Next Error (Ctrl+Shift+>)	Move to next (compilation or execution) error or warning

Options

Editor	Define Editor options
Document	Define Document options
View Program bar	Show/Hide the Program bar
View Document bar	Show/Hide the Document bar
View Status bar	Show/Hide the Status bar
Panes Auto-Size	Resize panes automatically when changing the active pane
Ignore Compiler Warnings	Ignore compiler warnings and notices
View Document Ruler	Show/Hide the Ruler in the current document
View Document Margins	Show/Hide the top/bottom margins in the current document
View Document Whole page	Show the whole page of the current document
View Document Special characters	Show special characters (spaces, tabs, etc.)
Update Document fields	Update fields (figure numbers, equation numbers, etc.)
Set Document Units	Set the measurement units (inch, cm)

Window

Cascade (Shift+F5)	Cascade Program windows on the desktop
Tile (Shift+F4)	Tile Program windows on the desktop
Minimize All	Minimize all Program windows
Arrange Icon	Arrange icons of minimized windows
Horizontal Split (Shift+F9)	Split the current program horizontally
Vertical Split (Shift+F8)	Split the current program vertically
Maximize Pane (Shift+F7)	Maximize the current pane
Toggle Panes (Shift+F6)	Toggle between the Editor and the Document panes
Input	Select the Input file for editing
Extra	Select the Extra module for editing (Professional Edition)
Editor	Select an open module for editing
Document	Select the Report or the Draft for editing
Report/Draft (Ctrl+D)	Check to show the Report in the Document pane; Uncheck to show the Draft
Message (Ctrl+M)	Show the Message pane

Help

Contents (F1)	Show contents of online help
Search (Ctrl+F1)	Search for help on keyword
About Numerit	Show version number, copyright, registration info, etc.

The following menus appear when editing the document:

Insert

Viewer (F11)	Insert a Viewer
Field (F12)	Insert a Field
Figure caption	Insert a numbered Figure caption header
Table caption	Insert a numbered Table caption header
Reference caption	Insert a numbered Reference caption header
Equation	Insert a numbered Equation
Expression	Insert an Expression
Picture	Insert a Picture from disk
Date	Insert a Date in a specific format
Page break (Ctrl+Enter)	Insert a new page (force a page break)
Line break (Shift+Enter)	Insert a new line within the same paragraph (force a line break)
Nonbreaking space (Ctrl+Shift+Spacebar)	Insert a nonbreaking space

Format

Character	Change character appearance (Bold, Font size, etc.)
Paragraph	Change paragraph appearance (Alignment, Indentations, Tabs, etc.)
Style	Change paragraph style (Normal, Title, etc.)
Page	Change page appearance (Size, Margins, etc.)
Define Style	Change definition of styles

Tools

Document Spelling	Check spelling of the document from the current position
Word Spelling (Ctrl+W)	Check spelling of the word at the current position
Dictionary	Edit personal dictionaries
Character Map	Open a character map for the current font

The Program Bar

The Program Bar is the main tool bar. It is located below the menu bar at the top of the Desktop. The buttons in the Program Bar give faster access to some of the menu commands and are arranged in six groups, each matches a set of menu commands. These buttons are relevant to the program as a whole and to both the Editor and the Document panes. The buttons are also used as indicators for various states. For example, when the program is running, the Run button - ▶ is depressed and shown as - ▶.

Note that the buttons 'Modules' and 'Save Module' are only available in ***Numerit Pro***.

Use	To Invoke
	File: New, Open, Modules, Save, Save Module, Print.
	Edit: Undo, Redo, Cut, Copy, Paste, Block.
	Search: Find, Find Again, Replace, Replace Again.
	Run: Run, Step Over, Step Into, Pause, Stop, Calling Function, Called Function.
	Window: Vertical/ Horizontal Split, Maximize Pane, Toggle Panes, Message, Report/Draft.
?	Help: Contents

The Program Bar may be hidden by unchecking the item **View Program bar** in the **Options** menu.

The Document Bar

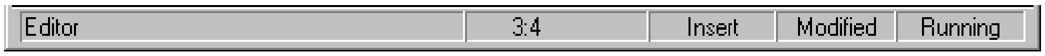
The Document Bar is used for operations that are related to document editing. It is located below the Program Bar at the top of the Desktop and is active only when editing a document. The buttons in the Document Bar are also used as indicators for various states. For example, when editing a bold text in the document, the Bold button - **B** is depressed and shown as - **B**.

Use	To
Normal	Select a style for the current paragraph (Ctrl+T).
Times New Roman	Select a font.
12	Select a character height (point size).
B *I* U α	Toggle bold/italic/underline/Greek attribute (Ctrl+B/I/U/G).
X_2 X^2	Set subscript/superscript attribute (Ctrl+[, Ctrl+]).
	Align paragraph: left/center/right/justified (Ctrl+L/E/R/J).
	Set left/center/right aligned tab.
¶	Show/Hide paragraph marks and other special characters.
	Show the whole page in the window.
	Insert an equation at the current position (Ctrl+Q).
	Open a character map for the current font.
	Check spelling of the document from the current position.
	Check spelling of the word at the current position (Ctrl+W).

The Document Bar may be hidden by unchecking the item **View Document bar** in the **Options** menu.

The Status Bar

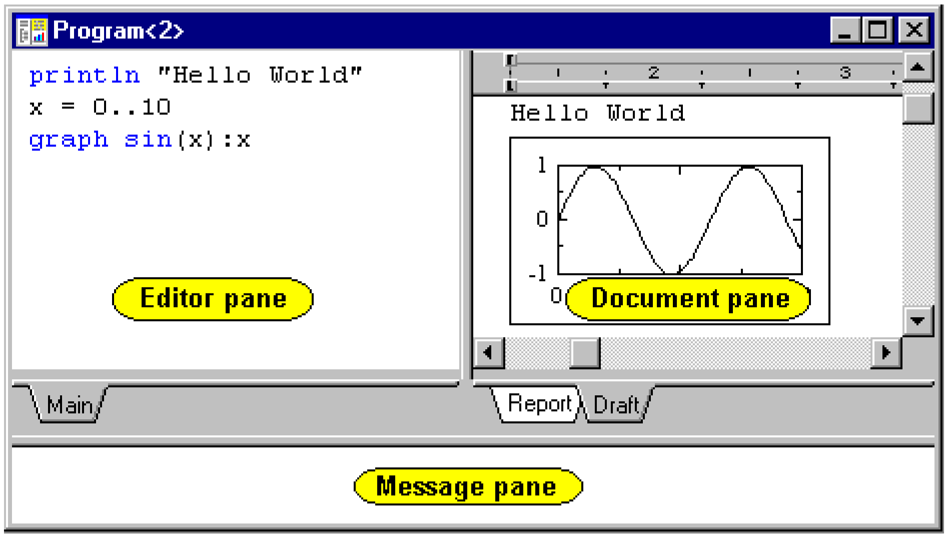
The status bar is used for displaying information about the current status of the program. It is located at the bottom of the Desktop and is divided into five fields:



The first field at the left shows the current position in the program (Editor, Report, etc.).
The second field shows the position in the Editor (line no.:column no.) or the Document (page no./number of pages) whichever is currently active.
The third field shows the current Insert mode (Insert/Overwrite).
The fourth field shows the Modified state. When the item "Modified" appears in the status bar it means that the text in the currently active pane (i.e. the program’s code in the Editor pane, or the report or the draft in the Document pane) has been modified since the last save.
The last field shows the Run state (Running, Pause, etc.).
When the cursor is pointing to a menu command or to a button in one of the tool bars, the status bar displays a help message for that command.
The status bar may be hidden by unchecking the item **View Status bar** in the **Options** menu.

The Program Window

A new program is created by opening a Program window using the menu command **File:New** (or Ctrl+N). An existing program is loaded from the disk by the command **File:Open** (or Ctrl+O).
Each Program window is divided into panes. The two main panes are the Editor pane and the Document pane. The Editor pane is where the program's code is written and the Document pane is where the program's output is displayed. In addition, there is a third pane, the Message pane at the bottom of the program window. This pane is used for displaying errors and warnings.



```
println "Hello World"
x = 0..10
graph sin(x):x
```

```
Hello World
```

It is possible to arrange the panes in two configurations. The first, vertical (shown above), puts the Editor pane on the left and the Document pane on the right. The second, horizontal, puts the Document pane at the top and the Editor pane below it. The Message pane is always at the bottom. To change configuration, use either the commands in the **Window** menu, or the corresponding buttons in the Program Bar.
The panes may be resized by changing the position of the separators between them (pressing the mouse button and dragging). There are two such separators, one between the Editor and the Document panes, and one above the Message pane. When the configuration is changed or when the active pane is maximized, the separators are relocated automatically.
Whenever one of the Editor or the Document panes becomes active and it is smaller than the other, the two panes swap sizes so that the active pane takes the larger size. This makes it easy to work with the Editor and the Document panes without the need for manual resizing. It is possible to disable panes auto-sizing by unchecking the menu item **Options:Panes Auto-Size**. The pane sizes are saved with the program and are restored when the program is loaded from disk.

Printing programs and documents

To print the program's code or the document use the menu command **File:Print** or the Print button in the Program Bar. This opens a dialog for entering the print specifications. The program's code is printed when the Editor pane is the active pane. The document is printed when the Document pane is the active pane.
The menu command **File:Printer Setup** opens a dialog for selecting a specific printer from a list of currently installed printers. The button **Properties** in the dialog opens a dialog (usually supplied by the printer's vendor) for changing the settings of the selected printer.

Chapter 3

The Editor

Introduction

The program is written and edited in the Editor pane by ***Numerit***'s program editor. The editor was especially designed to help you edit and debug your program. In ***Numerit Pro*** the Editor pane is used for editing the Main module as well as other modules of the program. To edit a module: open the Modules dialog (menu command **File:Modules** or the Modules button in the Program bar), select the module, and click Open Module. This opens the module and brings it into the Editor. At the bottom of the Editor pane there are tabs that allows you to select which of the currently open modules you want to bring into the Editor. The Editor pane is also used for editing the Input File (see **Input File** below for details).
To print the Editor's content use the **File:Print** menu command or click the Print button in the Program Bar when the Editor pane is active.

Program Blocks

Program blocks are used to group together several instructions in the program's code. There are many commands that require blocks, for example, **`if`**, **`for`**, and **`function`**.
Unlike other languages, a block in ***Numerit*** is not constructed by separate Begin and End statements, but is a single entity. It appears to the user as a frame which wraps the instructions on the left. The block frame adjusts itself automatically to the number of instructions that are inserted into it.
For example:

```
 if x > 0
┌y = x^2
│if y > 100
│┌u = x
│└v = y
│w = u + v
└
```

This blocking technique allows easy manipulation of program blocks, and has many advantages over conventional blocking, including:

- You get a much better view of the program's structure, especially when there are several levels of nested blocks.
- You don't have to worry about matching Begin and Ends, nested blocks, block indentation etc.; it is all done automatically.
- Blocks may be folded, which makes it possible to get an overview of the program without the details of inner blocks.
- A group of instructions may be easily put inside a block by selecting it and inserting a block.
- A whole block may easily be commented by a single key press.
- Commenting out a group of instructions is very simple: just select it, insert a block, and comment the block. If such a block is also folded, it becomes almost invisible.

Blocks are inserted by the menu command **Edit:Block** (Ctrl+[) or by the Block button in the Program Bar. The shortcut key ctrl+[is the most useful way for inserting blocks. The same command is used to insert a block around a selected group of instructions and to insert an empty block (when no text is selected).
A block frame may be deleted by pressing **Delete** in front of it.
Blocks are folded and unfolded by using the menu command **Edit:Fold** (Ctrl+Shift+F) or by clicking the right mouse button while pointing to the block.
A block is commented by putting a comment character (`) in front of its frame. Deleting this character or pressing the same key again, restores the block to normal.
Blocks may be split and joined. By pressing **Enter** in front of a block frame, the block splits to two at the point of insertion. By deleting the line between two blocks they are joined together and form a single block.

Highlighting

In order to make it easier to work with the editor, and make the program's code more readable, the editor's normal text, language keywords, and comments, are all displayed in different colors. You can set these colors by using the menu command **Options:Editor**. Language keywords may also be displayed in bold face to make them more distinguishable.

Comments

Comments may be inserted anywhere in the code and are ignored when the program runs. A comment starts with the comment character (`) and is terminated either at the end of the line or by a second comment character on the same line. Thus, several comments may appear on the same line, each enclosed between two comment characters. Since the end of line is considered to be a closing comment character, the number of comment characters in a line may be odd, and the last comment is terminated at the end of line.

To convert all the text from some point to the end of the line into a comment, regardless of the comment characters that appear after this point, insert two successive comment characters (` `). This ensures that everything from that point to the end of the line is a comment. This is useful while debugging the program to comment out a line of code or a portion of it even when it includes comments (inserting a single comment character will start a comment that will be terminated by the next comment character in the line, and will not comment out the whole text that follows it).

To enter a multiline comment, put it in a block and comment the whole block. A whole block is commented by putting the comment character in front of it. The block goes back to normal when the key is pressed again, or when the character is deleted. Besides being the best way for entering a long comment, this is also very useful while debugging the program when you want to comment out portions of the code.

Comments are displayed in a distinctive color to make the program more readable. This color may be set by the menu command **Options:Editor**.

Undo/Redo

Operations in the editor may be undone. For example, a deleted text may be recovered. This is done by the menu command **Edit:Undo** (Ctrl+Z) or the Undo button in the Program Bar.

Numerit keeps a list of recent operations. This makes it possible to restore the editor to its state before each of these operations took place. Undone operations may be redone by the menu command **Edit:Redo** (Ctrl+Shift+Z), or the Redo button in the Program Bar. Thus, it is possible to move back and fourth through the operation list. New operations are inserted at the current position in the list, so operations that were made after this point are lost.

The size of the operation list may be set with the menu commands **Options:Editor**. The maximum size is 99 operations; the default is 25 operations. When the list becomes full the oldest operations are pushed out.

Some editing operations are grouped into one operation in the list. For example, when deleting successive characters, they are all restored at once.

Before undoing or redoing an operation, the cursor position is checked and if it is not where it was when the operation took place, it is first moved to that position, and only the next undo/redo command will actually undo/redo the operation.

Editor Options

The **Editor** command in the **Options** menu opens a dialog in which some of the Editor properties may be customized:

The font may be selected from a list of the available fixed pitch fonts (this font is also used in the Message pane.).

- The font size may be set. Note that some fonts may not be scaled.
- Language keywords may appear in bold text by checking the **Bold keywords** box.
- The number of undo levels may be set. The default is 25, the maximum is 99.
- Colors of normal text, keywords, comments, and the background may be selected.

When the **OK** button is clicked, the selected options are applied immediately to all the currently open programs and become the default options from now on.

When ***Numerit*** is closed, these options are saved and will be used in next sessions of ***Numerit***

Chapter 4

The Document

Introduction

Every program has two output documents, the Draft and the Report. The two documents allow two different approaches for displaying the program's output. The Draft is used as a scrolling output sheet to which the program sends output as if it was a printer, while the Report is a document in which the user embeds objects that display the value of program variables in different forms (e.g., tables, graphs). When the program is running, the Draft accumulates output, while the Report's content remains unchanged, and only the embedded objects are updated with new values. The Report and the Draft are both edited in the Document pane. The Document pane is actually a word processor with all the standard features plus special features that were especially designed for technical reports.

To print the Document's content use the **File:Print** menu command or click the Print button in the Program Bar when the Document pane is active. This will print the Report or the Draft, whichever is currently displayed in the Document pane.

The Report

The Report is brought into the Document pane by either selecting the Report tab at the bottom of the Document pane, or by checking the **Report/Draft** item in the **Window** menu (or clicking the Report/Draft button in the Program Bar).

Various objects (i.e., graphs, tables, pictures, equations) may be inserted into the Report by commands in the **Insert** menu. Output objects like graphs and tables are linked to the program's code and display the value of the program's variables. These objects are called Viewers. Viewers are updated whenever the program pauses, stops, or reaches a **`refresh`** instruction.

The report is an integral part of the program. It is saved on disk and is loaded from the disk together with the program. The report may also be saved as a separate document by the menu command **File:Save Report As**. In this case its is saved as a file of type "rep". ***Numerit*** automatically identifies this file as a document when it is opened.

When the Report is saved as a separate document it is saved together with the current data that is shown in its Viewers. When it is saved as part of the program, the Viewers are saved only with the names of the variables that are specified in them, and the variables' data is not saved, unless the Viewers are in the Freeze Data mode.

You can save the Report as a Rich Text Format file (see section below) which is a portable format that is recognized by most word processors. This allows you to continue editing the document in your favorite word processor.

When the Report is saved, its previous version on the disk (if exists) is kept as a backup with the character ~ added as the last character of the file name, leaving the extension unchanged..

The Draft

When the **Report/Draft** item is unchecked (or the button released) the Draft is brought into the Document pane.

The Draft is mostly used as an output sheet for the program's output statements (e.g., **`print`**, **`graph`**, **`table`**, etc.).

After the program sends its output to the Draft, it can be edited like any other document. But, the Draft is considered to be a temporary storage for text, and as such it is not saved with the program on the disk. It can, however, be saved as a separate document (using the menu command **File:Save Draft As**). In this case its is saved as a file of type "drf". ***Numerit*** automatically identifies this file as a document when it is opened.

The Draft is cleared either through the **Edit:Clear Draft** menu command or by the instruction **`clear`** in the program's code.

The output remains in the Draft until it is explicitly erased, so, one can accumulate the results of several runs. It is a good practice to clear the Draft when this information is no longer needed. If such accumulation is not required, it is better to put a **`clear`** instruction at the top of the program's code so that each run will first clear the old data.

You can save the Draft as a Rich Text Format file (see section below) which is a portable format that is recognized by most word processors. This allows you to continue editing the document in your favorite word processor.

When the Draft is saved, its previous version on the disk (if exists) is kept as a backup with the character ~ added as the last character of the file name, leaving the extension unchanged..

Rich Text Format (RTF)

As mentioned above, you can save a document (Report or Draft) as a Rich Text Format (RTF) document. The RTF specification provides a format for text and graphics interchange that can be used with different output devices, operating environments, and operating systems.

So, you can save your program's output document as an RTF file and continue the editing in your favorite word processor, such as Microsoft Word, without losing formatting or character attributes information.

To create an RTF file select the Rich Text Format type in the **Save As** dialog of the commands **File:Save Report As** or **File:Save Draft As**.

Notes:

1. ***Numerit***'s Graphs, Images, and Equations are converted to pictures (i.e., Windows metafiles) when saved in an RTF file.
2. ***Numerit***'s Tables are converted to standard RTF tables. However, two (or more) tables that are inserted side by side in the same paragraph in a ***Numerit*** document will appear in different paragraphs in the RTF file.
3. ***Numerit*** cannot read RTF files.

Document formatting

The printer's page size and other properties are used while formatting a document so that it will be printed properly. In most cases the displayed document looks exactly as it will look when printed. However, the printer properties determine the exact formatting and since the printer and the screen do not display all characters exactly the same way (in some fonts some characters have different relative sizes), there might be minor differences between the displayed document and the printed one. For example, a line may fit the current page width in the printer but appears to be too wide or too narrow on the screen.
The document page size is, by default, taken as the current printer's page size, and when this page size is changed (through the **Printer Setup** command) the documents are reformatted to reflect this change. It is possible to break this link by unchecking the box **Printer Page Size** in the Page Format dialog (invoked by the menu command **Format:Page**). In such a case you may set a different page size. Other formatting properties, however, are still taken from the current printer (e.g., characters sizes).

Page Formatting

Page formatting determines the appearance of the page. It may be set by the menu command **Format:Page**. The dialog fields are:

- Page size. Normally, the document is formatted according to the current setting of the printer page. So in order to change the printer page size or orientation use the menu command **File:Printer Setup**. It is possible however, to unlink the page size from the printer's page by unchecking the **Printer page size** box in the Page formatting dialog. If this case the page width and height may be set to any value.
- Margins. Each of the top, bottom, left, and right margins may be set. Note that the top and bottom margins are normally hidden. To make them visible check the menu item **Options:View Document Margins**.
- Header and Footer. A header is displayed at the top margin and a footer at the bottom margin of every page. To specify a different Header or Footer for odd and even pages, or for the first page, check the appropriate boxes. In addition, a space above the Header and below the Footer may be specified. To edit the header or the footer, make the top and bottom margins visible (check the menu item **Options:View Document Margins**) and move into the header or the footer.
- Pagination options: Avoid Widow lines, i.e., do not allow the last line of a paragraph to be the first line on the page (moves at least two lines to the next page), and Avoid Orphan lines, i.e., do not allow the first line of a paragraph to be the last line on the page (moves the whole paragraph to the next page).

Paragraph Formatting

Paragraph formatting determines the appearance of paragraphs. It may be set by the menu command **Format:Paragraph**. Some parameters may also be set in the Ruler or by buttons in the Document Bar. The dialog fields are:

- Alignment: Left, Center, Right, Justified.
  May also be set by buttons in the Document Bar.
- Indentations: From left - specifies the amount of indentation from the left margin. From right - specifies the amount of indentation from the right margin. First line - specifies the amount of indentation of the first line from the left margin. May also be set in the Ruler.
- Spacing between paragraphs and line spacing within paragraphs: Before - specifies the amount of space before the first line of the paragraph. After - specifies the amount of space after the last line of the paragraph. Line spacing - specifies the vertical space between lines.
- Tab stops: Left-aligned, Centered, Right-aligned.
  May also be set in the Ruler and by buttons in the Document Bar.
- Pagination options: Keep with next (do not insert a page break between the paragraph and the next paragraph) and Keep together (do not insert a page break inside the paragraph).

- When a new paragraph format is set while text is selected, it is applied to all the paragraphs inside the selection, otherwise, it is applied to the current paragraph only (where the insertion point is).
- A paragraph's format information is copied with the paragraph mark (at the end of the paragraph).
- Note that when a paragraph is formatted, the current page width is used. This is normally taken from the current setting of the printer so that when the document is printed the paragraph will be aligned properly in the page. The page width may be manually set to a different value using the menu command **Format:Page**.

Character Formatting

Character formatting determines the appearance of characters. It may be set by the menu command **Format:Character**. Except for the Color, all the parameters may also be set in the Document Bar. The dialog fields are:

- Font and Font size.
- Attributes: Bold, Italic, and Underline.
- Subscript and superscript.
- Color.
- When a new character format is set while text is selected, it is applied to the selected text, otherwise, it will affect the next characters that will be typed.

Paragraph Styles

It is possible to set a style to a paragraph. The style includes the default paragraph format settings, and the default character format settings for the text in the paragraph.

The style may be set by the menu command **Format:Style** or in the Document Bar. There are eight available styles. The first is the Normal style which is the default style. The second is the Main-Title style which may be used for titles. The third is the Sub-Title style which may be used for subtitles. The forth is the Caption styles which may be used for figure and table captions. The last four styles are User1, User2, User3, and User4, and may be used for any purpose.

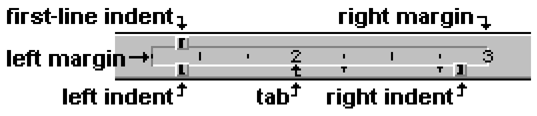
When a new style is set while text is selected, it is applied to all the paragraphs inside the selection.

Any style may be re-defined (i.e., its paragraph and character settings may be modified) by using the menu command **Format:Define Styles**. This menu command also lets you save on disk the current set of eight styles, or load from disk a set of previously saved styles. When a set of styles is saved in the main ***Numerit*** directory under the name DEFAULT.STY, it is used as the default set of styles from now on (deleting this file restores the original default set).

When a style is re-defined, all the paragraphs of that style in the document are reformatted.

The Ruler

At the top of the Document pane there is a ruler which provides quick setting for some of the document's parameters. It also contains a scale which shows dimensions in the current unit of measurement (which is either inch or centimeter, and is set by the menu command **Options:Set Document Units**).



- To change the document margins: point with the mouse at the left margin (or left to it) or at the right margin (or right to it) and drag it to the new position. This is equivalent to setting the margins with the **Format:Page** menu command.
- To change one of the paragraph indents (first-line, left, or right): point to the marker with the mouse and drag it to the new position. To move the left indent with the first-line indent together: point to the area between the markers, above the left indent marker, and drag them to the new position. This is equivalent to setting the indents with the **Format:Paragraph** menu command.
- To set a tab stop at a certain position: select the tab type in the Document Bar, point with the mouse to the lower part of the ruler, and drag the tab stop marker to the position. To remove a tab stop: point to its marker and drag it out of the ruler. This is equivalent to setting tab stops with the **Format:Paragraph** menu command.
- The Ruler may be hidden by unchecking the item **View Document Ruler** in the **Options** menu.

Character Map

The menu command **Tools:Character Map** (or the corresponding button in the Document Bar) opens a character map for the current font. This allows you to insert extended characters not found on most keyboards into the document. The character map also allows inserting symbols and other characters (e.g., Greek) whose key positions are not known to you.

Undo/Redo

Operations in the document may be undone. For example, a deleted text may be recovered, or a formatting operation may be reversed. This is done by the menu command **Edit:Undo** (Ctrl+Z) or the Undo button in the Program Bar.
Numerit keeps a list of recent operations. This makes it possible to restore the document to its state before each of these operations took place. Undone operations may be redone by the menu command **Edit:Redo** (Ctrl+Shift+Z), or the Redo button in the Program Bar. Thus, it is possible to move back and fourth through the operation list. New operations are inserted at the current position in the list, so operations that were made after this point are lost.
The size of the operation list may be set with the menu command **Options:Document**. The maximum size is 99 operations; the default is 25 operations. When the list becomes full the oldest operations are pushed out.
Some editing operations are grouped into one operation in the list. For example, when deleting successive characters, they are all restored at once.
Before undoing or redoing an operation, the cursor position is checked and if it is not where it was when the operation took place, it is first moved to that position, and only the next undo/redo command will actually undo/redo the operation.

Spelling Check

To check the spelling of the document use the **Tools:Document Spelling** menu command or the corresponding button in the Document Bar. This checks the document from the current position to the end of the document. To check the spelling of the word at the current position use the **Tools:Word Spelling** (Ctrl+W) menu command or the corresponding button in the Document Bar. If a misspelled word is found, a dialog opens and displays the misspelled word with the type of error and a list of optional replacements. Possible errors are: Word not in dictionary, Capitalization (one or more letters in the word need capitalization), and Abbreviation (an error in an abbreviation). The word shown in the **Replacement** field is the one that appears at the top of the list. The list is ordered according to similarity to the misspelled word. Selecting a word from the list copies it to the **Replacement** field. Double clicking a word in the list selects it and also substitutes it for the misspelled word in the document.
You may do one of the following:

- Click **Ignore** to skip the current word.
- Click **Ignore All** to ignore all the occurrences of that word during the current check.
- Click **Replace** to replace the word with the one that appears in the **Replacement** field.
- Click **Replace All** to replace every occurrence of the word with the specified replacement during the current check.

- Click **Add** to add the misspelled word to the personal dictionary. The word will be treated as correctly spelled from now on (until it is explicitly removed from the personal dictionary).
- Click **Add Correction** to add the word and the specified replacement to the personal dictionary. The specified replacement will replace the word from now on, every time the word will be checked (until it is explicitly removed from the personal dictionary).
- Click **Undo** to move to the last visited word and undo the last replacement.

It is possible to manually modify the word that appears in the **Replacement** field. In this case, clicking the button **Check Replacement** will check the modified replacement.

Personal Dictionaries

Personal dictionaries contain words that do not appear in the standard dictionary. Words are added to a personal dictionary by either clicking the buttons **Add** or **Add Correction** during a spelling check or by opening the dictionary using the **Tools:Dictionary** menu command. This opens a dialog for specifying the current personal dictionary. It also lets you add new words or remove existing entries from the dictionary. During a spelling check ***Numerit*** is looking in the currently open personal dictionary in addition to the standard dictionary. A word that appears in the dictionary without a correction next to it is considered a correctly spelled word; a word that has a correction will be replaced by the correction when checked.

Auto-numbered Captions

Graphs and tables are usually accompanied by captions. It is possible to insert caption headers (i.e., Figure nn or Table nn, where nn is a number) anywhere in the document (normally under graphs and above tables). It is also possible to insert a caption header for a reference anywhere in the document (references are normally inserted as a list at the end of the document). A reference header is just a number.
The relevant caption's text is typed in by the user next to the header. The headers are automatically numbered according to their position in the document relative to other captions. Whenever a caption is inserted or removed, all the captions are renumbered automatically according to the new order.
It is possible to refer to captions in the text by inserting a number field (a Figure number, a Table number, or a Reference number). The fields are also renumbered when captions are inserted or removed, so, a number field that refers to a specific caption will always refer to the same caption.
Captions are inserted by commands in the **Insert** menu.

Equations

Numerit supports two types of equations which are both created and edited by its built-in Equation Editor. The first type is a numbered equation which occupies a whole line and is inserted by the menu command **Insert:Equation** (or the Equation button in the Document Bar). The second type is an unnumbered equation which is inserted inside the line by the menu command **Insert:Expression**.
The Equation Editor is invoked by either selecting one of the menu commands to insert a new equation or by double-clicking an existing equation in the document.
Numbered equations are centered in the line and are automatically numbered according to their position in the document relative to other equations. Whenever a numbered equation is inserted or removed, all the equations are renumbered automatically.
It is possible to refer to a numbered equation in the text by inserting an Equation number field. The number fields are also renumbered when equations are inserted or removed, so, a number field that refers to a specific equation will always refer to the same equation.

Fields

A field is a code that is inserted in the document and displays a number that is related to some other object in the document. There are five types of fields: Figure number, Table number, Reference number, Equation number, and Page number. All are inserted through the menu command **Insert:Field** (F11).
In the text, we refer many times to figure numbers, table numbers, reference numbers, and equation numbers (e.g., "See Fig. 3 ...", or "In Eq. (5) ...", etc.). Instead of writing these numbers as normal text, it is possible to insert a field as a place-holder for entering the number. The inserted field is automatically linked to the figure caption, the table caption, the reference caption, or the equation whose number is entered. When captions or equations are inserted, removed, or moved, their number is automatically updated according to their new position in the document, and following this, all the fields that refer to them are also automatically updated.
For example, suppose we refer to Figure 3 in different places in the text, and another figure (actually a figure caption) is inserted before it. The caption header **Figure 3** will first be converted to **Figure 4** (caption renumbering), and then all

references to Figure 3 in the text will be converted to references to Figure 4 (field update). All the higher numbers will be also updated accordingly.
When a caption (or equation) is deleted from the text, the references to it are no longer valid. If the caption was cut (moved to the clipboard), it is considered to be temporary missing (as long as it is still in the clipboard) and all references to it are shown as question marks (?). When the caption is inserted back (pasted from the clipboard), its new position is calculated, its number is updated, and the references to it are updated according to the new number. If the caption was deleted (not cut) or is no longer available (clipboard was changed), the references to it are cleared and replaced by empty fields (only a place-holder).
Automatic field update may be disabled by unchecking the menu item **Options:Update Document fields**.
The fifth field type, Page number, displays the page number of the page on which it appears and is especially useful in the header and the footer.

Date

A date is inserted in the text by the menu command **Insert:Date**. The date may be inserted as either a text or a field. As a text, the date is inserted as a sequence of characters as if you typed it in. You may modify it and edit it as a normal text. When inserted as a field, the date appears as a single object which cannot be edited but is controlled by the system. In this case the date is linked to the system clock and is updated whenever the system date changes.
Use the first method when you want to keep the inserted date unchanged and the second method when you want it to always reflect the current date. In both cases you may select the format of the inserted date from a list.

Nonbreaking-space, Line-break, and Page-break

Nonbreaking-spaces are inserted between two words that you want to keep together so that they will not be separated at the end of line. To insert a nonbreaking space press Ctrl+Shift+Spacebar.
A line-break is inserted in a paragraph to end the current line and start a new line without starting a new paragraph. To insert a line break press Shift+Enter.
A page-break is inserted to force a new page regardless of ***Numerit***'s mechanism of dividing the text into pages which tries to fit as much text as possible on each page according to the current page size and margins setting. To insert a page break press Ctrl+Enter.

Nonbreaking spaces and line breaks are not visible normally. They become visible when the item **View Document Special characters** in the **Options** menu is checked, or when the corresponding button in the Document Bar in clicked. The page break is always visible. None of these marks are shown on the printed page.

Viewers

Viewers are objects that are inserted in a document for displaying program variables. There are currently four types of Viewers: the Variable Viewer, the Table Viewer, the Graph Viewer, and the Image Viewer.
The menu command **Insert:Viewer** (F11) opens a menu of Viewers. The Variable Viewer displays a scalar variable, Numeric, Staring, or Boolean as a simple formatted text, the Table Viewer displays variables as a table (each variable in a column), the Graph Viewer displays variables as a Y-X graph, and the Image Viewer displays a two-dimensional array as an image (each array element represents either intensity or color).
From the point of view of document editing, a Viewer behaves like a single character, and as such it is just part of the text.
Whenever the program pauses or stops, the Viewers are updated (if the variables listed in them have new values). It is also possible to explicitly update the viewers while the program is running by the command **refresh** in the program's code. This make it possible to display data in real-time, as it is changing.
All Viewers have default parameters that define their appearance (e.g., size, color, etc.). These parameters may be modified in the Viewer editing dialog by double clicking the Viewer.
It is possible to freeze the current value of a Viewer's variables so that further changes in the variables will not affect the Viewer. This is done by checking the **Freeze Data** box in the Viewer's dialog.

Variable Viewer

The Variable Viewer displays a scalar variable, Numeric, String, or Boolean, as a formatted text according to user's specifications. Inserting a new Viewer or double clicking an existing one opens a dialog in which the variable's **Name**, the **Width** (in characters), and the numeric **Format** are specified.
A numeric format specifies how a number is displayed. It consists of a **Format** type which may be either Auto, Decimal, or Scientific, and a **Precision** which is a number. When there is not enough room for the specified numeric format (width too small), the viewer displays asterisks instead of the number.
Auto is a free format that uses the precision to determine the number of significant digits that appear after the decimal point. In this format with a precision of 2, for example, the number 0.0348 is displayed as 0.035 (3 and 5 are the first two significant

digits), the number 3.412 is displayed as 3.41, and the number 3.004 is displayed as 3). Decimal and Scientific are fixed formats of the form 99.99 and 99.99×10^{99} respectively. Both use the precision as the number of digits that must appear after the decimal point (even if not significant).
When a Variable Viewer is used for displaying a String or a Boolean variable the **Format** and **Precision** fields are ignored and the only relevant field is the **Width**. However, you must note that although the width is specified in characters the resulted Viewer's width does not always fit the displayed string (even if the number of characters is correct) and you might have to adjust the width by increasing or decreasing the specified number of characters. The reason is that the actual width of the Viewer (in pixels) is independent of the content of the variable and is determined before this content is known. So, ***Numerit*** uses an average character size of the font at the Viewer's position to determine the actual width, but since most fonts are proportional (i.e., different characters have different widths) the final string might come out shorter or longer than the Viewer's width.

Table Viewer

The Table Viewer displays up to 16 columns of data, each corresponding to one program variable. The displayed variables may be of any type (numeric or string), and can be either scalars or one-dimensional. A row of a two-dimensional array may be specified with a dot followed by a number (e.g., x.2 represents the second row of x) and is treated as a one-dimensional variable. Note that in this case the specified row is shown as a column in the table. To display a column of x as a column in the table define the transpose of x.
Inserting a new Viewer or double clicking an existing one opens a dialog in which the variables names are specified in addition to other parameters that define the appearance of the Table Viewer. The variables are entered in a four column table. Each line of this table represents a column in the Table Viewer.
The specified parameters are the variable's name, the column title, the column width (in characters) and the number of padding spaces. Numeric data is aligned to the right and the specified padding spaces are inserted to the right of the number. Strings are aligned to the left and the specified padding spaces are inserted to the left of the string. The column title may be entered as a simple text or as a formatted title (see **Formatted Titles** below for details).
The first row in the Table Viewer is normally row 1 (which corresponds to the first element of the specified variables). However, it is possible to start from any other row by specifying its number in the field **First row**.
In addition, the number of rows in the Table must be specified in the field **Number of rows** so that the Viewer's size can be determined. If the variables that appear in the Table Viewer have already been computed, it is possible to find the maximum number of rows by clicking the button **Calculate** next to the **Number of rows** field.
Note that a Table Viewer is a single object and as such cannot be split between pages. If you need to display a long table you must change the page height accordingly (menu command **Format:Page**).
The other controls in the dialog are: the numeric format which is specified for each column, the font, the border, the background, the shadow, and the grids.
For details on the numeric format see the section **Variable Viewer**.

Graph Viewer

The Graph Viewer displays up to 16 traces inside the same frame, each corresponding to one or two program variables. Inserting a new Viewer or double clicking an existing one opens a dialog (the graph edit dialog) in which the variables' names are specified in addition to other parameters that define the appearance of the graph.

Traces Table

The traces are specified in a four column table. Each line of this table represents one trace. The first column is for the Y variable and the second column is for the X variable. The third column may specify an Error variable, which defines the size of error bars (peak to peak). The fourth column is used for a text which will appear as a label in the legend for this trace. A legend will be displayed by default if at least one of the legend labels is specified (only the traces for which labels are specified will appear in the legend).
The displayed variables can be either scalars, one-dimensional, or two-dimensional. A two-dimensional array is considered as one trace although all of its rows are drawn. If it is specified with a dot followed by a number, only the specified row is drawn (e.g., x.2 represents the second row of x).
When both the Y and the X variables are two-dimensional, each row of Y is drawn versus the corresponding row of X. If they don't have the same number of rows, the number of rows in Y is taken, and if there are not enough rows in X, its last row will be used for the rest of Y's rows. The same applies when the Error variable is two-dimensional.

Size

The default size of the graph window is 3×2 inch. It may be changed by specifying the width and the height in the graph edit dialog or by selecting the Viewer with a single click and dragging the borders.

Style
The **Style** controls allow you to customize the border, the background, and the shadow of the graph window.

Customization Dialogs
From the graph edit dialog you open other dialogs to customize the graph. This includes the scales, the titles, the axes, the tick marks, the tick labels, the grids, and the legend. The **Trace** dialog is used for editing each individual trace. The other dialogs refer to the whole graph. Note that in these dialogs there are two small buttons, labeled **All**, next to the standard **OK** and **Cancel** buttons. Clicking such a button accepts or cancels the current setting and closes all the dialogs. The **OK** and **Cancel** button themselves return you back to the previous level (the main dialog) so you can continue with other settings.

Trace
Each graph trace in a Graph Viewer may be customized by pointing to its line in the graph edit dialog and clicking the button **Trace**. This opens a dialog where various properties may be specified:

- Trace Type: Line, Step, or Bar. In the Line type - points are connected with straight lines. In the Step type - points are connected with steps, namely, a vertical line is drawn from the point to the Y position of the next point, and a horizontal line is then drawn from this position to the next point. In the Bar type - a bar is drawn on each point to the height that represents the Y value. The width of the bars is determined by the system according to the graph width, the number of bars, and the number of bar traces. Bars are drawn from the bottom axis. Normally, the bottom axis is at the minimum Y value, and the bars are drawn upward. If the axis position is not at the minimum (may be set by the button **Axes**), the bars are drawn on both sides of the axis, depending on the Y value.
- Line attributes in the **Line** group: Checking the **Show** Box will result in data points being connected by lines. The **Style** field determines which type of line will be drawn (solid, dashed, etc.). The **Width** field specifies the line width in mm. The **color** field specifies the line's color.
- Symbol attributes in the **Symbol** group: Available for the Line and Step trace types only. Checking the **Show** Box will result in a symbol appearing at each data point. The **Style** field specifies which symbol will be used (Circle, Diamond, etc.). The **Size** field specifies the size of the symbol in mm. The **Color** field specifies a color for the symbol. With closed shape symbols (e.g., Circle or Square) this color is used to fill the shape while the line color is used to frame it. With open shapes (e.g., Cross, Star) this color is used to draw the symbol itself.
- Fill attributes in the **Fill** group: Available for the Bar trace type only. Checking the **Show** box will define a fill for the bars of the bar graph. The **Style** field defines the fill pattern and the **Color** field specifies the fill color.
- Note that if the variable that is specified for a given trace is two-dimensional, all its rows are drawn with the same trace attributes.

Scale
The scales of the graph in a Graph Viewer are, by default, set to automatic mode. In this mode the X and Y scales are linear and are dynamically adjusted to the data that the graph displays. By clicking the button **Scale** in the graph edit dialog, it is possible to set a specific range for each scale, and to select it as linear or logarithmic.
In addition, the interval between the major tick marks (and tick labels) is set in the field **By**, and the number of sub-divisions between the major tick marks is specified in the field **Sub-Divisions**. Next to the field **By** there is an **Auto** checkbox. When checked, it instructs ***Numerit*** to adjust the interval between tick marks automatically even when the X and Y scales are set manually.
When the scale is logarithmic, the specified interval between major tick marks (the field **By)** is used as the ratio between two successive labels and must be a power of 10 (if set differently, it will be forced to such a value). The number of sub-divisions in this case may be 1,2, or 9 (other values will be forced to one of these). When the check box **Automatic** is not set, the scales remain fixed regardless of changes in the displayed data.
The scales limits may be set graphically. When the graph is selected (a single click) you can mark a rectangle inside the graph and the scales will automatically be set to fit this rectangle. To cancel the operation before releasing the mouse button, press the Esc key. To restore the previous settings after the mouse button is released, press Undo.

Titles
Each axis of the graph in a Graph Viewer may have a title. The button **Titles**, in the graph edit dialog, opens a dialog which allows you to specify the titles text and the font. The left and right axes' titles are written in a 90 degrees orientation (i.e., along the vertical axis). It is possible to show the left and right titles at 0 degrees orientation (i.e., same as the bottom title) by checking the **Upright** checkbox next to the title. The title may be entered as a simple text or as a formatted title (see **Formatted Titles** below for details).

Axes
Each axis of the graph in a Graph Viewer may be customized. The button **Axes**, in the graph edit dialog, opens a dialog the allows you to specify which of the axes will be shown, and for each axis, its position, line-width, and color.

By default all the axes are shown in the minimum and maximum positions (i.e., at the ends of the scales).

Tick marks

The tick marks on each axis of the graph in a Graph Viewer may be customized. The spacing between the tick marks is set in the scale specification. The size, line-width, and type of the major and the minor ticks may be specified by clicking the button **Ticks** in the graph edit dialog. The type of the ticks on each axis may be set to either ‘in’ (facing inside), ‘out’ (facing outside), or ‘cross’ (crossing the axis). Tick labels may appear next to each major tick. The type of the labels is set by clicking the button **Labels**.

Tick labels

Each major tick mark in a graph may be labeled by a number which identifies the value it represents. The labels may be customized by clicking the button **Labels** in the graph edit dialog. For each axis, the labels may be shown or hidden. Their numeric format may be set to Auto (default), Decimal, or Scientific (see the section **Variable Viewer** for details on the numeric format). In addition, a factor that multiplies the label values, may be specified for each axis (its default value is 1).
It is possible to hide the labels that appears near axes. Namely, in the bottom and top axes, the labels that appear near the left and right axes may be skipped, and in the left and right axes, the labels that appear near the top and bottom axes may be skipped. By default, the left and bottom axes are labeled, and the labels at the right and top axes are hidden.

Grids

Grids extend the tick marks across the graph. The button **Grids**, in the graph edit dialog, opens a dialog which allows you to define the grids. You may select a horizontal grid and/or a vertical grid. For each grid it is possible to specify if the gridlines pass through major and/or minor tick marks. The line style of the grid, its line-width, and its color may also be set.

Legend

It is possible to specify a legend label for each graph trace in a Graph Viewer. This is done in the same table where the trace's variables are entered. Only traces that are defined with legend label will appear in the legend. A legend will be displayed by default if at least one trace has a legend label. A legend may be hidden (even if legend labels are specified). To customize the legend, click the button **Legend** in the graph edit dialog.
There are two type of legends, Vertical and Horizontal. In a vertical legend the traces are listed vertically (top to bottom). In a horizontal legend the traces are listed side by side (left to right).
The legend's style (border, color, font, etc.) and position may also be set. The legend may be inside or outside the graph . It may be positioned at one of the corners or at the center of one of the sides. The position may be fine-tuned by specifying a vertical and/or horizontal shift in millimeters (positive or negative).
The legend text may be entered as a simple text or as a formatted title (see **Formatted Titles** below for details).

Image Viewer

The Image Viewer displays a two-dimensional array as an image. By default, the value of each element in the array is interpreted as the intensity of the image. The array is drawn from left to right and from bottom to top. Namely, each row of the array is drawn as a raw in the image from left to right, where the first row is drawn at the bottom of the image and the last row at the top of the image. The maximum value of the array is displayed in white and the minimum value in black. The other values are mapped linearly between these two values and are displayed in 256 levels of gray.
The user can control many of the default properties of the Image Viewer. Inserting a new Viewer or double clicking an existing one opens a dialog in which the name of the two-dimensional array variable is specified in addition to other parameters that define the appearance of the image.

Image Dialog Controls

X, Y

The **X** field allows you to specify the name of a one-dimensional array whose minimum and maximum values define the range of the X axis scale. Note that only the minimum and maximum values are used and all the other elements are ignored, so, it is sufficient to specify only two elements in the array. Similarly, the **Y** field allows you to define a scale for the Y axis. If no variable is specified in the **X** or **Y** field, the scale is taken from 1 to the number of elements in the corresponding dimension of the image array. If the variable that is specified in one of these fields is not valid (e.g., a scalar, a 2-d array, etc.) the scale is not drawn.

Color Table

The **Color Table** field allows you to change the default interpretation of the data in the image array. The default interpretation (if no variable is specified in the **Color Table** field) is to create an internal table by dividing the range that is defined by the minimum and maximum values of the image array to 256 equidistant values. The position (0 to 255) of the value of each

element in this table is used as the color of the drawn pixel, where 0 is drawn in black, 255 is drawn in white, and any integer between these values is drawn in a corresponding gray (from a very dark gray to a very light gray).
When you specify a name of a variable in the **Color Table** field the values of the image array are drawn differently, depending on this variable. The first option is to specify a *scalar variable* whose value is a positive integer smaller or equal to 256. In such a case the interpretation is very similar to the default interpretation; the only difference is that the number of levels is taken as the value of this variable. Thus, if the value of this scalar variable is 256 the result is equivalent to the default interpretation. See the sample program: image bw.num.
The second option is to specify a *one-dimensional array* in the **Color Table** field. In this case an internal table is created by dividing the range that is defined by the minimum and maximum values of the image array to N equidistant values, where N is the number of elements of the specified color table. The position (1 to N) of the value of each element in this table is used as an index to the color table and the pixel is drawn in the color specified at that position. The specified color must be an integer between 0 to 255, where 0 is drawn in black, 255 is drawn in white, and any integer between these values is drawn in a corresponding gray. Thus, the value of each element in the image array is actually a pointer to the color table. In this case, if the color table consists of 256 elements ranging from 0 to 255 the result is equivalent to the default interpretation (i.e., when no variable is specified in the **Color Table** field). See the sample program: image bw.num.
The third option is to specify a *two-dimensional array* in the **Color Table** field. In this case the color table must have three columns and any number of rows (i.e., a $N \times 3$ array). As before, an internal table is created by dividing the range that is defined by the minimum and maximum values of the image array to N equidistant values, where N is the number of rows of the specified color table. The position (1 to N) of the value of each element in this table is used as an index to the color table and the pixel is drawn in the color specified at that position. The color is specified as three integers (one in each column) of value between 0 to 255. The drawn color is a combination of the three basic colors Red, Green, and Blue, where the weight of each basic color is determined by the value in the corresponding column of the color table (first column is Red, second column is Green, and third column is Blue). As in the previous option, here too, the value of each element in the image array is actually a pointer to the color table. See the sample program: image color.num.

RGB

Next to the **Color Table** field there is a check box entitled **RGB**. When this box is checked the **Color Table** field is disabled and ignored. In this mode values of image array elements are interpreted as colors rather than pointers to a color table. Each element in the image array is constructed with the following formula: $2^{16} \cdot R + 2^{8} \cdot G + B$ (i.e., $65536 \cdot R + 256 \cdot G + B$). R, G, and B must be integers between 0 to 255, where R specifies the weight of red, G the weight of green, and B the weight of blue in the pixel's color. The **RGB** check box allows you to directly specify a color for each pixel and easily create true color images. When you define the color image array you must make sure that the values of R, G, and B are integers between 0 and 255 (you can use the **round** or **floor** functions to create integers). You can also read an image from a bitmap file and convert it to an image array in the Image Viewer RGB format using ***Numerit***'s **bitmap** function.
To manipulate the different colors of the RGB color image (after it has been defined by the program or read from a bitmap file) you may need to extract the values of each basic color, namely, to inverse the operation that defines the image array. This is achieved by defining the following functions in ***Numerit***:

```
func red(z) = floor(z/65536)
func green(z) = floor((z mod 65536)/256)
func blue(z) = floor(z mod 256)
```

where z is the image array. Each function returns a two-dimensional array, of the same dimensions as the image array, containing integer values between 0 to 255. Remember that when you modify the values of elements in these arrays you must make sure that they remain in the range 0 to 255 and that they are integers (by using the **round** or **floor** functions) before you sum them up to create the new image array. Reconstruction of the image array may be done by defining the following ***Numerit*** function:

```
func RGB(r,g,b) = 65536*round(r) + 256*round(g) + round(b)
```

where r, g, and b are the basic color arrays. Note that the functions `red()`, `green()`, `blue()`, and `RGB()` are not internal functions of ***Numerit*** and you must define them as ***Numerit*** functions if you want to use them (see the sample program: image rgb.num).

Orientation

The **Orientation** field has four options: 0°, 90°, 180°, and 270°. The image is rotated clockwise by the specified value. The scales are rotated with the image, so, if at the default orientation (0°) the horizontal scale is defined by the variable in the **X** field from left to right and the vertical scale by the variable in the **Y** field from bottom to top, in a 90° orientation the vertical

scale is defined by **X** from top to bottom and the horizontal scale by **Y** from left to right (note, however, that the tick labels are still drawn as specified in the **Labels** dialog, which by default is on the left and bottom axes; see below).
Below the **Orientation** field there are two check boxes, **Flip Horizontal** and **Flip Vertical**, that allow flipping of the image in the horizontal and the vertical directions. The scales are also flipped.

Zoom and Window Size

The **Zoom** field, inside the **Window Size** frame, allows you to set a magnification factor for the image. When you select this factor you can see the actual size of the window in pixels and in inches (or cm). Initially, ***Numerit*** sets a zoom value so as to get a window whose dimensions are as close as possible to 256 × 256 pixels. So, if for example the image size is 100 × 100 pixels, ***Numerit*** will choose a zoom value of 3 and will display a 300 × 300 pixels image. If the drawn image is not defined yet the zoom value is set to 1 and the window size is set to 256 × 256 pixels. When the image array is defined or changed and the option **Fixed** is not checked, the zoom value is adjusted to keep the viewer window size as close as possible to its current size. If the option **Fixed** is checked, the zoom value will remain fixed and the size will change as necessary. You may set any zoom value between 1/99 to 99. Pressing the **Reset** button will return the zoom setting to the initial value (i.e., making the size as close as possible to 256 x 256 pixels).
When the image dimensions or scales are changed during the run ***Numerit*** does not update the Viewer's window size immediately and waits for the next program pause or stop to do it. This may result in a temporary mismatch between the displayed image and the size of the Viewer's window (you can manually pause the program and resume execution if you want to readjust the window size).

Style

The fields inside the **Style** frame allow you to set the border thickness and a shadow for the viewer window.

Customization Dialogs

The buttons at the middle of the dialog box are used to open customization dialogs for defining the axes, the tick marks, the tick labels, the grids, the scales, and the titles. Note that in these dialogs there are two small buttons, labeled **All**, next to the standard **OK** and **Cancel** buttons. Clicking such a button accepts or cancels the current setting and closes all the dialogs. The **OK** and **Cancel** button themselves return you back to the previous level (the main dialog) so you can continue with other settings. The image customization dialogs are:

Axes

Each axis of the image in an Image Viewer may be customized. The button **Axes**, in the image edit dialog, opens a dialog to specify which of the axes will be shown, and for each axis, its position, line-width, and color.
By default all the axes are shown in the minimum and maximum positions (i.e., at the ends of the scales).

Tick marks

The tick marks on each axis of the image in an Image Viewer may be customized. The spacing between the tick marks is set in the scale specification. The size, line-width, and type of the major and the minor ticks may be specified by clicking the button **Ticks** in the image edit dialog. The type of the ticks on each axis may be set to either in (facing inside), out (facing outside), or cross (crossing the axis). Tick labels may appear next to each major tick. The type of the labels is set by clicking the button **Labels**.

Tick labels

Each major tick mark in an image may be labeled by a number which identifies the value it represents. The labels may be customized by clicking the button **Labels** in the image edit dialog. For each axis, the labels may be shown or hidden. Their numeric format may be set to Auto (default), Decimal, or Scientific. In addition, a factor that multiplies the label values, may be specified for each axis (its default value is 1).
It is possible to hide the labels that appears near axes. Namely, in the bottom and top axes, the labels that appear near the left and right axes may be skipped, and in the left and right axes, the labels that appear near the top and bottom axes may be skipped. By default, the left and bottom axes are labeled, and the labels at the right and top axes are hidden.

Grids

Grids extend the tick marks across the image. The button **Grids**, in the image edit dialog, opens a dialog to define the grids. You may select a horizontal grid and/or a vertical grid. For each grid it is possible to specify if the gridlines pass through major and/or minor tick marks. The line style of the grid, its line-width, and its color may also be set.

Scale

The scales of the image in an Image Viewer are, by default, set to automatic mode. In this mode the X and Y scales are set to the maximum range of the variables specified in the **X** and **Y** fields in the main dialog (see above) and the Image scale (the range of image values that are displayed) is dynamically adjusted to the data of the image array. By clicking the button **Scale** in

the image edit dialog, it is possible to set a specific range for each scale. If no variables are specified in the **X** or **Y** field the scale is set from 1 to N, where N is the number of elements of the image in the corresponding dimension.
When the X scale or Y scale are set manually, the image is displayed in the given range relative to the maximum range as specified in the **X** and **Y** fields of the main dialog. In such a case, the image magnification (see **Zoom** in Image Viewer) is set to a value that fits the specified range while trying to keep the viewer's window as close as possible to its current size. In addition, the interval between the major tick marks (and tick labels) is set in the field **By**, and the number of sub-divisions between the major tick marks is specified in the field **Sub-Divisions**. Next to the field **By** there is an **Auto** checkbox. When checked, it instructs ***Numerit*** to adjust the interval between tick marks automatically even when the X and Y scales are set manually.
When the Image scale is set manually (bottom field), the values of the image that exceed the higher scale value are clipped to this value and the values of the image below the lower scale value are clipped to this value. This allows you to map the whole color dynamic range (from black to white) to a limited range of image values. When the image is drawn in **RGB** mode (see above) the Image scale is set to the range 0 to 16777215 (the maximum available value in this mode) and cannot be modified.
When the check box **Automatic** is not set, the scales remain fixed regardless of changes in the displayed data.
The scales limits may be set graphically. When the image is selected (a single click) you can mark a rectangle inside the image and the scales will automatically be set to fit this rectangle. To cancel the operation before releasing the mouse button, press the Esc key. To restore the previous settings after the mouse button is released, press Undo.

Titles

Each axis of the image in an Image Viewer may have a title. The button **Titles**, in the image edit dialog, opens a dialog to specify the titles' text and font. The left and right axes' titles are written in a 90 degrees orientation (i.e., along the vertical axis). It is possible to show the left and right titles at 0 degrees orientation (i.e., same as the bottom title) by checking the **Upright** checkbox next to the title. The title text may be entered as a simple text or as a formatted title (see **Formatted Titles** below).

Formatted Titles

The Table, Graph, and Image viewers allow you to add titles to the columns in Tables, and to axes and legends in Graphs and Images. These titles have a default font type and font size and if entered as simple text are displayed using these default settings. The default font and size may be modified by the user.
In addition, you can embed instructions inside the text of the title to control some of its formatting and character attributes. These instructions are entered as control words. A control word consists of a backslash followed by a sequence of lowercase alphabetic characters and digits. The first space that follows the control word is ignored and is used as a delimiter of the sequence. Any characters following the first space, including spaces, will appear in the title.
The control properties of most control words (except \up*N*, \dn*N*, and \n; see below) have two states. When such a control word ends with 0 (zero) it turns off the property. For example, \b turns on Bold, whereas \b0 turns off Bold. Note, however, that the control words \sb and \sp that define subscript and superscript attributes are both set off by the control word \s0, so the sequences \sb0 and \sp0 do not exist (if entered, the 0 is interpreted as part of the subscript or superscript text).
The space delimiter may be omitted in many cases if the character that follows the control word cannot be interpreted as part of it. Take, for example, the control word \b that turns on the Bold attribute. The sequence \bXY displays **XY** (bold XY), and so does the sequence \b XY (where the space after the \b is ignored). In this case the space is not required since \bX cannot be interpreted as a valid control word sequence. The sequence \b01, on the other hand, displays a normal 1 rather than a bold **01**since the control word is interpreted as \b0 (bold off) and the 0 is not displayed. In order to display a bold 0 you *must* put a space delimiter after the \b. So, \b 01 displays **01** (bold 01). In case of a doubt always put a space after the control word.
The following control words are defined:

\b	Bold	\b0	Bold off
\i	Italic	\i0	Italic off
\u	Underline	\u0	Underline off
\g	Greek font	\g0	Default font
\sb	Subscript	\s0	Subscript/Superscript off
\sp	Superscript	\s0	Subscript/Superscript off
\up*N*	Shift baseline up by *N* half-points (*N* is a positive integer)		
\dn*N*	Shift baseline down by *N* half-points (*N* is a positive integer)		
\n	New line		

Notes:

- Subscript and superscript commands shift the text and also shrink its size. Both commands are turned off by the same control word, \s0, that restores normal text.

- \up*N* and \dn*N* are not turned on and off but rather define a new baseline for the text. So, for example, to restore the baseline to its previous position after a \up10 command put a \dn10 command. Note that \up or \dn that are not followed by an integer are ignored.
- There is only one level of subscript and superscript, so, a second \sp, for example, is ignored and the text remains at the same level of superscript (you can still use \up*N* and \dn*N* commands to get higher levels of subscript and superscript, but without further shrinkage of the font size).
- Remember, the commands \b, \i, \u, \g, \sb, and \sp, remain in effect until turned off. So, for example, if you write X\sp2 + Y\sp2 the result is $X^{2 + Y2}$ rather than $X^2 + Y^2$ which should be written as X\sp2\s0 + Y\sp2\s0. Note that the first space after \s0 in this example is ignored and interpreted as a delimiter of the \s0 sequence, so you must put a second space to see a space before the +.

Pictures

A picture may be inserted in the document by loading it from a disk file using the menu command **Insert:Picture** or by pasting it from the clipboard. Two types of pictures are supported: Bitmap and Metafile.
A picture that is loaded from a disk file may be kept in the document in two forms. The first is embedding the picture itself in the document. The second is inserting a link to the disk file by keeping only the file name in the document. If the second form is chosen then the picture file must exist in the same location on the disk in order to be displayed by the document.
A picture that is pasted from the clipboard is always saved inside the document (equivalent to the first form above).
Double clicking a picture, opens a dialog which allows to set a new size for the picture or restore the original size. It is also possible to change the picture size by dragging its borders when it is selected (use single click to select).

Document Options

The **Document** command in the **Options** menu opens a dialog in which some of the Documents properties may be customized:

- Default setting of various parameters may be set. These will be applied to new documents only. Parameters of current documents are set explicitly in each document.
- The number of undo levels may be set. The default is 25, the maximum is 99.
- Foreground, background, and fields colors may be selected.

When the **OK** button is clicked, the selected colors are applied immediately to all the currently open programs and become the default colors from now on. Default settings are applied to new programs only.
When ***Numerit*** is closed, the options are saved and will be used in next sessions of ***Numerit***.

Chapter 5

The Equation Editor

Introduction

Numerit's Equation Editor is invoked when a new equation is inserted or when an existing equation in the document is double-clicked.
The Equation Editor is actually a dialog window, so, clicking the **OK** button closes the dialog and inserts or updates the currently edited equation in the document, and clicking the **Cancel** button closes the dialog without updating the document. Note that when you click **Cancel**, all editing operations are lost (after your confirmation).
The Equation Editor's window consists of two parts: a control panel at the top and a display pane at the bottom.
The left side of the control panel contains buttons that are used for performing various editing actions. The right side consists of a set of dialog pages that are accessed through a corresponding set of tabs at the bottom of the panel. These pages are used for inserting templates and special symbols into the equation and for controlling the equation appearance.
The equation is edited in the display pane by typing characters and inserting templates or symbols from the control panel. The Equation Editor formats the equation automatically as items are added to the equation. It controls the position of items, the space between them, and the font of the typed characters.
When characters are typed or symbol are inserted, their attributes are determined by the current setting in the **Format** dialog page. To change the size or attribute after insertion, select the characters and set new values in the **Format** dialog page.

Equation Structure

An equation is constructed from blocks of items. The currently active block (the one with the insertion cursor) has a gray frame around it. A block may be split to two stacked blocks (one above the other) by pressing the **Enter** key inside the block.
When a new equation is inserted it has a single empty block, which is the root block. A multiline equation is created by splitting the root block as many times as necessary. The rows are aligned according to the Alignment setting in the **Format** dialog page. An empty block appears as a filled gray rectangle (when it is not the active block).
Blocks are usually created when inserting templates. Each template has a set of blocks which are placed by the Equation Editor in default positions. You may adjust the position of blocks by the **Shift controls** in the Equation Editor's control panel.

Editing the Equation

Blocks

Items are inserted into the currently active block of the equation by typing and by inserting templates and symbols from the Equation Editor's control panel. While inserting templates, the Equation Editor creates empty blocks which are place-holders for other items including other templates. Each block has a baseline which determines the vertical position of items in it. The insertion cursor has a small horizontal bar which shows the position of the active block's baseline.

Scale

The equation scale is determined by the base-size whose value is specified in points in the **Format** dialog page. By default, the base-size is taken from the current font size in the document where the equation is inserted and this is the normal size of characters and symbols in the equation. In many templates some of the blocks have smaller size (e.g., limits in an integral). When the base-size is changed, all other sizes are scaled accordingly so as to keep the same proportions in the equation. Note that changing the view scale does not affect the actual size of the equation, so, in order to obtain an enlarged equation (for presentation purposes for example) one must set a new value to the equation's base-size (this can be easily done by selecting the equation in the document and setting the font size).

Movement

To move from one block to another click with the mouse on the required block or move the insertion point with the keyboard's arrow keys. While you move between blocks, the Equation Editor surrounds the currently active block with a gray frame to make it clear to which block you are currently inserting items.

Display refresh

The equation display pane is refreshed constantly while you insert new items. It is also possible, however, to refresh the display without inserting items by pressing the **Esc** key (though this is normally not needed).

Modes
Usually you will edit the equation in the Math mode. In this mode the Equation Editor automatically defines the appropriate font for each item and controls its position. If you want to insert a normal text or a character of a specific font you may change to the Text mode.

Spacing
The Equation Editor spaces the items as necessary while you insert them. However, you may insert spaces of different sizes from the **Spaces** dialog page. You may also insert a tab when you want to ensure alignment of the text to a specific position. See Section **Spaces** below for more details.

Selection
To copy equation items to the clipboard or to modify their attributes they must be selected. Items are selected by either pressing the mouse button and dragging or by holding down the Shift key and pressing one of the arrow keys.
It is not possible to move out of the active block when selecting items.
To select a whole block double-click inside it. To select the whole equation, double-click anywhere in the area outside the blocks or press Ctrl+A.

Action Buttons
The Equation Editor's control panel contains buttons that are used for performing various editing actions. At the upper left corner of the panel there are two shift controls and underneath them three rows of buttons.

Shift controls
The items in the equation are automatically positioned while the equation is edited. The shift controls allow you to make adjustments to the position of specific items in the equation when you are not satisfied with the automatic positioning. Each shift control consists of two positioning buttons which surround a display field that shows the shift value (in screen pixels) relative to the default position. The upper control is used for horizontal adjustment and the lower control is used for vertical adjustment.
To shift a whole block, make it the active block and use the shift controls to move it. To shift items inside a block, select the items (with the mouse or the keyboard) and then use the shift controls to move the selected items. When nothing is selected, the shift controls display the shift values of the active block. When items are selected, the shift controls display the shift values of the selected items (a control becomes disabled when not all the selected items have the same shift).
Note that the shift is performed in screen pixels at the current scaling, so by scaling the view up you get a finer positioning grid. When the equation is printed, these shifts are translated to physical distances which match as closely as possible the distances shown on screen.
Shortcut keys for the positioning buttons: Ctrl+LeftArrow, Ctrl+RightArrow, Ctrl+DownArrow, and Ctrl+UpArrow.

Undo/Redo
The left button in the first row of buttons is the Undo button and the button next to it is the Redo button. The Equation Editor uses the same Undo/Redo mechanism that is used in the document, only that here the operations are undone or redone immediately when the buttons are clicked regardless of the insertion cursor position. The Equation Editor keeps a list of the recent 25 operations. The Undo and Redo buttons make it possible to move back and fourth through the operations list and restore the edited equation to its state before or after each of these operations took place. New operations are inserted at the current position in the list, so operations that were made after this point are lost.
Shortcut keys for Undo and Redo: Ctrl+Z and Ctrl+Shift+Z respectively.

Help
The third button in the first row shows the Equation Editor Help topic.
Shortcut key: F1.

Cut
The first button in the second row cuts the currently selected items. This deletes the selected items and copies them to the clipboard. See **Copy** below for available formats.
Shortcut key: Ctrl+X.

Copy
The second button in the second row copies the selected items to the clipboard. The information is sent to the clipboard in two formats, an internal format for use by the Equation Editor, and a Metafile picture for use by other Windows applications.
Shortcut key: Ctrl+C.

Paste
The third button in the second row pastes data from the clipboard to the current insertion position.
Shortcut key: Ctrl+V.

Scale up
The first button in the third row scales the view up. This does not affect the size of the equation in the document. The maximum scale is 4 times the equation's true scale.

Scale down
The second button in the third row scales the view down. This does not affect the size of the equation in the document. The minimum scale is the equation's true scale.

Greek
The third button in the third row lets you type Greek letters from the keyboard. Releasing this button restores the keyboard to English. Note that Greek letters may also be inserted from the **Greek** dialog page in the Equation Editor's control panel.
Shortcut key: Ctrl+G.

Math and Text Modes
The mode is selected in the **Format** dialog page. Most of the equation editing is done in the Math mode. In this mode the Equation Editor automatically defines the appropriate font for each item and controls its position. Variables appear in italics and spaces are automatically inserted around operators. In addition, functions names (e.g., sin, cos, etc.) are not italicized and so are long names which are assumed to be words rather than variable names (the maximum variable name length is set by default to 4 but may be set to any other value in the **Options** dialog page). It is possible to force the characters Italic and Bold attributes (see Format). The Italic check box has three possible states: Set, Not Set, and Auto. When the check mark in the box has a gray background, the state is Auto and then the Equation editor is responsible for italicizing the characters according to the above rule (this should be the normal state in the Math mode).
The Text mode is used for inserting normal text, or a character of a specific font. In this mode, the Equation Editor behaves like a word processor and you control the font and its attributes. Note that by inserting a template or a symbol while in Text mode, the mode at the insertion point is converted to Math.

Templates
Most of the editing activity in the Equation Editor involves inserting templates and filling in the template's empty blocks. The Template page in the Equation Editor's control panel lets you choose from a variety of templates. To insert a template click on it with the mouse.
The templates at the top row offer various combinations of parenthesis and brackets. They enclose one or two blocks and adjust their own size to the size of the enclosed blocks.
The second row includes various types of fractions, radicals, superscript, subscript, overscript, underscript, overbars, underbars, and two types of arrays, a normal size array and a small size array. The dimensions and elements alignment of the inserted array are defined in the array definition panel located at the right side of the Template page. This panel is also used to modify the size and alignment of existing arrays. To do this, click on one of the array's elements and make it the active block. This will turn the X sign between the dimension specifiers into a button and will display the dimensions in red. Clicking the button **X** sets the dimensions of the array to the specified values (by adding or removing rows and columns). Note that when removing rows, the last row and the last column are removed first and when expanding the array, rows and columns are added at the bottom and on the right side respectively.
The third templates row includes various types of sums, products, and labeled arrows.
The last two rows includes various types of integrals.

Symbols
Symbols are inserted from the **Symbols** dialog page of the Equation Editor's control panel. To insert a symbol at the current position in the equation, click on it with the mouse. Symbols may have any font size and also Italic and Bold attributes. When a symbol is inserted its size and attributes are determined by the current setting in the **Format** dialog page. To change a symbol's size or attribute after insertion, select it and change the size or the attribute in the **Format** dialog page.

Greek Letters

Greek letters may be inserted in two ways: 1. Click the Greek button in the Equation Editor's control panel (or Ctrl+G). When this button is pressed, Greek letters are typed with the keyboard. Releasing the button restores the keyboard to English. 2. Insert Greek letters from the **Greek** dialog page in the Equation Editor's control panel.
Greek letters may have any font size and also Italic and Bold attributes. When a Greek letter is inserted, its size and attributes are determined by the current setting in the **Format** dialog page. To change the size or attribute after insertion, select the text and change the size or the attribute in the **Format** dialog page.

Hats

Hats are symbols such as bars, arrows, or dots, that are inserted above characters. To insert a hat, move the insertion point after the character to which you add the hat, open the **Hats** dialog page in the Equation Editor's control panel, and click on the desired symbol. Up to four hats may be inserted above the same character. When more hats are added they replace the upper hat. The button **Remove** is used to remove the upper hat from the character on left.

Spaces

The Equation Editor spaces the items as necessary while you insert them. However, there are cases when you want to control the exact spacing. To do that, open the **Spaces** dialog page and click on one of the spaces in the list (or press the corresponding key combination).
When a space is inserted manually it replaces the automatic space that usually appears around operators (like +,=, etc.) and templates.
The automatic spacing mechanism may be disabled by unchecking the box in the **Options** dialog page.
The various spaces that may be inserted in the equation are of different sizes which are related to the current value of the base-size. When this value is changed, the spaces are scaled accordingly (see **Scale** in the section **Editing the Equation**).
Besides spaces, you may also insert a tab. This will align the inserted items to the closest tab stop. The distance between tab stops is equal to half the base-size (note that unlike the document, here you cannot set tab stops manually).
In order to see the spaces in the equation check the **Show Spaces** box. The space marks will not appear in the printed equation.

Format

The **Format** dialog page lets you determine the current mode, character attributes, and row alignment.

Mode

The Mode group lets you set the editing mode (Math or Text).

Font name

The first control in the Font group is the font name which becomes available only in the Text mode and makes it possible to select the desired font for inserting text. To modify the font of an exiting text select the text and enter the desired font name.

Font size

The second control is the font size field which is always available and lets you control the size of inserted characters and symbols. To modify the size of existing items, select the items and enter the desired value.

Base size

The third control in the Font group lets you set the base-size whose value, which is specified in points, determines the scale of the equation (see **Scale** in the section **Editing the Equation**). Normally the base-size is taken from the current font size in the document where the equation is inserted. You can change the base-size value by either changing the point size at the insertion point in the document, or you can break this link by resetting the check box "Inherit Base Size from document" in the **Option** dialog page and then change the value manually. Note that if the check box is set and the font size in the document is modified by either selecting the text and setting a new font size, or by changing the style, the base-size of all the equations in the range will be updated.

Attributes

The Attributes group has three controls: Italic, Bold, and Underline. In the Text mode each of the controls has two possible states: Set and Not Set. In the Math mode the Underline control is not available, the Bold control has two possible states: Set and Not Set, and the Italic control has three possible states: Set, Not Set, and Auto. The Auto state means that the Equation Editor is responsible for italicization (see Math and Text modes).

Alignment

The last group of controls in the Format page is the Alignment group. The controls in this group are used for specifying the alignment type for stack of blocks that are generated by splitting a block with the **Enter** key. Any block may be split to several rows of blocks that can be aligned in various ways.

Options

The **Options** dialog page lets you define several parameters for the currently edited equation. These parameters also become the defaults when inserting new equations. Existing equations keep their parameters and are not affected by the new setting.
The first two parameters are the name of the fonts that are used for normal characters and for Greek characters. Normally, in Windows, these are the Times New Roman font and the Symbol font, however, if there is a better choice it is possible to change the default fonts. Note that this is only related to the Math mode since in the Text mode the font is explicitly set in the **Format** dialog page.
The third parameter is the space between rows in a stack of blocks. This is given as a percentage of the base-size and its default value is 20%.
The next parameter is the maximum length of a variable's name. This is used in the Math mode to determine when to italicize a word and when not to italicize it. By default, this parameter is set to 4, so names that are 4 letters long or less are considered to be variable names and are italicized, while longer names are considered to be words and are not italicized. Note that function names are identified (the most common names) and are not italicized even when they are short. When you want to insert normal text regardless of words length, do it in the Text mode.
The last parameter is a check box that specifies whether to do auto-spacing or leave all the spacing to the user. This box is normally set, to allow the Equation Editor to automatically add spaces around operators and templates.

Chapter 6
Writing Programs

Writing the program

A new program is created by opening a Program window using the menu command **File:New** (Ctrl+N). An existing program is loaded from the disk by the command **File:Open** (Ctrl+O). Each Program occupies a window which has its own Editor that is used for writing the program's code, and two documents, the Report and the Draft, that display the program's output. Several programs may be open at the same time and run concurrently.
The program's code consists of a main body and functions. When the program is running the instructions in the main body are executed in order. Functions are declared by the **`function`** statement. Instructions in the main body and in functions may call any function that is defined inside the program.
In the Standard Edition of ***Numerit*** a program can only call functions that are defined inside the program itself. In ***Numerit Pro***, on the other hand, it is possible to call functions of other programs as well. See the section **Modules** in Chapter 7 for details.
The program's code is normally written one statement a line. The end of line terminates the statement but it is possible to continue a long statement into the next line by putting a backslash (\) at the end of line. It is also possible to put several statements on the same line and separate them by a semicolon (;).
Lines may be labeled. Labels allow to jump from one statement to another using the **`goto`** command. Too many **`goto`** statements and labeled lines tend to complicate the program, but in some cases the use of a **`goto`** is better than other alternatives and even simplifies the code. A labeled line consists of a label name (or number) followed by a colon and the instructions, for example:

```
 L1: x = 10
 L2: y = 20; x = y
 45: x = 25
```

Saving the program

The menu command **File:Save** (or the Save button in the Program bar) saves the program, including its code, the Report, and the name of the current Input File (if exists). The menu command **File:Save As** renames the program and then saves it under the new name.
In ***Numerit Pro***, the menu command **File:Save Module** (or the Save Module button in the Program bar) saves the code (source and compiled) of the module that is currently visible in the Editor. The menu command **File:Save Module As** saves a copy of the currently visible module under a new name. Note that unlike the **File:Save As** command, in this case the open module itself is not renamed and its original name is kept. This is done so because the module may appear as a sub-module in different levels of the modules list. So, if you want to continue working with the new module file you must change the module's name in the modules list manually, otherwise, you are still working with the original module (see the section **Modules List** in Chapter 7 for details on the sub-modules tree).
The shortcut key combination Ctrl+S is used to save the program, a module, or the Input File. Its interpretation depends on what is currently displayed in the Editor pane. When the Main module (i.e., the program's code) is displayed it is interpreted as a **File:Save** command; when a sub-module is displayed it is interpreted as a **File:Save Module** command; and when the Input File is displayed in the Editor pane, Ctrl+S is interpreted as a **File:Save Input File**.
When a file is saved, its previous version on the disk (if exists) is kept as a backup with the character ~ added as the first character of the file name.
Before a program is executed it is compiled (either by the user or by ***Numerit***; see Chapter 7). When a program or a module is saved, the compiled code is saved too, but only if it is updated (i.e., the program or module has not been modified after compilation). When a program and all its modules are saved together with their compiled code, the program is ready to run right after it's read from the disk (without compilation).
The command **File:Save** does not save the modules that are linked to the program. The menu command **File:Save All** (Ctrl+Shift+S) saves the program and all its open modules that have not been saved yet, including their source code and their compiled code; this is normally done after a Make or Run (see Chapter 7).

The language

Numerit's programming language is a high-level language that is based on conventional programming paradigm like popular programming languages (i.e., BASIC, C, FORTRAN, or PASCAL).

In addition to the versatility that such languages provide, ***Numerit***'s programming language offers many enhancements that make it much more powerful and easy to learn and use. Variables are defined by assignment with the right type and the right dimensions. Complex numbers are automatically created when necessary. Operators and functions (built-in as well as user-defined) work with multidimensional arrays the same way they work with scalars. Most array operations are performed without the use of loop statements (but a wide variety of loop types is still available).
In addition, the language provides many functions for data analysis, including: line and polynomial fitting, linear and cubic-splines interpolation, multidimensional fast Fourier transform, numerical integration and differentiation, root finding, solution of linear algebraic equations, matrix/vector arithmetic, special functions, and more.
Some of the basic concepts in the language are described below:

Identifier

An identifier is a name that is used for designating a variable or a function. A legal identifier consists of a sequence of letters and digits but must start with a letter. The underscore character `_` is considered a letter. The number of characters in an identifier cannot exceed 64.

Type

Every variable or constant has one of the following four types: Numeric, String, Boolean, and File. Different types cannot be mixed in expressions and a variable cannot change its type during the run.
The Numeric type is divided into two sub-types Real and Complex. Real and Complex may be mixed and a Complex is automatically created when necessary. There are some cases where only a Real is acceptable, especially as a parameter in some built-in functions.
A complex number is specified as a sum of a real number and an imaginary number, e.g., `2+1i`, `3+4i`, etc. (`j` may be used instead of `i`). To create a complex variable `c` from two real variables `a` and `b` write: `c = a + 1i*b`.
Strings are formed by a sequence of characters and are used for titles, file names, etc. a String constant must appear between quotes (`"..."`). Strings may be concatenated by the operator **`&`**.
Boolean expressions may have only two values **`true`** and **`false`**. The Boolean type is obtained from comparisons of Numeric, String, or Boolean expressions or by a direct assignment of one of the two Boolean constants **`true`** and **`false`**.
Booleans may form logical expressions with the operations: **`and`**, **`or`**, **`xor`**, and **`not`**.
File variables are defined by the commands **`file`** and **`binfile`**. See the section **Files** for more details.

Kind

Every variable has one of the following four kinds: Scalar, Array, Vector, and Matrix. The kind is inherited upon assignment, so, a variable's kind changes during the run when it is assigned a value of a different kind.
Most operations ignore the kind. Some operations are affected by the kind, for example, a multiplication of two matrices is not carried out on matching elements as with simple 2-dimensional arrays, but according to matrix multiplication rules.
By default, a variable's kind is Scalar or Array (depending on the dimension). Vectors and Matrices are explicitly defined as such, or inherit the kind by assignment.

Expression

An expression is: a variable, a constant, a function call of a function that returns a value, or any combination of these with operators.
An expression has a specific type and kind and usually appears at the right side of an assignment or as an argument to a function.

Dummy Variable

A dummy variable is designated by the character `$` (dollar). It is actually a placeholder for a real variable. Expressions that contain a dummy variable are called formal expressions and are not evaluated when they are specified but are sent as arguments to functions that evaluate them by replacing the dummy variable with real values.

Formal Expression

A formal expression is an expression that contains at least one dummy variable which is represented by the character `$` (dollar). Such an expression is sent to a function as an argument (see for example, Integ), and is evaluated inside this function by replacing the dummy variable with actual values. The other variables and constants in the formal expression are used with their current values.
The dimension of the value that is returned when the formal expression is evaluated must be the same as that of the dummy variable in this expression. This will normally be the case if the expression consists of ordinary operations that involve the dummy variable. Make sure that this dimension relation is also kept when the formal expression calls a user-defined function. Such a function should not set a specific dimension to the returned value but rather inherit the dimension of the dummy variable sent to it as an argument.

Note that while a formal expression is evaluated the program is busy and does not respond to external events. This means that the other programs are temporarily put on hold and that user action is not processed. Normally, a single evaluation takes a very short time so this effect is not noticed, however, if the expression contains a call to a user-defined function, and this function takes a long time to complete, it looks like the program is stuck. To interrupt the process in such a case press **Esc**.

Arrays

Arrays are variables that have dimensions. An array may have up to 8 dimensions and may be of type Numeric, String, or Boolean.

Array definition

An array is defined using the following syntax:

```
x[m,n,...]:v
```

Where `x` is the array's name, `m,n,...` the number of elements in each dimension, and `v` the initial value that is set to all the array's elements. The type of `v` defines the type of the array. For example, the instruction

```
x[3,3]:1
```

defines `x` as a 2-dimensional 3×3 Real array with all its elements set to 1; the instruction

```
z[2,2,2]:0+0i
```

defines `z` as a 3-dimensional 2×2×2 Complex array with all its elements set to 0; and the instruction

```
s[7]:""
```

defines `s` as a 1-dimensional String array, with 7 empty strings.
If the array already exists, the array definition instruction redefines it with the newly specified size and initialization.
Another way to define an array is to assign a value to its last element, assuming it's not defined yet. For example, if `x` is not defined than `x[10,10,10] = 0` defines a 3-dimensional Real array with 10 elements in each dimension. Note that in this case the array is defined using its last index rather than the number of elements. If the assigned value is not 0 and the array does not exist, only the specified element is set to the given value while the other elements are 0; if the array exists, all the other elements remain unaltered. Moreover, you must be aware of the fact that if arrays do not start with index 1 (by setting **`firstindex`** to a different value) the last index is not equal to the number of elements.

Arrays initialization with lists and ranges

An array may also be defined by assigning different values to its elements. For a 1-dimensional array this is done by specifying its elements as a list of numbers (or strings, or Boolean constants) separated by commas, for example, `x = 1.1,2.5,3.6` or `x = "red","green","blue"`. It is possible to put a 1-dimensional array instead of one of the numbers in this list and thus merge 1-dimensional arrays into one array. For example, if `x` is defined as above then `y = 1,2,x,3` defines `y` as: `1,2,1.1,2.5,3.6,3`.
One-dimensional Numeric arrays may also be defined by using one of the range statements, e.g., `x = 0` **`to`** `10`. A range enclosed in parenthesis may also be specified in a list that defines a 1-dimensional array. So, for example, the following definition is legal: `x = (0` **`to`** `10),15,(20` **`to`** `30).`
For arrays of higher dimensions you must use blocks. For example, a 2-dimensional 3×3 array is defined by the following statement:

```
x =
  ┌
  │11,12,13
  │21,22,23
  │31,32,33
  └
```

and a 3-dimensional 2×4×3 array is defined by:

```
 x =
 ┌
 │ ┌
 │ │111,112,113
 │ │121,122,123
 │ │131,132,133
 │ │141,142,143
 │ └
 │ ┌211,212,213
 │ │221,222,223
 │ │231,232,233
 │ │241,242,243
 │ └
 └
```

Arrays of higher orders are analogously defined with higher levels of blocks.
These definitions determine both the size of the array and the value of its elements. The length (number of elements) in a specific dimension is determined by the longest list along this dimension. For example, if in a definition of a 2-dimensional array one row has 3 elements while the others have 1 and 2 elements, the length in this dimension is set to 3 and the missing elements in the other rows are set to 0. Thus the definition:

```
 x =
 ┌11,12
 │21,22,23
 │31
 └
```

Yields the following 3×3 array:

```
 11,12, 0
 21,22,23
 31, 0, 0
```

Dynamic length

A length of an array is dynamically expanded when a value is assigned to an element with index that is greater than the length. For example, if `x` is a 1-dimensional array with 10 elements, then the statement `x[20] = 1` will expand the array and set element no. 20 to 1. This applies to any dimension of the array.

Dimensions

The leftmost dimension of an array is the first dimension, the next is the second, and so on. When defining an initialized array whose dimension is greater than 1, the first dimension is along the leftmost (higher level) block, the second is along the next level of blocks and so on. The last dimension is along the row, from left to right. For example, in the 3-dimensional array above, the element `x[2,4,3]` is the third element in the fourth row of the second block (with value 243).
The number of dimensions of an array is obtained by the built-in function **`dim`**. The length in each dimension is obtained by the built-in function **`length`** which for a multidimensional array returns a 1-dimensional array of lengths.
All the operations and functions are performed on variables of any dimension. Operations that involve two variables (e.g., addition or multiplication) are done on matching elements in both operands, so, the operands must have the same number of dimensions (and, of course, the same type). There are two exceptions to this rule: 1. Scalars may be involved in expressions with arrays of any dimension. In such a case the operation is carried out between the scalar and each of the array's elements.
2. Outer operations take two 1-dimensional arrays and produce a 2-dimensional array whose elements are combinations of each element of the first operand with each element of the second operand. In may cases, outer operations may be used instead of two-dimensional loops (which are much slower).
The number of dimensions is inherited upon assignment, thus, assignment of a 3-dimensional array into a 1-dimensional variable redefines it as 3-dimensional.
When two arrays are involved in an operation, the length in each dimension of the result is the maximum of the two. Missing elements in the shorter array are considered to be zeros. These virtual zeros behave like true zeros so that a division by such an array, for example, will generate a 'division by zero' error.

Minimum and maximum index

The first index in each dimension is by default 1. It can be set to another value by the command **`firstindex`**, for example, the instruction **`firstindex`** `0` will set the first index to 0 from now on. The maximum index is shifted by the same amount. The maximum number of elements that may be specified in each dimension is about 500,000,000 but in reality the amount of

free memory limits the size of arrays. A 2-dimensional array of 10000×10000 elements, for examples, requires about 800 MB of free memory.
Note that when all the physical memory of the computer is used up the system allocates more space from the virtual memory (taken from the disk), so working with very large arrays might degrade the performance significantly.

Sub-Arrays

Array elements are accessed with indices. A simple index, like `x[2,1,3]`, extracts one element (a scalar) from an array. However, it is possible to extract sub-arrays from a given array by using indices that are themselves 1-dimensional arrays. For example, the following code:

```
x = 10,20,30,40,50,60,70,80,90,100
i = 2,5,8
y = x[i]
```

defines `y` as a 1-dimensional array with elements 20,50,80.
The index may also be specified by using a range directly (must appear inside parentheses), so

```
y = x[(4 to 7)]
```

defines `y` as a 1-dimensional array with elements 40,50,60,70.
If `x` is a 2-dimensional array, then the code

```
x =
⎡11,12,13
⎢21,22,23
⎣31,32,33
i = 1,2
j = 2,3
y = x[i,j]
```

defines `y` as the following 2-dimensional array

```
12,13
22,23
```

Here again you can use a range directly, so the code:

```
y = x[i,(1 to 2)]
```

defines `y` as a 2-dimensional array, given by

```
11,12
21,22
```

It is also possible to extract sub-arrays with less dimensions. For example, `y = x[i,i]` (same index variable for both indices) defines `y` as the 1-dimensional array 11,22 taken from the diagonal of `x`. Or, another example, if `x` is the 3-dimensional array that was defined before, then the following code

```
y = x[i,2,j]
```

(where one index is a scalar) defines `y` as a 2-dimensional array, given by

```
122,123
222,223
```

The opposite assignment is also possible, namely, `z[i,2,j] = y` defines `z` as a 3-dimensional array in which the elements in the positions specified by `i` and `j` are taken from `y`. Or `x[i,i] = 1,2` sets `x[1,1]` to 1 and `x[2,2]` to 2. For such assignments to be valid, the effective dimensions of both sub-arrays must match.
It is possible to extract all the elements in a specific dimension by using `*` (asterisk) as the index, for example, `y = x[2,*]` defines `y` as a 1-dimensional array by extracting all the elements in the second row of `x`, and `x[*,2] = 3,4` defines `x` as a 2×2 array with 3,4 in its second column.

Vectors and Matrices

Vectors and matrices are special cases of one-dimensional and two-dimensional arrays. They are created by the declarations **`vector`** and **`matrix`**. For example, **`vector`** `v[3]:0`, creates a vector with 3 elements, and **`matrix`** `m[3,3]:0`, creates a matrix with 3 rows and 3 columns. When a variable is a vector or a matrix we say that its kind is Vector or Matrix (as opposed to the Array kind).
Vectors and matrices obey the matrix/vector arithmetic rules. When a vector multiplies a matrix from left (vector · matrix), it is taken as a row vector and its length must match the number of rows in the matrix. When the matrix is on the left (matrix · vector), the vector is taken as a column vector and its length must match the length of the matrix row.

When multiplying two matrices, the length of the left matrix row must match the number of rows of the right matrix. The multiplication of an N×M matrix (N rows M columns) by an M×K matrix produces an N×K matrix. A square matrix can be raised to an integer power (positive or negative). Raising a square matrix to a positive power `n` is equivalent to multiplying the matrix `n` times. Raising a square matrix to a negative power `-n` is equivalent to multiplying the matrix `n` times and taking the inverse of the result. Raising a square matrix to the power 0 returns the identity matrix.
Multiplication of two vectors is carried out as a scalar product, so their lengths must match and the result is a scalar.
Operations that involve matrices or vectors with regular arrays are performed as regular array operations (element by element). The kind of the result is inherited from the first operand, namely, matrix + array yields a matrix while array + matrix yields an array. It is possible to convert an array to a vector or a matrix by explicit declarations. For example if `v` is a 1-dimensional array, the statement **`vector`** `v` converts it into a vector. Similarly, 2-dimensional arrays may be converted to matrices by the command **`matrix`**. The opposite conversion is also possible, namely, vectors and matrices may be converted to regular arrays by the command **`array`**. Several functions are available for doing matrix analysis like **`Det`**, **`Inv`**, **`Trace`**, and **`Svd`**.

Outer operations

Outer operations operate on 1-dimensional arrays and produce 2-dimensional arrays. The `[i,j]` element of the result is the outcome of the operation on the i'th element of the first operand and the j'th element of the second operand. Outer operations are defined for the five basic operators: addition, subtraction, multiplication, division, and power. They are specified by putting the corresponding operator inside parenthesis.
For example, if `x` and `y` are 1-dimensional arrays of lengths N and M respectively, then `x(+)y` produces an N×M array whose `[i,j]` element is `x[i]+y[j]`.
Using an outer operation is much faster than using ***Numerit***'s loops to perform the same operation.
Notes:
1. If `x` and `y` are defined as vectors the result is a matrix.
2. If either `x` or `y` is a scalar, the outer operation is interpreted as the corresponding standard operation.

Strings

In ***Numerit*** a string is a variable of type String; it is not an array of characters. As with any other variable, you can define an array of strings of any dimension.
Strings may be concatenated by the operators `&` and `&=`. In addition, ***Numerit*** provides various functions for manipulating strings. These functions also accept an array of strings as an argument.
`strlower``(s)` (or **`strlow`**`(s)`): Returns a version of the string where all upper-case characters have been replaced with lower-case characters.
`strupper``(s)` (or **`strupp`**`(s)`): Returns a version of the string where all lower-case characters have been replaced with upper-case characters.
`strlen``(s)`: Returns the length of the string.
`strlen``(s,n)`: Sets the length of the string `s` to `n` (expands or shrinks as necessary).
`strpos``(s,t)`: Returns the position of the substring `t` in the string `s`. The position of the first character in `s` is `0`. If the substring is not found in `s` the returned value is `-1`.
`strsub``(s,p,n)` (or **`substr`**`(s,p,n)`): Returns a substring of `s` with length `n` starting at position `p`. The position of the first character in `s` is 0. If the third parameter `n` is omitted, the substring from position `p` to the end of the string is returned.
`strdel``(s,p,n)`: Deletes `n` characters of `s` starting at position `p`. The position of the first character in `s` is 0. If the third parameter `n` is omitted, the characters from position `p` to the end of the string are deleted. Note that this function does not return a value but rather operates on the string itself.
`strins``(s,p,t)`: Inserts the string `t` into `s` at position `p`. The position of the first character in `s` is 0. If `p` is greater than the length of `s`, the string `t` is appended to `s`. Note that this function does not return a value but rather operates on the string itself.
`strtonum``(s)`: Converts the string `s` to a number and returns its value.
`numtostr``(x)`: Converts the value of the numeric expression `x` to a string and returns the string.

In addition to these functions there are commands for reading and writing from and to strings (similar to files):

`read`	Reads data into variables from a String.
`write`	Writes data to a String.

Files

Numerit supports two types of files for input and output of data: a text file which contains information as a formatted text, and a binary file which contains information in binary representation.

Text files are used when the data they contain should be displayed, printed, or transferred to other applications. Binary files are used when the full precision of the data should be maintained, and are usually more compact.

Files are defined as variables of type File in the program's code by the instructions

`file` `f = "file-name" -` for a text file,

or

`binfile` `f = "file-name" -` for a binary file.

`f` is the variable and the file's name is specified as a string.

The file's name may contain the full path-name of the file, including the drive (e.g., `"c:\data\my.dat"`). If a full path-name is not specified the current working directory is used. The working directory is normally the directory where ***Numerit*** was installed but this may be changed by the operating system. You can set the working directory in the program using the command **`directory`** (see Chapter 14: **Reference to *Numerit*'s commands**).

After the file is defined it is still closed. A file is automatically opened the first time it is actually accessed in the program (e.g., when reading from it or writing to it). When a file is no longer needed it is a good practice to close it, by the instruction **`close`** `f`. This ensures that the file is updated in the disk and frees system resources that are associated with the file. The next access to the file will open it again and set the position at the beginning. Files are automatically closed at the end of the run (so, for short runs there is no need to close files explicitly).

Files in ***Numerit*** are opened for both reading and writing. If the first file operation after the file is defined or is closed (the one that actually opens the file) is writing to the file (e.g., **`write`**, **`writeln`**, **`writetab`**), and the file contains data, the data is first cleared and the position is set to the beginning of the file. To keep the old data, you must use the **`pos`** command to set the current position in the file before writing. The data in the specified position will be overwritten but the rest of the data in the file will remain untouched. Be careful when you overwrite existing data, replace it with matching data so that the rest of the data in the file will not be damaged. To append data to the file, first move the current position to the end by the instruction **`pos`** `f -1`, and then write the data. It is possible to explicitly clear the file's content by the instruction **`clear`** `f` (where `f` is the file variable). This also moves the current position to the beginning. Note that writing to the file after closing it, will first clear its content, as described above.

When a file variable is redefined (e.g., with a new name, or even with the same name) and the file is open, it is closed before the new definition takes place.

When a file variable is freed by the **`free`** command, the file is closed and is no longer available to the program (unless it is defined again). A file may be deleted from the disk by the instruction **`delete`** `f`.

The commands **`fopendlg`** and **`fsavedlg`** open Windows standard dialogs for selecting a file or specifying a new file name. This is a useful tool to allow the user to control the names of the files that are read or written by the program.

There are several commands that allow you read from files and write to files (see Chapter 14: **Reference to *Numerit*'s commands** for details):

`read`	Reads data into variables from a text or binary file.
`readln`	Reads lines of data into a string variable from a text file.
`readtab`	Reads data into variables, as a table, from a text file.
`write`	Writes data to a text or binary file.
`writeln`	Writes data to a text file and adds a new-line character.
`writetab`	Writes data to a text file as a table.

There are also functions that return information about files:

`size``(f)`	Returns the file size in bytes.
`pos``(f)`	Returns the current position in the file (in bytes).
`exists``(f)`	Returns **`true`** if the file exists and **`false`** if it doesn't.
`eof``(f)`	Returns **`true`** if the end of file has been reached and **`false`** if not.

Notes:

1. When writing data to a text file, the format is controlled by the current setting of **`outwidth`**, **`outprec`**, **`outformat`**, and **`outzero`**.
2. When writing data to a binary file, the exact representation is:
   Number - written and read as a double precision floating point number (8 bytes).
   String - written and read as a sequence of characters (1 byte each).
   Boolean - written and read as a byte.

3. A complex number is written and read with the real part first and the imaginary part next to it, as two distinct numbers.
4. When reading numeric data from a text file anything that appears between the numbers is skipped (actually is considered to be a delimiter) except the characters: `- + .` (minus, plus, and dot) which cannot be used as delimiters and will cause a read error if they are not part of a number. Strings should be separated by spaces unless they are read by readln.

Input File

A ***Numerit*** program can read data from a disk file and in many cases this is a convenient way to define parameters for the program. However, in some cases it is desired to modify the parameters from run to run and doing that with a disk file requires opening the file, making the modifications, and saving the file before running the program. In order to shorten this procedure ***Numerit*** allows you to open an Input File in the Editor pane and read data from this file directly.
Reading from the Input File is similar to reading from an ordinary text file and the same instructions can be used, namely, **`read`**, **`readln`**, **`readtab`**, and **`pos`**, only that **`input`** should be specified instead of a File variable. For example, to read the two variables `x` and `y` from the Input File, write: **`read input`** `x,y`. The instruction **`pos input`** should be followed by two numeric parameters that set the current position in the Input File to the specified line and column; the next read will start at this position. For details on these commands see Chapter 14: **Reference to *Numerit*'s commands**.
The menu command **File:New Input File** opens an empty Input File or clears the Input File pane if a file is already open. The menu command **File:Open Input File** opens a disk file and copies its content to the Input File pane. The menu commands **File:Save Input File** and **File:Save Input File As** save the current Input File as an ordinary text file on the disk. When the program is saved, the name of the current input file is also saved and this file is opened again as the Input File the next time the program is opened. If the input file and the program file are in the same directory, the name of the input file is saved without the full path and it is assumed that it resides in the same directory as the program. This allows you to move the program and input file from one directory to another, or from one computer to another, and when the program is opened it can find and open the input file. This is especially useful when you write a program with an input file for someone else without knowing in advance where it will be installed.
Note that when the Input File is displayed in the Editor pane, the shortcut key combination Ctrl+S is interpreted as a **File:Save Input File**. When a file is saved, its previous version on the disk (if exists) is kept as a backup with the character ~ added as the first character of the file name.
Note that the Input File pane can be used as an ordinary text editor for editing text files. So even if you prefer to read data from disk text files, rather than directly from the Input File, you can still use it to open, edit, and save these files.
The Input File also exists in ***NumRun***, the package that executes compiled ***Numerit*** programs outside the ***Numerit*** environment (See the section **NumRun: The Numerit Executor**). This allows users that run ***Numerit*** programs with ***NumRun*** to easily modify program input parameters.

User-Defined Functions

Functions let you execute the same piece of code many times and with different parameters. A function is defined by the declaration **`function`** (or **`func`**) followed by the function's name, the parameter list inside parentheses (which are empty when there are no parameters), and the function's body. A function definition may appear anywhere in the program (even after the statement that calls it) but only at the top level, namely, it cannot appear inside a block of another function or another command (i.e., **`if`** or **`for`**).
There are two possible forms:
1. Single line. The function header is followed on the same line by an equal sign and an expression which will be evaluated and returned when the function is called. For example:

```
function foo(a,x) = a*x^2
```

2. Multiline. The function header is followed on the next line by a block of statements. For example:

```
function foo(a,x)
 y = a*x^2
 return y
```

A function's name may be any valid identifier and the parameter list consists of identifiers that are separated by commas.

Type of arguments and the return value

The function receives arguments and may return a value. Since the language is type sensitive it determines the type of the arguments and the return value during compilation, so it is not possible to call the function once with an argument of one type, and then with a different type. However, other variable attributes (kind and number of dimensions) may change from call to call; thus, the same function automatically works with variables of different dimensions and kinds (just like the built-in functions).

Arguments: "by reference" or "by value"

A function may return a value or modify the arguments sent to it. Normally, a variable that is sent as an argument to a function is sent by reference, namely, the function receives the variable itself and not a copy of it, and can change its value; however, the function itself does not care how the variable is sent, and there is no need to specify if a parameter is "by reference" or "by value" when the function is defined. This is determined when the function is called. In order to send a variable "by value" (a copy, rather than the variable itself) it should be preceded by a `#` when sent as an argument, e.g., `y = f(#x)`, this tells the system to send a temporary copy of `x` to the function `f`. Sending arguments "by reference" is more efficient. Sending a variable "by value" is only required when you know that the function modifies the argument and you want to protect the variable (i.e., keep its original value).

There are cases where temporary arguments are automatically constructed by the system: 1. the argument is an expression, e.g., `y = f(a+b)`, 2. the argument is an indexed variable, e.g., `y = f(a[3])`, 3. the argument is a constant, e.g., `y = f(3)`. Temporary arguments can be modified by the function, but since they are temporary, they are actually local variables of the function.

Returning from a function

Functions that do not return a value cannot appear in expressions or assignments. Functions can return values in two ways:
1. The 'C' style, by using **`return`**. 2. The 'Pascal' style, by assigning a value to the function's name.
A function with no parameters must still be called with parenthesis (just as it is defined), i.e., `y = f()`.
When you want to return more than one variable from a function you can send variables as arguments to the function and the function may assign values to them.

Local and Global variables

Variables that are defined in the function are local to the function and are not available to other functions or to the main program. A local variable may have any name and there is no conflict between variables of the same name in other functions or in the main program.

Local variables exist only while the function is active, however, for efficiency considerations, ***Numerit*** does not free and does not initialize the local variables when exiting a function so that repeated calls to the same function will not create and destroy variables which is a time consuming operation (especially with large variables). So, when the function is called again, its local variables are already defined as they were on the last call, and with the same value. If there is a shortage in memory, or if you want to make sure that you start with new variables on every call (e.g., when variables have different dimensions on different calls), use the **`free`** command to free local variables before you return from the function or when you enter the function. If you only want to make sure that values from the last call are cleared and that you start with an initialized variable on every call, use the **`clear`** command either when you exit the function or when you enter the function. This is normally needed with local variables that are arrays. For example, a statement **`a[1000] = 0`**, inside a function, defines a local variable **`a`** as an array with all its elements set to 0. On the first call **`a`** is not defined yet so the assignment statement also allocates a memory for it. On the next call the memory will already be allocated so the same statement will only set the 1000th element (which is a much simpler operation), but the other elements in **`a`** will have the same values they had when the function was last exited. So, by putting a **`clear a`** command after the assignment, you make sure that all the elements are 0. If you want to make sure that the array is both initialized and has the right size, use array definition instead of assignment, e.g., **`a[1000]:0`**. This will make sure that the array size is indeed 1000 (in case it has been changed) and that all its elements are set to 0.

A function has access to the variables of the main program (globals). This is done by preceding the variable's name by %; namely, when the variable **`%x`** appears inside a function it refers to the variable **`x`** of the main program. Note that the function may modify the values of the global variables, so this is another method to return values from a function (this method is not very recommended from a programming style point of view but may be more convenient and more efficient when you want to modify many variables by a function or when there are many functions that should receive the same variables as arguments).

Global variables are not accessible to different modules of the same program. To share variables between modules use the command **`common`**.

Calling a function from another function

All the functions that are defined inside the same program may call each other. In ***Numerit Pro*** it is also possible to call functions that are defined in other programs (See the section **Modules** in Chapter 7).

Functions may call themselves directly or indirectly. This process is called recursion. Recursive functions must contain a proper termination to avoid an infinite loop. The use of recursive calls should be done with care. If not terminated correctly, the repeated call to the same function will exhaust system resources (especially the memory). A proper termination is a conditional statement that skips the recursive call. You must ensure that the condition is indeed satisfied some time. Recursion may be direct (function calls itself directly) or indirect (function A calls function B which calls function A).

Function rerouting

In many situations it is desirable to allow "reassignment" of functions at runtime. Consider, for example, the following case: you have a function, `Client()`, that performs a certain operation and needs the services of another function, `Server()`, to fulfill its task. Suppose that you have different versions of `Server()`, say `Server1()`, `Server2()`, `Server3()`, etc., and you want to control which specific version is used by `Client()` in a specific call. `Client()` doesn't know in advance what are (and will be) the possible `Server()` functions, so you can't simply send it a flag to tell it which function to call. What you really want is to somehow "reassign" `Server()` to the actual version that you want `Client()` to call so that when it calls `Server()` it actually calls the specific version that was assigned to `Server()`. A more specific example is an optimization function. Suppose that you have developed a general optimization function `Optimize()` that calls a merit function `Merit()` during the optimization process to evaluate the optimized function at a specific point. You want to be able to use the same `Optimize()` function with different `Merit()` functions, even functions that you will create in the future. ***Numerit*** offers a solution to this problem with "function rerouting". In your `Client()` function you call the function `Server()`. The function `Server()` is a normal ***Numerit*** function which is defined by the instruction **`function`** with any number of parameters. During the run `Client()` calls `Server()` and the instructions of `Server()` are executed. However, ***Numerit*** allows you to reroute the function `Server()` to another function, say `Server1()`, at runtime, so that whenever `Server()` is called it is `Server1()` that is actually executed. Thus, if you have a set of server functions and you want to control which of them is actually called by the function `Client()`, all you have to do is to reroute the function `Server()` to one of them before you call `Client()`. The function `Server()` itself may do some default action or may even be empty (if it's never actually used but serves only as a model).
The rerouting operation is very simple, you just write the instruction:

```
Server -> Server1
```

before calling `Client()`. We use here the names `Client()` and `Server()` but you can, of course, use any other name. In fact, any ***Numerit*** function can be rerouted to any other ***Numerit*** function provided that their parameter lists are identical (same number of parameters and same parameters' types). Note that in the rerouting instruction you should only specify the functions' names without the parameters or the parentheses.
You can also reroute a function in one module to a function in another module (if you are running ***Numerit Pro***). Returning to the optimization example mentioned above, you may build an optimization module, called `optlib` for example, where you define a function `Optimize()` which calls a function `Merit()`. The function `Merit()` is defined in the module `optlib` as an empty function and just serves as a model. The module can be compiled without errors. In your main module you have a specific merit function `MyMerit()` which you want `Optimize()` to call instead of the empty function `Merit()` (the functions `Merit()` and `MyMerit()` will normally have one or more parameters). To do so you write:

```
optlib.Merit -> MyMerit
optlib.Optimize()
```

You may even omit the "`optlib`" from these instructions and write:

```
Merit -> MyMerit
Optimize()
```

and ***Numerit*** will find the functions `Merit()` and `Optimize()` in the module `optlib` (provided that functions of the same names are not defined elsewhere in the Main module or in one of the other modules).
A function may be rerouted as many times as necessary to different functions. To return a function to its initial state you reroute it to itself, for example:

```
Merit -> Merit
```

The functions `Merit()` and `MyMerit()` in this example may have any number of parameters (as long as it's the same for both). They may return a value or modify the value of the parameters sent to them. In other words, the rerouting mechanism is general enough to allow any type of information exchange between the function that acts as a client and the function that acts as a server. It allows you to compile a module and replace its functions at runtime according to your needs without recompiling the whole module.

Dynamic-link Libraries

A dynamic-link library (DLL) is a collection of routines (functions) that can be called by applications. It is actually a separately compiled executable that is linked at runtime to the programs that use it. In addition to functions, a DLL may also make its global variables available to applications.
To distinguish DLLs from standalone executables, files containing compiled dynamic-link routines are named with the .dll extension. The dynamic link at runtime makes it possible to extend the usability of applications by adding to them functionality without making any modifications to the source code or recompiling the application. Using DLLs also helps reduce memory overhead because many applications can share the same code.
Starting with version 1.7, ***Numerit*** supports calling of DLL exported functions as if they were ***Numerit*** built-in functions. ***Numerit*** also allows accessing DLL exported variables.
Besides extending ***Numerit***'s functionality with the large collection of available DLLs (either commercially or as public domain packages), the ability to call DLL functions allows ***Numerit*** users that know how to program in low-level languages such as C or C++, to write time consuming pieces of code in the low-level language and compile them into a DLL. In some cases, calling this code from ***Numerit*** instead of writing it in ***Numerit*** may decrease execution time by orders of magnitude.
In order to call DLL functions or access DLL variables you just have to declare the DLL and the functions and variables that you want to access and you can use them right away. For example, suppose you have a numerical analysis library (DLL) called numlib.dll, and you want to call the functions `gamma(x)` and `beta(a,b)` and read the value of the variable `pi_half` from this library. All you have to do is declare the library and then the functions and variables:

```
dll "numlib.dll"
dfunc double gamma(double)
dfunc double beta(double,double)
dvar double pi_half()
```

and call them (note that in ***Numerit*** a DLL variable is accessed as if it is a function):

```
x = 1
y = gamma(x)
a = 0.5
b = 1.5
c = beta(a,b)
pi2 = pi_half()
```

When you declare a function you specify its name and the types of its return value and parameters (see **Types** below). When you declare a variable you specify its name and type. See **Dynamic-link Functions** below and also the commands **`dll`**, **`dfunc`**, and **`dvar`** in Chapter 14: **Reference to *Numerit*'s commands**, for a detailed description of these instructions.

Calling Convention

Numerit assumes that the functions in the DLL are C or C++ functions that were compiled into a standard Win32 (32-bit) DLL. Functions written in other languages may also be called if their argument passing mechanism matches the C convention (pushed on the stack from right to left) and they support the same parameter types. Two calling conventions are supported, the 'cdecl' (standard C) calling convention and the 'stdcall' calling convention. The second is more efficient since it relies on the called function itself to clean up the stack after execution. When a function is compiled into a DLL with the 'stdcall' calling convention it must be declared in ***Numerit*** with **`sdfunc`** instead of **`dfunc`**. This tells ***Numerit*** that it doesn't have to take care of the stack after the function returns. It is very important to use the correct declaration, otherwise ***Numerit*** might terminate abnormally.

Types

Unlike low-level languages (such as C) ***Numerit*** does not distinguish between different types of numeric variables. The standard numeric variable in ***Numerit*** is a double precision floating point number. A complex number consists of two such double precision numbers (real and imaginary). A double precision number occupies eight bytes in memory. Low level languages such as C support other numeric types that occupy less bytes. In 32-bit implementations the different numeric types in C are:

`char`	A single-byte signed integer
`unsigned char`	A single-byte unsigned integer
`short`	A two-byte signed integer
`unsigned short`	A two-byte unsigned integer

`int`	A four-byte signed integer
`unsigned int`	A four-byte unsigned integer
`long`	A four-byte signed integer (equivalent to **`int`** in 32-bit implementations)
`unsigned long`	A four-byte unsigned integer (**`unsigned int`** in 32-bit implementations)
`float`	A four-byte floating point number
`double`	An eight-byte floating point number

In many implementations there is also an additional floating point type: **`long double`** that occupies more than eight bytes. Another type that is supported in many implementations is **`complex`** that consists of two **`double`**'s packed together (so it occupies 16 bytes).
When a DLL function is declared in ***Numerit***, the types of its parameters must be specified so that ***Numerit*** will know what are the arguments types that the function expects. Since different types have different sizes and representations, it is very important to specify the correct types so that ***Numerit*** will be able to convert the variables that are sent as arguments to the actual types that are expected by the DLL function. The types that are specified in the DLL function or variable declaration in ***Numerit*** (i.e., in the **`dfunc`**, **`sdfunc`**, or **`dvar`** instructions) are the same types as shown in the list above (namely, the C types) including **`complex`**. The type **`long double`** is not supported in ***Numerit*** so DLL functions that expect a **`long double`** argument cannot be called from ***Numerit***. In ***Numerit***, unsigned types can be abbreviated by attaching **`u`** to the beginning of the type, namely, **`uchar`** is equivalent to **`unsigned char`**, **`ushort`** is equivalent to **`unsigned short`**, **`uint`** is equivalent to **`unsigned int`**, and **`ulong`** is equivalent to **`unsigned long`** (and to **`unsigned int`**).
Numerit also allows you to pass pointers between different DLL functions (pointers have no meaning inside ***Numerit***). In such a case you can declare a parameter of type **`ptr`** (or **`pointer`**). In 32-bit implementations (such as the current version of ***Numerit***) this is actually equivalent to an **`unsigned int`**. See **Dynamic-link Functions** below for information about the use of pointers.

Error Handling

If you write your own DLL you can use ***Numerit*** 's error handling to report about errors in a DLL function back to ***Numerit***. In such a case when the DLL function returns to ***Numerit*** the error handling mechanism of ***Numerit*** stops the currently running ***Numerit*** program and reports about the error. This is done the same way it is done when an error occurs in the ***Numerit*** code, namely, a dialog box pops up and shows the error message and the message is also shown in the Message pane.
To allow this option you need to add the following code in your DLL:

```
typedef void __stdcall (*ErrorHandler)(int errcode, char* message);
__declspec(dllexport) ErrorHandler NumeritError = NULL;
```

The first line defines the type `ErrorHandler` as the type of a function with two parameters that is called using the 'stdcall' calling convention. The first parameter is an integer that receives the error code, and the second is a string that receives the error message. The second line defines an exported pointer to a function of type `ErrorHandler`. This pointer, `NumeritError`, is set by ***Numerit*** to its internal error handler so that you can call it in your DLL code to report errors back to ***Numerit***. Note that it is important to initialize it to NULL so that you will be able to check if it is actually set (see below).
When you need to report an error to ***Numerit*** you call `NumeritError` with the error code and an optional message. If the error code is positive and corresponds to one of ***Numerit*** 's execution errors (see **Errors and Warnings**) the message that corresponds to the given error code will be displayed and the second argument (message) will be ignored (and can actually be NULL). If the error code is negative ***Numerit*** will display the specified error message (second argument). For example:

```
NumeritError(12,NULL);
NumeritError(-1,"Argument must be greater than 1");
```

The first call will display the message "File Read error" while the second call will display the specified message.
Note that it is important to export the function pointer `NumeritError` so that ***Numerit*** can find it when it loads the DLL and set it to the actual error handler. ***Numerit*** will first look for the pointer using the name `NumeritError` and if this is not found it will look for it using the name `_NumeritError` since many compilers attach an underscore to global names. If both attempts fail error reporting is not enabled, so before you call the error handler function you must check that the pointer is actually set and is not NULL, for example:

```
if (x <= 1)
{
 if (NumeritError) NumeritError(-1,"Argument must be greater than 1");
 return 0;
}
```

Note that earlier versions of some compilers use the keyword `__export` instead of `__declspec(dllexport)` to export functions and variables from a DLL, and some compilers may use other keywords as well.

Dynamic-link Functions

Numerit allows you to call dynamic-link library (DLL) functions as if they were ***Numerit*** built-in functions. The DLL functions are assumed to be C or C++ functions that were compiled into a standard Win32 (32-bit) DLL. Functions in a DLL that are available to applications are called exported functions. In the DLL source code (which shouldn't concern you if you are just using the DLL), a function is normally exported by the keyword `__declspec(dllexport)` (see also **Dynamic-link Libraries** above).
Before calling a DLL function the function must be declared in ***Numerit***. This declaration serves two purposes: 1) it allows ***Numerit*** to find the specified function in the DLL, and 2) it specifies the function's return type and the number and types of its parameters, so that ***Numerit*** can convert the variables that are sent as arguments to the actual types that are expected by the DLL function. The function is searched for in the library that has been declared on a previous **`dll`** statement.
The declaration in ***Numerit*** is done with the commands **`dfunc`** and **`sdfunc`**. The first is used to declare functions that have been compiled with the 'cdecl' calling convention and the second is used to declare functions that have been compiled with the 'stdcall' calling convention (other than that both commands are identical). See **Dynamic-link Libraries** above for information about calling conventions.
The **`dfunc`** (or **`sdfunc`**) statement includes a DLL function prototype that specifies the type of the function's return value and the types of the function's parameters. The specified types are the C types as described in **Dynamic-link Libraries** above. For example, the following statement:

```
 dfunc double sqr(double x)
```

declares a DLL function, `sqr`, that accepts one parameter of type **`double`** (a double precision floating-point number) and returns a value of type **`double`**.
The DLL function is called just like a ***Numerit*** function, for example, the following code calls the function `sqr` that was declared above:

```
 x = 1.5
 y = sqr(x)
```

A DLL function may either return a value (as the function `sqr` above) or not. The following statement declares a function that doesn't return a value but has two parameters the first of type **`int`** (integer) and the second of type **`double`**:

```
 dfunc void set(int n, double x)
```

For the exact syntax of a DLL function declaration see the command **`dfunc`** in Chapter 14: **Reference to *Numerit*'s commands**.

Return type

A function return type must be of one of the following: **`void`**, **`char`**, **`uchar`** (or **`unsigned char`**), **`short`**, **`ushort`** (or **`unsigned short`**), **`int`**, **`uint`** (or **`unsigned int`**), **`float`**, **`double`**, **`complex`**, and **`ptr`** (or **`pointer`**). The specified type should match, of course, the actual return type of the function. For more information about types in DLL function declarations see **Dynamic-link Libraries** above. The type **`void`** is used to specify that the function does not return a value. If the type is omitted it is assumed to be **`void`**. When the return type is **`ptr`** (or **`pointer`**) it represents an address of a variable or an object and cannot be used in ***Numerit*** but can be transferred to another DLL function (see also **Pointers** below).
Note that the return value is always interpreted as a scalar in ***Numerit***, and the function call actually generates a ***Numerit*** scalar.

Parameters

The parameters are listed inside the parenthesis as a list of types, separated by commas, one for each actual function parameter. Each type can be followed by an optional parameter name, and the allowed types are the same as listed above except **`void`** (which can only be used for the return value).

Unlike the return value which is always a scalar, a parameter can have a much more complicated structure. ***Numerit*** allows a DLL function parameter to be a scalar, an array, a structure, or a function. When you want to call a function from a DLL you must know how it expects to receive each of its parameters.

Note that the DLL does not provide any information about the number and types of its functions parameters. So, it is the responsibility of the user to supply this information accurately since there is no way to check its correctness. Specifying the wrong number of parameters or a wrong type will, in most cases, crash the program and result in an abnormal termination of ***Numerit***.

Since the DLL function prototype in ***Numerit*** is very similar to the prototype that is specified in the C header file, it is not too difficult to avoid errors in the ***Numerit*** declaration. You will usually copy the C function prototype to ***Numerit***. In many cases you will use the C prototype as-is and in other cases you'll have to make some adjustments (see below).

Pointers

Before entering into a detailed description of typed parameters it is important to clarify how ***Numerit*** refers to C pointers. ***Numerit*** has to deal with C pointers only when a pointer is specified as a return value or as a parameter of a DLL function. In C a pointer may represent different objects: a scalar, an array, a structure, etc. The actual interpretation of the pointer is done in the function itself. Since there are no pointers in ***Numerit***, you must know how a parameter that is declared as a pointer in C is actually interpreted in the function before you can specify its type in the **dfunc** statement. If the return value is a pointer you must specify it as **ptr** (or **pointer**). If a parameter is a pointer, the type you specify depends on the interpretation of the pointer. For example, if it represents a scalar that is sent "by reference" you must declare it with a type and an ampersand, e.g., **double**&, and if it represents a one-dimensional array you should declare it with a type and square brackets, e.g., **double**[] (see a more detailed description of each of these cases below).
When the pointer represents an address to an object that is not supported by ***Numerit***, it should be declared as **ptr** (or **pointer**). The use of a **ptr** parameter is mainly to transfer an object's address from one DLL function to another. The value of such an address will normally be returned from a DLL function and will not be manipulated or processed inside ***Numerit***, but will be transferred to another DLL function. For example, the following C functions are defined in a DLL, where the first function allocates memory for some object (whose internal structure is hidden from us) and returns the address of the allocated memory block, the second function receives the address of the object as a parameter, uses it and returns a value, and the third function frees the allocated memory.

```
obj * alloc_obj()
int proc_obj(obj *)
void free_obj(obj *)
```

In ***Numerit*** these functions will be declared in the following way:

```
dfunc ptr alloc_obj()
dfunc int proc_obj(ptr)
dfunc void free_obj(ptr)
```

And will then be used as follows:

```
p = alloc_obj()       ` allocate
if p = 0 stop         ` allocation failed
n = proc_obj(p)       ` process
free_obj(p)           ` free
m = n+1               ` use returned value
```

The value of `p` has no meaning in ***Numerit*** (although it can be tested for the success of the allocation as shown above), and `p` is used only to transfer the address of the memory block to the functions `proc_obj` and `free_obj`. The value of the integer `n` returned from `proc_obj` is numeric and can be used in subsequent ***Numerit*** operations.
Another use of a **pointer** parameter is when one needs to send a NULL pointer to a DLL function. For example, a DLL function that expects an array as one of its parameters may check if the array is actually sent (i.e., the address is valid) and if it receives a NULL pointer instead it will perform a different operation. When you declare such a function in ***Numerit*** you *must* specify an array in the parameter list, and since ***Numerit*** checks that the argument you send is indeed an array (when you call the function) this does not allow you to send a NULL value, namely 0, which is a scalar. The solution to this is to declare two versions of the same function with two different ***Numerit*** names, one version with an array parameter and the other version with a **pointer** parameter (see the command **dfunc** in Chapter 14: **Reference to *Numerit*'s commands**, for the exact

syntax of naming DLL functions). Both versions will refer to the same DLL function, but since they have different ***Numerit*** names you can call the first when you send an array and the other when you send a NULL pointer.
For example, in the following declarations (note that here we use an array parameter which is described in detail later),

```
dfunc foo1(double a[]):foo
dfunc foo0(ptr):foo
```

both `foo1` and `foo0` refer to the same DLL function `foo` but the first is called with an actual array and the second with 0, for example,

```
x = 1 to 10
foo1(x)
foo0(0)
```

A **`pointer`** parameter may also be used to transfer an address of one DLL function as an argument to another DLL function (more details below).

A scalar parameter

A scalar parameter is specified by its type and an optional name. When the function is called, ***Numerit*** checks that the argument sent is actually a scalar. A scalar can be sent to the function in two ways: "by value" or "by reference". When a variable is sent to a DLL function "by value" its value is copied to the function's stack and the function reads the value from the stack but has no access to the variable itself. When a variable is sent to the function "by reference", its address is copied to the function's stack and the function has full access to the variable itself; it can read its value and also modify it.
A C function receives a scalar variable as a "by value" parameter when it is specified with only the type, for example, in the function:

```
void foo(int n, double x)
{
 ...
}
```

the parameters `n` and `x` are passed "by value". The same rule is used to specify a scalar parameter "by value" in the DLL function declaration, namely, the above DLL function will be declared in ***Numerit*** as:

```
dfunc void foo(int n, double x)
```

A C function receives a variable "by reference" when the parameter is specified with a type and an address mark (an ampersand), for example:

```
void set(double& x)
{
 x = 1.23;
}
```

or when it is specified as a pointer, for example:

```
void set(double* x)
{
 *x = 1.23;
}
```

Numerit does not distinguish between the two and when specifying a scalar parameter "by reference" in ***Numerit*** you should always use the ampersand syntax even when the C function expects a `*` (specifying **`double`**`*` will generate an error). So, the above (equivalent) DLL functions will be declared in ***Numerit*** as:

```
dfunc void set(double& x)
```

Note that if you copy the function declaration from a C header file you must change the `*`'s to `&`'s wherever the `*` is used for sending a scalar "by reference".

A real (not complex) ***Numerit*** scalar can be sent as an argument when the expected function type in the C function is **`char`**, **`unsigned char`**, **`short`**, **`unsigned short`**, **`int`**, **`unsigned int`**, **`float`**, **`double`**, or a pointer. A complex ***Numerit*** scalar can be sent as an argument only when the expected C type is **`complex`** (representing two **`double`** values, real and imaginary). Since the internal representation of variables in ***Numerit*** is as double precision floating-point numbers, ***Numerit*** needs to convert the scalar to the expected type before it passes it to the function and then to convert it back (if passed "by reference") after the function returns. The conversion is not required with parameters of type **`double`** and **`complex`**. Thus, passing a **`double`** parameter is more efficient than passing an integer parameter (i.e., **`char`**, **`short`**, or **`int`**).
A scalar can also be sent as an argument to a function that expects a one-dimensional array (see below). In such a case the scalar will be sent "by reference" and will actually be treated as an array with one element.

An array parameter

An array parameter is specified by its type, an optional name, and square brackets. The allowed types are - **`char`**, **`uchar`** (or **`unsigned char`**), **`short`**, **`ushort`** (or **`unsigned short`**), **`int`**, **`uint`** (or **`unsigned int`**), **`float`**, **`double`**, and **`ptr`** (or **`pointer`**). ***Numerit*** complex arrays cannot be passed to DLL functions (unlike complex scalars), but a complex array can be converted to a packed-array with the function **PackComplex** and passed as a **`double`** array to the DLL function. The packed-array is converted back to a ***Numerit*** complex array with the function **UnpackComplex**.
A one-dimensional array is specified with a single pair of brackets. For example,

```
dfunc double sum_1d(double x[])
```

declares a function that receives a one-dimensional numeric array as an argument. Since the name of the parameter is optional this could also be declared as:

```
dfunc double sum_1d(double[])
```

The above ***Numerit*** declaration corresponds to the following C function prototype:

```
double sum_1d(double x[])
```

As mentioned above, in C one can also use a pointer to pass a one-dimensional array, so the above ***Numerit*** declaration also corresponds to the following C function prototype:

```
double sum_1d(double* x)
```

provided that `x` is indeed used as an array of **`double`**'s in the function.
As before, when the type of the array is **`double`**, ***Numerit*** doesn't need to convert its values and the array is sent to the DLL function as is. However, if the C function requires an array of another type, say **`int`**, the ***Numerit*** array is converted from **`double`** (***Numerit***'s internal representation) to **`int`** before it is passed to the DLL function, and is converted back to **`double`** when the function returns. Such a conversion will take place, for example, when calling the following C function:

```
void qsort(int x[])
{
 ...
}
```

which is declared in ***Numerit*** as:

```
dfunc void qsort(int x[])
```

The conversion is, of course, a time consuming operation, especially with large arrays and should be avoided if possible. Note however, that although passing **`double`** arrays is more efficient, in many cases the actual execution of the function is the dominant time consuming process and the conversion time is negligible, so one should not make too much effort to use **`double`** arrays in a DLL function if using other types of arrays is more natural. This is especially true when the function is already written and changing types of arrays requires making modifications to existing code.
Note: you can send a scalar as an argument to a function that expects a one-dimensional array. In such a case the scalar will be sent "by reference". This is actually equivalent to sending an array with one element.

A multidimensional array parameter is declared in ***Numerit*** with two pairs of square brackets, for example:

```
 dfunc double sum_2d(double x[][])
```

Note that this declaration is used whenever a multidimensional ***Numerit*** array (not only two-dimensional) needs to be passed to a DLL function. Arrays with more than two dimensions are “flattened” to two dimensions before they are passed to the DLL function. For example, the following three-dimensional 3x2x4 array

```
┌
│┌
││111,112,113,114
││121,122,123,124
│└
│┌
││211,212,213,214
││221,222,223,224
│└
│┌
││311,312,313,314
││321,322,323,324
│└
└
```

will be flattened to

```
┌
│111,112,113,114
│121,122,123,124
│211,212,213,214
│221,222,223,224
│311,312,313,314
│321,322,323,324
└
```

and passed to the DLL function as such. The number of rows in the resulted flattened array is the multiplication of all the higher dimensions up to the last one (6 in the above example), and the number of columns is the last dimension (4 in the above example). Note that multidimensional arrays are stored in ***Numerit*** in such a way that the “flattening” process doesn’t require an extra effort and is actually not a time consuming operation.
It is very important to note that in C two-dimensional arrays may take two forms. The first is where the array is allocated as a single contiguous block of memory of length (m x n) where m is the number of rows and n is the number of columns. In the second form, the array is allocated in two stages; in the first stage an array of pointers (addresses) of length m is allocated, and then each row of the array is (dynamically) allocated as a one-dimensional array of length n and its address is saved in the corresponding element of the array of addresses. In ***Numerit***, multidimensional arrays are allocated using the second form. So, multidimensional ***Numerit*** arrays can only be passed to DLL functions that expects two-dimensional arrays of the second form. Such an array is specified in the list of parameters of the C function with either two pointer operators (e.g., `double** x`), or with one pointer operator and a square bracket (e.g., `double* x[]`). In both cases the declaration in ***Numerit*** is the same, with two square bracket, namely, `x[][]` (which is actually not a legal specification in C). As before, if you copy the function declaration from a C header file you must change the `**`'s and `*[]`'s to `[][]`'s.
When a DLL function expects a parameter as a contiguous array (with two or more dimensions), it can only be called with a ***Numerit*** one-dimensional array. So, this parameter must be specified in the ***Numerit*** declaration as a one-dimensional array, namely, with a single `[]`. In the C declaration such a multidimensional array is specified as `x[][n]` or `x[m][n]` (or even `x[][m][n]`) where `m` and `n` are numbers, and in general, all the dimensions except the first must be specified. The size of the ***Numerit*** one-dimensional array must be equal to the multiplication of all the dimensions of the target array and the interpretation of the elements is left to the user (remember that in C, elements of multidimensional arrays are stored row-wise).

A string parameter

Strings are treated in C as one-dimensional arrays of characters. When a DLL function has a parameter that is a one-dimensional array of type **`char`** (namely, `char[]` or `char*`), you can send to it a ***Numerit*** String as an argument. Note that although in ***Numerit*** a string is considered to be a scalar when sent as an argument to a DLL function it is sent as an array of **`char`**'s. And as with other types of arrays what is actually sent is the address of the array so the DLL function may modify its content. A one-dimensional array of strings can be sent as an argument to a function that expects a two-dimensional array of **`char`**'s, namely, `char**` or `char*[]`.

A Boolean parameter

In ***Numerit*** a Boolean variable (namely, a variable that can have only one of two values, **`true`** or **`false`**) occupies one byte. So, you can only send it as an argument to a DLL function that expects a parameter of type **`char`** (or **`unsigned char`**). This is true for Boolean scalars as well as Boolean arrays. Note that in other languages (e.g., C) Boolean variables are defined as integers (namely variables of type **`int`**), so if you want to pass an argument to a DLL function that expects such a Boolean variable you must use **`int`** in the DLL function declaration and send to it a numeric variable as an argument rather than a ***Numerit*** Boolean. Such a numeric variable should be defined as 0 for **`false`** and 1 for **`true`**.

A structure parameter

Structures in C are objects that include several variables of different types that are grouped together. Unlike C, ***Numerit*** doesn't support structures but it allows you to group together several variables and send them as a single structure to a DLL function. Such a structure is sent "by reference", namely, only the address of the structure is sent. C also allows to send structures "by value" but this is actually equivalent to sending the variables of the structure as argument one after another, so ***Numerit*** doesn't provide special means for sending structures "by value".
A structure is declared in the **`dfunc`** statement with braces surrounding a list of types. For example, the following structure

```
struct person
{
 int age;
 double height, weight;
};
```

may appear, for example, in a C parameter list as:

```
void insert(struct person* p, int n)
```

but will be declared in ***Numerit*** as:

```
dfunc void insert({int a, double h, double w}, int n)
```

The **`void`** and the names `a`, `h`, `w`, and `n`, are of course optional and the above declaration is equivalent to:

```
dfunc insert({int,double,double},int)
```

This declaration tells ***Numerit*** that the first parameter it passes to the DLL function is a structure whose first member is an **`int`** and second and third members are **`double`**'s. The function also has a second parameter which is an **`int`** (a simple integer).
In the ***Numerit*** instruction that calls the DLL function, the relevant variables should also be grouped together with braces, for example:

```
age = 25
weight = 80
height = 180
n += 1
insert({age,height,weight},n)
```

Note that C++ expands the notion of structures to classes, however, the data structure of class objects is in most cases the same as that of simple structures so the same syntax of sending structures can be used for sending arguments to DLL functions that expect class objects.
When you specify an array as a member of a structure you can do it with the array size or without it. If you do specify the size, the array's elements are copied to the structure when the structure is created. If you don't specify the array size, its address is copied to the structure. For example in the following declaration,

```
dfunc foo({int[10],double},int)
```

the structure consists of 10 **`int`** values and one **`double`** value, and corresponds to the C structure:

```
struct
{
 int n[10];
 double x;
};
```

while in the following declaration,

```
dfunc foo({int[],double},int)
```

the structure consists of an address to an `int` array and a `double` value and corresponds to the following C structure:

```
struct
{
 int* n;
 double x;
};
```

The same is true with two-dimensional arrays, namely, when you specify an array without its size (e.g., `int[][]`), the address of an array is copied to the structure. To copy the actual array elements you must specify the size in both dimensions (e.g., `int[4][5]`).
Since the structure itself is passed to the DLL function "by reference" it is expected that the DLL function can modify the values of the structure's members, including arrays that are specified with dimensions. However, when ***Numerit*** sends such an array as a structure member it actually sends a copy of the ***Numerit*** array rather than the array itself, so it copies the array's elements back to the ***Numerit*** array after the function returns. If you know that the DLL function does not modify the array, this back copy is not actually required and you can tell ***Numerit*** not to do it (and save execution time). This is done by adding the '~' mark after the square brackets of the array that need not be copied back, namely,

```
dfunc foo({int[10]~,double},int)
```

Note that with arrays that are specified in the structure without dimensions the array address is copied to the structure so the array can be directly modified by the DLL function and need not be copied back; thus, the '~' is not relevant in this case.
Numerit does not allow you to specify a structure inside a structure. If this is required you must specify the members of the inner structure as members of the same structure, for example, the C structure "`circ`" shown below:

```
struct coordinate
{
 double x,y;
};

struct circ
{
 struct coordinate center;
 double radius;
};
```

is specified as a parameter to a DLL function with a single structure, namely,

```
dfunc foo({double x, double y, double r})
```

Note that this doesn't cover all possible cases. For example, the following structure cannot be specified as a DLL function parameter,

```
struct circ
{
 struct coordinate* center;
 double radius;
};
```

since a member of the structure is a pointer to another structure (the above specific case can be worked around by specifying a `double` array (i.e., `double center[]`) instead of a pointer to "`coordinate` "; but in the general case (a more complex

structure than "`coordinate`") this cannot be done). A structure may also include a member that is a function (see **A function parameter** below).
As mentioned above, the ***Numerit*** structure parameter can also be used for sending variables to DLL functions that expect C++ objects. For example, in most C++ implementations an array is declared as a class whose data part consists of the array length and address such as the following simple array class:

```
class array
{
 int len;
 double* x;

 public:
 array(int n){x = new double[len=n];}
 ~array(){delete[] x;}
 int size(){return len;}
 double& operator[](int);
 .
 .
 .
};
```

Such an array appears in a C++ function parameter list as:

```
double sum1d(array& a)
```

The DLL function `sum1d` is declared in ***Numerit*** as

```
dfunc double sum1d({int,double[]})
```

and is called with a ***Numerit*** array `x` in the following way:

```
x = 1 to 10
s = sum1d({length(x),x})
```

A function parameter

In many situations C functions receive other functions as arguments. For example, a function that calculates the integral of another function will normally receive the function that it integrates as an argument. ***Numerit*** allows you to send ***Numerit*** functions as arguments to DLL functions. It also allows you to send a DLL function as an argument to another DLL function. Such functions are referred to as “callback” functions since they are called from the called DLL function.
Notes:
1. A ***Numerit*** built-in function (e.g., **`sin`**) cannot be specified directly as a callback function. If this is required a user-defined function must be defined that calls the built-in function (e.g.: **`func`** `sinf(x) =` **`sin`**`(x)`).
2. When a ***Numerit*** function is called from a DLL it should be called using the 'cdecl' convention, namely, the default C calling convention. See **Dynamic-link Libraries** above for information about calling conventions.
A ***Numerit*** function is declared as a parameter of a DLL function by declaring it as **`function`** (or **`func`**) in the list of parameters of the function and by specifying its own list of parameters. This list specifies the types of the parameters of the callback function. The allowed types are the same as the types that can be specified in a DLL function declaration except a function parameter (namely, a callback function does not accept another function as an argument).
For example, the following C code defines a function `sqr` that receives a double scalar and returns its value squared. The function `integral` receives the function `sqr` as an argument and returns the integral. Whenever the function `integral` needs to evaluate the value of its integrand at a specific point in the given range it calls `sqr` with an argument and gets a value back.

```
double sqr(double x)
{
 return x*x;
}
```

```
typedef double (*integrand)(double);

double integral(integrand f, double a, double b)
{
 ... calculate integral of f from a to b
}
```

The function `integral` will be declared in ***Numerit*** as follows:

```
dfunc integral(func double f(double x), double a, double b)
```

Namely, the first parameter is declared as a function that returns a **`double`** value and receives one argument of type **`double`** (as before, the names f,`x`,`a`, and `b` are optional). The second and third parameters are declared as **`double`**'s. The ***Numerit*** callback function in this case must also expect a single scalar and return a scalar. For the above example such a ***Numerit*** function looks like:

```
func sqr(x) = x*x
```

Here we used the single line version of a ***Numerit*** function.
When a callback function is actually sent as an argument to a DLL function it must have a prefix `@`. So, when the function `integral` is called, the ***Numerit*** function `sqr` is sent to it in the following way,

```
a = 0
b = 1
c = integral(@sqr,a,b)
```

A callback function may also be specified as a member of a structure, For example, the following declaration:

```
dfunc foo({func double f(double x), double p}, double a)
```

passes the function `f` and the variable `p` as a structure to the DLL function `foo`. When `foo` is called in ***Numerit***, the callback function should be specified inside the braces with the prefix `@`, namely,

```
foo({@fun,p},x)
```

The callback process is carried out in the following way. The DLL function receives an address of a function as the callback function. When it needs it to evaluate a value, it pushes arguments to the stack and then calls the function in the given address. This call returns control to ***Numerit*** which retrieves the arguments from the stack. ***Numerit*** converts the arguments to ***Numerit*** variables according to the types that are specified in the callback function declaration in the **`dfunc`** statement. The ***Numerit*** function is then called with these variables as arguments and returns a value. The value is converted to the type that should be returned by the callback function (again, according to the **`dfunc`** declaration), and is returned to the calling (DLL) function. As before, it is very important to provide an accurate description of the expected callback function including the return type, the number of parameters, and the types of the parameters. It is also important that the ***Numerit*** function will match the expected callback function. Any error here will result in unpredictable results. ***Numerit*** has no way to verify how many arguments and of what type were actually sent by the DLL function in the callback process, so it must rely on the information that is specified in the declaration of the callback function that appears inside the declaration of the DLL function.
A somewhat more complex situation exists when the argument of the callback function is not a scalar but an array. Since ***Numerit*** must convert the argument to a ***Numerit*** variable before the callback function is executed, it must know the exact size of the array (and of course its type). That's why an array parameter in the declaration of a callback function must also include its length. This can be done in one of the following three ways:
1. The array has a fixed known size. In such a case the size of the array should be specified inside the square brackets, for example:

```
dfunc integ_3d(func double f(double[3]), double a[], double b[])
```

Now, when the callback function is called by `integ_3d`, ***Numerit*** will know that the argument sent to it represents an array. It will create a ***Numerit*** array of length 3 and copy to it the content of the array that was sent as an argument. Note that the arrays `a` and `b` are sent <u>to</u> the DLL function and do not require specification of the length.

In some cases the DLL function expects the callback function to modify the array it sends to it. That's why, before returning to the DLL function the ***Numerit*** array is copied back to the C array. Since this is not always required the user may let ***Numerit*** know that a back copy is not required (to save time) by adding the '~' mark after the square brackets of the array that need not be copied back, namely,

```
dfunc integ_3d(func double f(double[3]~), double a[], double b[])
```

This tells ***Numerit*** that the values of the array elements are not needed by the calling function and can be discarded. If a two-dimensional array is sent as an argument to a callback function all the lengths must be specified, for example:

```
dfunc foo(func double(double[3][5]))
```

Note that here, as with multidimensional arrays that are sent to a DLL function, multidimensional arrays are expected be allocated by rows and not as a single block of memory although the syntax here resembles the syntax of an array that was allocated as a single block.
2. The second method to transfer the array length is to use one of the arguments before the array to specify the length. In such a case the number of the argument is specified instead of the actual length, inside the square brackets. This is done by adding the prefix '#' to the number, for example, the declaration

```
dfunc integ_nd(func double f(int,double[#1]~), double a[], double b[])
```

tells ***Numerit*** that the length of the array is sent in the first argument (which is **`int`**). A two dimensional array in such a declaration can look like this:

```
dfunc foo(func double(int,int,double[#1][#2]))
```

Here the argument of the callback function is a two-dimensional array where the number of rows is sent in the first argument and the number of column in the second argument. A combination of methods 1 and 2 is also possible, for example:

```
dfunc foo(func double(int,double[3][#1]))
```

3. The third method to transfer an array length is to use a global variable for the length. For example:

```
n = 0
dfunc integ_nd(func double f(double[n]~), double a[], double b[], int n)
```

The global variable `n` will be assigned a value before the DLL function is called and this value will be used for the length of the array. This method is useful when the length parameter `n` is under the control of the ***Numerit*** program and in the above example it is also sent as a parameter to the DLL function (last parameter) so that it will know the size of the array it needs to pass to the callback function.
The argument of a callback function can also be a structure. In this case ***Numerit*** assumes that the DLL function pushes the address of the structure to the stack as an argument. However, since ***Numerit*** doesn't have structures, the actual callback ***Numerit*** function will receive the structure members as separate variables. For example, the following declaration:

```
dfunc foo(func double({int,double[#1]}), double)
```

tells us that the DLL function `foo` receives two parameters, the first is a callback function of the form

```
func double({int,double[#1]})
```

and the second is a **`double`** scalar. When `foo` will call the callback function it will pass to it the address of a structure whose members are a scalar of type **`int`** and an array of type **`double`** whose length is specified in the previous integer scalar. The ***Numerit*** callback function (that cannot work directly with structures) will actually receive two arguments, a scalar and an array. This conversion from a single structure sent by the DLL function to the two arguments that are actually sent to the ***Numerit*** callback function is carried out internally by ***Numerit***. All you have to do is to make sure that the number of arguments of the callback function matches the number of members in the structure sent by the DLL function (plus, of course, any other arguments that are sent in addition to the structure).

Note that when the array size is specified inside a structure as the number of a parameter (i.e., with the prefix `#`), the numbering is done from the beginning of the specific structure and not from the beginning of the parameter list. For example, in the following declaration:

```
dfunc foo(func double({int,double[#1]}, {int,double[#1]}), double)
```

the DLL function `foo` calls the callback function with two arguments, each representing a structure of two members where the first member provides the size of the second array member. In both structures we see `#1` as the number of parameter that provides the size because the number of parameter is counted from the beginning of each structure. In general, parameters of a callback function are counted locally inside structures and globally when specified outside a structure. The corresponding ***Numerit*** callback function, in the above case, will have four parameters, for example,

```
func bar(m,x,n,y)
```

where `x` and `y` are arrays whose lengths are `m` and `n` respectively.
In another example we can see how a structure specifying a two-dimensional array is sent to a callback function,

```
dfunc foo(func double({int,int,double[#1][#2]}), double)
```

The callback function will now receive a structure with two integer scalars and an address of a two-dimensional array whose length is given by the two integers. The ***Numerit*** callback function will look like

```
func bar(m,n,x)
```

where `x` is a two-dimensional array whose number of rows given in `m` and number of columns in `n`.
Note that the above callback structure parameters are especially useful when the DLL function is written in C++ and calls the callback function with arrays that are defined as classes. As mentioned above, array classes in C++ are usually defined as structures with the first member(s) specifying the length(s) and the next member specifying the address of the actual array.

A DLL function as a callback function

As already mentioned, a DLL function can be sent to another DLL function as an argument to serve as a callback function. This is the most efficient way from an execution point of view since ***Numerit*** doesn't have to deal with the parameters of the callback function at all. It just sends its address to the DLL function that calls it directly without involving ***Numerit*** in the process. And since the callback function is itself a C (or C++) function it also runs much faster than the ***Numerit*** function. The drawback is, of course, the need to prepare in advance a C (or C++) version of the callback function and to compile it into a DLL.
So, if you know in advance what are the callback functions that you want to use and you know how to write and compile C functions into a DLL it is highly recommended to use this approach. You can use a ***Numerit*** version of the callback function during the development stage, and then when everything works as expected you can write a C version of the function and use it to save execution time.
When a DLL function is specified as a parameter to another DLL function, its type in the function declaration should be **`ptr`** (or **`pointer`**). You can declare two versions of the same DLL function where the first receives a ***Numerit*** function as an argument and the second receives a DLL function as an argument. Returning to the function `integral` discussed above, the first version that expects a ***Numerit*** function for the integrand can be declared as:

```
dfunc integ_num(func double f(double), double a, double b):integral
```

and the second version that expects a DLL function for the integrand as:

```
dfunc integ_dll(ptr f, double a, double b):integral
```

Since the second version doesn't involve any argument passing to ***Numerit***, ***Numerit*** doesn't need to know the returned type or parameters types of the callback function. To use the second version, a C function such as `sqr` should be compiled into a DLL and declared in ***Numerit***, for example:

```
dfunc double sqr(double)
```

In the instruction that calls the function `integ_dll`, the callback function is specified as an argument the same way as a ***Numerit*** function is specified, namely, using the `@` prefix:

```
 c = integ_dll(@sqr,a,b)
```

And since the first parameter of `integ_dll` is a **`ptr`**, ***Numerit*** will send to it the address of the DLL function `sqr`. Note that here again, it is the user's responsibility to make sure that the DLL callback function matches in types and number of parameters the expected callback function.

Chapter 7
Executing and Debugging

Running the program

After typing in the program's code, the program is ready to run. This is done by the menu command **Run:Run** (F9), or the Run button in the Program Bar. The program's code is first compiled and its syntax is checked. If invalid instructions are found, a notification window pops up and a list of errors is displayed in the Message pane. ***Numerit Pro*** also verifies that all the modules that are linked to the program are compiled. If all the modules are compiled and there are no compilation errors ***Numerit Pro*** links external functions from the modules to the main program (see **Modules** below). If one or more functions are not found, a notification window pops up and a list of link errors is displayed in the Message pane.
If there are no errors, the program is executed. Any modification in the Editor will cause a re-compilation of the program when it is run again. If the program's code is not modified it can be run again without compilation. It is possible to compile the program without running it (e.g., for syntax checking) by the menu command **Run:Compile**. In ***Numerit Pro*** this is done with the menu command **Run:Compile Module** which is used to compile the Main module or any other module of the program. It is also possible to compile all the uncompiled modules and link external functions without running the program by the menu command **Run:Make** (see **Modules** below).
Besides error messages, the compiler also issues warnings when it detects suspicious situations, for example, when a variable is assigned a value but is not used by the program. For a complete list of compilation errors and warnings see the chapter **Errors and Warnings**.
A program may be paused or stopped any time using the **Pause** or **Stop** commands in the **Run** menu, or the corresponding buttons in the Program Bar. When a program pauses or stops, it does so after completing all the instructions in the current code line. When a program pauses, a marker (horizontal line) appears above the line which will be executed next when the run is resumed.
It is possible to run the program one line at a time by either the **Run:Step Over** menu command (F8) which executes the line without tracing into functions, or by the **Run:Step Into** (F7) command which executes the line while tracing into functions when they are called during the execution of the line. Both commands have corresponding buttons in the Program Bar. After the execution of each line, the program is in the Pause state. This mode of execution is very useful for debugging. Other debugging aids are the Inspector and breakpoints. See the section **Debugging** below for more details.
When errors occur during the run, a notification window pops up and displays the error. A corresponding error message is also sent to the Message window and the erroneous line appears in the Editor. Some execution errors do not allow the program to continue (Fatal errors) and stop it. Other errors (non-fatal) just pause the program so it may continue to run. Non-fatal errors may also be handled by the program which can trap the error, check it, and decide how to proceed. This is done with the help of three commands: **`HideError`**, **`ShowError`**, **`ResetError`**, and the system variable: **`errval`**. See more details and an example in the description of the command **`HideError`** in Chapter 14: **Reference to *Numerit*'s commands**. See the chapter **Errors and Warnings** for a complete list of execution errors.
When the menu item **Options:Panes Auto-size** is set, the Document pane becomes active whenever the program is executed, in order to make it larger for displaying the output. This does not happen when panes auto-sizing is not enabled or while stepping through the program.
If the program sends output to the Draft (by **`print`**, **`graph`**, etc.) and the Report has no viewers in it, the Draft is brought into the Document pane to display the output. If the Report has viewers in it, it is not replaced by the Draft.

Modules

In ***Numerit Pro*** you can link external modules to a program. When program A is declared as a module of program B, all the functions of A are available to B. This makes it possible to create programs that serve as libraries of reusable functions.
Every program in ***Numerit Pro*** has a list of modules attached to it. This list, which is initially empty, is opened by the menu command **File:Modules** or by clicking the Modules button in the Program bar. See **Modules List** for details about the list and how to edit it.
The code of the program itself is considered to be the Main module. The modules that are listed in its modules' list are the submodules. A submodule is itself a ***Numerit*** program and as such may have submodules of its own. A submodule of a submodule is not accessible to the Main module but it may also be declared as a submodule of the Main module if necessary (***Numerit*** makes sure that only one copy of this submodule is actually loaded).

Compiled Library

As mentioned above a submodule is an ordinary ***Numerit*** program whose functions are made available to the main program. Normally, a program that serves as a submodule is added to the modules list with its source, namely, the program itself (".num") is specified in the list. In some cases it might be desired to add a compiled version of the program to the list. This allows, for example, to develop a library of functions and distribute it to ***Numerit*** users without providing the source code. To create a compiled version of a program, use the menu command **File:Create Library**. The resulted file (with the default extension ".nlb") includes only the compiled code of the program without the source code or the Report.
Note: do not confuse between compiled libraries and ***Numerit*** executables. Executables are also compiled programs, but are intended to be run by ***NumRun*** (see **NumRun: The Numerit Executor**). ***Numerit*** compiled libraries can only be used by ***Numerit Pro***.

Linking

When a function is called by an instruction in the Main module (either in the body or in one of the functions) and this function is not found in the program's code, ***Numerit*** looks for the function in the modules that appear in the program's list of modules. The search is conducted from top to bottom until the function is found. This process is called Linking and is carried out before the program is executed. It is redone whenever the Main module or one of the submodules is modified. If a function is not found in any module an error is issued and the program cannot run.

Making

Before external functions are linked to the Main module ***Numerit*** verifies that all the submodules (in all levels) are compiled. If it finds an uncompiled module it opens it and compiles it (note that from now on, until the module is closed, ***Numerit*** will use the open version of it rather than its disk version; see **Open Modules** below). After compiling all the modules that need compilation ***Numerit*** links the external functions in each module and the program is ready to run. This process of compiling and linking is called Making. It is possible to Make a program without running it using the menu command **Run:Make** (Ctrl+F9).
If during the compilation ***Numerit*** finds errors in a module, it brings it into the Editor and lists the errors in the Message pane. You can correct the errors and try to run the program again (***Numerit*** will use the corrected version of the module rather than its disk version, so you don't have to save the module after the modifications; see **Open Modules** below).
After a successful Make you can use the **File:Save All** menu command (Ctrl+Shift+S) to save the current version of all the modules together with their compiled code (only modules that have been modified after the last Save are actually saved).

Open Modules

A submodule may be opened inside the program window either by ***Numerit*** (for compiling the module or when stepping into one of its functions) or by the user (through the Modules List window). An open submodule remains open until it is closed manually (using the menu command **File:Close Module** or through the Modules List window). Every open module has a tab at the bottom of the Editor pane that allows you to bring it into the Editor. In addition, open modules appear in boldface in the Modules List window. A module remains open as long as it is not explicitly closed, even if it is not currently visible in the Editor. Note that a compiled library cannot be opened as a module and its source is not accessible from within the program's window. To modify the code in such a module you must open it independently in its own window, make the changes, create a new compiled version, and then click **Reload** in the modules list window (see **Modules List** below).
When ***Numerit*** is making the program and there are open modules, it uses the version of the code that is currently in the open modules. Thus, you may change the code in an open module without saving it and the version that is used is the one that is currently open rather than the one on the disk.
A ***Numerit*** program may be open simultaneously in several windows, either as an independent program (if opened with the **File:Open** menu command) or as a submodule of another program (if opened through the Modules List window). All the versions of the program refer to the same file. As long as the file is not saved, each version of it is independent of the others and each program ignores the changes in the source code of other programs and uses its own copy. When the file is saved (either as an independent program or as a module) a message is sent to all the programs that use this file. If a program uses this file as a submodule there are two possible cases: 1) if the submodule is not open (the file is listed in the Modules List but is closed), ***Numerit*** will use the disk version when the program is executed; and 2) if the submodule is open ***Numerit*** will display a message asking you to either reload the module from the disk or to ignore the disk version and continue with the current source code. Note that if you continue with the current version and save it, it will rewrite the disk version.
If the file is open as a program in its own window (not a submodule) and is saved by another program (as a program or as a module), the current version will continue to be used but when you'll try to save it ***Numerit*** will issue a warning to remind you that the same file was also saved by another program. If you confirm the Save it will rewrite the disk version.

Saving Modules

You may save a modified module using the **File:Save Module** menu command or the Save Module button in the Program bar. When a module is saved, only its source code and compiled code are updated in the file (remember that as a ***Numerit*** program the module also has a Report document which is ignored when it is used as a module).
If you save the module using the menu command **File:Save Module As**, a copy of the module is saved under a new name but you continue to work with the original module. This is done so because the module may appear as a sub-module in different levels of the modules list. If you want to continue working with the new module file you must change the module's name in the Modules List window manually.
Note that when a module is displayed in the Editor pane, the key combination Ctrl+S is interpreted as a **Save Module** command; when the Main module is displayed, the Ctrl+S combination saves the whole program (including the Report document and the modules list, but not including the modules themselves). To save the program and all its open modules that have not been saved yet, use the menu command **File:Save All** (Ctrl+Shift+S).

Using Modules

Calling a module's function is not different from calling a function that is defined inside the program. When ***Numerit*** encounters a call to a function and cannot find the function in the Main module (the program itself) it looks for it in the submodules. The search starts with the first module in the list and continues from top to bottom until the function is found. A function may appear in more than one module but the first to be found is the one that's actually used. It is possible to change the order of modules in the list and thus control which function is used (see **Modules List** below). A better control is achieved by specifying the name of the module in the calling instruction. The syntax is:
`module_name.function_name(arguments)`, i.e., the module name precedes the function name with a dot between them (note that the module name is not the file name; see **Modules List** below). For example, if a module named `MyLib` contains a function `MyFunc(x)`, then the instruction `MyLib.MyFunc(x)` will call this specific function even if a function `MyFunc` also appears in another module which is located above `MyLib` in the list of modules.
In addition, it is possible to call the main body of a module using the instruction: `module_name.main()`; for example, to call the main body of `MyLib` you write `MyLib.main()`. Calling the main body of a module may be very useful for defining global variables for the module. As in any ***Numerit*** program, all the functions in a module have access to the variables that are defined in the module's body using the % prefix in front of the variable name (see **Local and Global variables** in the previous chapter). Thus, the module's functions may share variables that are defined in the module's main body. These variables are normally defined by calling the main body of the module once before the module's functions are called. To share variables between modules use the instruction **common**.

Modules List

Every program in ***Numerit Pro*** owns a list of modules. The list is initially empty and you can add as many modules as necessary to it. To open the list select the menu command **File:Modules** or click the Modules button in the Program bar.
The Modules window displays the list of modules that are currently linked to the program. The lines in the list may appear in either the default color (normally black), or in a strong color (normally bold black) or in an error color (red). The normal color represents an existing module that is not open in the editor. The strong color represents an open module (see **Open Modules** above and **Opening and Closing Modules** below). The error color represents a module that either doesn't exist or is not a valid ***Numerit*** program. Opening a module that doesn't exist creates a new module of the specified name (after confirmation).

Adding and Removing Modules

To add a module to the list, select an empty line in the list and click the top **Browse** button (next to the File field). This opens a window that allows you to select a ***Numerit*** program (remember that any ***Numerit*** program may serve as a module) or a ***Numerit*** compiled library. You must point to an empty line in the list when you add a module. To insert an empty line above the current line click the button **Insert**. To remove a module or an empty line from the list, select it and click the button **Remove**. Note that there is always an empty line at the end of the list that is available for adding a new module.
You can also add a module (or change the name of a listed module) manually, without using the button **Browse**. Just type the file name of the module in the File field and press the Enter key (you *must* press the Enter key after entering or modifying the name, otherwise the operation is ignored).

Submodules List

If the added module has submodules of its own you'll see a + (plus) sign preceding its file name in the list. Double-clicking the line displays the list of submodules; double-clicking the first line of the submodules list (which is marked by two dots [..] and an up arrow) returns you to the parent's modules list. Note that a submodule's list cannot be modified; you must open the module as a program (using **File:Open**) in order to modify its modules list.

Module Name and File Name

Each module has a name in addition to its file-name. This name is displayed at the left column of the Modules window. A module is known to ***Numerit*** only by its name. This name is used both in the program code when you want to call a function from a specific module instead of letting ***Numerit*** search for it in the list (see section **Modules** above), and as a tab title to identify the module when several modules are open. When you add a new module to the list, ***Numerit*** creates a name for it that is derived from the file-name. The fields Name and File at the top always reflect the name and file-name of the currently selected module. These fields allow you to specify and modify both the module's name and file-name manually. Note that the actual name and file-name are not modified while you are editing them in the Name and File fields but only when you press the Enter key. If you don't press the Enter key after entering the name or file-name, the list will not be updated and the names you entered will be ignored.

Opening and Closing Modules

To open a module in the Editor pane select it and click the button **Open Module**. If you try to open a module that does not exist on the disk, ***Numerit*** gives you an option to create a new module of the specified name. To close a module select it and click the button **Close Module**. An open module appears in boldface in the list. For each open module there is a tab at the bottom of the Editor pane which allows you to bring it into the Editor. When you close a module its tab is removed. You may also close a module using the menu command **File:Close Module** which closes the module that is currently displayed in the Editor. A module remains open as long as it is not explicitly closed, even if it is not currently visible in the Editor.
Note that when a module is open ***Numerit*** is using its current open version when running the program rather than the disk version (see **Open Modules** in the section **Modules** above).
Note that if a module doesn't exist (displayed in red), opening it will first create a new module of the specified name on the disk (after confirmation).

Modules Directory

At the bottom of the Modules window you may see the **Directory** field where you can specify the modules directory. This allows you to use short file names (i.e., not including the full path) for modules in the list. If you want to add a module that is not in the specified directory you must put its full file-name in the list (including the path). Note that when you add a module using the top **Browse** button and the module is inside the specified modules directory, ***Numerit*** automatically puts its short file-name in the list.
If you change the modules directory, the short file-names will remain, which means that ***Numerit*** will now search for the modules in the new directory. So, if you rename the modules directory (in **Windows**) you only need to change the modules directory in the **Directory** field and not each individual module in the list. A module file-name that is specified with its full path name will always refer to the same file regardless of the modules directory.
The **Auto** checkbox next to the **Directory** field allows you to specify that the modules directory is not set by you manually but rather is taken as the directory of the main program. This allows you to move the program and the modules from one directory to another, or from one computer to another, and the modules directory will be updated automatically. This is especially useful when you write a multi-module program for someone else without knowing in advance where it will be installed.

Reload

When you open a program with modules, ***Numerit*** searches for the modules and creates the modules tree (remember that each module may have submodules of its own). At this stage ***Numerit*** doesn't care if a module file is not found, only when you run the program ***Numerit*** tries to find all the modules and compile the uncompiled ones. If then a module is not found ***Numerit*** issues an error message.
In some rare situations you may want to force a reload of all the modules. For example, if ***Numerit*** already loaded all the modules and found no errors it will not try to reload the modules again each time you run the program (to save time). In such a case if you replace a module's file on the disk with a new file, outside the ***Numerit*** environment, ***Numerit*** will keep using the currently loaded version. You may then click the button **Reload** to force ***Numerit*** to reload the modules and thus to update the replaced module. This situation is rare since normally you will modify modules inside the ***Numerit*** environment and ***Numerit*** will be aware of these modifications. Another case is when you modify the source of a compiled library (see above) and recreate the library. In such a case you must click **Reload** in order to load the new version of the compiled library.

Reordering the list

When ***Numerit*** looks for functions in the modules it starts with the first module in the list and searches down. Functions with the same name may appear in two or more modules but only the first that is found is the one that's actually used. Thus, the order of modules in the list might be important. You can change the position of a module in the list by selecting it and clicking the Up or Down arrows at the bottom-right corner of the Modules window, next to the title "Move".

Extra Module

The Extra module is intended to be used when you run a program and then need to execute some extra code without changing the program or rerunning it. It is available only in ***Numerit Pro***. The code in the Extra module is executed as if it was part of the Main program. For example, you run a program that computes a lot of data and then realize that after the long run you didn't save some of the data in a file. You can add the code to save the data in the Extra module and run it. The code will be considered as a continuation to the code of the Main module. You can then copy the code to the main program for the next run (or leave it in the Extra module for optional execution). Note that in ***Numerit*** when a program has ended its variables are still defined and that's why the Extra module can use them. The variables are initialized only when the program is run again.
To open the Extra module select **File:Open Extra Module**. To close the Extra module select **File:Close Extra Module**. When the Extra module is open you will see an Extra tab at the bottom of the Editor pane that will allow you to move to other modules and back to the Extra module with a simple click. The code you put in the Extra module is saved when you save the program, so, the next time you open the program this code is loaded into the Extra module (the Extra code is saved inside the program file and not separately). Note that the Extra code is executed only when you bring the Extra module to the Editor and run it (namely, when you press Run while the Extra module is visible).
In addition to providing a way to continue the program of the Main module after it was ended, the Extra module has other uses. For example, you can use it for debugging. Suppose that you run a program and at some point while the program is still running you want to pause it and examine the value of some variables. After you pause the program you can open the Extra module and print or plot the values of any of the main program's variables, or the result of any operation you do with these variables.
Since the Extra module has full access to all the variables and functions of the Main module, you can put in it any code that you would put in the Main module. You can, for example, pause the main program, write a code in the Extra module that *changes* the value of some of the program's variables, run the Extra code, and then resume the execution of the main program with the new values.

Debugging

When a program produces errors or does not behave as expected, in needs debugging. Usually, debugging is done by pausing the program and inspecting the variables and the output in the Report or the Draft. There are several ways to pause a program: The first is by using the menu command **Run:Pause** (or the Pause button in the Program Bar) while the program is running. This will pause the program after completing the execution of the current line of code.
When a better control is required, the program may be run one line at a time. In this mode the program pauses after each line. There are two commands that allow stepping through the program, both are invoked from the **Run** menu or by buttons in the Program Bar. The first command is **Step Over** (F8) which executes all the instructions in the current line and pauses before the next line. The second command is **Step Into** (F7) which executes the instructions in the current line and steps into functions when they are called. If no functions are called in the line, the two commands are equivalent.
Another way to pause a program is to put breakpoints at specific lines and run the program normally (**Run:Run** or F9). When a line with a breakpoint is encountered, the program pauses just before the line is executed. A breakpoint is set or reset by the menu command **Run:Breakpoint** (F4), or by clicking the left mouse button at the left edge of the line (where the mouse cursor changes its shape into a right pointing arrow). A line with a breakpoint is highlighted. To set a breakpoint inside a module (in ***Numerit Pro***), open the module (through the Modules List window) or make it visible by clicking its tab if it's already open (see **Open Modules** in the section **Modules** above), and set the breakpoint. Note that in order to keep the breakpoint active the module must remain open (but not necessarily visible) during the run; when the module is closed, the breakpoint is removed. The menu command **Run:Clear Breakpoints** clears all the breakpoints.
The last method for pausing a program is by putting the instruction `pause` inside the program's code. This will pause the program before the next instruction is executed.

When the program pauses inside a function it is possible to move the inspection point back to the calling statement and inspect the variables at that point. This is done by the menu command **Run:Calling Function** (Ctrl+<) or by clicking the Calling-Function button in the Program bar. The menu command **Run:Called Function** (Ctrl+>), or the Called-Function button, return the inspection point to its original position. This back and forth movement can be performed through all levels of function calls and across modules. When the inspection point is not at the current Pause position it is marked by a dashed horizontal line (as opposed to a solid line at the actual Pause position).

Note that during debugging the program runs slower than normal. ***Numerit*** enters the debugging mode automatically when it identifies one of the following situations:
1. The program is run one line at a time (i.e., stepping).
2. The program has at least one breakpoint set.
3. The inspector is open and is in **Watch** mode.

The Inspector

The Inspector is a utility for inspecting program's variables. Each program has its own inspector. To open the Inspector:

- Position the insertion point on a variable's name in the Editor. Choose **Inspect** from the **Run** menu (or press F12).
  or
- Click the right mouse button on a variable's name in the Editor.

The inspector window is displayed and the variable's name appears in the Name field. When pointing to an empty space inside the Editor pane the Name field is empty and you can type any variable name in it. The Inspector looks for a variable of the specified name that belongs to the function at the cursor position.
The Inspector displays the variable's attributes (type, number of dimensions, etc.), and its content. When the variable is dimensioned, its content is displayed as a list. When it has two or more dimensions, you choose along which dimension the list is displayed by selecting one of the radio buttons that appear under the title **Dim**. The index of the first element in the list is specified in the fields that appear under the title **Index**. The window which displays the variable content has two scroll bars. The vertical scroll bar is used for scrolling the window up and down along the dimension that is currently selected for viewing. The horizontal scroll bar is used for scrolling the window to the left or to the right when the displayed data is too wide to fit the window (mainly long strings).
Note that when displaying a multidimensional array, scrolling to the left or to the right will not display different columns of the array but rather shift the currently displayed list so as to fully view its data (if it's wider than the window). To see different columns you should select the appropriate indices in the index fields under the **Index** title. For example, if you inspect a 2-dimensional array M, you see two index fields that allow you to select the index of the first element in the list. By default, the first element is M[1,1] and the list shows the first column (because the first dimension is selected in the **Dim** radio buttons and this dimension is along the rows of the array from the top row downward). To display the next column, increase the index in the second index field. This will start the list from M[1,2] which is the first element of the second column. Increasing the second index field to 3 will start from element M[1,3] and display the third column, and so on. To see the rows, you should follow a similar procedure but select the second dimension for viewing in the **Dim** radio buttons. Then, increasing the first index will display each row in turn.
The Inspector runs in the background and may remain open while the program is running. It may also remain open when you change the name of inspected variable. This is done by either changing the name in the variable's name field or by clicking the right mouse button on a new variable and it will replace the current one.
When the **Watch** check-box is set, the Inspector is updated whenever the watched variable changes its value. If the **Watch** check-box is not set, the update is done manually by clicking the button **Update**.
When the program pauses inside a function it is possible to move the inspection point back to the calling statement and inspect variables at that point. The inspection point is marked by a dashed horizontal line (unless it coincides with the execution point which is marked by a solid horizontal line). At the bottom of the Inspector's window you can see the name of the function whose variable is currently displayed and the name of the function where the program was paused (the execution point).
Note that when you right click on a variable's name, the Inspector looks for the variable in the function that is currently at the cursor position. In order to uniquely identify this function the program must be compiled. Therefore, when you open the Inspector after the program is modified ***Numerit*** first compiles the modified program and only then opens the Inspector.

NumRun: The *Numerit* Executor

NumRun is an environment that allows execution of ***Numerit*** programs outside the ***Numerit*** environment. It can execute compiled ***Numerit*** programs or ***Numerit*** executables (which do not include the source code). ***Numerit*** executables are created by the menu command **File:Create Numerit Executable**. Inside ***NumRun*** the program's code is not visible to the user and the output documents cannot be edited.

In order to make a program executable by ***NumRun*** you must save the program ***after*** it is compiled so that the compiled code is included in the saved file, or create a ***Numerit*** executable. If your program includes modules you ***must*** create a ***Numerit*** executable in order to allow ***NumRun*** to execute it. A ***Numerit*** executable has the extension ".nex" as opposed to the ".num" extension of a ***Numerit*** program.

Users of ***Numerit*** may develop programs and distribute them to customers. The customers can download ***NumRun*** from http://www.numerit.com/download.htm, for free, and use it to run the programs. No royalties are charged from users that develop ***Numerit*** programs and distribute or sell them.

Chapter 8

Errors and Warnings

Introduction

The Message pane, located at the bottom of the Program window, is used for displaying compilation and execution errors and warnings. Errors and warnings are listed in the Message pane along with the line numbers to which they refer.
When pointing to a specific message in the Message pane, the message is highlighted and so is the corresponding line in the Editor. To edit this line press the **Enter** key or move to the Editor and make it the active pane; the insertion point will move to the highlighted line. To move the insertion point to the next or previous error without going back to the Message pane, use the menu commands **Run:Next Error** (Ctrl+Shift+>) and **Run:Previous Error** (Ctrl+Shift+<).
Pressing F1 inside the Message pane, displays a help topic for the currently highlighted error or warning.

Compilation messages are issued when the program is compiled. There are three types:

- An Error - is issued when the compiler detects an error that does not allow the program to run. Such an error must be corrected before that program may run.
- A Warning - is issued when the compiler detects a situation that is most likely, but not necessarily, an error. Such a situation involves, in most cases, variable's dimensions which may be modified at run time. So, for example, when the compiler detects that a 2-dimensional array is used where a 1-dimensional array is expected, it cannot be sure that this is actually an error. Another situation is when the compiler detects that a variable is used in an expression before it has been assigned a value. In this case it cannot be sure if at run time the variable will not be initialized by a call to a function. When warning are issued, the program may still run, but if these warnings are actually errors (which is most likely), the program will stop due to execution errors.
- A Notice - alerts you to a possible programming error and is issued when a variable or a function parameter is defined but is never used in the program. A notice has no effect on the program.

Execution errors are issued when an error is detected at run time. There are two types of errors:

- A fatal error - stops the program (for example, where there is not enough memory to continue).
- A non-fatal error - pauses the program. In most cases you should stop the program after such an error and correct it. Non-fatal errors may also be handled by the program which can trap the error, check it, and decide how to proceed. This is done with the help of three commands: **`HideError`**, **`ShowError`**, **`ResetError`**, and the system variable: **`errval`**. See more details and an example in the description of the command **`HideError`** in Chapter 14: **Reference to *Numerit*'s commands**.

Compilation Errors

'*type1*' used as '*type2*'
An expression of type *type1* is used where an expression of type *type2* is expected.

'*name*' - Nested formal expression is not allowed
The specified function requires a formal expression as an argument and cannot be itself part of another formal expression.

Common variables cannot be declared inside a function
Common variables can only be declared in the main body of a program.

Constants cannot be declared inside a function
Constants can only be declared in the main body of a program.

Declaration of DLL function must be preceded by a 'dll' statement
A declaration of a DLL function has been encountered before a **dll** statement. The declarations of DLL functions must be preceded by a **dll** statement that defines the library where the functions should be found.

Different return types in function '*name*'
The type of the value that is returned from a function must be unique. When more than one **return** instructions are specified in a function they must return values of the same type.

Function with no return value is used in expression
A function that does not return a value was used as if it does, namely, inside an expression or in an assignment.

Illegal constant '*xxx*'
The specified constant cannot be evaluated. When an item starts with a digit it is assumed to be a numeric constant. This error occurs if it is not a legal constant (e.g., contains letters).

Illegal parameter
A global variable prefix (#) cannot be used with a parameter name in a function definition.

Incomplete instruction
This error occurs when the compiler identifies a beginning of a valid instruction but cannot find the rest of it.

Insufficient memory to complete compilation
There is not enough memory to complete the compilation. Close other programs to free memory and try again.

Loc - Arguments are of different types
The arguments of the function **Loc** must be of the same type.

Multiple declaration of constant '*name*'
The specified constant has already been declared.

Multiple declaration of function '*name*'
The specified function name is already used in a previous function definition.

Multiple declaration of label '*name*'
The specified line label is already used. Line labels must be unique.

Name too long
The name of a variable or function cannot have more than 64 characters. This error occurs when a longer name has been specified.

Number of dimensions cannot exceed n (m)
An array of dimension m was defined but the maximum number of dimensions is n.

String too long
The maximum length of a string constant is 1024 characters. This error occurs when a longer string is defined.

Syntax error
The compiler found an unidentified syntax error. This usually does not occur, but if it does, it means that the compiler cannot supply more information about the error.

Type mismatch in a call to '*name*': argument 'n' (*type1*) -> parameter '*name*' (*type2*)
The n'th argument in a call to the specified function was of type *type1* which does not match the corresponding parameter's type - *type2*.

Type mismatch in assignment (*type1* <- *type2*)
A variable of type *type1* was assigned an expression of a different type - *type2*.

Type mismatch in comparison (*type1* <=> *type2*)
Expressions of different types cannot be compared. In comparison operations (e.g., x > y) the operands must have the same type.

Undefined label '*name*'
An undefined label has been specified in a **goto** statement.

Undefined function '*name*'
A call to an undefined function has been encountered.

Unexpected '}' | Unexpected '{' | Unterminated '{' | Missing '{'
Unmatched structure marks or nested structures. These messages indicate an error in the specification of a structure in a DLL declaration or a call to a DLL function. Either a mismatch between a '{' and a '}', or a nested structure ("{}" within "{}" which is not allowed).

Unexpected '*item*'
An unexpected item has been encountered. This usually indicates that a previous item is missing or incorrectly spelled.

Unexpected 'break'
The instruction **break** is allowed only inside a loop.

Unexpected 'next'
The instruction **next** is allowed only inside a loop.

Unexpected end of program
This error occurs when the compiler is expecting more instructions but reaches the end of the program. This is usually an indication for an incomplete instruction at the end of the program.

Unterminated String
A string constant is defined between two quotes("..."). This error occurs when no terminating quote was found.

Wrong number of arguments in a call to '*name*'
The call to the specified function contains a wrong number of arguments. Check the function definition.

Inconsistent number of parameters - rerouting "*func1(n1)* -> *func2(n2)*"
When a function is rerouted to another function, both must have exactly the same number of parameters. n1 and n2 are the number of parameters of each function.

Compilation Warnings

'*name*' - Argument seems to be Complex (must be Real)
The arguments of the specified function must be Real but at least one seems to be Complex.

'*name*' - Arguments must be 1-dimensional (n1,n2)
The arguments of the specified function should be 1-dimensional arrays but seem to have dimensions n1 and n2.

'*name*' - Argument must be 2-dimensional (n)
The argument of the specified function should be a 2-dimensional array but seems to be n-dimensional.

'*name*' must be m-dimensional (n)
The specified variable seems to be an n-dimensional array but should be m-dimensional.

'*name*' must be a scalar (n)
The specified variable seems to be an n-dimensional array but should be a scalar.

'*name*' seems to be a scalar and cannot be indexed
The specified variable seems to be a scalar and cannot be referenced with an index.

'*name*' seems to n-dimensional but its index does not match
The specified variable seems to be n-dimensional but is referenced with an index of a different dimension.

'*name*' - Formal expression cannot have dimensions (n)
The formal expression sent as an argument to the specified function seems to be n-dimensional but should be a scalar.

'*name*' - First argument must be 2-dimensional
The first argument of the specified function should be 2-dimensional. It seems to have a different dimension.

'*name*' - Second argument must be 1-dimensional
The second argument of the specified function should be 1-dimensional. It seems to have a different dimension.

'*name*' - Fitted arrays must be 1-dimensional (n)
Fitting can only be done to a set of points represented as 1-dimensional arrays. It seems that an n-dimensional expression was specified.

'*name*' - Order must be a scalar
The order of the fitting polynomials should be an integer scalar.

An index item cannot be n-dimensional
In an index, each item can be either a scalar or a 1-dimensional expression, but a higher dimension (n) was detected. For example, in `x[i,j]`, `i` and `j` must be either scalars or 1-dimensional arrays.

BessX - Order must be a scalar
The first argument of a **Bessel** function is the order and should be an integer scalar.

Defint - Arguments must have the same dimension (n1,n2)
The arguments that define the limits in the function **`Defint`** should have the same dimensions but seem to have different dimensions n1,n2.

Dimension greater than 1 (n)
An n-dimensional expression was detected where a 1-dimensional expression was expected.

Dimension mismatch (n1 <> n2)
An indexed variable whose index defines an item with dimension `n1` is assigned an expression which seems to be of a different dimension n2. For example, if `i` is a 1-dimensional array then `x[i] = y` is valid only if `y` is also a 1-dimensional array.

Dimension mismatch in comparison (n1 <=> n2)
An expression of dimension `n1` was compared to an expression with a different dimension n2. In comparison operations (e.g., `x > y`) the operands must have the same dimensions.

Dimension mismatch in for-loop
The limits and the step in a for loop may be arrays, but if they are, they must have the same dimensions.

Dimension mismatch in operation (n1 <> n2)
Two expressions that are involved in an operation (e.g., `x+y`) must have the same dimensions, unless one is a scalar.

Expecting a scalar (n)
An n-dimensional expression is detected where a scalar is expected.

Expressions in Range definition cannot have dimensions
When defining a range, all the parameters must be scalars but one of the specified parameters seems to be an array.

FFT/IFFT/Shifft - Argument should be an array
The argument of the functions **`FFT`**`(x)`, **`IFFT`**`(x)`, **`FFTS`**`(x)`, **`IFFTS`**`(x)`, **`Shifft`**`(x)` (Transform/Shift of `x` over all the dimensions) should be an array.

FFT/IFFT/Shifft - First argument should be an array
The first argument of the functions **`FFT`**`(x,n)`, **`IFFT`**`(x,n)`, **`FFTS`**`(x,n)`, **`IFFTS`**`(x,n)`, **`Shifft`**`(x,n)` (Transform/Shift of `x` over dimension `n`) should be an array.

FFT/IFFT/Shifft - Second argument should be a scalar
The second argument of the functions **`FFT`**`(x,n)`, **`IFFT`**`(x,n)`, **`FFTS`**`(x,n)`, **`IFFTS`**`(x,n)`, **`Shifft`**`(x,n)` (Transform/Shift of `x` over dimension `n`) should be an integer scalar.

FitBasis - Basis must be 2-dimensional (n)
The basis sent to the function **`FitBasis`** should be represented by a 2-dimensional array but an n-dimensional expression was detected.

Integ - Argument must be 1-dimensional (n)
The second argument of the function **`Integ`**, which specifies the array where the integral is evaluated, must be 1-dimensional but an n-dimensional expression was detected.

Interp/Linterp - Wrong dimensions in arguments (n1,n2,n3)
The arguments of the function **`Interp`** or **`Linterp`** must have matching dimensions.

It looks like '*name*' is used before it is defined
The specified variable is probably used before it was defined. It is illegal to use a variable before a value is assigned to it.

Loc - Wrong dimensions in arguments (n1,n2)
The first argument of the function **`Loc`** must be a scalar and the second argument an array; n1,n2 are the actual dimensions found.

Locmin/Locmax - Argument must be an array
The argument of the function **`Locmin`** or **`Locmax`** must be an array but seems to be a scalar.

Looks like a numeric type mismatch (*type1* <> *type2*)
The numeric type of the expression involved in the operation must be the same (both Real or both Complex).

Matrix must be 2-dimensional (n)
A matrix must be 2-dimensional but was defined as n-dimensional.

Operands of outer operation should be 1-dimensional
Two expressions that are involved in an outer operation (e.g., `x (+) y`) should be 1-dimensional. (They can also be scalars, but then the outer operation is the same as the corresponding standard operation).

PackComplex - Argument must be a Complex/Real array
The first argument of **`PackComplex`** must be a Complex array and the second argument, if specified, must be a Real array.

Poly - First argument must be 1-dimensional (n)
The first argument of the function **`poly`**, which specifies the coefficients array, must be 1-dimensional but seems to be n-dimensional.

Tolerance must be a scalar
The tolerance argument must be a scalar.

Tolerance seems to be Complex (must be Real)
The tolerance argument seems to be Complex. It must be a real scalar.

UnPackComplex - Argument must be a Real/ Complex array
The first argument of **`UnpackComplex`** must be a Real array and the second argument, if specified, must be a Complex array.

Vector must be 1-dimensional (n)
A vector must be 1-dimensional but was defined as n-dimensional.

Compilation Notices

Parameter '*name*' is not used
The specified parameter appears in the function header but is not used inside the function. This is allowed but is usually an indication of an error.

Variable '*name*' is not used
The specified variable was defined but not used anywhere in the program. This is allowed but is usually an indication of an error.

Link Errors

Unresolved function '*name(n)*'
Numerit couldn't find the specified function in the current program or in any of its modules. n is the number of parameters.

Different number of parameters rerouting "*func1(n1)* `->` *func2(n2)*"
When a function is rerouted to a function in another module, both must have exactly the same number of parameters. n1 and n2 are the number of parameters of each function.

Cannot open DLL: '*name*'
The specified dynamic-link library (DLL) could not be opened. Either the library doesn't exist or is not in the DLL search path.

Unresolved DLL function '*name*' or **Unresolved DLL variable '*name*'**
The specified function or variable was not found in the specified dynamic-link library.

Execution Errors

Fatal Errors

Compiler internal error
The program was not compiled. This error should not occur normally. It might occur when there is not enough memory; if it persists contact technical staff.

Index error
Internal indexing error. This error should not occur normally; if it persists contact technical staff.

Internal error
Internal error. This error should not occur normally; if it persists contact technical staff.

Memory allocation failed
Not enough memory to complete the operation. To free memory, use the **`free`** instruction inside the code whenever possible. Closing other applications might also help.

Stack error
Internal stack error. This error should not occur normally; if it persists contact technical staff.

Non-Fatal Errors

Argument must be monotonic (increasing or decreasing)
The specified argument is expected to be a monotonic increasing or decreasing array, for example, the x-table that is sent to interpolation functions.

Array is empty
An illegal operation has been performed on an empty array. An empty array may be returned from a call to **`loc`**(b) where b is a Boolean array whose elements are all **`false`**.
Some operations (like **`sum`** or **`max`**) cannot be performed on empty arrays since they must return a valid scalar value; other operations are simply ignored.

Complex is illegal
A Complex variable or expression is used as an argument or an operand where Complex is not allowed. Check your code.
Degree must be an integer - smaller than the number of data points
When fitting polynomials to a set of data points (**`FitPoly`** and **`FitCheby`**), the first parameter is the degree and must be an integer smaller than the number of data points. When fitting a basis of functions (**`FitBasis`**) to a set of data points, the number of basis functions must be smaller than or equal to the number of data points.

Dimension error
Wrong dimension, or dimension mismatch in operation.

Division by zero
A division by zero is a numeric error. The program may continue to run and the result of the operation is set to `+INF` or `-INF`. `INF` is an IEEE standard representation for infinity. The value `INF` propagates through expressions, namely, if `x` becomes `INF`, then an expression like `x+1` also yields an `INF`.

End of file
The end of file was reached while reading from the file.

File Open error
File not found or couldn't been opened.

File Position error
File position could not been set. This usually occurs when the specified position is invalid or larger than the file size.

File Read error
An error occurred while reading from the file.

File Write error
An error occurred while writing to the file. Possible reason: the disk is full.

Index not integer
A non-integer value was specified as an array index.

Index out of range
The specified array index is either smaller than the first index (which is 1 or the value set by the instruction **`firstindex`**), or greater than the index of the last element of the array.

Kind error
A variable or expression with an unexpected kind was encountered.

Length is not a power of 2
The length of the array that is transformed by the functions **`FFT`** and **`IFFT`** must be an integer power of 2.

Length mismatch
Some operations require equal lengths of the operands. This error occurs when the lengths are different.

Matrix must be square
A non-square matrix was specified as an argument to a function that expects a square matrix (e.g., **`Det`** , **`Inv`**).

Must be integer
Some functions and operations expect an integer argument (e.g., when a matrix is raised to a power n, n must be an integer). This error occurs when the argument is not an integer.

Must be positive
Some functions expect a positive argument (e.g., the standard deviation argument sent to the fitting functions). This error occurs when this argument was not positive.

Negative value not allowed
Some functions do not accept negative arguments (e.g., **`Bessy`**).

Non-integer order
Bessel functions can only be evaluated for integer orders.

Not converging
Some functions (like **`Integ`**, **`Deriv`**, and **`Root`**) use an iterative process. This error occurs when the process does not converge. This might indicate that the specified tolerance is too small.

Numeric error
An error occurred as a result of an invalid numerical operation (e.g., `0/0`, `log(0)`, etc.). The program may continue to run and the result of the operation is set to `NAN`. `NAN` (or `-NAN`) is an IEEE standard representation for Not-A-Number. The value `NAN` propagates through expressions, namely, if `x` becomes `NAN`, then an expression like `x+1` also yields a `NAN`.

Numeric overflow
A numeric overflow occurred. The program may continue to run and the result is set to `+INF` or `-INF`. `INF` is an IEEE standard representation for infinity. The value `INF` propagates through expressions, namely, if `x` becomes `INF`, then an expression like `x+1` also yields an `INF`.

Output error
An error occurred while writing output to the Draft. This usually indicates that the Draft is full. Use the menu command **Edit:Clear Draft** to clear the Draft, or delete some text from the Draft. In most cases the Draft becomes full when the output of different runs is accumulated. If this accumulation is not required, put the **`clear`** instruction at the head of the program's code. This will clear the Draft at the beginning of each run.

Path not found
The specified path in the **`directory`** statement was not found.

Singular matrix
A singular (or almost singular) matrix has been detected. Some operations (e.g., **`Det`**) cannot be completed in such a case.

Type error
An unexpected type has been encountered during execution. This error should not occur normally since types are checked during the compilation; if it persists contact technical staff.

Undefined variable
An undefined variable is used in an expression. Variables must be assigned a value before they are used.

Function didn't return a value
A function that was expected to return a value has terminated without returning a value.

Variable capacity exceeded
Arrays are limited in the number of elements that they may have in each dimension. This error indicates that too many elements have been defined in one dimension.

Execution Errors Codes

1	Numeric overflow
2	Division by zero
3	Not converging
4	Numeric error
5	Argument must be monotonic (increasing or decreasing)
6	Complex is illegal
7	Degree must be an integer - smaller than the number of data points
8	Dimension error
9	End of file
10	File Open error
11	File Position error
12	File Read error
13	File Write error
14	Index not integer
15	Index out of range
16	Kind error
17	Length is not a power of 2
18	Length mismatch
19	Matrix must be square
20	Must be positive
21	Negative value not allowed
22	Non-integer order
23	Output error
24	Singular matrix
25	Type error
26	Undefined variable
27	Function didn't return a value
28	Variable capacity exceeded
29	Array is empty
30	Path not found
31	Must be integer

Chapter 9
Keyboard and Mouse

Keyboard Shortcuts

Press	To
Ctrl+A	Select All
Ctrl+B	Set Bold text in the Document
Ctrl+C	Copy the selected text to the clipboard
Ctrl+D	Toggle the Document pane between the Report and the Draft
Ctrl+E	Center a paragraph in the Document
Ctrl+F	Find text
Ctrl+Shift+F	Fold/Unfold a block in the Editor
Ctrl+G	Set Greek font in the Document
Ctrl+H	Search and Replace text
Ctrl+Shift+H	Set character Height (point size) in the Document
Ctrl+I	Set Italic text in the Document
Ctrl+J	Justify a paragraph in the Document
Ctrl+L	Align a paragraph to the Left in the Document
Ctrl+M	Show the Message pane
Ctrl+N	Create a New program
Ctrl+O	Open an existing program
Ctrl+P	Print the current program or document
Ctrl+Q	Insert an equation at the current position
Ctrl+R	Align a paragraph to the Right in the Document
Ctrl+S	Save the current program, module, or Input File
Ctrl+T	Set a paragraph style in the Document
Ctrl+U	Set Underline text in the Document
Ctrl+V	Paste from the clipboard to the current position
Ctrl+W	Check spelling of the Word at the current position in the Document
Ctrl+X	Cut the selected text to the clipboard
Ctrl+Y	Delete the current line in the Editor
Ctrl+Z	Undo the last editing operation
Ctrl+Shift+Z	Redo the last undone operation
F1	Display Help Contents
Ctrl+F1	Search for a keyword and display its help topic
F2	Repeat the last Replace operation
Shift+F2	Search and Replace text
F3	Repeat the last Find operation
Shift+F3	Find text
F4	Set/Clear a breakpoint at the current line
Ctrl+F4	Close the current program
Alt+F4	Exit ***Numerit***
Shift+F4	Tile Program windows
F5	Stop the current program
Shift+F5	Cascade Program windows
F6	Pause the current program
Shift+F6	Move between the Editor and the Document panes in the current Program window
F7	Step Into: execute the current line while entering into functions
Shift+F7	Maximize the current pane in the current Program window
F8	Step Over: execute current line without entering to functions
Shift+F8	Split the current Program window vertically
F9	Run or resume running of the current program
Ctrl+F9	Compile/Make the current program

Press	**To**
Shift+F9	Split the current Program window horizontally
F10	Make the menu bar active
F11	Insert a Viewer in the current document
F12	Invoke a right-mouse-button operation
Ctrl+ +	Increase the font size in the Document
Ctrl+ -	Decrease the font size in the Document
Ctrl+[	Insert a block around the selected text or insert an empty block (if not text is selected) in the Editor or Set subscript text in the Document
Ctrl+]	Set superscript text in the Document
Ctrl+>	Move inspection position to the calling function
Ctrl+<	Move inspection position to the called function
Ctrl+Shift+>	Find the next error in the Editor
Ctrl+Shift+<	Find the previous error in the Editor
Enter	Start a new line in the Editor or Start a new paragraph in the Document
Shift+Enter	Move to the beginning of the next line in the Editor or Start a new line within the same paragraph in the Document
Ctrl+Enter	Start a new page in the document (force a page-break)
Tab	Insert a tab
Ctrl+Tab	Move to the next Program window
Shift+Tab	Move between the Editor and Document panes in the current Program window
Ctrl+Shift+Spacebar	Insert a nonbreaking space
Back Space	Delete a character on left
Delete	Delete the current character or the current selection
Insert	Toggle Insert/Overwrite mode
Home	Move to the beginning of the current line
Ctrl+Home	Move to the beginning of the Editor or the Document
End	Move to the end of the current line
Ctrl+End	Move to the end of the Editor or the Document
Page Up	Move up one screen-page
Ctrl+Page Up	Scroll Left
Page Down	Move down one screen-page
Ctrl+Page Down	Scroll Right
Up Arrow	Move up one line
Ctrl+Up Arrow	Scroll Up
Down Arrow	Move down one line
Ctrl+Down Arrow	Scroll Down
Left Arrow	Move one character to the left
Ctrl+Left Arrow	Move one word to the left
Right Arrow	Move one character to the right
Ctrl+Right Arrow	Move one word to the right

Mouse Operations

The left mouse button is used for:

- Moving the cursor around.
- Selecting menu commands.
- Selecting text.
- Selecting objects.
- Resizing frames.
- Setting the Document's ruler.
- Setting breakpoints.
- Clicking buttons.

Double clicking with the left mouse button on a Viewer opens the Viewer's edit dialog.

The right mouse button is used for:

- Folding blocks in the Editor; when pointing to a block.
- Displaying help on keywords; when pointing to a keyword in the Editor.
- Opening the Inspector; when pointing to a variable or to an empty space in the Editor.
- Opening the Insert Field menu; when inside the Document.

Pressing the function key F12 is equivalent to pressing the right mouse button.

Chapter 10

A Quick Reference to *Numerit*'s built-in functions

(Language keywords that return a value or modify variables' value)

abs(x)	Absolute value
abs2(x)	Absolute value squared
acos(x)	Inverse cosine
acosh(x)	Inverse hyperbolic cosine
acot(x)	Inverse cotangent
acoth(x)	Inverse hyperbolic cotangent
adefint(y,x,a,b)	Definite integral of array - by interpolation
aderiv(y,x)	Derivative of array - by interpolation
ainteg(y,x)	Integral of array - by interpolation
angle(x)	Angle of a complex number (same as arg, phase)
arg(x)	Argument of a complex number (same as angle, phase)
aroot(y,x)	Zero of array - by interpolation (same as azero)
asin(x)	Inverse sine
asinh(x)	Inverse hyperbolic sine
atan(x)	Inverse tangent
atan2(y,x)	Inverse tangent of y/x
atanh(x)	Inverse hyperbolic tangent
azero(y,x)	Zero of array (same as aroot)
bessj(n,x)	Bessel function of the first kind of order n
bessy(n,x)	Bessel function of the second kind of order n
ceil(x)	Rounded up value
ceil(x,n)	Rounded up value with accuracy 10^n
conj(x)	Complex conjugate
cos(x)	Cosine
cosh(x)	Hyperbolic cosine
cot(x)	Cotangent
coth(x)	Hyperbolic cotangent
csc(x)	Cosecant
csch(x)	Hyperbolic cosecant
defint(exp,x,y,tol)	Definite integral of an expression
deriv(exp,x)	Derivative of an expression
det(M)	Determinant of square matrix
dim(x)	Dimension of variable
eof(x)	End of file detection
erf(x)	Error function
even(x)	Test for an even number
exists(x)	Existence of file
exp(x)	Exponent function
fitbasis(b,y,s)	Fit a basis of functions to a set of data points
fitcheby(n,x,y,s)	Fit Chebyshev polynomials to a set of data points
fitline(x,y,s)	Fit a straight line to a set of data points
fitpoly(n,x,y,s)	Fit a polynomial to a set of data points
fft(x)	Fast Fourier transform
ffts(x)	Shifted fast Fourier transform
floor(x)	Rounded down value
floor(x,n)	Rounded down value with accuracy 10^n
gamma(x)	Gamma function
gammaln(x)	Logarithm of gamma function
ident(n)	Identity matrix ($n \times n$)
ifft(x)	Inverse fast Fourier transform

iffts(x)	Shifted inverse fast Fourier transform
imag(x)	Imaginary part
index(b)	indices where a 1-dimensional Boolean array is **true** (same as loc)
index(a,x)	index of scalar in an array (same as loc)
integ(exp,x,tol)	Integral of an expression
interp(y,x,xx)	Cubic splines interpolation
inv(x)	Inverse of a square matrix
length(x)	Length of variable
linterp(y,x,xx)	Linear interpolation
linsol(M,b)	Linear system solution using LU decomposition
linsolsvd(M,b)	Linear system solution using singular value decomposition
ln(x)	Natural logarithm (same as log)
loc(b)	indices where a 1-dimensional Boolean array is **true** (same as index)
loc(a,x)	index of scalar in an array (same as index)
locmax(x)	index of maximum element in an array
locmin(x)	index of minimum element in an array
log(x)	Natural logarithm (same as ln)
log10(x)	Base 10 logarithm
max(x)	Maximum element of an array
mean(x)	Mean of elements in array
mean(x,n)	Mean of elements in array over dimension n
min(x)	Minimum element of an array
numtostr(x)	Convert a number to String
odd(x)	Test for an odd number
packcomplex(z)	Packs a complex array
packcomplex(z,x)	Packs a complex array
phase(x)	Phase of a complex number (same as arg, angle)
poly(c,x)	Polynomial evaluation
pos(x)	Position in a file
prod(x)	Product of elements
product(x)	Same as prod(x)
prod(x,n)	Product of elements over dimension n
product(x,n)	Same as prod(x,n)
rand(x)	Random number
real(x)	Real part
root(exp,x)	Zero of an expression (same as root)
round(x)	Rounded value
round(x,n)	Rounded value with accuracy 10^n
sec(x)	Secant
sech(x)	Hyperbolic secant
sign(x)	Sign
shifft(x)	Shift of a fast Fourier transform
shifft(x,n)	Shift of a fast Fourier transform over dimension n
sin(x)	Sine
sinh(x)	Hyperbolic sine
size(x)	Size of file
sort(x)	Sorting an array
sqrt(x)	Square root
stdev(x)	Standard deviation
stdev(x,n)	Standard deviation over dimension n
strdel(s,p,n)	Delete characters in string
strins(s,p,t)	Inserts string into string
strlen(s)	Length of string
strlow(s)	Lowercase of string
strlower(s)	Lowercase of string
strpos(s,t)	Position of substring in string
strsub(s,p,n)	Extract substring from string
strtonum(s)	Convert string to number

strupp(s)	Uppercase of string
strupper(s)	Uppercase of string
substr(s,p,n)	Extract substring from string
sum(x)	Sum of elements of array
sum(x,n)	Sum of elements of array over dimension n
svd(M)	Singular value decomposition of a matrix
tan(x)	Tangent
tanh(x)	Hyperbolic tangent
trace(M)	Trace of a square matrix
transp(x)	Transpose of a multidimensional array
transpose(x)	Same as transp
trunc(x)	Integer part
trunc(x,n)	Integer part with accuracy 10^n
unpackcomplex(z)	Unpacks a complex array
unpackcomplex(z,x)	Unpacks a complex array
var(x)	Variance
var(x,n)	Variance over dimension n
zero(exp,x)	Zero of an expression (same as root)

Chapter 11

A Quick Reference to *Numerit*'s commands

(Language keywords that define objects, perform operations on objects, or control program execution)

array	Converts Vector or Matrix into a simple Array
beep	Generates a beep
binfile	Defines a binary file
bitmap	Reads a bitmap file and converts to Image
break	Breaks a loop
by	Specifies an increment
case	Defines conditional execution of statements
clear	Clears a variable, a file, or the Draft
clock	Gives access to the computer internal clock
close	Closes a file
common	Declares common variables in modules
const	Declares a constant
delete	Deletes a file
dfunc	Declares a dynamic-link library function
dfunction	Same as dfunc
dvar	Declares a dynamic-link library variable
directory	Sets the current working directory
dll	Declares a dynamic-link library
do	Used in various loop statements (optional)
downto	Specifies the limit in a decreasing for loop
draft	Displays the Draft
else	Used in if, case, and where statements for optional execution
fopendlg	Opens a dialog to select a file name for opening
file	Defines a text file
firstindex	Sets the starting index of arrays
for	Defines a loop with a running counter
free	Frees the memory occupied by a variable
freedll	Frees a dynamic-link library
fsavedlg	Opens a dialog to select a file name for saving
func	Starts a function definition
function	Same a func
goto	Transfers execution to a labeled line
graph	Inserts a graph at the current position in the Draft
if	Defines conditional execution of statements
input	Gets input from the user
len	Specifies an array length
loop	Defines an infinite loop
matrix	Defines a 2-dimensional array as a Matrix
menu	Opens a menu of options for user selection
message	Displays a message
next	Causes a jump to the next loop round
ocmessage	Displays a OK/Cancel message
outf	Specifies the format of numbers on output
outformat	Same as outf
outp	Specifies the precision of numbers on output
outprec	Same as outp
outw	Specifies the width allocated to items on output
outwidth	Same as outw
outz	Specifies a zero threshold for output (numbers below the threshold appear as 0)
outzero	Same as outz

pause	Pauses the program
pos	Sets the current position in a disk file or the Input File
print	Writes output to the Draft
println	Writes output to the Draft and moves to a new line
printpg	Writes output to the Draft and moves to a new page
randomize	Initializes the random number generator
read	Reads data from a file or a string
readln	Reads a line of text from a text file
readtab	Reads tabular data from a text file
refresh	Refreshes the Viewers in the Document
repeat	Defines a loop with a condition test at its end
report	Displays the Report
return	Returns from a function
sdfunc	Declares a dynamic-link library function
sdfunction	Same as sdfunc
smessage	Displays a Stop message
stop	Stops the program
system	Executes a system (Windows) command
table	Inserts a table at the current position in the Draft
then	Used in an if statement (optional)
time	String containing the current date and time
to	Specifies the limit in an increasing for loop
until	Defines a loop that runs until a condition becomes true
vector	Defines a 1-dimensional array as a Vector
wait	Waits for the specified amount of time
where	Assigns values to array elements according to a condition
while	Defines a loop that runs while a condition is true
wmessage	Displays a Warning message
write	Writes output to a file or a string
writeln	Writes output to a text file and moves to a new line
writetab	Writes tabular output to a text file
ynmessage	Displays a Yes/No message
yncmessage	Displays a Yes/No/Cancel message

Chapter 12

A Quick Reference to *Numerit*'s operators

Operators are listed in order of precedence (highest to lowest). Operators in each group have the same precedence.

Operator	Meaning	Types
^	power	Numeric
(^)	outer power	Numeric
!	factorial	Numeric
*	multiplication	Numeric
(*)	outer multiplication	Numeric
/	division	Numeric
(/)	outer division	Numeric
mod	modulo	Numeric
+	addition	Numeric
(+)	outer addition	Numeric
−	subtraction	Numeric
(−)	outer subtraction	Numeric
&	concatenation	String
>	greater than	All
<	less than	All
>=	greater than or equal	All
<=	less than or equal	All
=	equal	All
<>	not equal	All
in	element in array	All
not	logical NOT	Boolean
and	logical AND	Boolean
or	logical OR	Boolean
xor	logical exclusive OR	Boolean
=	assignment	All
+=	increase by	Numeric
−=	decrease by	Numeric
*=	multiply by	Numeric
/=	divide by	Numeric
&=	concatenate to	String
−>	function rerouting	Function

Chapter 13

Reference to *Numerit*'s built-in functions

abs

Syntax	**`abs`**`(x)`
Operands	`x` is a numeric expression
Description	Returns the absolute value of `x`. If `x` is complex the returned value is the square root of the sum of squares of the real part and the imaginary part.

abs2

Syntax	**`abs2`**`(x)`
Operands	`x` is a numeric expression
Description	Returns the square of absolute value of `x`. If `x` is complex the returned value is the sum of squares of the real part and the imaginary part.

acos

Syntax	**`acos`**`(x)`
Operands	`x` is an expression
Description	Returns the inverse cosine of `x`. If `x` is in the range −1 to 1, the returned value is Real, in the range 0 to π.

acosh

Syntax	**`acosh`**`(x)`
Operands	`x` is an expression
Description	Returns the inverse hyperbolic cosine of `x`.

acot

Syntax	**`acot`**`(x)`
Operands	`x` is an expression
Description	Returns the inverse cotangent of `x`. If `x` is Real, the returned value is also Real, in the range –π/2 to π/2.

acoth

Syntax	**`acoth`**`(x)`
Operands	`x` is an expression
Description	Returns the inverse hyperbolic cotangent of `x`.

adefint

Syntax	**`adefint`**`(y,x)`
	`adefint``(y,x,a,b)`

Operands	`y` and `x` are 1-dimensional Real expressions `a` and `b` are scalars
Description	The first form returns a scalar which is the definite integral of `y` with respect to `x` using cubic splines interpolation. The integration is carried out from the first element of `x` to its last element. Both `x` and `y` must be 1-dimensional Real arrays. The second form returns the definite integral from `a` to `b`. If `a` or `b` are outside the `x`-

range they are clipped to this range (in other words, the array `y` is considered to be 0 outside the range).

aderiv

Syntax **`aderiv`**`(y,x)`
Operands `y` and `x` are 1-dimensional Real expressions
Description Returns an array which is the derivative of `y` with respect to `x` using cubic splines interpolation. Both `x` and `y` must be 1-dimensional Real arrays.

ainteg

Syntax **`ainteg`**`(y,x)`
Operands `y` and `x` are 1-dimensional Real expressions
Description Returns an array which is the integral of `y` with respect to `x` using cubic splines interpolation. Both `x` and `y` must be 1-dimensional Real arrays.

angle, arg, phase

Syntax **`angle`**`(x)` or **`arg`**`(x)` or **`phase`**`(x)`
Operands `x` is an expression
Description Returns the argument (or phase) of `x`. The argument is the angle, in radians, of `x` in the complex plane.

aroot, azero

Syntax **`aroot`**`(y,x)` or **`azero`**`(y,x)`
Operands `x` and `y` are 1-dimensional Real expressions
Description Returns the value of `x` which corresponds to the zero of `y`. Both `x` and `y` must be 1-dimensional Real arrays. The search for zero starts from the first element of `y`. When a sign change is detected the zero crossing position is calculated by cubic splines interpolation of `y` vs. `x`.

asin

Syntax **`asin`**`(x)`
Operands `x` is an expression
Description Returns the inverse sine of `x`. If `x` is in the range –1 to 1, the returned value is Real, in the range $-\pi/2$ to $\pi/2$.

asinh

Syntax **`asinh`**`(x)`
Operands `x` is an expression
Description Returns the inverse hyperbolic sine of `x`.

atan

Syntax **`atan`**`(x)`
Operands `x` is an expression
Description Returns the inverse tangent of `x`. If `x` is Real, the returned value is also Real, in the range $-\pi/2$ to $\pi/2$.

atan2

Syntax	**`atan2`**`(y,x)`
Operands	`y` and `x` are Real expressions
Description	Returns the inverse tangent of `y/x`. `x` and `y` must be Real, and the returned value is in the range –π to π.

atanh

Syntax	**`atanh`**`(x)`
Operands	`x` is an expression
Description	Returns the inverse hyperbolic tangent of `x`.

azeryo, aroot

Syntax	**`azero`**`(y,x)` or **`aroot`**`(y,x)`
Operands	`x` and `y` are 1-dimensional Real expressions
Description	Returns the value of `x` which corresponds to the zero of `y`. Both `x` and `y` must be 1-dimensional Real arrays. The search for zero starts from the first element of y. When a sign change is detected the zero crossing position is calculated by cubic splines interpolation of `y` vs. `x`.

bessj

Syntax	**`bessj`**`(n,x)`
Operands	`n` is an integer; `x` is a Real expression
Description	Returns the Bessel function of the first kind of order `n` of `x` (`x` must be Real).

bessy

Syntax	**`bessy`**`(n,x)`
Operands	`n` is an integer; `x` is a positive Real expression
Description	Returns the Bessel function of the second kind of order `n` of `x` (`x` must be Real and positive).

ceil

Syntax	**`ceil`**`(x)`
	`ceil``(x,n)`

Operands	`x` is an expression; `n` is an integer
Description	The first form returns the rounded up value of `x` (nearest integer value not smaller than `x`). The second form returns the rounded up value of `x` with accuracy 10^n, namely, `n` determines which digit is rounded. For example: **`ceil`**`(1234.567,2)` is `1300` (rounded to hundreds), and **`ceil`**`(1234.567,-2)` is `1234.57` (rounded to hundredths).

conj

Syntax	**`conj`**`(x)`
Operands	`x` is an expression
Description	Returns the complex conjugate of `x`.

cos

Syntax **`cos`**`(x)`
Operands `x` is an expression
Description Returns the cosine of `x`.

cosh

Syntax **`cosh`**`(x)`
Operands `x` is an expression
Description Returns the hyperbolic cosine of `x`.

cot

Syntax **`cot`**`(x)`
Operands `x` is an expression
Description Returns the cotangent of `x`.

coth

Syntax **`coth`**`(x)`
Operands `x` is an expression
Description Returns the hyperbolic cotangent of `x`.

csc

Syntax **`csc`**`(x)`
Operands `x` is an expression
Description Returns the cosecant of `x`.

csch

Syntax **`csch`**`(x)`
Operands `x` is an expression
Description Returns the hyperbolic cosecant of `x`.

defint

Syntax **`defint`**`(`*`exp`*`,x,y)`
`defint``(`*`exp`*`,x,y,tol)`
Operands *`exp`* is a formal expression of one variable
`x` and `y` are Real expressions
`tol` is a number
Description Returns the definite integral of the expression *`exp`* from `x` to `y`. *`exp`* has one dummy variable over which the integration is done. It is represented by the `$` (dollar) character. `x` and `y` must have the same dimensions. The integration is carried out between every two matching pairs of `x` and `y`. If, for example, `x` and `y` are 1-dimensional, then the i'th element of the result is the integral of the expression from `x[i]` to `y[i]`.
The integral is calculated by an iterative process (adaptive Simpson's rule) which terminates when the relative error of two successive iterations is smaller than some tolerance. The tolerance may be specified in the fourth argument. If not specified, it is taken as 10^{-6}.
For example, assuming that the variables `a`, `b`, `c`, `x`, `y` are known:

```
z = defint(a*$^2 + b*sin($) + c, x, y)
```

calculates the integral of the expression from `x` to `y`, where `$` represents the integration variable.
The expression used as the integrand can be just a function call, for example:

```
  z = defint(fun($), x, y)
```

where `fun(x)` is any ***Numerit*** function (built-in or user's).

deriv

Syntax `deriv(exp,x)`

Operands *`exp`* is a formal expression of one variable
`x` is an expression

Description Returns the derivative of the expression *`exp`* at `x`. *`exp`* has one dummy variable over which the differentiation is done. It is represented by the `$` (dollar) character.
For example, assuming that the variables `a`, `b`, `c`, `x` are known:

```
z = deriv(a*$^2 + b*sin($) + c, x)
```

calculates the derivative of the expression at `x`, with respect to `$`.
The derived expression can be just a function call, for example:

```
y = deriv(fun($), x)
```

where `fun(x)` is any ***Numerit*** function (built-in or user's).

det

Syntax `det(x)`

Operands `x` is a square matrix

Description Returns the determinant of the matrix `x`.

dim

Syntax `dim(x)`

Operands `x` is an expression

Description Returns the number of dimensions of `x`.

eof

Syntax `eof(f)`
`eof(input)`

Operands `f` is a file

Description Returns **`true`** if the end of file has been reached, **`false`** otherwise. `f` should be previously defined by **`file`** or by **`binfile`**.
The second form checks for end of file in the Input file.

erf

Syntax **`erf`**`(x)`
Operands `x` is an expression
Description Returns the error function of `x`.

exists

Syntax **`exists`**`(f)`
Operands `f` is a file
Description Returns **`true`** if the file exists, **`false`** otherwise. `f` should be previously defined by **`file`** or by **`binfile`**.

even

Syntax **`even`**`(x)`
Operands `x` is an expression
Description Returns **`true`** if the expression is an even integer, **`false`** otherwise.

exp

Syntax **`exp`**`(x)`
Operands `x` is an expression
Description Returns the exponential `e` to the `x`.

fitbasis

Syntax **`fitbasis`**`(b,y)`
`fitbasis``(b,y,s)`
Operands `b` is a 2-dimensional Real expression
`y` and `s` are 1-dimensional Real expressions
Description Returns a 1-dimensional array whose elements are the coefficients in the linear combination of basis functions (represented by `b`), that best fits the set of data points given by `y`.
`b` is a user-supplied 2-dimensional Real array with number of rows that is equal to the number of data points and number of columns that is equal to the number of basis functions. The i'th column of the array `b` is the value of the i'th basis function evaluated at `x` (where `x` is an array of x-coordinates that correspond to the given `y`).
For example, when fitting a Fourier series to the set of data points, the first five base functions are: `1`, `cos(x)`, `sin(x)`, `cos(2x)`, `sin(2x)`, so, the third column of `b`, for example, is `sin(x)`.
The i'th element in the returned array is the coefficient of the i'th basis function.
The second form of **`fitbasis`** allows to add a third array `s`, which specifies relative standard deviations of the data points. All the elements of `s` must be positive numbers. A larger `s[i]` means that the corresponding data point has a larger error and consequently less weight. A very small value will force the curve to go through (or very close to) the corresponding data point. Note that in the first form of the function all the points have the same weight.

fitcheby

Syntax **`fitcheby`**`(n,x,y)`
`fitcheby``(n,x,y,s)`
Operands `n` is an integer
`x`, `y`, and `s` are 1-dimensional Real expressions

Description Returns a 1-dimensional array with `n+1` elements which are the coefficients in the linear combination of Chebyshev polynomials up to degree `n` that best fits the set of data points given by `x` and `y`. `x` and `y` are 1-dimensional arrays of the same length and must be Real.
The i'th element in the returned array is the coefficient of the Chebyshev polynomial T_{i-1}.
Note that the Chebyshev polynomials are defined on the interval `[-1,1]`, so, the function **`fitcheby`** first maps the data points into this interval (the smallest `x` is mapped to `-1` and the largest `x` to `1`) and then calculates the coefficients. Thus, you should use these coefficients to evaluate the fit in the interval `[-1,1]`.
The second form of **`fitcheby`** allows to add a third array `s`, which specifies relative standard deviations of the data points. All the elements of `s` must be positive numbers. A larger `s[i]` means that the corresponding data point has a larger error and consequently less weight. A very small value will force the curve to go through (or very close to) the corresponding data point. Note that in the first form of the function all the points have the same weight.

fitline

Syntax **`fitline`**`(x,y)`
`fitline``(x,y,s)`

Operands `x`, `y`, and `s` are 1-dimensional Real expressions

Description Returns a 1-dimensional array with two elements such that the straight line `a[1]+a[2]·x` is the best linear fit to the set of data points given by `x` and `y`. `x` and `y` are 1-dimensional arrays of the same length and must be Real.
The second form of **`fitline`** allows to add a third array `s`, which specifies relative standard deviations of the data points. All the elements of `s` must be positive numbers. A larger `s[i]` means that the corresponding data point has a larger error and consequently less weight. A very small value will force the straight line to go through (or very close to) the corresponding data point. Note that in the first form of the function all the points have the same weight.

fitpoly

Syntax **`fitpoly`**`(n,x,y)`
`fitpoly``(n,x,y,s)`

Operands `n` is an integer
`x`, `y`, and `s` are 1-dimensional Real expressions

Description Returns a 1-dimensional array with `n+1` elements which are the coefficients of a polynomial of degree `n` that best fits the set of data points given by `x` and `y`. `x` and `y` are 1-dimensional arrays of the same length and must be Real.
The first element in the returned array is the free coefficient the second is the linear coefficient and so on. Use the function **`poly`** to evaluate the polynomial (not required but recommended).
The second form of **`fitpoly`** allows to add a third array `s`, which specifies relative standard deviations of the data points. All the elements of `s` must be positive numbers. A larger `s[i]` means that the corresponding data point has a larger error and consequently less weight. A very small value will force the curve to go through (or very close to) the corresponding data point. Note that in the first form of the function all the points have the same weight.

fft, ffts

Syntax **`fft`**`(x)`
`ffts``(x)`
`fft``(x,n)`
`ffts``(x,n)`

Operands `x` is an expression
`n` is an integer

Description The first and second forms return the fast Fourier transform of `x` over all the dimensions. The third and fourth forms return the fast Fourier transform of `x` over the dimension `n`. The number of elements in each transformed dimension must be an integer power of 2. With the **`fft`** version the zero frequency component is the first element. The **`ffts`** version performs a shift after the transform so that the zero frequency component appears at the center (the `N/2+1` element). Calling the function **`shifft`** after **`fft`** is equivalent to calling **`ffts`**.

Notes:

- **`fft`** and **`ffts`** may either return the transform of `x` and keep `x` unchanged (when specified as part of an expression or on the right side of an assignment, e.g., `y =` **`fft`**`(x)`), or may be applied on `x` itself (when specified as statements, e.g., **`fft`**`(x)`). The second method is recommended if the original `x` is not needed since it uses less memory.
- No normalization is performed on the transform. There are two common methods to normalize the transform: 1. divide the transform by the multiplication of the number of elements in each of the transformed dimensions and leave the inverse transform as is, or 2. divide both the transform and the inverse by the square root of the above normalization factor. Both methods ensure that after performing a transform and its inverse we return to the original array.

The functions **`ifft`** and **`iffts`** are the inverse transforms of **`fft`** and **`ffts`** respectively.

floor

Syntax **`floor`**`(x)`
`floor``(x,n)`

Operands `x` is an expression; `n` is an integer

Description The first form returns the rounded down value of `x` (nearest integer value not greater than `x`).

The second form returns the rounded down value of `x` with accuracy 10^n, namely, `n` determines which digit is rounded. For example: **`floor`**`(1234.567,2)` is `1200` (rounded to hundreds), and **`floor`**`(1234.567,-2)` is `1234.56` (rounded to hundredths).

gamma

Syntax **`gamma`**`(x)`

Operands `x` is an expression

Description Returns the gamma function of `x` (for integer `x` this is equivalent to the factorial of `x-1`).

gammaln

Syntax **`gammaln`**`(x)`
Operands `x` is an expression
Description Returns the logarithm of the gamma function of `x`. This function may be used with large values of `x` where the gamma function overflows.

ident

Syntax **`ident`**`(n)`
Operands `n` is an integer
Description Returns an $n \times n$ Identity matrix.

ifft

Syntax
`ifft``(x)`
`iffts``(x)`
`ifft``(x,n)`
`iffts``(x,n)`

Operands
`x` is an expression
`n` is an integer

Description
The first and second forms return the inverse fast Fourier transform of `x` over all the dimensions.
The third and fourth forms return the inverse fast Fourier transform of `x` over the dimension `n`.
The number of elements in each transformed dimension must be an integer power of 2.
The **`ifft`** version is the inverse of **`fft`**, namely, it assumes that the zero frequency component is the first element. The **`iffts`** version is the inverse of **`ffts`**, namely, it assumes that the zero frequency component appears at the center (the `N/2+1` element), so it performs a shift before the inversion. Calling the function **`shifft`** before **`ifft`** is equivalent to calling **`iffts`**.
Notes:

- **`ifft`** and **`iffts`** may either return the transform of `x` and keep `x` unchanged (when specified as part of an expression or on the right side of an assignment, e.g., `y =` **`ifft`**`(x)`), or may be applied on `x` itself (when specified as statements, e.g., **`ifft`**`(x)`). The second method is recommended if the original `x` is not needed since it uses less memory.
- No normalization is performed on the inverse transform. See **`fft`** for more details.

imag

Syntax **`imag`**`(x)`
Operands `x` is an expression
Description Returns the imaginary part of `x`.

Index, loc

Syntax **index**(b) or **loc**(b)
index(x,y) or **loc**(x,y)

Operands b is a 1-dimensional Boolean array
x is a scalar expression
y is an expression

Description The first form returns an array of indices where b is **true**. Note that this is applicable only to 1-dimensional arrays. For example, if you need to operate on the positive elements of a 1-dimensional array x and leave the other elements intact you define an index array i = **loc**(x > 0) and operate on x[i]. If all the elements of b are **false** the returned array is empty, namely, the value returned by **length**(i) is 0.

The second form returns the index of the scalar x in the array y. Note that the returned index is the first that's found, namely, if x appears in y in different locations only the first is returned. If x does not appear in y the returned index is the last-index + 1 (outside the range). If y is multidimensional the returned value is a 1-dimensional array where the first element is the index in the first dimension, the second element is the index in the second dimension, etc.

integ

Syntax **integ**(*exp*,x)
integ(*exp*,x,tol)

Operands *exp* is a formal expression of one variable
x is a 1-dimensional Real expression
tol is a number

Description Returns the integral of the expression *exp* at x. *exp* has one dummy variable over which the integration is done. It is represented by the $ (dollar) character. x must be a 1-dimensional array. The i'th element of the result is the integral of the expression from the first element of x to x[i].
The integral is calculated by an iterative process (adaptive Simpson's rule) which terminates when the relative error of two successive iterations is smaller than some tolerance. The tolerance may be specified in the third argument. If not specified, it is taken as 10^{-6}.
For example, assuming that the variables a, b, c are defined constants:

```
y = integ(a*$^2 + b*sin($) + c, x)
```

calculates the integral of the expression at x, where $ represents the integration variable. The expression used as the integrand can be just a function call, for example:

```
y = integ(fun($), x)
```

where fun(x) is any ***Numerit*** function (built-in or user's).

interp

Syntax **interp**(y,x,xx)

Operands y, x, and xx are Real expressions. y and x are 1-dimensional.

Description Returns the cubic splines interpolation of (y,x) at xx. y and x are 1-dimensional arrays of the same length. All the arguments must be Real. x should be monotonic increasing or decreasing. Boundary conditions are set with zero second derivative.

inv

Syntax **`inv`**`(x)`
Operands `x` is a square matrix
Description Returns the inverse of the matrix `x`.

length

Syntax **`length`**`(x)`
Operands `x` is an expression
Description Returns the length of `x`, namely, the number of elements in each dimension. If `x` is multidimensional the returned value is an array.

linterp

Syntax **`linterp`**`(y,x,xx)`
Operands `y`, `x`, and `xx` are Real expressions. `y` and `x` are 1-dimensional.
Description Returns the linear interpolation of `(y,x)` at `xx`. `y` and `x` are 1-dimensional arrays of the same length. All the arguments must be Real. `x` should be monotonic increasing or decreasing.

linsol

Syntax **`linsol`**`(M,b)`
Operands `M` is a matrix, `b` is a vector
Description Returns the vector `x` that solves the linear system: `Mx = b`. `M` is a non-singular square matrix and `b` is a vector whose number of elements is equal to the number of rows of `M`. `M` and `b` may be Complex.
`M` and `b` may also be 2-dimensional and 1-dimensional arrays respectively rather than a matrix and a vector, in which case the returned value is a 1-dimensional array.
`linsol` uses LU decomposition to solve the system. The function **`linsolsvd`** which uses the singular value decomposition may be used for solving systems where the number of equations is different from the number of unknowns.

linsolsvd

Syntax **`linsolsvd`**`(M,b)`
Operands `M` is a Real matrix, `b` is a Real vector
Description Returns the vector `x` that solves the linear system: `Mx = b` using singular value decomposition. `M` is a matrix (not necessarily square) and `b` is a vector whose number of elements is equal to the number of rows of `M`. `M` and `b` should be Real.
`M` and `b` may also be 2-dimensional and 1-dimensional arrays respectively rather than a matrix and a vector, in which case the returned value is a 1-dimensional array.
If `M` has m rows and n columns, then the system represents m linear equations in n unknowns, so the returned solution vector has n elements.
When $m < n$ there are fewer equations than unknowns and there is no unique solution to the system. When $m > n$ there are more equations than unknowns and **`linsolsvd`** returns a least-squares solution.

ln, log

Syntax **`ln`**`(x)` or **`log`**`(x)`
Operands `x` is an expression
Description Returns the natural logarithm of `x`.

loc, index

Syntax	**`loc`**`(b)` or **`index`**`(x,y)`
	`loc``(x,y)` or **`index`**`(x,y)`
Operands	`b` is a 1-dimensional Boolean array
	`x` is a scalar expression
	`y` is an expression
Description	The first form returns an array of indices where `b` is **true**. Note that this is applicable only to 1-dimensional arrays. For example, if you need to operate on the positive elements of a 1-dimensional array `x` and leave the other elements intact you define an index array `i =` **`loc`**`(x > 0)` and operate on `x[i]`. If all the elements of `b` are **false** the returned array is empty, namely, the value returned by **`length`**`(i)` is 0.
	The second form returns the index of the scalar `x` in the array `y`. Note that the returned index is the first that's found, namely, if `x` appears in `y` in different locations only the first is returned. If `x` does not appear in `y` the returned index is the last-index + 1 (outside the range). If `y` is multidimensional the returned value is a 1-dimensional array where the first element is the index in the first dimension, the second element is the index in the second dimension, etc.

locmax

Syntax	**`locmax`**`(x)`
Operands	`x` is an expression
Description	Returns the index of the maximum element in the array `x`. If `x` is multidimensional the returned value is a 1-dimensional array where the first element is the index in the first dimension, the second element is the index in the second dimension, etc.

locmin

Syntax	**`locmin`**`(x)`
Operands	`x` is an expression
Description	Returns the index of the minimum element in the array `x`. If `x` is multidimensional the returned value is a 1-dimensional array where the first element is the index in the first dimension, the second element is the index in the second dimension, etc.

log, ln

Syntax	**`log`**`(x)` or **`ln`**`(x)`
Operands	`x` is an expression
Description	Returns the natural logarithm of `x`.

log10

Syntax	**`log10`**`(x)`
Operands	`x` is an expression
Description	Returns the base 10 logarithm of `x`.

max

Syntax	**`max`**`(x)`
Operands	`x` is an expression
Description	Returns the maximum element of `x`.

mean

Syntax	**`mean`**`(x)`
	`mean``(x,n)`

Operands	`x` is an expression; `n` is an integer
Description	The first form returns a scalar as the mean of all the elements in `x`. The mean is defined as the sum of elements in `x` divided by the number of elements. The second form returns the mean of elements of `x` over the dimension `n`. The returned value has one dimension less than `x`. For example, if `M` is a 2-dimensional array than **`mean`**`(M,1)` returns a 1-dimensional array which is the mean of the rows of `M`, while **`mean`**`(M,2)` returns a 1-dimensional array which is the mean of the columns of `M`.

min

Syntax	**`min`**`(x)`
Operands	`x` is an expression
Description	Returns the minimum element of `x`.

numtostr

Syntax	**`numtostr`**`(x)`
Operands	`x` is an expression
Description	Converts the value of the numeric expression `x` to a string and returns the string.

odd

Syntax	**`odd`**`(x)`
Operands	`x` is an expression
Description	Returns **`true`** if the expression is an odd integer, **`false`** otherwise.

packcomplex

Syntax	**`packcomplex`**`(z)`
	`packcomplex``(z,x)`

Operands	`z` is a Complex array; `x` is a Real array
Description	This function packs a complex array into a real array of twice the length of the complex array where the real and imaginary parts of each complex number are placed in alternate neighboring elements. If the array's dimension is greater than 1 only the last dimension is doubled and all the other dimensions are the same. The main use of packed arrays is as parameters to DLL functions. The first form accepts a complex array and returns the packed array (as a real array). The second form receives both arrays as arguments where the second argument is the packed array whose type must be Real and whose length must be twice that of the complex array (this form may be used if the function is called many times with the same arrays, to save memory allocation of the real array on each call).

phase , angle, arg

Syntax	**`phase`**`(x)` or **`angle`**`(x)` or **`arg`**`(x)`
Operands	`x` is an expression
Description	Returns the phase (or argument) of `x`. The phase is the angle, in radians, of `x` in the complex plane.

poly

Syntax **`poly`**`(c,x)`
Operands `c` is a 1-dimensional expression and `x` is an expression
Description Evaluates a polynomial with the given coefficients `c`, at `x`. `c` must be 1-dimensional and `x` may have any dimension. Both may be Real or Complex.
The degree of the evaluated polynomial is `n-1`, where `n` is the number of coefficients. The first element of `c` is the free coefficient, the second element is the linear coefficient, etc.
`poly``(c,x)` is equivalent to: $c[1]+c[2]\cdot x+c[3]\cdot x^2+c[4]\cdot x^3...$, but is much more efficient.

pos

Syntax **`pos`**`(f)`
Operands `f` is a file
Description Returns the current position in the file `f` (in bytes).
`f` should be previously defined by **`file`** or by **`binfile`**.

product, prod

Syntax **`product`**`(x)` or **`prod`**`(x)`
`product``(x,n)` or **`prod`**`(x,n)`
Operands `x` is an expression; `n` is an integer
Description The first form returns a scalar as the product of all elements of `x`.
The second form returns the product of elements of `x` over the dimension `n`. The returned value has one dimension less than `x`. For example, if `M` is a 2-dimensional array than **`prod`**`(M,1)` returns a 1-dimensional array which is the product of the rows of `M`, while **`prod`**`(M,2)` returns a 1-dimensional array which is the product of the columns of `M`.

rand

Syntax **`rand`**`(x)`
Operands `x` is an expression
Description Returns a random number between `0` to `x`. If `x` is an array, the result is also an array of the same size. If, for example, `x` is a 1-dimensional array, then the i'th element of the result is a random number between `0` and `x[i]`.

real

Syntax **`real`**`(x)`
Operands `x` is an expression
Description Returns the real part of `x`.

root, zero

Syntax **`root`**`(`*`exp`*`,x)` or **`zero`**`(`*`exp`*`,x)`
`root``(`*`exp`*`,x,tol)` or **`zero`**`(`*`exp`*`,x,tol)`
Operands *`exp`* is a formal expression of one variable
`x` is the initial guess
`tol` is a number
Description Returns the zero of the expression *`exp`*. *`exp`* has one dummy variable over which the search for the zero is done. It is represented by the `$` (dollar) character. `x` is the initial guess from which the search starts. If the expression has several zeros, the returned value is usually, but not necessarily, the closest to the initial `x`. `x` must be real but may have any dimension. When `x` is multidimensional, **`root`** returns an array of the same dimension as `x`, whose elements are the zeros that were obtained by taking the corresponding elements

in `x` as initial guesses. A tolerance may be specified as a third argument. If not specified, it is taken as `10`$^{-6}$.

For example, the following:

```
  x = 0,3
  z = root($^2 - 3*$ + 2, x)
```

defines `z` as a 1-dimensional array with two elements: `1` and `2` (which are zeros of the given expression obtained by taking `0` and `3` as initial guesses).
The expression can be just a function call, for example,

```
  y = root(fun($), x)
```

where `fun(x)` is any ***Numerit*** function (built-in or user's).

round

Syntax **`round`**`(x)`
`round``(x,n)`

Operands `x` is an expression; `n` is an integer

Description The first form returns the rounded value of `x` (the nearest integer value).

The second form returns the rounded value of `x` with accuracy 10^n, namely, `n` determines which digit is rounded. For example: **`round`**`(1234.567,2)` is `1200` (rounded to hundreds), and **`round`**`(1234.567,-2)` is `1234.57` (rounded to hundredths).
Tie-breaking when rounding 0.5: this function rounds “away from zero”, for example, `1.5` is rounded to `2` and `-1.5` is rounded to `-2`.

sec

Syntax **`sec`**`(x)`

Operands `x` is an expression

Description Returns the secant of `x`.

sech

Syntax **`sech`**`(x)`

Operands `x` is an expression

Description Returns the hyperbolic secant of `x`.

sign

Syntax **`sign`**`(x)`

Operands `x` is an expression

Description Returns the sign of `x`, i.e., `+1` if `x` is positive and `-1` if `x` is negative.
The sign of `0` is `0`. The sign of a complex number `z` is `z/|z|`.

shifft

Syntax	**`shifft`**`(x)`
	`shifft``(x,n)`

Operands	`x` is an expression `n` is an integer
Description	Performs a shift of a fast Fourier transform so that the zero frequency component appears at the center (the `N/2+1` element) rather than the first element. Apply again to restore the original array. The first form shifts all the dimensions. The second form shifts the dimension `n`. Note: **`shifft`** may either return the shifted `x` and keep `x` unchanged (when specified as part of an expression or on the right side of an assignment, e.g., `y = `**`shifft`**`(x)`), or may shift `x` itself (when specified as a statement, i.e., **`shifft`**`(x)`). The second method is recommended if the original `x` is not needed since it uses less memory.

sin

Syntax	**`sin`**`(x)`
Operands	`x` is an expression
Description	Returns the sine of `x`.

sinh

Syntax	**`sinh`**`(x)`
Operands	`x` is an expression
Description	Returns the hyperbolic sine of `x`.

size

Syntax	**`size`**`(f)`
Operands	`f` is a file
Description	Returns the file size in bytes. `f` should be previously defined by **`file`** or by **`binfile`**.

sort

Syntax	**`sort`**`(x)`
Operands	`x` is a 1-dim expression
Description	Returns a sorted version of the 1-dimensional array `x`.

sqrt

Syntax	**`sqrt`**`(x)`
Operands	`x` is an expression
Description	Returns the square root of `x`.

stdev

Syntax	**`stdev`**`(x)`
	`stdev``(x,n)`

Operands	`x` is an expression; `n` is an integer
Description	The first form returns a scalar as the standard deviation of all the elements in `x`. The standard deviation is defined as the square root of the variance of the elements in `x` (See var below for the definition of the variance). The second form returns the standard deviation of elements of `x` over the dimension `n`.

The returned value has one dimension less than `x`. For example, if `M` is a 2-dimensional array than **`stdev`**`(M,1)` returns a 1-dimensional array which is the standard deviation of the rows of `M`, while **`stdev`**`(M,2)` returns a 1-dimensional array which is the standard deviation of the columns of `M`.

strdel

Syntax	**`strdel`**`(s,p,n)` or **`strdel`**`(s,p)`
Operands	`s` is a string; `p` and `n` are integers
Description	Deletes `n` characters of `s` starting at position `p`. The position of the first character in `s` is 0. If the third parameter `n` is omitted, the characters from position `p` to the end of the string are deleted. Note that this function does not return a value but rather operates on the string itself. `s,p` and `n` may also be arrays.

strins

Syntax	**`strins`**`(s,p,t)`
Operands	`s` and `t` are strings; `p` is an integer
Description	Inserts the string `t` into `s` at position `p`. The position of the first character in `s` is 0. If `p` is greater than the length of `s`, the string `t` is appended to `s`. Note that this function does not return a value but rather operates on the string itself. `s,p` and `t` may also be arrays.

strlen

Syntax	**`strlen`**`(s)`
	`or`
	`strlen``(s,n)`

Operands	`s` is a string or array of strings; `n` is an integer
Description	The first form returns the length (number of characters) of the string `s`. The second form sets the length of the string `s` to `n` (expands or shrinks as necessary).

strlow, strlower

Syntax	**`strlow`**`(s)` or **`strlower`**`(s)`
Operands	`s` is a string or array of strings
Description	Returns a version of the string `s` where all upper-case characters have been replaced with lower-case characters.

strpos

Syntax	**`strpos`**`(s,t)`
Operands	`s` and `t` are strings or arrays of strings
Description	Returns the position of the substring `t` in the string `s`. The position of the first character in `s` is `0`. If the substring is not found in `s` the returned value is `-1`.

strsub, substr

Syntax	**`strsub`**`(s,p,n)` or **`strsub`**`(s,p)`
Operands	`s` is a string; `p` and `n` are integers
Description	Returns a substring of `s` with length `n` starting at position `p`. The position of the first character in `s` is 0. If the third parameter `n` is omitted, the substring from position `p` to the end of the string is returned. `s,p` and `n` may also be arrays.

strtonum

Syntax **`strtonum`**`(s)`

Operands `s` is a string or array of strings

Description Converts the string `s` to a number and returns its value.

strupp, strupper

Syntax **`strupp`**`(s)` or **`strupper`**`(s)`

Operands `s` is a string or array of strings

Description Returns a version of the string `s` where all lower-case characters have been replaced with upper-case characters.

substr, strsub

Syntax **`substr`**`(s,p,n)` or **`substr`**`(s,p)`

Operands `s` is a string; `p` and `n` are integers

Description Returns a substring of `s` with length `n` starting at position `p`. The position of the first character in `s` is 0. If the third parameter `n` is omitted, the substring from position `p` to the end of the string is returned.
`s,p` and `n` may also be arrays.

sum

Syntax **`sum`**`(x)`
`sum``(x,n)`

Operands `x` is an expression; `n` is an integer

Description The first form returns a scalar as the sum of all elements of `x`.
The second form returns the sum of elements of `x` over the dimension `n`. The returned value has one dimension less than `x`. For example, if `M` is a 2-dimensional array than **`sum`**`(M,1)` returns a 1-dimensional array which is the sum of the rows of `M`, while **`sum`**`(M,2)` returns a 1-dimensional array which is the sum of the columns of `M`.

svd

Syntax **`svd`**`(M)`

Operands `M` is a Real matrix

Description Returns the singular value decomposition of a Real matrix `M` (`M` may also be a 2-dimensional array rather than a matrix). Assuming that `M` is an m×n matrix (m rows, n columns), its singular value decomposition is given as a multiplication of three matrices: `M = U W V'`, where `U` is an m×n matrix, `W` is a diagonal n×n matrix, and `V'` is the transpose of an n×n matrix. The matrices `U` and `V` are each orthogonal.
`svd` returns a matrix with m+n+1 rows and n columns. The first m rows form the matrix `U`, the next n rows form the matrix `V`, and the last row consists of the diagonal elements of `W`.
To learn how to extract the sub-matrices see **Sub-Arrays** in the section **Arrays**.

tan

Syntax **`tan`**`(x)`

Operands `x` is an expression

Description Returns the tangent of `x`.

tanh

Syntax **`tanh`**`(x)`
Operands `x` is an expression
Description Returns the hyperbolic tangent of `x`.

trace

Syntax **`trace`**`(x)`
Operands `x` is a square matrix
Description Returns the trace (sum of diagonal elements) of the square matrix `x`.

transpose, transp

Syntax **`transpose`**`(x)` or **`transp`**`(x)`
Operands `x` is an expression
Description Returns the transpose of `x`. If `x` is a scalar or 1-dimensional, the operation has no effect. If `x` is 2-dimensional, the rows and columns are interchanged. If `x` has a higher dimension, the operation is generalized in the following way: each dimension is shifted forward, and the last dimension moves to the first position (e.g., a 3×5×7 array is converted to a 7×3×5 array).

trunc

Syntax **`trunc`**`(x)`
`trunc``(x,n)`
Operands `x` is an expression; `n` is an integer
Description The first form returns the integer part of `x`.

The second form returns the truncated value of `x` with accuracy 10^n, namely, `n` determines from which digit the value is truncated. For example: **`trunc`**`(1234.567,2)` is `1200` (truncated to hundreds), and **`trunc`**`(1234.567,-2)` is `1234.56` (truncated to hundredths).
The function **`trunc`** behaves like **`floor`** with positive numbers and like **`ceil`** with negative numbers; namely, it always rounds toward 0.

unpackcomplex

Syntax **`unpackcomplex`**`(x)`
`unpackcomplex``(x,z)`
Operands `x` is a Real array; `z` is a Complex array
Description This function unpacks a packed complex array (represented as a real array) back into a complex array, reversing the operation of **`packcomplex`**. The length of the complex array is half the length of the real array. The first form accepts a real array and returns the complex array. The second form receives both arrays as arguments where the second argument is the complex array whose length must be half that of the real array (this form may be used if the function is called many times with the same arrays, to save memory allocation of the complex array on each call).

var

Syntax **`var`**`(x)`
`var``(x,n)`
Operands `x` is an expression; `n` is an integer
Description The first form returns a scalar as the variance of all the elements in `x`. The variance is defined as

$\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2$, where N is the length of `x`. `x` may have any dimension.

Note that we divide by N rather than N-1, so if you need a variance with a denominator of N-1 you should multiply the result by the appropriate factor.
The second form returns the variance of elements of `x` over the dimension `n`. The returned value has one dimension less than `x`. For example, if `M` is a 2-dimensional array than **`var`**`(M,1)` returns a 1-dimensional array which is the variance of the rows of `M`, while **`var`**`(M,2)` returns a 1-dimensional array which is the variance of the columns of `M`.

zero, root

Syntax **`zero`**`(`*`exp`*`,x)` or **`root`**`(`*`exp`*`,x)`
`zero``(`*`exp`*`,x,tol)` or **`root`**`(`*`exp`*`,x,tol)`

Operands *`exp`* is a formal expression of one variable
`x` is the initial guess
`tol` is a number

Description Returns the zero of the expression *`exp`*. *`exp`* has one dummy variable over which the search for the zero is done. It is represented by the `$` (dollar) character. `x` is the initial guess from which the search starts. If the expression has several zeros, the returned value is usually, but not necessarily, the closest to the initial `x`. `x` must be real but may have any dimension. When `x` is multidimensional, **`zero`** returns an array of the same dimension as `x`, whose elements are the zeros that were obtained by taking the corresponding elements in `x` as initial guesses. A tolerance may be specified as a third argument. If not specified, it is taken as 10^{-6}.

For example, the following:

```
 x = 0,3
 z = zero($^2 - 3*$ + 2, x)
```

defines `z` as a 1-dimensional array with two elements: `1` and `2` (which are zeros of the given expression obtained by taking `0` and `3` as initial guesses).
The expression can be just a function call, for example,

```
  y = zero(fun($), x)
```

where `fun(x)` is any ***Numerit*** function (built-in or user's).

Chapter 14
Reference to *Numerit*'s commands

array

Syntax **`array`** `x`
`array` `y = x`
Operands `x` is a vector or a matrix
Description Converts `x` into an array. The first form changes the kind of `x` itself while the second form defines a new variable `y` and copies the content of `x` into it.

beep

Syntax **`beep`**
Description Generates a beep.

binfile

Syntax **`binfile`** `f = s` or **`binfile`** `f s`
Operands `f` is an identifier
`s` is a string
Description Defines `f` as a binary file; the string `s` specifies the file name.
For example:

```
binfile f = "c:\bin.dat"
```

bitmap

Syntax **`bitmap`**`(s)`
Operands `s` is a string
Description Returns a two-dimensional array containing a bitmap image. The string `s` specifies the bitmap file name.
For example:

```
b = bitmap("c:\images\mypicture.bmp")
```

The bitmap file is converted to the Image Viewer RGB format, so that it can be displayed using the Image Viewer (See the section **Image Viewer** subsection **RGB**). This format allows you to manipulate each of the three color components of the image (red, green, and blue).

break

Syntax **`break`**
`break` `n`
Operands `n` is a number
Description Used for breaking a loop. It stops the loop and transfers the execution point to the first statement after the loop block.
The second form is followed by a number which specifies how many nested loops to exit.

For example:

```
n = length(b)
i = 0
```

```
loop
  i += 1
  if i > n break        `` end the loop
  if b[i] < 0 next      `` next loop round
  a[i] = sqrt(b[i])
```

or

```
for i = 1 to m
  for j = 1 to n
    if a[i]^2 + b[j]^2 > max_val break 2  `` exit both loops
    if b[j] < 0 next 2     `` exit this loop and go to next i
    println a[i]-b[j]
```

by

Syntax `by`

Description Used in a range definition to specify an interval, and in a **`for`** loop to specify the step.

case

Syntax **`case`** *`expression  cases-block`*
`case` *`conditional-cases-block`*

Description Used for a conditional execution of statements. The first form looks like the following example:

```
case x
  8.5: y = x
  -2.2: a = x+1; y = a*x+1
  1.3,1.4: y = x-1
  5.5:
    a = x+1
    b = x-1
    y = a^2-b^2
  6.2:;
  else: y = 0
```

In this form, there is an expression at the head (`x`). The statements of each case in the *`cases-block`* are preceded by a constant (or an expression) followed by a colon. If the value of the expression in the head (`x`) is equal to one of the constants (or expressions) of one of the cases, the statements of this case are executed and no more cases are checked. If no case matches, the optionally **`else`** case is executed.
It is possible to specify more than one expression for a case. These expressions should be separated by commas (like the `1.3,1.4:` case above). The statements of such a case will be executed if a match to one of the expressions is found.
Several statements may be specified for each case. They may be written in the same line separated by semicolons (as in case `-2.2:` above), or, as a block on the next line (as in case `5.5:`). To define an empty case, a semicolon must be inserted (as in case `6.2:` above).

The second form of **`case`** looks like:

```
case
  x = 4: y = 1
  x > 1 and x < 3: y = 2
  x >= 5: y = -1
  else: y = 0
```

Here, the head is empty and each case starts with a Boolean expression which is followed by statements. The expressions are evaluated one by one and the first one that yields a **`true`** (and only the first), is executed. The optional **`else`** case is executed if no condition was satisfied. This is actually a substitute for a series of **`if-else-if`** statements.

clear

Syntax **`clear`**
`clear` `x,y,z,...`
Operands `x,y,z,...` are variables
Description The first form clears the Draft document.
The second form clears all the elements of the specified variables (set the value to 0 or clear the strings). Note that the variables are still defined and occupy the same amount of memory as before. To free the memory occupied by variables use **`free`**.
If the variable is of type File, this command clears all the file's data and puts the current position at the beginning of the file.

clock

Syntax **`clock`**
Description Returns the processor time elapsed since the beginning of the program invocation in miliseconds. This can be used to determine the time interval between two events. For example the following code:

```
t = clock
 .
 .
 .
 statements
 .
 .
 .
delta_t = clock - t
```

sets `delta_t` to the time it takes to execute the statements between the two calls to **`clock`**.

close

Syntax **`close`** `f`
Operands `f` is a file
Description Closes the file `f`. `f` should be previously defined by **`file`** or by **`binfile`**.

common

Syntax **`common`** `x,y,z,...`
Operands `x,y,z,...` are identifiers (variables names)
Description Variables that are defined in the main body of a program can be accessed by all the functions in this program using the **`%`** prefix (see **Local and Global variables** in the section **User-Defined Functions** in Chapter 6). This makes it possible for functions to share variables either as predefined constants or for transferring information between them. However, these variables are not accessible to different modules of the same program.
The command **`common`** is used to share variables between modules. It allows you to specify a list of variables that you want to share between two or more modules. A variable that appears in the **`common`** list of any two modules that are linked to the same program (in any level) is shared by these modules, i.e., both actually use the same variable. Note that you can only declare common variables in the main body of a module or a program and you

still need to use the `%` prefix to access these variables from within a function.
The lists that are specified in the **`common`** instructions of two different modules need not be identical; only those variables that appear in both lists will be shared. For example, if you put **`common`** `x,y,z` in module A, **`common`** `x,y` in module B, and **`common`** `y,z` in module C, then `x` is shared between A and B, `z` is shared between A and C, and `y` is shared between A, B, and C.

const, constant

Syntax **`const`** `x = c`

Operands `x` is an identifier, `c` is a Numeric, Boolean, or String constant

Description By declaring a constant you can use the name of the constant instead of the constant itself in any ***Numerit*** instruction. This is useful when you want to use the same constant in various instructions or expressions and be able to change it with a single assignment. In most cases you can use a standard ***Numerit*** variable for the same purpose, however, the use of a declared constant ensures that it will not be modified by the program. In some cases a variable cannot replace a constant, for example, when specifying the parameters of output formats such as **`outprec`** or **`outwidth`**. Constants can only be declared outside functions but can be used inside functions (without the global prefix %). The actual constant replaces the declared name in the code at compilation time.

Examples:

```
const N = 12
x[N]:0
y[N]:0

const NAME = "c:\data\"
directory NAME
```

delete

Syntax **`delete`** `f`

Operands `f` is a file

Description Deletes the file `f` from the disk. `f` should be previously defined by **`file`** or by **`binfile`**.

dfunc

Syntax **`dfunc`** *`function-prototype`*
or
`dfunc` *`block of function-prototypes`*

Operands *`function-prototype`* is a DLL function prototype

Description Declares a function of a dynamic-link library (DLL). The function is assumed to be a C or C++ function that was compiled into a standard Win32 (32-bit) DLL. The function is searched for in the library that has been declared on a previous **`dll`** statement. For a detailed description of DLL functions in ***Numerit***, including examples, see **Dynamic-link functions** (in Ch. 6).
The **`dfunc`** statement is used to declare functions that were compiled with 'cdecl' calling convention. A function that was compiled with the 'stdcall' calling convention should be declared with the statement **`sdfunc`**.
The **`dfunc`** (or **`sdfunc`**) statement includes a DLL function prototype on the same line or a number of function prototypes in a subsequent block. The format of such a prototype is:

```
ret_type name(parameters):actual_name
```

where *`ret_type`* is optional and is the function's return type, *`name`* is the name that the function will have in ***Numerit***, *`parameters`* is the list of parameter types that the

function expects to receive, and `actual_name` is the actual name of the function in the library. The colon and the `actual_name` that follows it may be omitted if the actual function's name is `name` (i.e., the name that you want to use when you call the function from ***Numerit*** is the same as its name in the library). Note that if `name` (or `actual_name`) is not found in the library ***Numerit*** will look for it using the name preceded by an underscore since many compilers attach an underscore to global names. For example, if `name` (or `actual_name`) is `foo` then ***Numerit*** first searches for the function `foo` and if it is not found ***Numerit*** searches for the function `_foo`.
If `ret_type` is specified it must be one of the following: **`void`**, **`char`**, **`uchar`** (or **`unsigned char`**), **`short`**, **`ushort`** (or **`unsigned short`**), **`int`**, **`uint`** (or **`unsigned int`**), **`float`**, **`double`**, **`complex`**, **`ptr`** (or **`pointer`**). **`void`** is specified when the function doesn't return a value. If `ret_type` is omitted it is assumed to be **`void`**.
The `parameters` are listed inside the parenthesis as a list of types, separated by commas, one for each actual function parameter. Each type can be followed by an optional parameter name, and the allowed types are the same as listed above except **`void`** (which can only be used for `ret_type`).
Functions that belong to the same DLL that was declared in a previous statement can be grouped together into a single **`dfunc`** block, namely,

```
dll "mylib.dll"

dfunc
  [void set(int n, double x)
  [double sqr(double x):nlib_pow2
```

A scalar parameter
A scalar parameter is specified by its type and an optional name. A scalar can be sent to the function in two ways: "by value" or "by reference". When a variable is sent to a DLL function "by value" the function reads the value from the stack but has no access to the variable itself. When a variable is sent to the function "by reference", the function has full access to the variable itself; it can read its value and also modify it. A scalar is sent "by value" when the parameter is specified with only the type (e.g., **`double`** `x`). A scalar is sent "by reference" when the parameter is specified with a type and the address mark `&` (e.g., **`double`**`& x`). A scalar can also be sent as an argument to a function that expects a one-dimensional array (see below). In such a case the scalar will be sent "by reference" and will actually be treated as an array with one element.

An array parameter
An array parameter is specified by its type, an optional name, and square brackets. One-dimensional arrays are specified with a single pair of brackets (e.g. **`double`** `x[]`). A scalar can be sent as an argument to a function that expects a one-dimensional array. In such a case the scalar will be sent "by reference". This is actually equivalent to sending an array with one element. Multidimensional arrays are specified with two pairs of square brackets (e.g. **`double`** `x[][]`). All numeric types are allowed except **`complex`**. A complex array should be converted to a packed-array with the function **`PackComplex`** and passed as a **`double`** array to the DLL function. The packed-array is converted back to a ***Numerit*** complex array with the function **`UnpackComplex`**. ***Numerit*** arrays with more than two dimensions are "flattened" to two dimensions and are passed to the DLL function as two-dimensional arrays. See **Dynamic-link functions** for details about the structure of two-dimensional arrays that are sent as arguments to DLL functions.

A string parameter
Strings are treated in C as one-dimensional arrays of characters. When a DLL function has a parameter that is a one-dimensional array of type **`char`** you can send to it a ***Numerit*** String as an argument. Note that although in ***Numerit*** a string is considered to be a scalar when sent as an argument to a DLL function it is sent as an array of **`char`**'s. A one-dimensional array of strings can be sent as an argument to a function that expects a two-

dimensional array of **char**'s.

A Boolean parameter
In ***Numerit*** a Boolean variable (namely, a variable that can have only one of two values, **true** or **false**) occupies one byte. So, you can only send it as an argument to a DLL function that expects a parameter of type **char** (or **unsigned char**). This is true for Boolean scalars as well as Boolean arrays.

A structure parameter
Structures in C are objects that include several variables of different types that are grouped together. Unlike C, ***Numerit*** doesn't support structures but it allows you to group together several variables and send them as a single structure to a DLL function. ***Numerit*** actually creates a structure at runtime and sends its address to the DLL function ("by reference").
A structure is declared in the **dfunc** statement with braces surrounding a list of types, for example the parameter {**int,double,double**} represents a structure whose first member is an integer and the second and third members are **double**'s.
An array can be specified as a member of a structure with a size or without it. If the size is specified (e.g., {**int**[10]**,double**}), the array's elements are copied to the structure when the structure is created. If the size is not specified (e.g., {**int**[]**,double**}), the array's address is copied to the structure. The same is true with two-dimensional arrays, namely, when you specify an array without its size (e.g., **int**[][]), the address of an array is copied to the structure. To copy the actual array elements you must specify the size in both dimensions (e.g., **int**[4][5]). On return ***Numerit*** copies back the values of arrays that were specified with size. If this is not required you can save execution time and instruct ***Numerit*** to skip the copy operation by adding the '~' mark after the square brackets of the array (e.g., {**int**[10]~**,double**}).
Numerit does not allow you to specify a structure inside a structure. If this is required you must specify the members of the inner structure as members of the same structure (when possible).

A function parameter
Numerit allows you to send (user-defined) ***Numerit*** functions as arguments to DLL functions. Such functions are referred to as "callback" functions since they are called from the called DLL function (you can also send a DLL function as an argument to another DLL function; see **A DLL function as a callback function** below). Note that the ***Numerit*** callback function should be called from the DLL using the 'cdecl' convention.
A ***Numerit*** function is declared as a parameter of a DLL function by declaring it as **function** (or **func**) in the list of parameters of the function and by specifying its own return type and list of parameters types, for example in:

```
   dfunc double foo(func double f(double x))
```

the function f is declared as a parameter of the DLL function foo. The callback function f is declared as a function that receives and returns a **double** scalar. The allowed types of the callback function parameters are the same as the types that can be specified in a DLL function declaration except a function parameter (namely, a callback function does not accept another function as an argument). The ***Numerit*** function should match the specified callback function (in number and types of arguments); a mismatch will result in unpredictable results. When the DLL function is called the callback function should be sent to it as an argument with a prefix @ (e.g., foo(@f)).
A callback function may also be specified as a member of a structure, for example,

```
  dfunc double foo({func double f(double x), double p})
```

Here too, when foo is called the callback function is specified inside the braces with the prefix @ (e.g., foo({@f,p})).
When a ***Numerit*** function is called back by a DLL function, the arguments that are sent by the DLL function are converted to ***Numerit*** variables and are then sent to the ***Numerit***

function. When the argument is an array ***Numerit*** must know its size in order to carry out the conversion. That's why an array parameter in the declaration of a callback function must also include its length. This can be done in one of the following three ways:
1. The array has a fixed known size. In such a case the size of the array should be specified inside the square brackets (e.g., `func double f(double[3])`).
2. One of the arguments before the array is used to specify the length. In such a case the number of this argument is specified instead of the actual length by adding the prefix '#' to the number (e.g., `func double f(int,double[#1])`, tells ***Numerit*** to take the length from the first argument).
3. The array's length is taken from a global variable (e.g.,
`func double f(double[n])`, where `n` is a global variable). This method is useful when the array's length is under the control of the ***Numerit*** program and in most cases will also be sent as an argument to the DLL function.
Two-dimensional arrays should be specified with both dimensions (e.g.,
`func double(double[3][5])`). Any combination of the above three methods is allowed in such a case (e.g., `func double f(int,double[3][#1])`).
The values of array elements are copied back to the DLL function argument when the callback function completes its calculation. If these values are not required by the DLL function it is possible to skip this copy operation (and save execution time) by adding the '~' mark after the square brackets of the array (e.g.,
`func double f(int,double[#1]~)`.
The argument of a callback function can also be a structure. The parameters in the declaration, in such a case, are specified inside braces (e.g.,
`func double f({int,double[#1]})`). In this case ***Numerit*** assumes that the DLL function pushes the address of the structure to the stack as an argument (i.e., that the structure is sent "by reference"). Note, however, that since ***Numerit*** doesn't have structures, the actual callback ***Numerit*** function will receive the structure members as separate variables. The user must make sure that the number of arguments of the callback function matches the number of members in the structure sent by the DLL function (plus, of course, any other arguments that are sent in addition to the structure).
When the array size is specified inside a structure as the number of a parameter (i.e., with the prefix `#`), the numbering is done from the beginning of the specific structure and not from the beginning of the parameter list (e.g., in
`func double({int,double[#1]},{int,double[#1]})` `#1` is used in both structures to specify that the length is the first member in each of them).

A DLL function as a callback function
A DLL function can be sent to another DLL function as an argument to serve as a callback function. In such a case ***Numerit*** sends the address of the callback function as an argument to the DLL function that calls it directly without involving ***Numerit*** in the process.
When a DLL function is specified as a parameter to another DLL function, its type in the function declaration should be **`ptr`** (or **`pointer`**). It is the user's responsibility to make sure that the DLL callback function matches in types and number of parameters the expected callback function. The callback function must also be declared with **`dfunc`** (or **`sdfunc`**) and is specified as an argument the same way as a ***Numerit*** function is specified, namely, using the `@` prefix.

directory

Syntax **`directory`** `s`
Operands `s` is a string
Description Sets the working directory from this point on to `s`.
For example:

```
directory "c:\data\"
```

Files that are defined after this statement without specifying their full path name (in the **file** or **binfile** statements) will be opened in this directory.

dll

Syntax **dll** s
Operands s is a string
Description Declares the dynamic-link library (DLL) where the succeeding DLL functions and DLL variables are searched for. For example:

```
dll "my.dll"
dfunc void set(int n, double x)
dfunc double sqr(double x)
dvar double x_glb()


dll "other.dll"
dfunc int mode()
dvar int flag()
dvar param(int)
```

The library 'my.dll' will be searched for the functions `set()`, `sqr()`, and the variable `x_glb`, while the library 'other.dll' will be searched for the function `mode()` and the variables `flag` and `param`.

The string 's' may specify the full path of the DLL, e.g.,

```
dll "c:\lib\my.dll"
```

or only its name, namely,

```
dll "my.dll"
```

In the first case ***Numerit*** will look for the DLL only in the given directory. In the second case it will look for it in the following directories (in the given sequence):
1. The ***Numerit*** directory (where ***Numerit*** was installed).
2. The current ***Numerit*** program directory.
3. The 32-bit Windows system directory (e.g., c:\windows\system32).
4. The Windows directory (e.g., c:\windows).
5. The directories that are listed in the PATH environment variable.

do

Syntax **do**
Description An optional **do** may appear after the condition of a **while**, **until**, or **for** loops to separate between the condition and the statement(s) that follow it. The **do** is redundant and is actually ignored by the compiler.

downto

Syntax **downto**
Description Used to specify the lower limit of a decreasing **for** loop.

draft

Syntax **`draft`**

Description Displays the Draft in the Document pane. When the program encounters the command **`draft`** it checks the Document pane and if it is displaying the Report it is switched to the Draft. This allows the program to send output to both the Report and the Draft and to control which of them is currently displayed.

dvar

Syntax **`dvar`** *`variable-prototype`*
or
`dvar` *`block of variable-prototypes`*

Operands *`variable-prototype`* is a DLL variable prototype

Description Declares a variable of a dynamic-link library (DLL). A DLL may contain exported variables in addition to functions. ***Numerit*** allows you to access such variables and read or set their value.
Note that a DLL variable is treated in ***Numerit*** as if it is a function and not a variable. So, to read or set its value you must call it like a function, namely, with parenthesis.

A variable prototype has one of the following forms:

  `type name``():`*`actual_variable_name`*
or
  `name``(`*`type`*`):`*`actual_variable_name`*
or
  `& name``():`*`actual_variable_name`*

The *`name`* is the name that will be used to access the variable in ***Numerit*** and *`actual_variable_name`* is the actual name of the variable in the library. The colon and the *`actual_variable_name`* that follows it may be omitted if the actual variable's name is *`name`* (i.e., the name that you want to use when you access the variable from ***Numerit*** is the same as its name in the library). Note that if *`name`* (or *`actual_variable_name`*) is not found in the library ***Numerit*** will look for it using the name preceded by an underscore since many compilers attach an underscore to global names. For example, if *`name`* (or *`actual_variable_name`*) is `xglb` then ***Numerit*** first searches for the variable `xglb` and if it is not found ***Numerit*** searches for the variable `_xglb`.
The *`type`* must be one of the following: **`char`**, **`uchar`** (or **`unsigned char`**), **`short`**, **`ushort`** (or **`unsigned short`**), **`int`**, **`uint`** (or **`unsigned int`**), **`float`**, **`double`**, **`complex`**, **`ptr`** (or **`pointer`**).
The first form should be used when you want to *read* the value of the variable. The second form should be used when you want to *set* the value of the variable. The third form should be used when you want to get the *address* of the variable rather than its value. Note that this address has no use in ***Numerit*** but might be required by some DLL functions as an argument. Such an argument must be specified as type **`ptr`** (or **`pointer`**) in the DLL function prototype. If you want to both read and set the value of a variable you must use two separate declarations with different *`name`*'s (see example below).
You can declare more than one DLL variable with a single **`dvar`** instruction followed by a block of variable prototypes.

Examples
The following code declares four **`dvar`**'s: `dparam()`, `flag()`, `set_flag()`, and `x_addr()`. The first is used to read the value of the DLL variable `dparam` of type **`double`**, the second and third are used to read and set the value of the same DLL variable `nlib_control_word` of type **`int`**, and the fourth returns the address of the DLL variable `x_global`:

```
dvar double dparam()
dvar int flag():nlib_control_word
dvar set_flag(int):nlib_control_word
dvar & x_addr():x_global
```

The following code reads the value of the DLL variable `dparam` and puts it in the ***Numerit*** variable `dp`:

```
dp = dparam()
```

The following code reads the value of the variable `nlib_control_word`:

```
c = flag()
```

and the following code sets its value to 7:

```
set_flag(7)
```

The following code reads the address of the DLL variable `x_global`:

```
xa = x_addr()
```

else

Syntax **`else`**
Description May appear at the end of **`if`**, **`where`**, or **`case`** statements. It is followed by statements that are executed if no condition was satisfied.

file

Syntax **`file`** `f = s` or **`file`** `f s`
Operands `f` is an identifier
`s` is a string
Description Defines `f` as a text file; the string `s` specifies the file name.

For example:

```
file f = "c:\txt.dat"
```

fopendlg

Syntax **`fopendlg`**`(t,fn,ft)`
Operands `t` and `fn` are strings, `ft` is a string array with two columns and any number of rows.
Description Opens a dialog box that allows the user to specify a file for opening. The user can select a disk drive, a directory path, and a file name. The returned value is a String that contains the selected file name including the full path. The parameter `t` specifies the title of the dialog box, The parameter `fn` specifies the default file name that appears in the dialog box. The parameter `ft` specifies the list of file types that will be available for the user. The first string in each row of `ft` is a description of the file type; the second string is a filter that specifies the file types that should be displayed in the dialog box when each description is selected.

For example:

```
st =
 ⌈"Documents","*.txt"
 |"Data","*.dat"
 ⌊"All","*.*"
```

```
  file f = fopendlg("Open","mydoc",st)
  read f a,b,c
```

firstindex

Syntax `firstindex n`

Operands `n` is a number

Description Sets the starting index of arrays. Valid values are integers in the range -32000 to 32000. The default value of `n` is 1.
Different values of **`firstindex`** may be set in different points of the same program. The specified value will be in effect until a new value is set.

for

Syntax
```
for var = exp to exp [by value] [do] statements
for var = exp downto exp [by value] [do] statements
```

Description Defines a loop that is controlled by a variable which goes between two limits. The first form is used when the control variable should be increased while looping, and the second form is used when the control variable should be decreased while looping. The default step is 1 and a different value may be specified with the keyword **`by`** (followed by a constant or a variable's name). The keyword **`do`** is optional (but is redundant and is ignored by the compiler).
The control variable starts from the first limit and terminates the loop when it passes the second limit. The limits and the step may have dimensions. In such a case the control variable is an array whose first element is used for controlling the loop (i.e., testing for termination).
When there is only one statement in the loop it may appear on the same line, for example:

```
  for i = 1 to 100  x[i] = i^2
```

Otherwise, a block should be used, for example:

```
  for i = 1 downto 2 by 2
  ⎡x[i] = i^2
  ⎣if i > 50  print x[i]
```

free

Syntax `free x,y,z,...`

Operands `x,y,z,...` are variables

Description Free the memory occupied by the specified variables. The memory is returned to the operating system so that more memory becomes available to the program. If the variable is a file this command also closes the file.
The specified variables become undefined but still keep their type (Numeric, Boolean, String, or File). So, if they are assigned a new value, it must be of the same type.

freedll

Syntax
```
freedll s
freedll
```

Operands `s` is a string

Description Frees the Dynamic-link library (DLL) whose name is specified in the string `s`, or free all open DLLs if no name is specified. For example:

```
freedll "my.dll"
```

fsavedlg

Syntax `fsavedlg(t,fn,ft)`

Operands `t` and `fn` are strings, `ft` is a string array with two columns and any number of rows.

Description Opens a dialog box that allows the user to specify a file for saving. The user can select a disk drive, a directory path, and a file name. The returned value is a String that contains the selected file name including the full path. The parameter `t` specifies the title of the dialog box, The parameter `fn` specifies the default file name that appears in the dialog box. The parameter `ft` specifies the list of file types that will be available for the user. The first string in each row of `ft` is a description of the file type; the second string is a filter that specifies the file types that should be displayed in the dialog box when each description is selected. For example:

```
st =
 ┌"Documents","*.txt"
 │"Data","*.dat"
 └"All","*.*"

file f = fsavedlg("Save As","mydoc",st)
write f a,b,c
```

function, func

Syntax **`function`** *`name`*`(`*`parameters`*`) =` *`expression`*
`function` *`name`*`(`*`parameters`*`)` *`block`*

Description A function is defined by the declaration **`function`** (or **`func`**) followed by the function's name, the parameter list inside parentheses (which are empty when there are no parameters), and the function's body. A function definition may appear anywhere in the program (even after the statement that calls it) but only at the top level, namely, it cannot appear inside a block.
There are two possible forms:
1. Single line. The function header is followed on the same line by an equal sign and an expression which will be evaluated and returned when the function is called. For example:

```
function foo(a,x) = a*x^2
```

2. Multiline. The function header is followed on the next line by a block of statements. For example:

```
func foo(a,x)
 ┌y = a*x^2
 └return y
```

A function's name may be any valid identifier and the parameter list consists of identifiers that are separated by commas.
See section **User-Defined Functions** in Chapter 6 for more details.

goto

Syntax **`goto`** *`label`*

Description Transfers the execution point to a labeled line whose label is specified.

graph

Syntax **`graph`** `y1:x1:e1:t1, y2:x2:e2:t2,...`

Operands
`y1,y2, ...` are scalars or 1-dim expressions
`x1,x2, ...` are optional scalars or 1-dim expressions
`e1,e2, ...` are optional scalars or 1-dim expressions
`t1,t2, ...` are optional strings

Description Inserts a graph at the current position in the Draft. Each group of four arguments (`y:x:e:t`) represents a trace in the graph. Up to 16 traces may be specified for the same graph. The first argument in each group (`y`) is drawn vs. the second argument (`x`); the third argument (`e`) represents sizes of error bars; the fourth argument (`t`) is a legend label for the trace.

Except for the first argument in each trace (`y`), any of the other arguments may be omitted. When an argument is omitted and there are no more arguments in the group, the colons that follows it may also be omitted. When the `x` argument is not specified, the index of the `y` argument is used instead.

Some valid forms of **`graph`**:

`graph` `y1,y2`	draws `y1` and `y2` vs. the index
`graph` `y1:x, y2:x`	draws `y1` and `y2` vs. `x`
`graph` `y1:x1::"trace1", y2:x2::"trace2"`	draws `y1` vs. `x1` and `y2` vs. `x2` with a legend

Note:

The command **`graph`** actually inserts a Graph Viewer in the Draft. As any other Viewer, this may also be edited to change its attributes. It can also be copied from the Draft to the Report. The data in this Viewer is "frozen", namely, the graph shows the data as it was at the moment when it was created. Further changes to the variables that appear in the Viewer will not affect the displayed data. Note that unlike manually inserted Viewers, here the arguments may be expressions and not only variables.

hideerror

Syntax **`hideerror`**

Description Disables the automatic execution error handling. While this command is in effect (may be canceled by **`showerror`**) the system doesn't pause the program on execution errors. Instead, the program continues to run and the user is responsible for handling errors. Use this command with care. Put it before an operation, check the error code right after the operation (by checking the value of the system variable **`errval`**), and resume automatic error handling using **`showerror`**.

If, after checking the error code, you decide to handle the error, you will normally reset the error flag (using **`reseterror`**) before you resume automatic error handling.

For example, suppose we call the function **`integ`** and we want to change the tolerance parameter when the process doesn't converge. In this case the execution error is '**Not converging**', and the value of **`errval`** is 3 (See table of execution error codes). Normally, this will pause the program and display the error message. In the following code we disable automatic error handling before computing the integral and check the error code right after it. If there is no error (**`errval`** is 0), or if the error is other than '**Not converging**', we go directly to the command **`showerror`** which resumes automatic error handling and generates an error message if the value is nonzero. If the error code is 3, we reset it to 0 (**`reseterror`**), increase the tolerance and try again. If the second attempt also fails, the error code is again 3 and will be displayed when automatic error handling is resumed.

```
x = 0..10
tol = 1e-9
HideError
y = integ(sin($),x,tol)
if errval = 3
 ┌ResetError
 │tol = 1e-6
 └y = integ(sin($),x,tol)
ShowError
```

if

Syntax **`if`** *`Condition`* `[`**`then`**`]` *`Statements`*
`if` *`Condition`* `[`**`then`**`]` *`Statements`* **`else`** *`Statements`*

Description Used for a conditional execution of statements. In the first form the statements are executed if the condition is **`true`**. In the second form the first group of statements is executed if the condition is **`true`**, otherwise, the second group that follows the **`else`** is executed. The keyword **`then`** may follow the condition, but it is redundant.
The condition is a Boolean expression with no dimensions (scalar).
If there is only one statement to execute, it may appear on the same line right after the condition, for example:

```
if a > b  a = b  else  a = -b
```

If there are several statements to execute, they must be put in a block, for example:

```
if a > b
 ⎡a = b
 ⎣b = 0
else
 ⎡b = a
 ⎣a = 0
```

input

Syntax **`input`** `tx x, ty y, tz z,...`
Operands `x,y,z,...` are variables
`tx,ty,tz,...` are optional strings
Description Opens a dialog for user's input into the variables. The dialog is opened for each variable in turn. If a string precedes the variable, it is used as the title for its dialog, otherwise, the variable's name is the title.
Only scalars and 1-dimensional variables may receive user's input. The input for a 1-dimensional variable is a list of items separated by spaces. If the variable is defined as complex before execution of the Input command, the values are read as pairs of real and imaginary values (i.e., R I R I …).
The variables should normally be defined before Input is executed, namely, their type and size should be known before they are read. Input into an undefined variable defines it as a Real scalar or 1-dimensional array according to the number of values entered.

len

Syntax **`len`**
Description Used in a range definition to specify the number of elements.

loop

Syntax **`loop`** *`Statements`*
Description Defines an infinite loop. The statement or block of statements that follow it are executed until the loop is broken by a **`break`**, or a **`goto`** statement.
For example:

```
x = 0
loop
 ⎡x += 1
 ⎢if x > 4 break
 ⎣print x
```

matrix

Syntax `matrix x`
`matrix y = x`
`matrix y[m,n]:v`

Operands `x` is a 2-dimensional array; `y` is an identifier; `m` and `n` are integers, and v is a scalar expression

Description The first form converts the 2-dimensional array `x` to a matrix by changing its kind.
The second form defines `y` as a matrix and copies the content of the 2-dimensional array `x` into it.
The third form defines an `m`×`n` matrix `y` and initializes its elements to the value `v`.

menu

Syntax `menu(i,j,s)`

Operands `i` and `j` are integers; `s` is a string array

Description Opens a menu box that presents to the user a list of options. When the user clicks one of the options the menu is closed and the number of the option is returned. If the user clicks outside the menu box the returned value is 0. `i` and `j` specify the position (in pixels) of the menu box relative to the top-left corner of the program window. `s` is an array of strings that specifies the options presented to the user. For example:

```
s = "Monday","Tuesday","Wednesday","Thursday","Friday"
n = menu(10,10,s)
```

message

Syntax `message(t,s)`
Operands `t` and `s` are strings

Description Opens a message window where the string `t` is the window's title and the string `s` is the message text. The user must click the **OK** button to continue program execution.

next

Syntax `next`
`next n`

Operands `n` is a number

Description Used for jumping directly to the next loop round.
The second form is followed by a number which specifies how many nested loops (not rounds!) should be skipped.

For example:

```
n = length(b)
i = 0
loop
 ⎡i += 1
 ⎢if i > n break        `` end the loop
 ⎢if b[i] < 0 next      `` next loop round
 ⎢a[i] = sqrt(b[i])
 ⎣
```

or

```
for i = 1 to m
 ⎡for j = 1 to n
 ⎢ ⎡if a[i]^2 + b[j]^2 > max_val break 2 `` exit both loops
 ⎢ ⎢if b[j] < 0 next 2   `` exit this loop and go to next i
 ⎢ ⎢println a[i]-b[j]
 ⎢ ⎣
 ⎣
```

ocmessage

Syntax **`ocmessage`**`(t,s)`

Operands `t` and `s` are strings

Description Opens an OK/Cancel message window where the string `t` is the window's title and the string `s` is the message text. The user must click either the **OK** or the **Cancel** buttons to continue program execution.
`ocmessage` returns `1` if the user clicked **OK** and `0` if the user clicked **Cancel**.

For example:

```
if ocmessage("AppName","OK to save x?") = 1 write f x
```

outformat, outf

Syntax **`outformat`** *`fmt`* (or **`outf`** *`fmt`*)
or
`outformat -`*`fmt`* (or **`outf -`***`fmt`*)

Operands *`fmt`* can be: `auto`, `fixed`, `scientific` (or `sci`), `point`, `left`, `right`

Description Sets the format of numbers when writing data to a file or to the Draft. The parameter *fmt* can be one of the following:
`auto` – the output format is determined automatically according to the data.
`fixed` – floating-point output in fixed-point notation.
`scientific` (or `sci`) – floating-point output in scientific notation.
`point` – forces a decimal point.
`left` – adjust the output to the left of the output field as defined by **`outwidth`**.
`right` – adjust the output to the right of the output field as defined by **`outwidth`**.

Note that specifying **`outformat`** without a parameter is equivalent to **`outformat`** `auto`, and resets all the other flags. Each of the flags can be reset individually by putting a minus sign (`-`) in front of it, e.g., to reset the 'fixed' flag write **`outformat`** `-fixed`. To set more than one flag you need to call **`outformat`** once for each flag, for example:

```
outformat fixed
outformat left
```

The default format is '`auto`'. In this setting numbers are printed according to their value and the current precision, and are adjusted to the right of the output field. Note that the exact form of the displayed numbers depends on the setting of **`outprec`** (see **`outprec`** below for more details).

Some examples:

the following code -

```
a = 1234.56789
outw 10
outp 4
outf fixed
println a
outf sci
println a
outf auto
println a
```

will produce -

```
 1234.5679
 1.2346e+03
    1235
```

See more examples in the description of **`outprec`**.

outprec, outp

Syntax **`outprec`** `n` (or **`outp`** `n`)

Operands `n` is a number

Description Sets the precision of numbers when writing formatted data to a file or to the Draft. The interpretation of n depends on the current setting of **`outformat`**. The default value of `n` is 4.

When **`outformat`** is not set (or is set to `auto`), `n` defines the number of significant digits that are displayed. If the displayed number is equal to or greater than 10^n, it will be displayed in exponential format. For example:

With **`outprec`** `4`:

π	will be written as 3.142
10·π	will be written as 31.42
12345	will be written as 1.234e+04

With **`outprec`** `6`:

π	will be written as 3.14159
10·π	will be written as 31.4159
12345	will be written as 12345

When **`outformat`** is set to `fixed` the parameter `n` specifies the number of digits after the decimal point and the number is always displayed in decimal format.

For example, with **`outprec`** `4`:

π	will be written as 3.1416
10·π	will be written as 31.4159
12345	will be written as 12345.0000

When **`outformat`** is set to `scientific` the parameter `n` specifies the number of digits after the decimal point and the number is always displayed in exponential format.

For example, with **`outprec`** `4`:

π	will be written as 3.1416e+00
10·π	will be written as 3.1416e+01
12345	will be written as 1.2345e+04

outwidth, outw

Syntax **`outwidth`** `n` (or **`outw`** `n`)

Operands `n` is a number

Description Sets the width of items when writing formatted data to a file or to the Draft.

The parameter `n` defines the minimum width (number of characters) allocated to each item. Data items that occupy less space than specified are left-padded with spaces.

A value of 0 (which is the default) sets no minimum width and adds one space between the items.

outzero, outz

Syntax **`outzero`** `n` (or **`outz`** `n`)
Operands `n` is a number
Description Sets a range of small numbers around zero that are written as 0 when writing formatted data to a file or to the Draft. The parameter `n` specifies the negative exponent of the numbers that define the range. Its default value is 15, which means that numbers in the range -10^{-15} to 10^{-15} are written as 0. The instruction **`outzero`** 0, disables this feature.

pause

Syntax **`pause`**
Description Pauses the program before executing the next instruction.

pos

Syntax **`pos`** `f n`
`pos input` `m,n`
Operands `f` is a file
m, `n` are numbers
Description The first form sets the current position in the file `f` to `n` (in bytes). To move the current position to the beginning of the file use: **`pos`** `f 0`; to move it to the end of the file use: **`pos`** `f -1` (or any other negative value). If `n` is greater than the file size, the current position moves to the end of the file.
The second form sets the current position in the Input File to the line `m` and column `n`. To move the current position to the beginning of the Input File use: **`pos input`** `0,0`
Subsequent read or write operations will start from the new position.
`f` should be previously defined by **`file`** or by **`binfile`**.

print

Syntax **`print`** `x,y,z,`...
Operands `x,y,z,`... are expressions
Description Writes the content of `x,y,z,`... to the Draft document. Any number of items may be specified. The items are written one after another into the current paragraph at the current position in the Draft. The format of the written data is controlled by the current setting of **`outwidth`**, **`outprec`**, **`outformat`**, and **`outzero`**.
Subsequent **`print`** commands send output to the same paragraph. Use **`println`** to send output to different paragraphs.

println

Syntax **`println`** `x,y,z,...`
`println`
Operands `x,y,z,`... are expressions
Description Writes the content of `x,y,z,`... to the Draft document and then moves the current position in the Draft to a new line (starting a new paragraph). Any number of items may be specified. The items are written one after another at the current position in the Draft. If no items are specified, the current position in the Draft is moved to a new line.
The format of the written data is controlled by the current setting of **`outwidth`**, **`outprec`**, **`outformat`**, and **`outzero`**.

printpg

Syntax **`printpg`** `x,y,z,...`
`printpg`

Operands `x,y,z,...` are expressions

Description Writes the content of `x,y,z,...` to the Draft document and then moves the current position in the Draft to a new page. Any number of items may be specified. The items are written one after another at the current position in the Draft. If no items are specified, the current position in the Draft is moved to a new page.
The format of the written data is controlled by the current setting of **`outwidth`**, **`outprec`**, **`outformat`**, and **`outzero`**.

randomize

Syntax **`randomize`**
`randomize` `x`

Operands `x` is an expression

Description Initializes the random number generator. The first form initializes the generator with a random value. The second form initializes the generator with the seed `x`. The seed is an integer between 1 and 2^{32} that determines which sequence of random numbers will be generated by the function **`rand`**; the same seed will always generate the same sequence.

range

Syntax `x..y`
`x` **`to`** `y`
`x..y` **`by`** `d` or `x` **`to`** `y` **`by`** `d`
`x..y` **`len`** `n` or `x` **`to`** `y` **`len`** `n`

Operands `x, y, d, n` are scalar expressions (`n` is an integer)

Description Defines a 1-dimensional array ranging from `x` to `y`.
The first form (`a = x..y`) defines an array with 101 elements, namely, the interval between the elements is one hundredth the full range.
The second form (`a = a` **`to`** `y`) defines an array with interval 1 between the elements.
The third form (`a = x..y` **`by`** `d` or `a = x` **`to`** `y` **`by`** `d`) defines an array with interval d between the elements.
The fourth form (`a = x..y` **`len`** `n` or a = `x` **`to`** `y` **`len`** `n`) defines an array with n elements.

read

Syntax **`read`** `f x,y,z,...`
`read input` `x,y,z,...`

Operands `f` is a file or a string
`x,y,z,...` are variables

Description Reads data from a file or a string `f` or the Input File into the specified variables. If `f` is a file it should be previously defined by **`file`** (a text file) or by **`binfile`** (a binary file).
Any number of variables may be specified. The variables are read one after another from the current position in the file or from the beginning of the string.
Since each variable consumes data which fits its type and size, the variables should be defined before the read, namely, their type and size should be known when they are read. Reading into an undefined variable defines it as a Real scalar by default.
When reading numeric data from a text file, a string, or the Input File, any character (or group of characters) may be used as a delimiter between the numbers except the characters: `- + .` (minus, plus, and dot) which must be part of a number. This actually means that any character that cannot be interpreted as part of a number is filtered out during the read.
So, for example, you may put any comments that do not include digits in a text file and

when you read numbers from the file these comments will be automatically skipped. When reading strings they should be separated by spaces (use **`readln`** to read a whole line including spaces as a single string).
Note: It's the user's responsibility to make sure that data is read from the file in the same way it was written to it, namely, that the file was defined the same way (text or binary) and that the variables are of the right type and size.

readln

Syntax
`readln` `f`
`readln input`
`readln` `f s`
`readln input` `s`

Operands
`f` is a text file
`s` is a String variable

Description
Reads a line or lines of text from a text file `f` or from the Input File. `f` should be previously defined by **`file`**. The first two forms skip one line in the file regardless of the text in that line. The third and fourth forms read text lines into the String variable `s`. The number of lines read depends on the dimensions of `s`. If `s` has no dimensions, one line is read into it. If `s` is a string array, lines are read one by one into its elements. If `s` is undefined it is automatically defined as a simple string and one line is read into it.
Note: It's the user's responsibility to make sure that the file is indeed a text file and that the string variable is of the right dimension.

readtab

Syntax
`readtab` `f x,y,z,...`
`readtab input` `x,y,z,...`

Operands
`f` is a file
`x,y,z,...` are variables

Description
Reads data from a text file `f` or from the Input File into the specified variables. `f` should be previously defined by **`file`**. Any number of variables may be specified. The variables may be scalars, 1-dimensional arrays, or 2-dimensional arrays. For scalars **`readtab`** has no special meaning and they are read from the first line of the file. 1-dimensional arrays are read one element per line, namely, the first line is read into the first elements of each array, the second line is into the second elements, and so on. 2-dimensional arrays are read one row per line, namely, the first line is read into the first row, the second line is read into the second row, and so on.
The variables must already be defined, namely, their type and size should be known, before they are read. Each variable consumes from the file data which fits its type and size. So, each 1-dimensional array consumes one item per line, and the number of items consumed by each 2-dimensional array is equal to its number of columns.
If the number of items read from a file's line is smaller than the number of items in the line, the unread items in the line are skipped. So, it is possible to read only the first few columns of data from a file which contains many columns.
An example:
Suppose the data in a file is arranged in the following way:

```
10  20  30  40  50
11      31  41  51
            32  42  52
```

We may define 4 variables: `p`,`q`,`r`,`s` where `p` is a 1-dim array with 2 elements, `q` is a scalar, `r` is a 3×2 2-dim array, and `s` is a 1-dim array with 3 elements. After the read: `p` contains the first column, i.e., 10,11, `q` is 20, `r` contains the next two columns and `s` contains the last column, i.e., 50,51,52.
When reading numeric data from a text file any character (or group of characters) may be used as a delimiter between the numbers except the characters: **`- + .`** (minus, plus, and

dot) which must be part of a number. Strings should be separated by spaces (use **readln** to read a whole line including spaces as a single string).
Note: It's the user's responsibility to make sure that data is read from the file in the same way it was written to it, namely, that the file is indeed a text file and that the variables are of the right type and size.

refresh

Syntax **`refresh`**

Description Refreshes Viewers. When the program encounters the command **`refresh`** it scans the documents (the Draft and the Report), looking for Viewers that need refreshing. A Viewer is refreshed (i.e., updated and redrawn) if at least one of the variables that it displays has been changed since the last refresh.
Viewers are automatically checked for refresh when the program pauses or stops. However, during the run, they are not checked even if the variables they display get new values (this check is a time consuming operation), unless an explicit request by **`refresh`** is encountered. Thus, when only the final result is needed, **`refresh`** is not required since the Viewers will be updated at the end of the run. But, for displaying intermediate results a **`refresh`** is required.
Note that by using the **`refresh`** command inside a loop where variables that are displayed by a Viewer are constantly changed, an animated view of these variables is produced.

repeat

Syntax **`repeat`** *`Statements`* **`while`** *`Condition`*
`repeat` *`Statements`* **`until`** *`Condition`*

Description Defines a loop in which the statements are repeatedly executed as long as the condition that follows the **`while`** is **`true`**, or until the condition that follows the **`until`** becomes **`true`**. Since the test is done at the end of the loop, the statements in the loop are always executed at least once.
When there is only one statement in the loop it may appear on the same line, for example:

```
x = 1
repeat x += 2 while x < 10
```

Otherwise, a block should used, for example:

```
x = 5
repeat
 ⎡println x^2
 ⎣x += 5
until x > 25
```

report

Syntax **`report`**

Description Displays the Report in the Document pane. When the program encounters the command **`report`** it checks the Document pane and if it is displaying the Draft it is switched to the Report. This allows the program to send output to both the Report and the Draft and to control which of them is currently displayed.

reseterror

Syntax **`reseterror`**

Description Resets the error code to 0 so that the last execution error will be ignored by the system. See the command hideerror for more details.

return

Syntax **`return`**
`return` *`expression`*

Description Used to return from a function. May appear anywhere in the function.
The first form is used to return from a function without a value (not required at the end of the function).
The second form is used to return from a function with a value.
Note that since the first form does not return a value, the function call in this case cannot be part of an expression but must be in a statement of its own.

sdfunc

Syntax **`sdfunc`** *`function-prototype`*
or
`sdfunc` *`block of function-prototypes`*

Operands *`function-prototype`* is a DLL function prototype

Description Declares a function of a dynamic-link library (DLL). The **`sdfunc`** statement is used to declare functions that have been compiled with the 'stdcall' calling convention. Other than that **`sdfunc`** is equivalent to **`dfunc`**.

showerror

Syntax **`showerror`**

Description Resumes the automatic execution error handling. See the command hideerror for more details.

smessage

Syntax **`smessage`**`(t,s)`

Operands `t` and `s` are strings

Description Opens a Stop message window where the string `t` is the window's title and the string `s` is the message text. The user must click the **OK** button to continue program execution.

stop

Syntax **`stop`**

Description Stops the program.

system

Syntax **`system`**`(s)`

Operands `s` is a string

Description Executes a system (Windows) command. For example:

```
system("dir")
```

Opens a Windows command window and executes the `dir` command (displays a list of files and sub-directories in a directory). In order to keep the command window open, add `"&& pause"` to the command, for example: **`system`**`("dir && pause")`.

table

Syntax `table x1:t1, x2:t2,...`

Operands `x1,x2,...` are scalars or 1-dim expressions
`t1,t2,...` are optional strings

Description Inserts a table at the current position in the Draft. Each group of two arguments (`x:t`) represents a column in the table. Up to 16 columns may be specified for the same table. The first argument in each group (`x`) is the column's data and the second argument (`t`) is a column's title. If the second argument, which is optional, is not specified, the variable's name is used as the column's title.

Note:
The command **`table`** actually inserts a Table Viewer in the Draft. As any other Viewer, this may also be edited to change its attributes. It can also be copied from the Draft to the Report. The data in this Viewer is "frozen", namely, the table shows the data as it was at the moment when it was created. Further changes to the variables that appear in the Viewer will not affect the displayed data. Note that unlike manually inserted Viewers, here the arguments may be expressions and not only variables.

then

Syntax **`then`**

Description An optional **`then`** may appear after the condition of an **`if`** statement to separate between the condition and the statement(s) that follow it. The **`then`** is redundant and is actually ignored by the compiler.

time

Syntax **`time`**

Description Returns the current time and date as a string. For example:

```
s = time
```

The string `s` looks like: Mon Jan 01, 2001 10:15:08

to

Syntax **`to`**

Description Used to define a range (see range for syntax and description), or to specify the upper limit of an increasing **`for`** loop.

until

Syntax `until` *`Condition`* `[do]` *`Statements`*

Description Defines a loop in which the statements are repeatedly executed until the condition becomes **`true`**. The keyword **`do`** is optional (but is redundant).
When there is only one statement in the loop it may appear on the same line, for example:

```
x = 0
until x = 100  x += 5
```

Otherwise, a block should be used, for example:

```
i = 10
until i > 25
 ┌x[i] = i^2
 └i += 1
```

vector

Syntax `vector x`
`vector y = x`

Operands `x` is a 1-dimensional array

Description Converts `x` into a vector. The first form changes the kind of `x` itself while the second form defines a new variable `y` and copies the content of `x` into it.

wait

Syntax `wait x`

Operands `x` is a scalar expression

Description Waits for the specified amount of seconds before executing the next instruction. `x` may be any expression that yields a Real value. Note that the accuracy of the delay is machine dependent.

where

Syntax `where` *`Condition Assignments`*
`where` *`Condition Assignments`* `else` *`Assignments`*

Description Used for conditional assignment of array elements. The condition should be a Boolean array followed by a single assignment or by a block of assignments, for example:

```
where x > y  x = -y
```

or

```
where x > y
 ⎡a = x
 ⎢b = y
 ⎣c = y-x
```

The assignments take place only in elements where the condition array is true. The other elements of the assigned variable remain unchanged. The assigned variables should have the same dimensions as the condition array.
The **where** statement may be followed by an **else** with a single assignment or a block of assignments which will take place in the elements where the condition array is false. For example, after the following code is executed:

```
i = 1,2,3,4
where i > 1 and i < 4  x = i  else  x = -1
```

the value of `x` is: `-1,2,3,-1`.

while

Syntax `while` *`Condition`* `[do]` *`Statements`*

Description Defines a loop in which the statements are repeatedly executed as long as the condition is **true**. The keyword **do** is optional (but is redundant).
When there is only one statement in the loop it may appear on the same line, for example:

```
x = 0
while x < 10  x = x + 1
```

Otherwise, a block should be used, for example:

```
x = 0
while x < 5
 ⎡print x
 ⎣x += 2
```

wmessage

Syntax **`wmessage`**`(t,s)`

Operands `t` and `s` are strings

Description Opens a Warning message window where the string `t` is the window's title and the string `s` is the message text. The user must click the **OK** button to continue program execution.

write

Syntax **`write`** `f x,y,z,`...

Operands `f` is a file or a string
`x,y,z,`... are expressions

Description Writes the content of `x,y,z,`... to the file `f`. If `f` is a file it should be previously defined by **`file`** or by **`binfile`**, if it is a string it should be defined as such (e.g., `f = ""`). Any number of items may be specified. The items are written one after another at the current position in the file or at the beginning of the string.
If `f` is a text file or a string, the format of the written data is controlled by the current setting of **`outwidth`**, **`outprec`**, **`outformat`**, and **`outzero`**. If `f` is a binary file the data is written in its binary form.
Note: If this is the first file operation (the one that actually opens the file) the old data in the file is first cleared and the position is set to the beginning of the file. To keep the old data, you must use the **`pos`** command to set the current position in the file before writing (see the section **Files**).

writeln

Syntax **`writeln`** `f x,y,z,`...
`writeln` `f`

Operands `f` is a text file
`x,y,z,`... are expressions

Description Writes the content of `x,y,z,`... to the text file `f` and adds a new-line character at the end. `f` should be previously defined by **`file`**. Any number of items may be specified. The items are written one after another at the current position in the file. If no items are specified, only a new-line character is written.
The format of the written data is controlled by the current setting of **`outwidth`**, **`outprec`**, **`outformat`**, and **`outzero`**.
Note: If this is the first file operation (the one that actually opens the file) the old data in the file is first cleared and the position is set to the beginning of the file. To keep the old data, you must use the **`pos`** command to set the current position in the file before writing (see the section **Files**).

writetab

Syntax **`writetab`** `f x,y,z,`...

Operands `f` is a text file
`x,y,z,`... are expressions

Description Writes the content of `x,y,z,`... to the file `f`. `f` should be previously defined by **`file`**. Any number of items may be specified. The items are written line by line as a table at the current position in the file, namely, the first elements of `x,y,z,`... are written to the first line, the second elements of `x,y,z,`... are written to the next line, and so on.
The format of the written data is controlled by the current setting of **`outwidth`**, **`outprec`**, **`outformat`**, and **`outzero`**.
Note: If this is the first file operation (the one that actually opens the file) and the file contains data, the data is first cleared and the position is set to the beginning of the file. To keep the old data, you must use the **`pos`** command to set the current position in the file before writing (see the section **Files**).
Note: If this is the first file operation (the one that actually opens the file) the old data in the

file is first cleared and the position is set to the beginning of the file. To keep the old data, you must use the **pos** command to set the current position in the file before writing (see the section **Files**).

ynmessage

Syntax **ynmessage**(t,s)
Operands t and s are strings
Description Opens a Yes/No message window where the string t is the window's title and the string s is the message text. The user must click either the Yes or the No buttons to continue program execution.
ynmessage returns 1 if the user clicked Yes and -1 if the user clicked No.

For example:

```
  if ynmessage("AppName","Stop program?") = 1 stop
```

yncmessage

Syntax **yncmessage**(t,s)
Operands t and s are strings
Description Opens a Yes/No/Cancel message window where the string t is the window's title and the string s is the message text. The user must click either the **Yes**, **No**, or **Cancel** buttons to continue program execution.
yncmessage returns 1 if the user clicked **Yes**, -1 if the user clicked **No**, and 0 if the user clicked **Cancel**.

For example, the following may be a test inside a loop:

```
  .
  .
  rc = yncmessage("AppName","Continue with current item?")
  if rc = 0 break else if rc < 0 next
  .
```

Numerit Software License Agreement

Acceptance

KEDMI Scientific Computing grants you a non-exclusive license to use ***Numerit*** - a software and accompanying documentation ("SOFTWARE"). Exercising your rights to use the SOFTWARE indicates your acceptance of the terms of this Agreement.

Copyright

All intellectual property rights in the SOFTWARE are owned by **KEDMI Scientific Computing** and are protected by international copyright laws and international treaty provisions.

Use of Evaluation Edition

You may use the Evaluation Edition of the SOFTWARE free of charge on one or more computers.

Use of Professional Edition or Standard Edition

You may install and use the Professional Edition or Standard Edition of the SOFTWARE on a single computer provided that you have obtained a license for it. Under no circumstances may you use the SOFTWARE on more than one computer at a time unless you have obtained a site license. If you have obtained a site license you may install and use the licensed Professional Edition or Standard Edition of the SOFTWARE on more than one computer provided that your site license is covering the number of people that use the SOFTWARE simultaneously. You may make a copy of the software for backup purposes only. Your Registration Code and Registration File should be kept confidential and you may not transfer them to third parties.

Restrictions

You may not reverse engineer, decompile, or disassemble the SOFTWARE.
You may not remove any component, copyright notices, or proprietary notices from the SOFTWARE.
You may not rent or lease the SOFTWARE, but you may transfer your rights under this License on a permanent basis provided you transfer the SOFTWARE and all documentation and media and you do not retain any copies, and the recipient agrees to the terms of this License. If the SOFTWARE is an upgrade, any transfer must include all prior versions of the SOFTWARE.

Limited Warranty

The SOFTWARE is provided without warranty of any kind, either express or implied, including, without limitation, fitness for a particular purpose.
Defective media or documentation will be replaced at no charge provided they are returned within 90 days of the date of delivery.

Termination

The license will terminate automatically if you fail to comply with the terms and conditions of this Agreement. On termination, you must destroy all copies of the SOFTWARE.

Limitation of liability

In no event will **KEDMI Scientific Computing** be liable for any damages including any lost profits, data or information, or other indirect, incidental, special, or consequential damages arising out of the use of or inability to use the SOFTWARE, even if advised of the possibility of such damages, or for any claim by any other party.