DS Lab Help Index

Product Support

What is DS Lab?
How do you Use DS Lab?
Who is DS Lab For?
Some Basic Concepts
Description of the Work Environment
The Script Language
Functions and Instructions
How to...
Common Questions and Answers

Product Support

DS Lab Help Index

Decision Support Laboratory
"The Visual Spreadsheet"

Registered licensees of DS Lab are entitled to 90 days of free technical support starting from the purchase date. Registered users of DS Lab Pro are entitled to lifetime technical support for the most current release of the software.

For Mexico, USA, Canada:

Technical support is available M-F 9:00 am to 5:00 pm **Eastern Time** at +1 (203) 861-1833, on CompuServe our ID is 71552,1645 or you can write to us.

DS Group, Inc.
474 North Street
Greenwich, CT 06830-3449 (USA)

For other countries:

Technical support is available M-F 9:00 am to 6:00 pm **Italy Time** at +39 (2) 26111981, on CompuServe our ID is 100113,2272 or you can write to us.

DS Group Srl
Viale Monza, 27
20125 Milano (ITALY)

We are sorry that we cannot afford to provide technical support for users of the working model but if you have questions about DS Lab contact us.

What is DS Lab?

Decision Support Laboratory is a visual spreadsheet. It provides an easier and faster way to build, edit and explain a spreadsheet model. The basic building block of traditional spreadsheets is a cell. The basic building block of DS Lab is an element. With DS Lab, you define your model the same way you would think through a problem, using symbols for elements such as Inputs, Variables, Constant, Series and Tables to define the data you work with.

DS Lab graphically shows the logic of the model by letting the user represent and manipulate the data with symbols connected by arrows. The symbols (elements) display the data one step at a time, while leaving all of the data accessible in an instant with the click of a mouse. The arrows display the relationship between the elements. DS Lab easily manages hundreds of elements over thousands of steps. Unlike traditional spread sheets that hide the logic of the model behind columns and rows of data, DS Lab makes it easy to understand how the numbers are generated, easy to manage large arrays of numbers, and easy to edit the logic of the model.

DS Lab simplifies and speeds building formulas through the use of an English language "script". A script can be as simple as a spreadsheet type formula or as complex as a program. DS Lab provides nearly 200 predefined mathematical, financial, time, goal-seeking and logical functions which are available with a click of the mouse. Each function or keyword is shown with a description. When selected, a function is inserted into the script including its argument syntax, in plain English. Each of the connected elements is also displayed, thus most of the script writing is a matter of pointing and clicking. Unlike traditional spreadsheets there is no risk of including the wrong cell or range reference in a formula, nor of missing one that should be included.

The logic of the model is automatically applied to all the steps of the model. Time is the parameter most commonly defined for these steps, so DS Lab provides a built in calendar to define days, weeks, months or years. Steps may be defined as something other than time, such as geography or product type by simply creating a list of items. The software will then automatically generate the values for each step of the user-defined parameters. With DS Lab there is no risk of incorrectly copying formulas across a range of cells.

The results can be printed in WYSIWYG graphical decision flow images or in tabular format. Using a Copy Paste or Paste Link command, they can also be automatically exported to any Windows program for further manipulation, publishing or graphing. Integration with Excel is further automated allowing for automatic export with the click of a single button. The plain English scripts can also be printed for clear and automatic documentation of the model. A much simpler process than trying to read lists of formulas with cell references and obscure function.

DS Lab is designed to function stand-alone and provide functionality not found in spreadsheets, but works best in conjunction with Windows spreadsheets (Excel in particular). DS Lab provides six different ways to export data (one as simple as one click of the mouse) and supports DDE links, both inbound and outbound, to any other Windows application which supports DDE. DS lab is designed to provide a complex modelling environment which then makes use of the strengths of spreadsheets and graphing packages for charting and publishing functions.

How do you Use DS Lab?

DS Lab allows you to draw flow-chart type models instead of having to define them by means of complex relationships between spreadsheet cells.

DS Lab splits the simulation process into four distinct parts:

- the graphical building of the model;
- its mathematical definition;
- the actual simulation;
- reporting the results.

DS Lab is a *visual model editor*. That is, the elements of the model are visually represented on a page and the logical and mathematical links between them are shown by connecting arrows. This flow-chart approach allows a clear overview of the model, making it easy to work on. In a spreadsheet, the model is hidden behind the cells. DS Lab puts it in the foreground. This characteristic makes DS Lab the ideal tool to help you lay out the logic of a situation. Changing the layout is a simple drag and drop process.

After drawing the model, it must be defined mathematically. Each graphical symbol represents an element in a calculation, together with the functions, instructions and operators of the DS Lab working environment.

DS Lab provides calendar, mathematical, trigonometric, statistical, bond, short term note, equity risk and value, portfolio, inventory, financial, cash flow, logical, DDE (Dynamic Data Exchange) and programing functions. All of the functions are readily accessible with a click and are displayed with a description. In addition, the on-line help fully documents each function (its use and error messages) and provides an example. DS Lab will also return context sensitive error messages which very quickly and clearly identifies the problem in a script.

Once the model has been drawn and mathematically defined, you can start the simulation. In this phase you will again appreciate how easy DS Lab is to use. Simply by using a scroll bar, you can move back and forth through the steps and instantly see the situation at any particular point in the simulation. It is equally easy to go back and change the initial data, or even the logic of the calculations, by adding new elements to the original problem.

The results of simulations can easily be transferred to any other Windows application, particularly to an Excel spreadsheet, where they can be formatted and printed out as reports. Similarly, it is easy to create charts to show the changes over time of the main elements in the model. Simply select the elements to be shown and, at the touch of a button, DS Lab automatically sends to Excel all the commands needed to build the chart. The result is a standard Excel chart, so all the tools provided by Excel are available.

Thus DS Lab is a new type of product in the world of computing. It is not intended to replace the spreadsheet, but to complement it. In fact, though both products can work separately, they are used to their best advantage together. DS Lab is software for the development and use of *Decision Support Systems*. It can be used as a CASE tool for the spreadsheet, or as a spreadsheet with graphic symbols in the place of cells.

Who is DS Lab For?

DS Lab can be used in project management, financial analysis, strategic or operative planning whenever it is necessary to prepare budgets, cash flows or any kind of forecasting model. In short, DS Lab can be used in any situation where you want to represent a system by means of a model that is, a series of linked elements and evaluate the changes in the model over time, showing the results in graphic form or as numerical reports. Its use, then, is not restricted just to the business world. DS Lab is suitable for problem-solving in all kinds of areas, including engineering, biological, physical and social simulations.

Users fall into into two major categories:

1. Traditional spreadsheets users who find the DS Lab graphical approach to modelling faster and easier to use when defining and presenting their models. For these users DS Lab was created as an adjunct to the spreadsheet. There are four significant advantages in using DS Lab over traditional spreadsheet models:

A. The graphical representation of the model makes it much **faster and easier** for the user **to develop and modify** the logic and flow of the model.

- The user can always see the logic of the model and focus on the flow and relationships of the data.
- The user is not overwhelmed by the vast number of columns and rows of data found in spreadsheets.
- All functions and elements are always available with the click of a mouse.
- Function syntax is built into the script functions so the user does not have to remember it or refer to a manual.
- Element names and functions are in plain English rather than cell references or undeciferable abreviations.
- Changes in element names, relationships and values are propagated automatically throughout the model.
- All the data is available through on screen reports with a cick of the mouse.

B. The graphical representation of the model makes it much easier for those who did not develop it to see and understand the logic and flow of the model. Those who need to read the results of the model can plainly see the relationship between the elements and are not simply confronted with tables of data.

C. The DS Lab method of managing time sequences in particular (or other unit of measure for the second dimension) is easier to manage than spreadsheets. It provides a more intuitive way of seeing the results of a model change over time, geography, product type or what ever the second dimension might be.

2. The second group includes users who require a sophisticated graphical tool for defining complex simulations involving many elements and many iterations (steps) and find spreadsheets inadequate.

DS Lab Pro was designed to meet the needs of this second category of user. Its goal is to assist the professional who needs to develop, simulate and demonstrate large and complex decision or process models in the financial, logistics, engineering, scientific, public administration and educational fields.

Examples of use by current customers include:

- A bank uses it for calculating complex bond yields.
- A petrochemical company uses it to optimize its tanker's scheduled departures.
- Several industrial companies use it for developing complex budgets.
- A carburettor manufacturer uses it to model the performance of its carburettors.
- A regional government uses it to model its budget.
- A business school uses it in its Decision Sciences courses to develop simulation analyses.
- A financial institution uses it to estimate municipal revenues and prepare offers on bonds.

Examples of applications for which DS Lab can be used are:

Investors
- Securities and options pricing and yield projections
- Investment choices
- Equities risk evaluation

Materials Analysts
- Material flow simulations
- Optimum shipping/delivery/processing scheduling

Bankers
- Business planning
- Complex loan amortization projections

Scientists & Engineers
- Simulating chemical reactions
- Fluid dynamics
- Mechanical dynamics
- Component stress modeling
- Statistical modelling

Financial & Business Analysts
- Business planning
- Actuarial Analysis
- Inventory planning

Social Scientists
- Quantitative behavioral models
- Econometric models

DS Lab makes modelling and modifying these complex simulations easier than using spreadsheets, and graphically displays the models to those who need to understand the results.

Some Basic Concepts

Before getting into the details of DS Lab, we will consider some concepts which are essential for first time users. DS Lab is theoretically based on *Dynamic Systems Analysis*. It is not necessary for a DS Lab user to be familiar with this theory in order to use the product, as the symbols and calculating methods used are simple enough for anyone to grasp.

In the following pages, nine basic concepts are introduced:

Models
Variables
Series
Tables
Constants
Inputs
Shadows
Links between models
DDE

Models

Models are simplified representations of real systems. The operations of a companys sales division can, for example, be represented by a model. A model can also be considered mathematically as a set of formulas describing the relationships between the *elements* involved. An example of an element is *Sales Forecast*, which in turn is a function of a series of other elements such as *Number of Agents*, *Sales Points*, and so on.

In DS Lab, models are defined in two ways:

- **Graphically:** by placing on a graphics page symbols representing the elements and linking them with arrows representing the causal relationships between the elements; and

- **Mathematically:** by associating with each element a formula or numeric value which defines it.

DS Lab models can contain five types of elements:

Variables
Series
Tables
Constants
Inputs

Shadows

A *shadow element* is a duplicate of some other element in the model. Any type of element, including another shadow, can be duplicated. There is a live link between the primary element and its shadow so that any changes (i.e. value, name, color) to either will be reflected in the other. The shadow element will have the same shape as the primary but with a double outline. For example, a shadow of a variable element will appear as two concentric circles.

Suppose the Sales and Marketing department builds two models representing the components of advertising costs and agent costs. When a model to represent total marketing costs is created on the same page, these two sub-models would become part of the main one. The *Advertising Costs* and *Agent Costs* elements can be duplicated and their shadow elements used as components of *Total Marketing Costs*.

Links Between Models

It is possible to make one model influence the input, output or execution of another. This can be done in three ways:

- An *incoming link* reads the value of a variable from another model during the simulation.
- An *outgoing link* sends data to another model during the simulation.
- *Executing a sub-model* causes one model to execute another.

These operations are carried out using the following functions in the script language: **Request**, **Poke** and **Execute**.

DDE

DDE (Dynamic Data Exchange) is the mechanism by which Windows applications can exchange data automatically (Copy and Paste operations are a manual method of obtaining the same result).

Three pieces of information are needed by DDE Links:

- *Application*: the name of the application with which to establish the link, for example DS Lab, Excel, etc.
- *Topic*: the name of the document or the applications work unit: DS Lab models, spreadsheet work sheets or word processor documents are Topics. A special kind of Topic is System, which represents the application itself.
- *Item*: identifies the desired data. In DS Lab it will be the name of an element (with or without step notation), in a spreadsheet it would be a cell, etc.

There are two types of DDE links:

- A *Hot Link* is a link that is always active. This means that changes in the data at the sending end will be reflected simultaneously at the receiving end. The Paste Link command found in many Windows applications is a common example of a hot link.
- A *Cold Link* is a permanent link that allows data to be sent or received by an application. The link is updated only on demand.

DS Lab supports both kinds of links when sending data, but when receiving, accepts only cold links. This choice was based on the typical use of DS Lab, which consists of a data acquisition phase followed by the *What-If* analysis. Since the analysis is based on factual data that remains unchanged until the next acquisition phase, there is no need for it to be automatically updated.

Description of the Work Environment

DS Lab presents a ***Work Environment*** in which a **Models**, **Texts** or **Reports** can be built.

The basic functions of this window are the same as in any Windows application. The name of the program is shown at the center of the title bar. The windows **Control Menu** box is in the upper left corner. This gives access to a menu from which you can **Restore, Move, Size, Minimize, Maximize** or **Close** the window and **Switch To** other active applications, leaving DS Lab running in the background. The usual **Maximize** and **Minimize** buttons (up and down arrows) are in the upper right-hand corner. If you have any queries about these functions, refer to your Microsoft Windows Users Guide for a fuller explanation.

The **Menus**, the **Toolbars**, the **Edit Bar** and the actual **Work Area** with the **Model**, **Report** and **Text Windows** are specific to DS Lab.

Menus
The Toolbars
The Edit Bar
Popup Menus
DS Lab Menus in Model Mode
DS Lab Menus in Text Mode
DS Lab Menus in Report Mode
Editing the Scripts of the Variables
Rules for Building DS Lab Models

Menus

Three different work environments exist in DS Lab:

- **Model mode**
- **Text mode**
- **Report mode.**

The menus are opened by moving the pointer to the desired item and clicking with the left mouse button, or by holding down the ALT key while pressing the underlined letter in the name of the required menu. The following sections describe in detail the function of each of the menu commands.

The Toolbars

DS Lab can have two toolbars: one just below the menu bar (*Horizontal Toolbar*), the other at the left edge of the screen (*Vertical Toolbar*). They contain tools representing the elements and the main commands of DS Lab in Model mode. The arrangement of the buttons on the toolbar can be customized. To move a button, position the pointer over it and, while holding down the right mouse button, drag the button to its new position.

Both toolbars are optional. The **Workspace...** command in the **Options** menu allows you to define your personal work environment. In the following sections, the various tools are described in their default positions.

The Horizontal Toolbar
The Vertical Toolbar

The Horizontal Toolbar

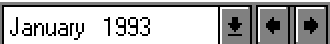
Starting from the left, the first four buttons correspond to the main commands of the standard Windows **File** menu:

New tool

Open tool

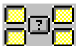
Save tool

Print tool.



To their right is the drop-down list box of the Simulation Steps, showing the **Current Step** of the model; clicking on the left and right arrows adjacent to it moves you to earlier or later steps.

Next comes a small colored circle called the **Traffic Light**; it can show red or green. Immediately after a simulation the Traffic Light is green, indicating that the values shown on the screen have been updated. If anything (such as editing a script or the values of a series) is done that could make the values on the screen invalid, the **Traffic Light** turns red and has an X through it as a reminder that a recalculation should be done to obtain updated data.



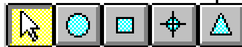
Moving to the right again there is the **Zones**

The four outer rectangles each represent a zone of the model. To associate a zone with one of these, select the elements to be included, then click with the right mouse button (or SHIFT + left button) on one of the four rectangles. From now on, clicking on that rectangle will move you to the zone of the model associated with it.

The central rectangle allows you to define zones and assign names to them. Thereafter, clicking on this rectangle will give access to a dialog box in which you can choose which pre-defined zone to move to.

The remaining buttons each correspond to a menu command that will be described more fully later. Here we give only a brief summary:

The **Selection Arrow** tool is used to select one or more elements, or to move them after they have been selected. It corresponds to the **Selection Arrow** command in the **Edit** menu.

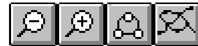


The **Variable**, **Constant**, **Series**, **Input** and **Table** tools represent the corresponding elements that can be used in DS Lab. These each correspond to a command of the same name in the **Model** menu.

The **Shadow** tool creates a copy of an element. It too corresponds to a command in the **Model** menu.

The **Connect** tool is used to connect the elements (**Connect** in the **Model** menu).

The **Comment** tool is used to place comments in the model (**Comment** in the **Model** menu).



Further to the right are four **Zoom** tools, used to give different views of the model. The first two correspond to **Zoom Out** and **Zoom In**; the third shows the model at the default enlargement (**Normal View** tool), and the fourth at maximum reduction, giving an overview of the whole model (**All Pages** tool).

The Vertical Toolbar

The **Export to Excel** tool allows integration with the Excel spreadsheet and corresponds to the **Export to Excel** command in the **Simulation** menu.

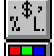
The **Report** tool is used to show the results of a DS Lab simulation in tabular form (**Report** in the **Simulation** menu).

The **Values for Current Step** tool shows the values of the variables and series at the current step (**View Values for Current Step** in the **View** menu).

The **Text** tool corresponds to the **Open Text** command in the **File** menu.

The **Recalculate** tool has the same function as the **Recalculate** command in the **Simulation** menu.

The **Calculate** tool corresponds to the **Calculate** command in the **Simulation** menu.



The **Number Format** and **Color** tools represent commands in the **Options** menu.

The **Find** tool corresponds to the **Find...** command in the **View** menu.

The **Simulation Parameters** tool corresponds to the **Parameters...** command in the **Simulation** menu.

The **DDE** tool corresponds to the **Update DDE Links...** command in the **Simulation** menu.

The **First Step** tool allows you to go immediately to the **First Step** of the simulation.

The **Set Print Size** tool corresponds to the **Set Print Size** command in the **Model** menu.

The **Print Size** tool corresponds to the **Print Size** command in the **View** menu.

The remaining four tools correspond to four commands in the **Model** menu:

Edit Constants..., **Edit Series...**, **Edit Starting Values...** and **Edit Past Steps...** .

The Edit Bar

The **Edit Bar** is located immediately below the Horizontal Toolbar. The name of the currently selected element appears on it. The name of the element can be modified simply by typing in the new name, which will immediately appear on the edit bar.

When only a part of the name needs to be changed, select the element and then move the mouse pointer on to the edit bar highlighting the part to be changed. You may also go into edit mode with the short-cut key F2. To make the name change permanent, press ENTER; otherwise the changes will be lost when you select another element.

The Edit Bar can be removed from the workspace by selecting **Workspace...** from the **Options** menu, then clicking the **Edit Bar** check box to remove the X.

Popup Menus

Pressing the right mouse button from within a DS Lab model window gives access to two reduced size, context sensitive menus containing the commands most commonly used in the particular situation from which they are accessed.

When one or more elements are selected, the menu is the following:

Cut	Shift+Del
Copy	Ctrl+Ins
Copy Link Total	Shift+F2
Copy Link Current	Shift+F3
Delete	Del
Export to Excel	F11
Report	F12
Number Format...	
Color...	

which correspond to the following commands in Model mode:

Cut command in the **Edit** menu

Copy command in the **Edit** menu

Copy Link Total command in the **Edit** menu

Copy Link Current command in the Edit menu

Delete command in the **Edit** menu

Export to Excel command in the **Simulation** menu

Report command in the **Simulation** menu

Number Format... command in the **Options** menu

Color... command in the **Options** menu

If no elements are currently selected, the menu is:

Paste	Shift+Ins
Parameters...	
Calculate	F9
Recalculate	F10
UpDate DDE Links...	
Workspace...	
Model Setup...	

which correspond to the following commands in Model mode:

Paste command in the **Edit** menu

Parameters... command in the **Simulation** menu

Calculate command in the **Simulation** menu

Recalculate command in the **Simulation** menu

Update DDE Links command in the **Simulation** menu

Workspace... command in the **Options** menu

Model Setup... command in the **Options** menu

By using the right mouse button and the Toolbars, it is rarely necessary to access the classic pull-down menus.

DS Lab Menus in Model Mode

When working with models, the menu bar offers the following choices:

The File menu
The Edit menu
The Options Menu
The View Menu
The Model Menu
The Simulation Menu
The Window menu
The Excel! menu
The Help menu

The File Menu

New...
Open...
Close
Close All
Save
Save As...
Open Text
Print
Print Setup...
Page Setup...
Last Four Models Used
Exit

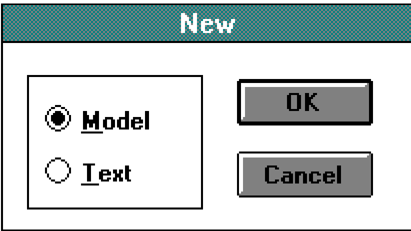
New...

This command allows you to create a new file. When **New...** is chosen, a dialog box appears in which you are asked whether you want to start a new model or a text file. The default is **Model**; when you click the **OK** button, DS Lab will present you with an empty window in which to build a new model. Several model and text files can be open at the same time: practically no operational limits exist apart from the structural ones of Windows.

By default, new model and text files assume the names MODEL and TEXT respectively, plus a serial number indicating how many new files were opened in the same work session. Model files have the extension .LAB identifying them as DS Lab files.

The **Text** option enables you to write texts, which can run to many pages, without having to leave DS Lab to run a separate text editor. Text files are in ASCII format, which can be read by any word processor or text editor, and are saved with the extension .TXT.

The **New** tool is a short cut for this command.

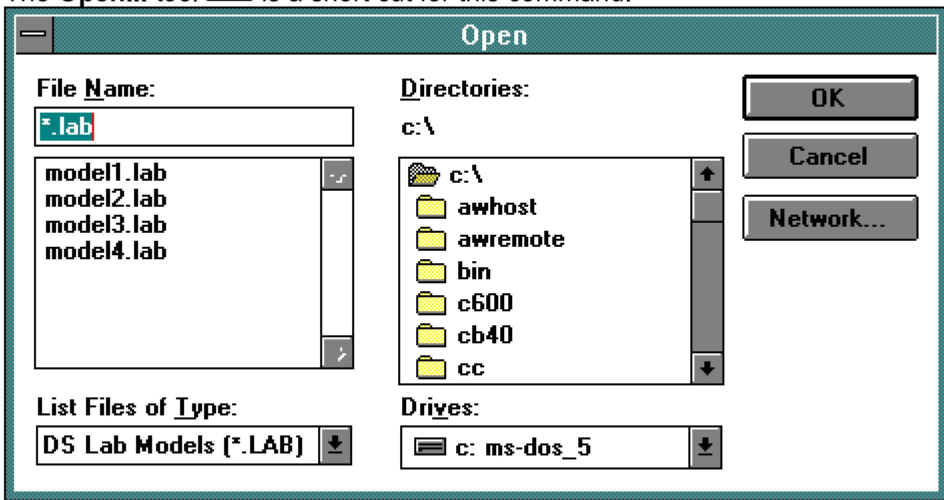


Open...

This command is used to open files previously saved to disk. The command works in much the same way as in any other Windows program. When you choose this command, a dialog box appears which allows you to look for the file you wish to open. The desired file can be loaded by double-clicking on it or by selecting it and then choosing the **OK** button.

To open a text file, change the extension in the **File Name** box to .TXT, either by replacing *.LAB with *.TXT or by selecting **Text Files** from the **List Files of Type** box. To exit this command without opening a file, choose the **Cancel** button.

The **Open...** tool is a short cut for this command.



Close

Clicking on this command closes the current model. If the model has been modified, DS Lab will notify you and ask if you wish to save the changes.

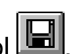
Close All

This command is self-explanatory: it closes all open windows, after asking you whether you want to save any models or texts that have been changed.

After all windows have been closed, the only menus available are **File**, which offers you the choice of starting a new model, opening an existing one or exiting from DS Lab to return to Windows, and **Help.**

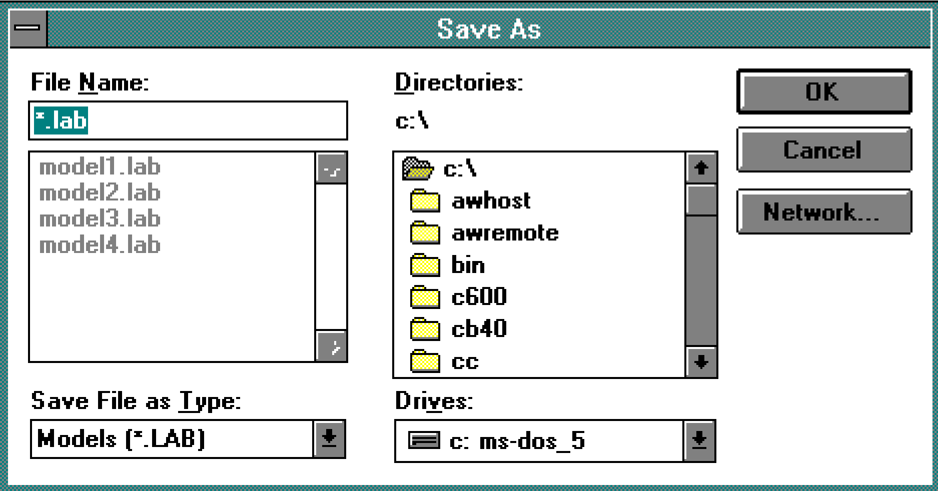
Save

Like the corresponding command in other Windows programs, this command allows you to save to disk a file that has been changed. The first time a model or text is saved, you are asked to give it a name.

There are two short cuts for this command: the SHIFT F12 key and the **Save** tool .

Save As...

This is similar to the preceding menu command, the only difference being that a dialog box appears in which you are asked for a new name. This is useful when, for example, you want to open and edit a model while keeping a copy of the file as it was before the changes. To do this, use the **Save As...** command to save the modified version under a new name.



Open Text

This command is used to open the Text File associated with the model. An ideal use of this feature is to write a description of a model as it is being built, which is associated with the model and is always available when working on it. The text is saved in a separate file with the same name as the currently active model but a .TXT extension.

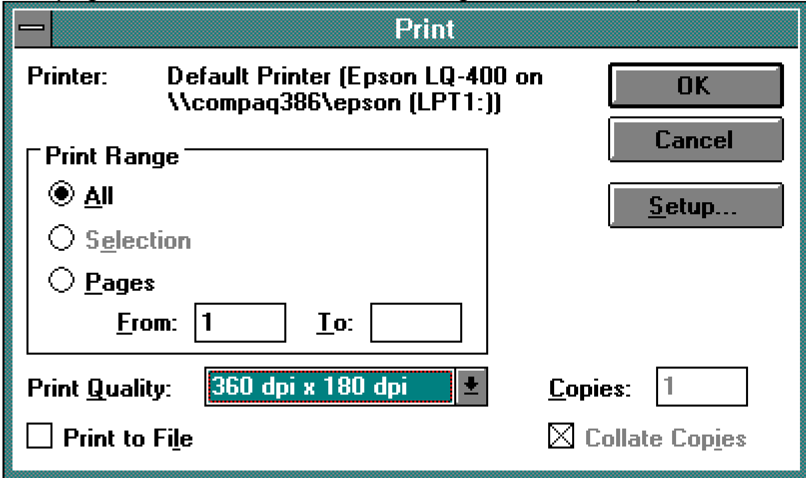
The **Text** tool T is a short cut for this command.

Print

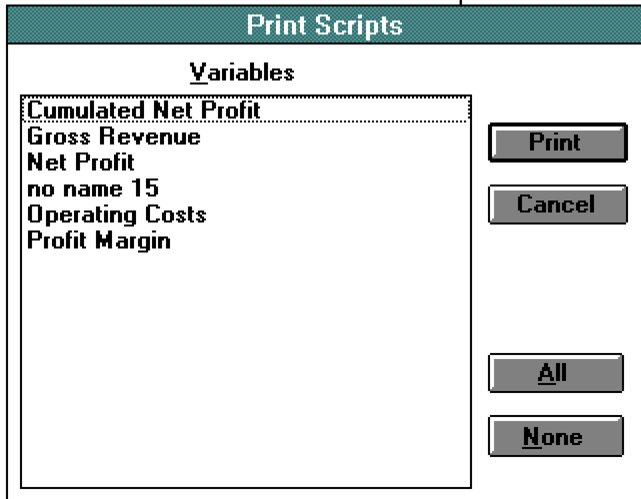
When you choose **Print**, a cascading menu appears with the following options:

- **Graphic**
- **Script**

The **Graphic** option is to print the model in graphic form. The dialog box which appears after selecting this option allows you to set the number of copies and the range of pages to print. The page breaks are indicated on the screen by a *pagination grid* and can be previewed with the **View All Pages** command. The pages are numbered from left to right and from top to bottom.



The **Script** print option calls up a dialog box containing a list of all the variables in the model, allowing you to choose the scripts to be printed. After completing the input fields, choose the **OK** button to start printing or the **Cancel** button to abandon the operation.



The **Print** tool is a short cut for this command.

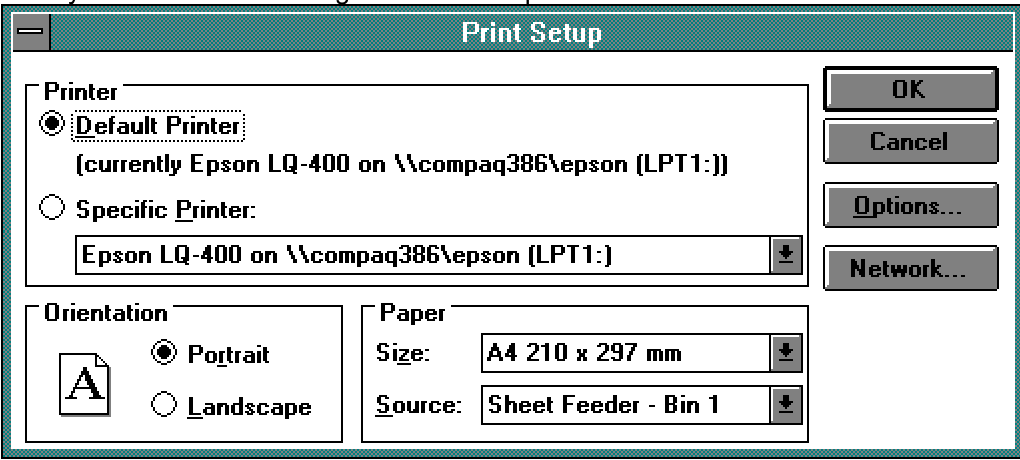
Print Setup...

Choosing this command from the **File** menu allows you to define standard parameters for the printer to be used. In particular, you can define:

- the printer connected to the system;
- the type of paper feed;
- the paper size

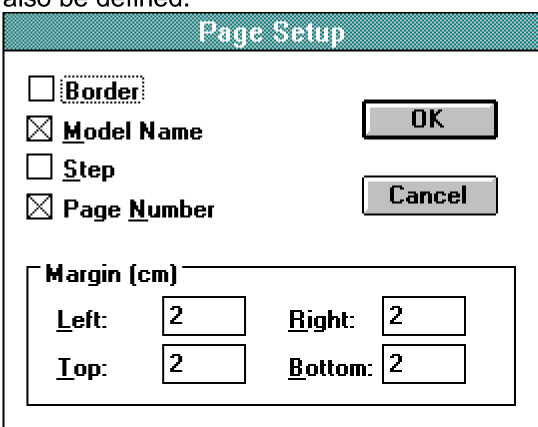
and, when using a laser or similar printer:

- the amount of printer memory;
- the print orientation: portrait (vertical) or landscape (horizontal);
- the graphics resolution: (75, 150 or 300 dots per inch);
- any additional font cartridges fitted to the printer.



Page Setup...

This command allows you to define the items which will appear with the printed model. By clicking on the appropriate check box, you can decide whether to include a border, the name of the model, the step in the simulation and/or the page number in the printed document. (**Note:** an option is enabled when an X appears in the check box, disabled when it is empty.) The left, right, top and bottom page margins can also be defined.



Last Four Models Used

After the **Page Setup...** command on the menu and before the **Exit** command, the names of the last four models used in DS Lab are shown. They can be opened directly by clicking on a name, bypassing the **Open...** command.

Exit

This command exits from DS Lab. If any open model or text files have been modified, you will be asked if they should be saved before exiting.

The Edit Menu

The commands of the Edit menu are discussed briefly below. Since the commands are common to most Windows programs, refer to your Microsoft Windows Users Guide for more detailed explanations.

Cut
Copy
Copy Link Total
Copy Link Current
Paste
Delete
Selection Arrow

Cut

This command allows the user to cut the selected element(s) or text object(s) from the current model and transfer it (them) automatically to the Windows clipboard. (For further information on the Clipboard, refer to your Windows Users Guide.) It will be helpful to keep in mind the following:

- In DS Lab, several models can be open at the same time, so that the **Cut** and **Paste** commands can be used to transfer sections of models from one to another.
- The Clipboard can also be used to transfer data to a Windows spreadsheet or word processor.
- More generally, anything that can be seen on the screen can be transferred via the Clipboard to other Windows applications.

When working in Model mode, selecting one or more elements and then choosing this command causes DS Lab to transfer to the Clipboard the names of the selected elements and their values for each step of the simulation.

Short Cut: This command may be used without accessing the menu simply by pressing SHIFT+DEL. It may also be accessed from the popup menu which appears on clicking the right mouse button with one or more elements selected.

Copy

This is similar to **Cut**, except that with **Copy**, the elements or text transferred to the Clipboard are also left on the page from which they have been copied. **Cut** transfers them to the clipboard, deleting them from the model, whereas **Copy** transfers them to the Clipboard but leaves the model intact.

When working in Model mode, selecting one or more elements and then choosing this command causes DS Lab to transfer to the Clipboard the names of the selected elements and their values for each step of the simulation.

Short Cut: This command may be used without accessing the menu simply by pressing CTRL+INS. It may also be accessed from the popup menu which appears on clicking the right mouse button with one or more elements selected.

Copy Link Total

The **Copy Link Total** command allows data to be exported in such a way that it will be updated automatically. **Copy Link Total** sets up a link for each step of the model: that is, if the model covers 24 steps, there will be 24 linked values for every element for which **Copy Link Total** has been invoked.

The procedure is identical to that of the **Cut** and **Copy** commands:

1. Select the elements to be linked to the other application.
2. From the **Edit** menu, choose **Copy Link Total**.
3. Bring up the destination application.
4. Select the insertion point for the data.
5. From the **Edit** menu, choose **Paste**.

Since each Windows application uses its own link format, you must choose the format in which the link is to be written before invoking this command. This is done with the **Copy Link Format** command in the **Options** menu. The default format is that used by Excel.

In practice, these commands are useful if, for example, you want to create report documents with another Windows application and have them automatically updated every time the data in the original model changes.

Short Cut: This command may be used without accessing the menu by pressing SHIFT+F2. It may also be accessed from the popup menu which appears on clicking the right mouse button with one or more elements selected.

Copy Link Current

This command is similar to **Copy Link Total**. The only difference is in the number of links set up; **Copy Link Current** sets up only one link per element, and that link is for the value at the current step; **Copy Link Total** creates a link at each step for each element.

To clarify the difference between **Copy Link Current** and **Copy Link Total**, let us consider an example. Suppose you want to transfer data for two elements from a model lasting 12 steps so that it will be updated automatically. If the current step in the model is the 12th, **Copy Link Current** creates a link for only two values (those of the two elements at the last step), while **Copy Link Total** creates a link for 24 values (2 elements * 12 simulation steps).

Short Cut: This command may be used without accessing the menu by pressing SHIFT+F3. It may also be accessed from the popup menu which appears on clicking the right mouse button with one or more elements selected.

Paste

This command transfers the contents of the Clipboard to the active window. It is used after something has been cut or copied onto the clipboard. To copy one or more elements from one model to another:

1. Select the elements to copy.
2. From the **Edit** menu, choose **Copy** (or press CTRL+INS).
3. Select the destination model (if it is not already open, open it now).
4. From the **Edit** menu, choose **Paste**.

If a part of the model is copied and pasted to the same model, the new elements take the same names as the originals with a 2 added at the end.

Short Cut: This command may be used without accessing the menu simply by pressing SHIFT+INS. It may also be accessed from the popup menu which appears on clicking the right mouse button when no elements are selected.

Delete

This command deletes a selection. To delete something, first select it and then choose this command. For safety, DS Lab asks for confirmation every time **Delete** is used.

Short Cut: This command may be used without accessing the menu simply by pressing DEL. It may also be accessed from the popup menu which appears on clicking the right mouse button with one or more elements selected.

Selection Arrow

Choosing this command is equivalent to clicking on the Selection Arrow tool .

If the **Selection Arrow** tool is already active, a check mark appears next to the **Selection Arrow** command in the **Edit** menu.

With the **Selection Arrow** tool active, it is possible to select:

- a single element by moving the pointer to it and clicking the left mouse button;
- all of the elements in a rectangular area of the model by placing the pointer at one corner of the desired area, pressing the left mouse button and dragging to the opposite corner before releasing the mouse button. All the elements in the selected area will be shown with a broken outline, showing they are currently selected;
- several elements even if they are not adjacent, by holding down the SHIFT key while clicking on them one at a time.

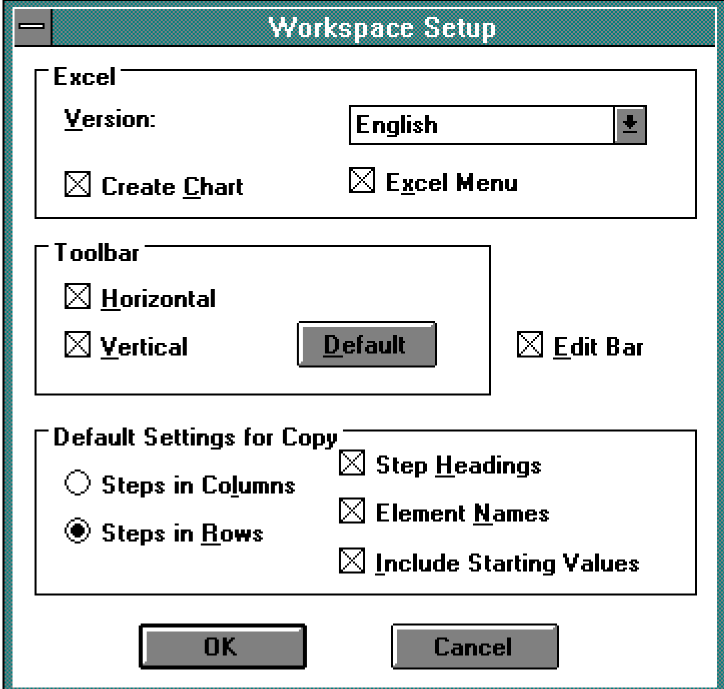
The Options Menu

The first part of this menu allows you to customize some of DS Labs tools.

Workspace...
Model Setup...
Copy Link Format...
Number Format...
Color...
Password...

Workspace...

This command may also be accessed from the popup menu which appears on clicking the right mouse button with no elements selected. It opens a dialog box where you can choose the components of your personal work environment. Each of these is active when the corresponding check box is selected (an X appears in it).



Excel

This part of the **Workspace...** dialog box is used to tell DS Lab which version of Microsoft Excel is installed on the system. This information is essential to the correct transfer of data from DS Lab to Excel.

- The **Create Chart** option, when enabled, causes Excel to automatically create a chart each time data is exported to the worksheet.

- An X in the **Excel Menu** check box will add the **Excel!** menu to the menu bar. This single item menu, when chosen, activates the Excel worksheet, launching the application if it is not already running.

Toolbar

The **Toolbar** section of this dialog box allows you to choose whether to display the **Horizontal** and/or **Vertical** toolbars. Choosing the **Default** button will cause all the tools to return to their default positions.

Edit Bar

The **Edit Bar** check box controls whether or not the Edit Bar is displayed.

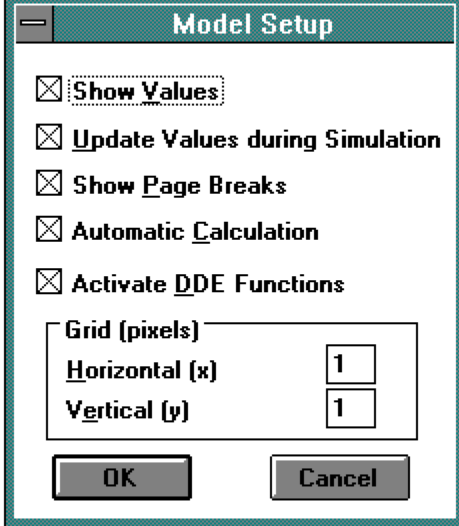
Default Settings for Copy

The options in this section allow you to customize the form in which data will be exported to another

application, usually a spreadsheet, by means of the **Copy** and **Paste** procedures.

Model Setup...

This command may also be accessed from the popup menu which appears on clicking the right mouse button with no elements selected. It opens a dialog box where you can choose the components that will characterize your model window. Each of these is active when the corresponding check box is selected (when an X appears in it).



Show Values

If this option is selected, the values of the elements for the current step or the error values returned by the program during calculation are shown below the names of the elements.

Update Values during Simulation

When enabled, this option updates the value of each element on screen at each step of the simulation. Disabling this option will display the calculated values only at the final step. With large, complex models, disabling this option will speed up the simulation, since the screen is no longer redrawn at each step.

Show Page Breaks

When this option is selected, a pagination grid is superimposed on the model window, indicating where the page breaks will occur when the model is printed. This option is useful when fixing the zoom level for the printed document before printing.

Automatic Calculation

Like a spreadsheet, DS Lab offers the option of recalculating the values of the formulas every time data is changed. With very large models, it may be better to disable automatic calculation in order to speed up maintenance operations on the model.

Activate DDE Functions

This option is the equivalent of the preceding one for DDE links with other applications. If it is selected, the links will be updated every time the models calculations require it (including each time a variable script

is modified). It has the drawback that, if the other application is not active, error messages will appear. It is, therefore, advisable to leave this option disabled during model building and maintenance, remembering to enable it before carrying out any simulation in which DDE functions are needed to obtain reliable results.

Grid

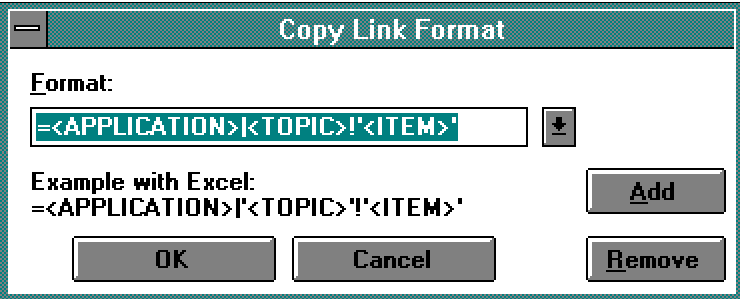
This option, similar to that found in most drawing programs, allows you to define two parameters, **Horizontal (x)** and **Vertical (y)**, which determine the size of the smallest distance in pixels by which elements can be moved on the two axes. The grid thus obtained is not shown on the screen.

For example, if the two values are set to 50 and 25 respectively, the elements in the model will be spaced at intervals of 50 pixels (or multiples of 50) horizontally and 25 pixels (or multiples of 25) vertically.

This option is useful when you wish to improve the appearance of models by aligning elements in horizontal or vertical rows.

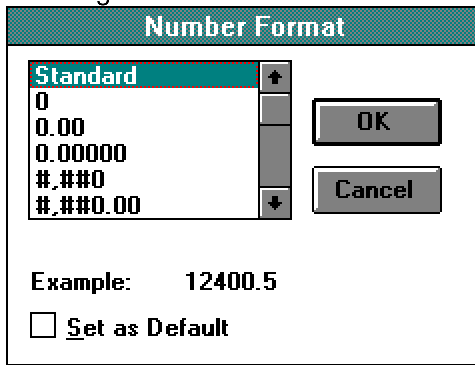
Copy Link Format...

This item is used to select the format of the data to be sent by the **Copy Link Total** and **Copy Link Current** commands. Select the appropriate format from the **Format** drop-down list box, or type in a new format and then choose the **Add** button. The default format is that used by Excel.



Number Format...

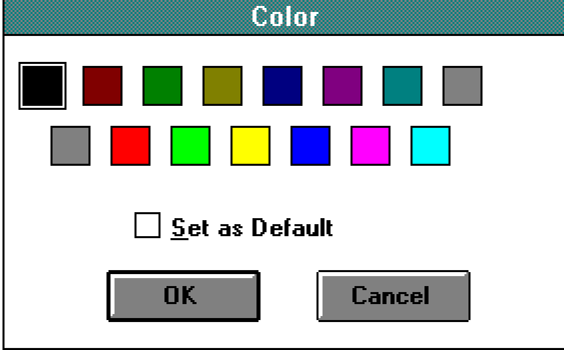
This command allows you to define the format in which the numerical values of the elements will be displayed. Select the desired element and then click on this command to access the dialog box in which you can choose the most appropriate format. A standard format for all new elements can be set by selecting the **Set as Default** check box.



The short cut for this command is the **Number Format** tool . It may also be accessed from the popup menu which appears on clicking the right mouse button with one or more elements selected.

Color...

This command allows you to choose the color in which the names and values of the selected elements will be displayed. Select the element to be colored, then choose this command to access the dialog box of available colors. A standard color for new elements can be set by selecting the **Set as Default** check box.



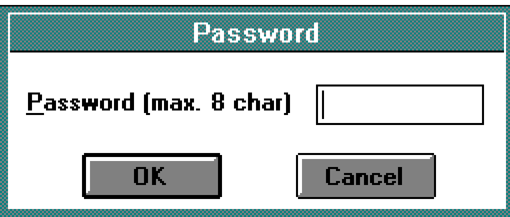
The short cut for this command is the **Color** tool . It may also be accessed from the popup menu which appears on clicking the right mouse button with one or more elements selected.

Password...

This command is used to assign a password (no longer than 8 characters) to a model. Every time an attempt is made to open the model, DS Lab will ask for the password, thus restricting access to the model and its data by unauthorized persons.

After the password has been entered, you will be asked to type it again. If the two versions correspond, the password is attached to the model.

Note: The password can be modified only after the model has been opened. If you subsequently forget it, the model will be irrevocably lost!



The View Menu

The **View** menu is made up of three distinct parts. The first duplicates the four buttons used to give different views of the model, described in ***The Horizontal Toolbar***. The second contains a command to redraw the model, and the third enables you to move around the model and to find elements with particular characteristics.

All Pages
Normal
Zoom In
Zoom Out
Print Size
Redraw
Find...
Next
Zones...
Undefined Variables
Run-Time Errors
View Values for Current Step
Search...

All Pages

This command allows an overview of the whole model. The short cut for this command is the **All Pages** tool .

This viewing mode is useful for a preview before printing.

Normal

This command activates the default level of magnification of the model. The short cut for this command is the **Normal View** tool .

Zoom In

This command gives an enlarged the view of the model. It can be used after **Zoom Out** to return step by step to the original viewing mode. The short cut for this command is the **Zoom In** tool .

Zoom Out

This command is the opposite of **Zoom In:** it gives a reduced view of the model. Choosing it repeatedly gives an increasingly reduced-scale view. The short cut for this command is the **Zoom Out** tool .

It should be noted that what is shown on screen at the lowest levels of magnification may not correspond exactly with what will be printed: names of variables or comments which appear superimposed on screen may not be so when printed, depending on the print size chosen and the printer used.

Print Size

This command returns you instantly to the zoom level set for printing out the model by means of the **Set Print Size** command in the **Model** menu. It provides a form of print preview, allowing you to see at a glance what elements will be printed together on the same page.

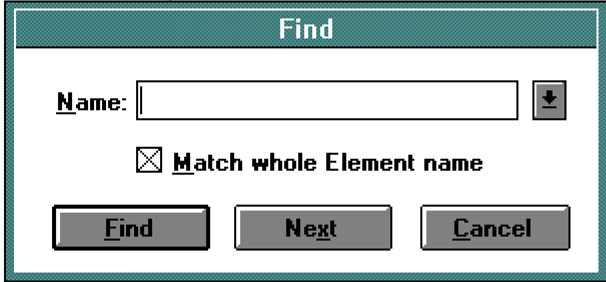
The short cut for this command is the **Print Size** tool .

Redraw

This command redraws the model on the screen. It can be used when, for any reason, the model is not displayed correctly. Though the time required to redraw depends on the complexity of the model, it should only take a few seconds even for a complex model.

Find...

This command is used to search for an element in a large model. A dialog box appears in which you are asked to type in the name of the element to be found or select it from a list of all the elements in the model. You can also choose whether to **Match Whole Element Name**, that is, the name entered corresponds exactly to that of the element being searched for, or if it represents only a part of it. The **Next** button allows you to search for the next occurrence of the element.



There are two short cuts for the **Find...** command: the F3 key and the **Find** tool

.

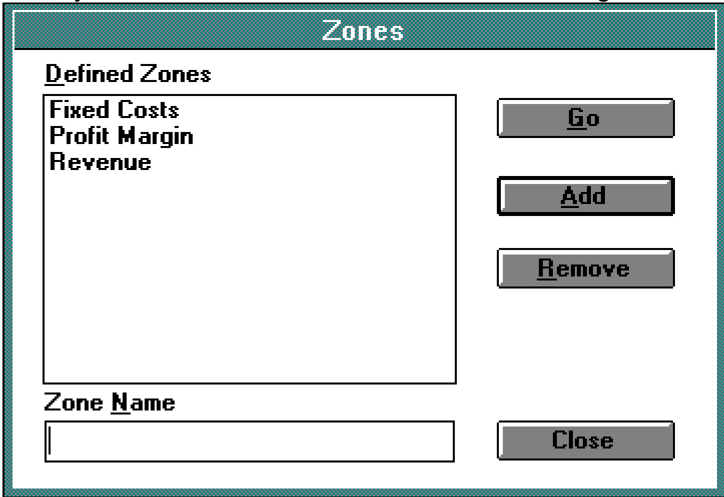
If a shadow element is selected, pressing F3 and then **Return** causes DS Lab to look for the primary element where the shadow is defined.

Next

This command duplicates the function of the **Next** button in the dialog box of the **Find...** command. The short cut to find the next occurrence of the element selected is the F4 key.

Zones...

This menu item gives access to the dialog box listing user-defined areas of the model. In simple terms, it is possible to define a name for a certain portion or zone of the model. There is no limit to the number of zones that may be defined. Once the zone has been defined, using this command will enable you to go directly to a desired area of the model without using the scroll bars.



This dialog box can also be accessed by clicking on the central rectangle of the **Zones** tool, or by pressing the F5 key.

Undefined Variables

This command is used to locate all variables that have an incorrectly defined or undefined script. Its short cut key is F6.

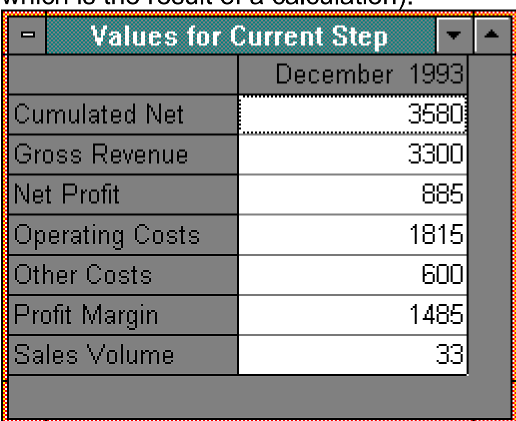
Run-Time Errors

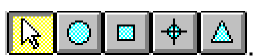
This command is used to find all variables that have given run-time errors, for example, division by zero or the logarithm of a negative number. Its short cut key is F7.

View Values for Current Step

This command displays the values of the models variables and series for the current step (shown in the drop-down list box on the toolbar). The values are shown in either a two-column or a two-row table, depending on the selection made under **Default settings for Copy** in the **Workspace Setup** dialog box in the **Options** menu. The display contains the names of the elements and their current values.

The values can be copied (and thus exported to other Windows applications), but not edited, because this could destroy the validity of the values (for example by arbitrarily changing the current value of a variable which is the result of a calculation).



The short cut for this command is the **Values for Current Step** tool .

Search...

This command is used to find, one after another, all the elements of the same type. On choosing this command, a cascading menu appears from which you choose the type of element to search for.

The Model Menu

This is undoubtedly the most important menu of DS Lab. It consists of three sections.

- The first part includes the five types of elements that can be used in DS Lab models (also represented on the Toolbar).

- The second part allows the values of the elements in the model to be changed without selecting them one by one. This point will be elaborated upon later.

- The third part presents two commands which allow you to convert an element to a different type and to define the size of the printed document in relation to that shown on the screen.

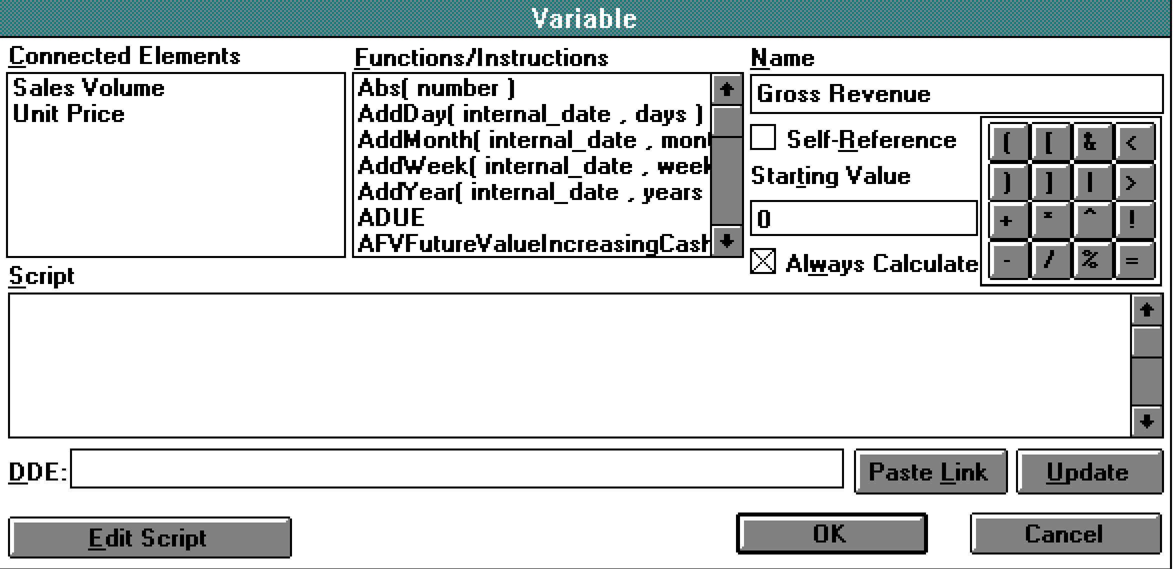
Variable
Constant
Series
Input
Table
Comment
Connect
Shadow
Edit Variables
Edit Constants
Edit Series
Edit Inputs
Edit Tables
Edit Starting Values
Edit Past Steps
Convert
Set Print Size

Variable

This command is used to insert a variable in the model and is equivalent to the **Variable** tool . A *variable* is an element whose value is defined by calculations within a DS Lab model. A *script* a kind of program written in the DS Lab language   is associated with each variable.

- **To insert a Variable in the model:**
1. From the **Model** menu, choose **Variable** (or click the**Variable** tool, or press ALT+F6).
2. Position the mouse pointer where the variable is to be inserted.
3. Click the mouse button.

Double-clicking on the variable accesses a *dialog box* where the variable is defined by means of its *script*.



This box shows:

- the name of the element;
- the elements connected to it which are used in its defining formula (**Connected Elements**);
- a list of the **Functions** and **Instructions** that can be used in the script;
- some of the operators that can be used (+, -, * , =, etc.);
- a check box for **Self-Reference**;
- a box where a **Starting Value** can be inserted if required;
- the **Script** edit box;
- an **Always Calculate** check box;
- an edit box and two buttons (**Paste Link** and **Update**) for **DDE**;
- the **OK, Cancel** and **Edit Script** buttons.

Self-Reference, and consequently **Starting Value**, will be briefly discussed here, whereas the others are discussed later in the manual.

Self-Reference enables the variable to be used as an input to itself. (Obviously, only values from previous steps can be used). This is useful in defining variables which express a running total, where the current value is equal to that at the previous step, plus the increase or decrease that has taken place: for example, a *Cash Balance*, whose in and out flows are *Income* and *Expenditure*.

Although by default the step immediately prior to the current one is taken as the input value, a different

interval can be specified by using *step notation*.

The **Starting Value** is the value given to the variable for step 0 and is required to calculate the first step when **Self-Reference** is enabled.

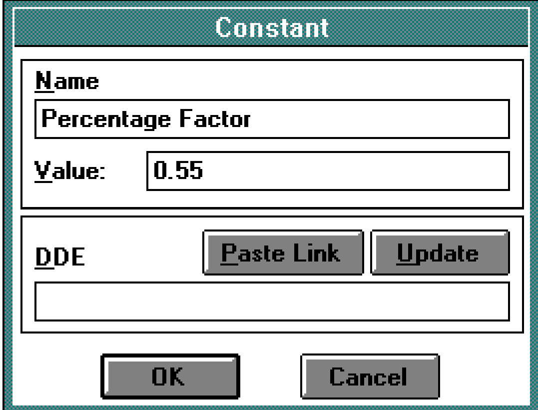
Constant

This command is used to insert a constant in the model and is equivalent to the **Constant** tool . A *Constant* is an element with a single value that remains unchanged throughout the simulation.

- **To insert a Constant in the model:**

1. From the **Model** menu, choose **Constant** (or click the **Constant** tool, or press ALT+F7).
2. Move the mouse pointer to where the element is to be inserted.
3. Click the mouse button.

Double-clicking on the element accesses a *dialog-box* in which you can specify the constants name and either its value or a DDE link which will import a value from another model or application.

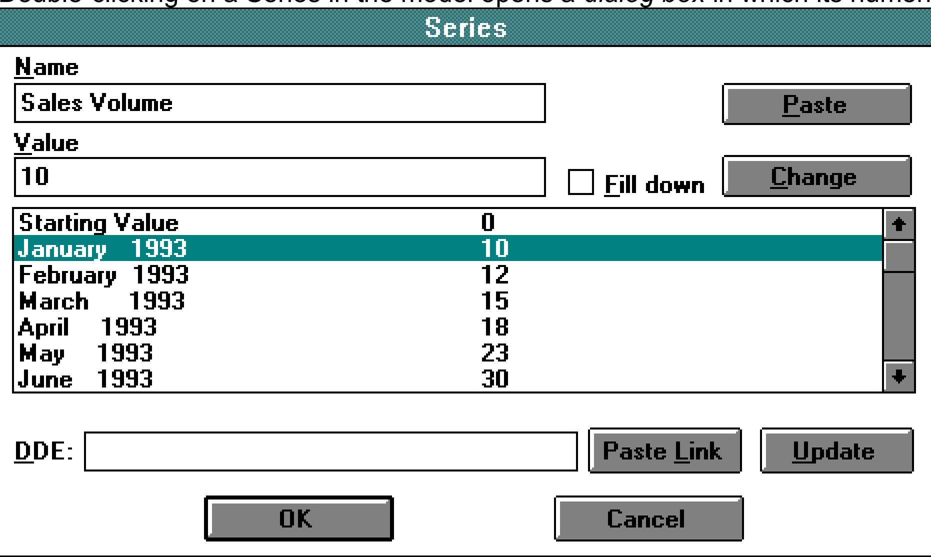


Series

This command is used to insert a series in the model and is equivalent to the **Series** tool . A series is an element that represents a *series of values*, one value for each step of the simulation. It can use either data from outside the model (e.g. the firms financial records) using DDE links or data listed in the model.

- **To insert a Series in the model:**
1. From the **Model** menu, choose **Series** (or click the **Series** tool, or press ALT+F8).
2. Position the mouse pointer where the element is to be inserted.
3. Click the mouse button.

Double-clicking on a Series in the model opens a *dialog box* in which its numerical values are defined.



The dialog box contains:

- an edit box containing the name of the series, where it may be changed if desired;
- an edit box headed **Value**, where the value for the step highlighted in the list box below is edited;
- a list box showing the steps of the simulation and the value assumed by the series for each of them;
- a **Paste** button, which inserts the data in the Clipboard as the value for the step highlighted in the list box (equivalent to SHIFT+INS);
- a **Change** button, which assigns the value currently in the **Value** edit box to the step highlighted in the list box (the same effect is produced by pressing the and keys);
- a **Fill Down** check box. If this is enabled before clicking the **Change** button, the value in the **Value** edit box is assigned to all the remaining steps to the end of the simulation;
- an edit box and two buttons (**Paste Link** and **Update**) for incoming DDE links;
- the usual **OK** and **Cancel** buttons.

Input

This command is used to insert an input in the model and is equivalent to the **Input** tool .

An input is an element whose value remains constant throughout a simulation, but will normally be changed between one simulation and another.

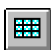
- **To insert an Input in the model:**

1. From the **Model** menu, choose **Input** (or click the **Input** tool, or press ALT+F9).
2. Position the mouse pointer where the input is to be inserted.
3. Click the mouse button.

Double-clicking on the element accesses a *dialog box* in which you can specify the inputs name and either an initial value or a DDE link which will import a value from another model or application.

Its value in subsequent simulations can be changed through the dialog box that appears every time the **Recalculate** command is given. This element makes it easy to define parameters in the model whose values are to be changed at each recalculation.

Table

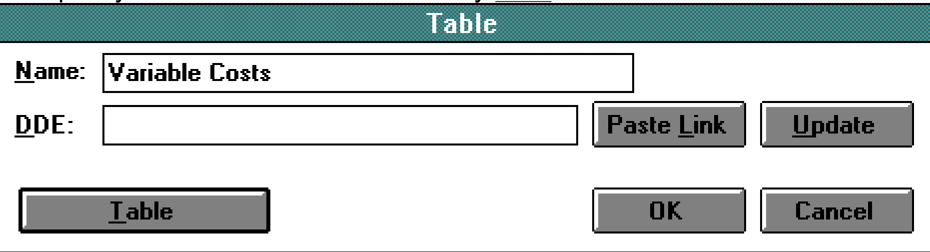
This command is used to insert a table in the model and is equivalent to the **Table** tool .

It is used to insert a table in the model. Tables allow you to access a number of data arrays, organized in a matrix, by means of a single element.

- **To insert a Table in the model:**

1. From the **Model** menu, choose **Table** (or click the **Table** tool, or press ALT+F10).
2. Position the mouse pointer where the element is to be inserted.
3. Click the mouse button.

To access the table and enter data in it, double-click on the element. This opens a *dialog box* in which you can specify the name of the table and any DDE links.



Click on the **Table** button to enter spreadsheet mode (cells, edit bar, etc.).

Variable Costs

Edit

PERIODS

	C1	C2	C3	
R1	PERIODS	Share A (dep.)	Index (indep.)	
R2	1	311.850	37.125	
R3	2	312.600	37.000	
R4	3	309.140	35.500	
R5	4	307.570	35.875	
R6	5	310.490	36.750	
R7				
R8				

To refer to the data contained in a table in the script of a variable, enter the name of the table followed by *RC notation* for the selected cells.

Comment

This command is used to insert a comment in the model and is equivalent to the **Comment** tool . It is used to add notes to the model.

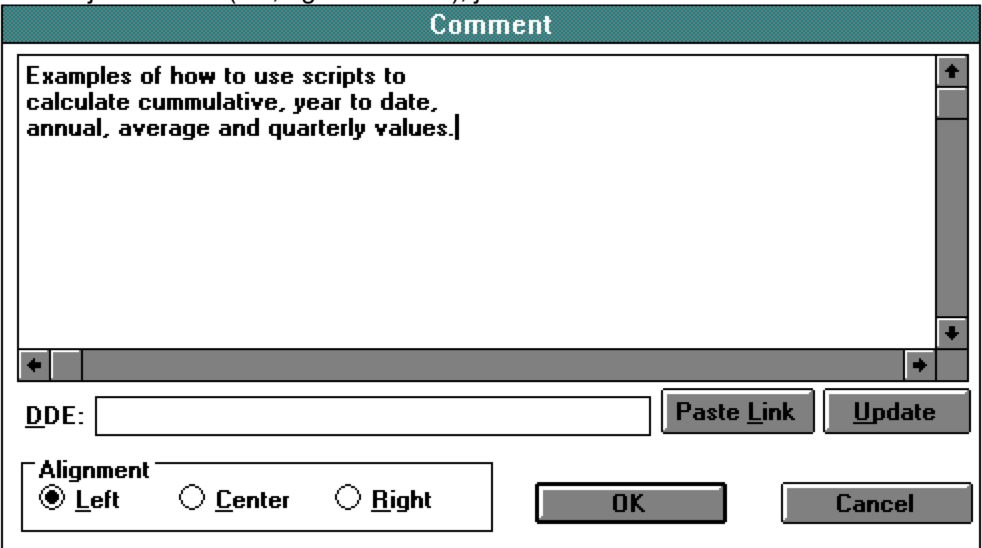
- **To insert a Comment in the model:**

1. From the **Model** menu, choose **Comment** (or click the Comment tool, or press ALT+F11).
2. Position the mouse pointer where the comment is to be inserted.
3. Click the mouse button.

A comment, unlike a **Text** describing the model, is shown on screen (and printed) as part of the model. It enables the user to add titles, notes, etc. to the model.

The text created using this tool is an object. When selected, the text frame is a dotted rectangle. Since the text is an object, it can be moved and placed anywhere in the model by dragging it with the mouse.

Double-clicking on a text frame opens a *dialog box* where you can edit the text, establish DDE links and set the justification (left, right or center), just as in most text editors.



Connect

This command is equivalent to the**Connect** tool. Its function is to connect two elements.

- **To make a connection between two elements:**

1. From the **Model** menu, choose **Connect** (or click the **Connect** tool, or press ALT+F12).
2. Position the mouse pointer to the first element (the start point).
3. Press the left mouse button and keep it pressed while dragging the pointer to the second element (destination).
4. Release the mouse button.

Only a variable may be the destination of a connection. Any other element (Series, Constant, Input or Table) by definition has a value that does not depend on other elements in the model, while Shadows represent elements defined elsewhere in the model.

Shadow

This command is used to insert a shadow in the model and is equivalent to the **Shadow** tool .

A shadow element is not an element in its own right, but is a copy of an element (variable, constant, series, table, input, comment or another shadow) already defined in the model.

- **To insert a Shadow in the model:**

1. From the **Model** menu, choose **Shadow** (or click the **Shadow** tool, or press ALT+F5).
2. Point to the element you want to duplicate.
3. Click the mouse button.
4. Position the mouse pointer where you want the Shadow.
5. Click the mouse button again.

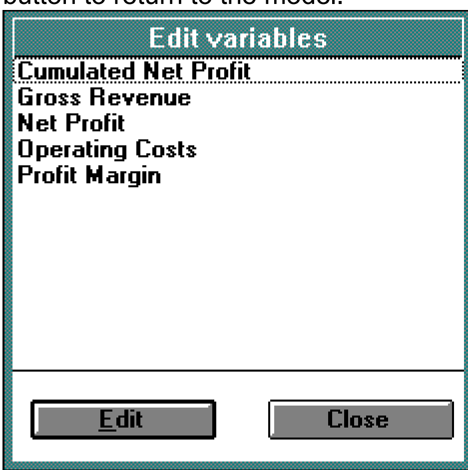
DS Lab does not distinguish a Shadow from its primary element. Any changes in the primary are reflected in the Shadow and vice versa. The model can contain more than one Shadow of the same element and, since you can create a Shadow of another Shadow, it is not necessary to copy the original every time.

Note: Keeping the SHIFT key pressed during the above process will cause the primary element to be exchanged with its Shadow. This means the primary will be moved to the new position and the Shadow will remain where the primary was. This is useful when building a model *top-down*. After building the basic model, you can subsequently decide to analyze the composition of a particular element in a sub-model.

Edit Variables

This command gives access to a list box for all the variables in a model without having to select them one by one on the screen. This is particularly useful in a large model when it becomes necessary to edit the script of a number of variables.

When this command is chosen, a dialog box appears containing a list of all the variables in the model. Once you have selected a variable, click on the **Edit** button to open a dialog box in which the variables script and other characteristics can be edited. After making the necessary modifications, choose the **OK** button. You are then returned to the list box where you can select another variable or choose the Close button to return to the model.

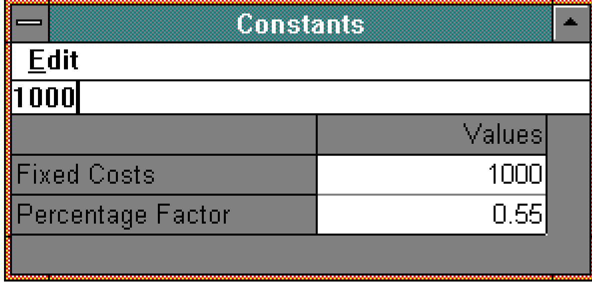


Edit Constants

This command facilitates the insertion of values for the models constants by making it possible to review all the constants in one table. They are presented in two columns; the first, containing the names of the constants, and the second, the value assigned to each.

Of course, the value of each constant can be edited individually by double-clicking on it to open its dialog box; however, the **Edit Constants** command provides a faster means to modifying several constant elements.

The values for the Current Step are updated on exiting from this dialog box, unles the **Automatic Calculation** option in the **Model Setup** dialog box is disabled.



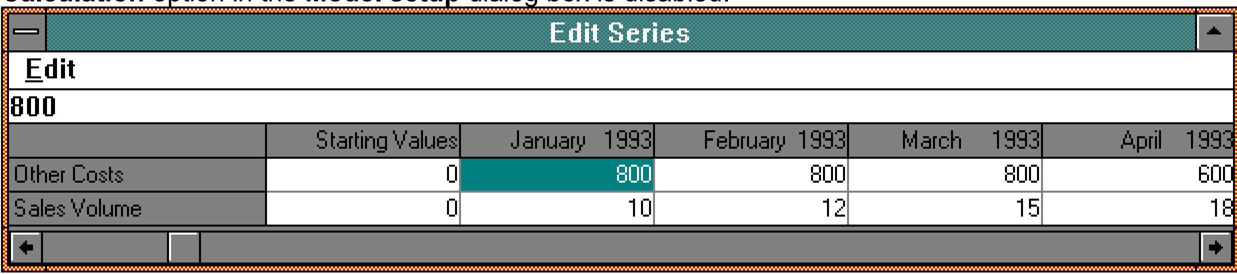
The short cut for this command is the Edit Constants tool .

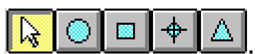
Edit Series

This command facilitates the insertion of values for the *Series* elements contained in the model by making it possible to access all such elements without selecting each one individually.They are presented in tabular form, with the names of the series in the rows and the simulation steps in the columns. At the row-column intersection, the value for the corresponding step may be inserted or modified.

It is also possible to modify any part of a series in its own dialog box. However, this method allows you to modify only one series at a time, so it is more time consuming when editing many elements, and it does not allow you to compare all the series in the model.

The values for the Current Step are updated on exiting from this dialog box, unles the **Automatic Calculation** option in the **Model setup** dialog box is disabled.



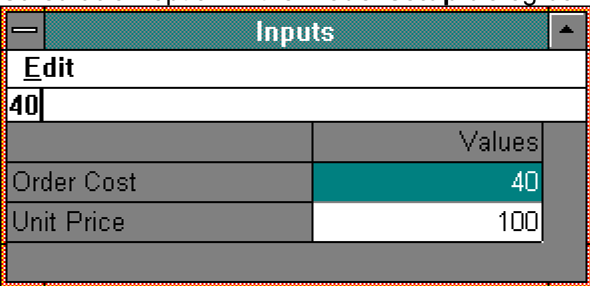
The short cut for this command is the **Edit Series** tool .

Edit Inputs

This command permits you to see or modify the inputs contained in the model.

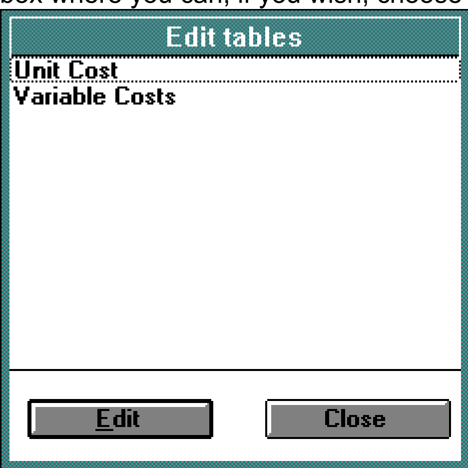
As with other similar commands, clicking on it gives access to a table containing two columns, one with the names of the inputs, the other with the corresponding values. This allows you to change the values of the inputs without using the **Recalculate** command.

The values for the Current Step are updated on exiting from this dialog box, unles the **Automatic Calculation** option in the **Model setup** dialog box is disabled.



Edit Tables

With this command you can edit all the table elements in the model without locating them in the model. This capability is particularly useful in a large model when it is necessary to change the table elements defined in it. Choosing this command opens a list box containing all the table elements in the model. Select the table element to be modified by clicking on it, then click the **Edit** button (or press ENTER). Make the required changes in the dialog box presented, then click the **OK** button. You will be returned to the list box where you can, if you wish, choose another table element to edit.

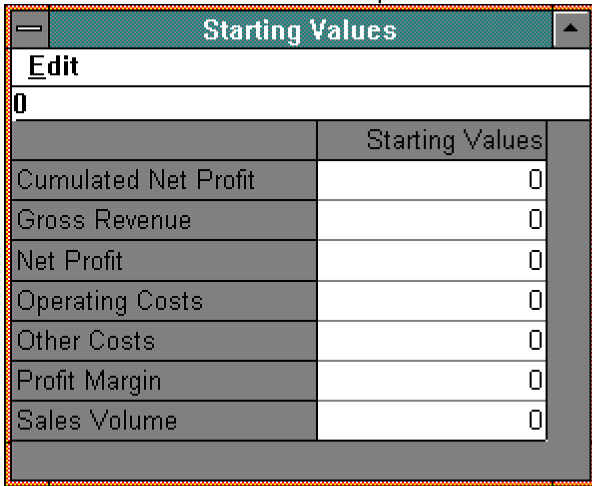


Edit Starting Values

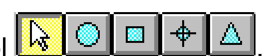
This command allows you to input starting values (those for step 0) for the variables and series in the model.

The values for the Current Step are updated on exiting from this dialog box, unles the **Automatic Calculation** option in the **Model setup** dialog box is disabled.

The values are entered in a simple table:



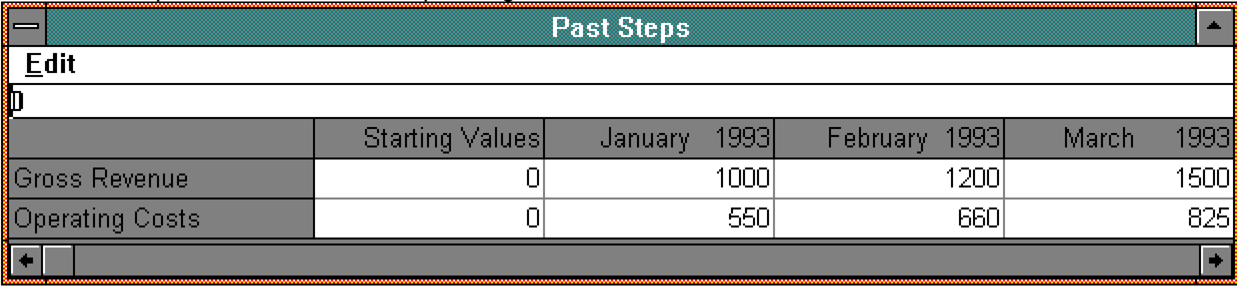
This command, which is most commonly used to enter starting values manually for the models variables, can also be used in other ways, such as importing values from other models or other Windows applications by means of the **Copy** and **Paste** commands.

The short cut for this command is the **Edit Starting Values** tool .

Edit Past Steps

This command allows you to input values for steps which are not simulated but have values that are already known. They are entered by means of a simple table.

The values for the Current Step are updated on exiting from this dialog box, unles the Automatic Calculation option in the Model setup dialog box is disabled.



	Starting Values	January 1993	February 1993	March 1993
Gross Revenue	0	1000	1200	1500
Operating Costs	0	550	660	825

The short cut for this command is the **Edit Past Steps** tool .

Convert

This function is used to convert an element in the model into a different type of element. Sometimes you may notice too late that you have used a series when the values are constant, or an input where a variable was required, etc.

In such cases this command allows you to change the element without having to delete it, which would cause the loss of all its connections with other elements.

In certain types of conversion, however (e.g. from a series to a constant), some data loss is inevitable. DS Lab will warn you when a conversion will result in lost data.

Set Print Size

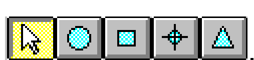
This command prepares the document for printing. DS Lab has a pagination reference grid which can be shown superimposed on the model by choosing the **Model setup...** command from the **Options** menu and enabling the **Show Page Breaks** check box.

The page size is defined by:

- The paper size selected;
- The margin settings;
- The orientation of the printer page;
- This menu command.

Clicking on the **Zoom In** and **Zoom Out** buttons changes the magnification level, showing a smaller or larger portion of the model on the screen. When you think you have obtained a satisfactory size for printing, choose **Set Print Size**. This command tells DS Lab that this is the scale at which the model is to be printed and that page breaks are to be set according to the current positioning of the page break lines.

To obtain a print preview, use the **View All Pages** command (**View** menu). This gives you a **whole model** print preview, typical of the Windows environment. If necessary, you can then select a different zoom level and reset the print size. In essence, you are able to change the way the model will be printed simply by using the zoom commands. You may then finalize the layout by moving any elements which lie on page breaks.

The short cut for this command is the **Set Print Size** tool .

The Simulation Menu

This is the core menu of the DS Lab. The functions contained in it control the actual simulation and *What if...* analysis once the model has been built. The basic concept behind DS Lab is that all the calculations defined in the model are repeated for each step of the simulation. The functions contained in this menu define the framework in which these repetitions will take place.

Parameters...
Calculate
Recalculate
Update DDE Links...
Export to Excel
Report
Link with Excel...

Parameters...

Choosing this command from the Simulation menu or the popup menu (or clicking the Simulation Parameters tool) opens a dialog box in which the following basic parameters are defined:

Step Unit

The **Step unit** is the unit in which the intervals between successive steps of the simulation will be measured. You can choose from **Day**, **Week**, **Month** or **Year**. You can also choose a generic **Unit** for cases not specifically provided. For example, a person following the evolution of the giraffe from its first appearance on the earth to the present time might want to use millennium steps.

In this manual we emphasize simulations over time, since most applications of DS Lab fall in this category, but of course models can also be built which use all kinds of other **Step units**. For this reason the **User Defined** option allows you to define your own step unit. This option is particularly useful for parallel simulations (e.g. different models of cars in a model to calculate running costs). When using the **User Defined** step unit, you must give a name to each step. Click the **User Defined Unit** button and enter the names.

Multiple Unit

Multiples of the selected step unit may be used in place of single units. For example, selecting **Month** as the step unit and **3** as the multiple means that the next step after January 1993 will be April 1993.

First Step

This establishes the first step in the simulation. For example, if the first step is the first month of 1993, just enter January 1993.

Number of Steps

Enter in this field the number of steps to be included in the simulation.

Last Step

DS Lab calculates this, using the formula:

Last Step = First Step + Number of Steps.

Current Step

DS Lab automatically increments this every time it executes the calculations for a step. The user may change it at any time. To move back and forth through the simulation steps, you can usually use the **Current Step** box on the toolbar: the right arrow will advance to later steps, the left arrow returns to earlier steps.

Simulation Start

This marks the step at which the program starts to calculate values according to the scripts of the

variables. It can be different from the **First Step**, as we shall see in the following example.

Suppose that in July a business sets up a model to predict sales for the year. **First Step** would be set to January and **Last Step** to December. Since the sales through June are known, we set **Simulation Start** to July. This way, *for some of the variables*, the program will consider data up to June as given and will calculate the values from July onwards. This is only true, however, for those variables whose **Always Calculate** check box is not selected. The known values of these variables are entered with the **Edit Past Steps** command in the **Model** menu.

Simulation Length

This sets the number of steps for which values are to be calculated during the simulation.

The values for the Current Step are updated on exiting from this dialog box, unles the **Automatic Calculation** option in the **Model setup** dialog box is disabled.

Simulation Parameters

Step Unit: Month — Multiple Unit: 1

First Step: January 1993 — N° of Steps: 12 — Last Step: December 1993

Current Step: January 1993

Simulation Start: January 1993

Simulation Length: 6

User defined Unit — OK — Cancel

To analyze monthly values over a two-year period (January 1993-December 1994), the following values must be assigned:

Step unit = Month

Multiple Unit = 1

First Step = January 1993

Number of Steps = 24

Last Step = December 1994 (calculated by the program).

To actually calculate values for the first half of 1993 only, enter the following values:

Current Step = January 1993

Simulation Start = January 1993

Simulation Length = 6

Now start the calculation with the **Calculate** command. The simulation will run through June 1993. At this point, having seen the results, it is possible to edit the script of a variable. Once the changes are made, reset the **Current Step** value in the dialog box to January 1993. The simulation many now be repeated from the beginning.

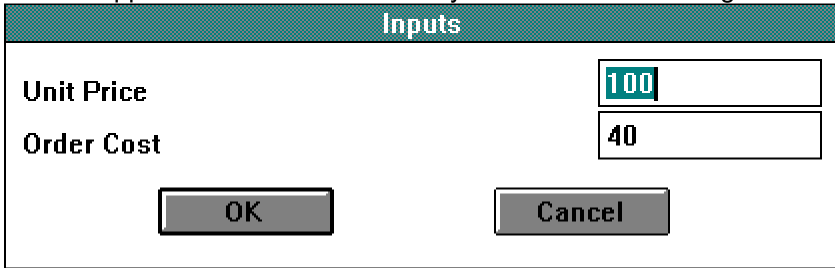
Calculate

This command is used to execute a simulation for the number of steps defined in **Simulation Length** (or to the **Last Step** if that is reached sooner), starting from the *Current Step*. What differentiates this command from **Recalculate** is that the simulation does not start again from the **First Step**, allowing it to be continued after pausing. Also the Inputs are not presented for redefinition.

This command can also be accessed by clicking the **Calculate** tool, or from the popup menu (click the right mouse button with no elements selected). Or press the short cut key F9.

Recalculate

This command executes a simulation for the number of steps defined in **Simulation Length**, starting from the **First Step**. Unlike the preceding command, **Calculate**, the simulation is run from the beginning, regardless of what the *Current Step* is. Also, if there are any Input elements defined in the model, a dialog box will appear where their values may be confirmed or changed.



If the value shown is correct, click on the **OK** button to confirm the current value. The simulation starts after the **OK** button is clicked.

While the simulation is running, the drop-down list box on the toolbar shows the steps as they are calculated. After the simulation has ended, you can review the results by moving back and forth through the steps: just click on the left and right arrows next to the **Current Step** drop-down list box, or open the list to select a specific step.

The time taken to execute calculations is directly proportional to:

- the number of steps,
- the number of variables,
- the complexity of the models structure, and
- the complexity of the scripts defining the variables.

However, DS Lab is very fast, and even for complex models calculation will rarely take more than a few minutes. The simulation may be interrupted at any time by pressing Esc.

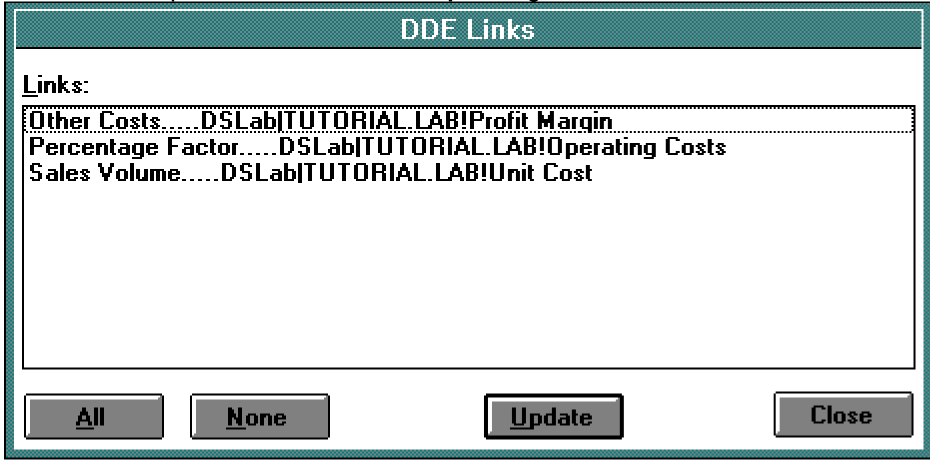
This command can also be accessed by clicking the **Recalculate** tool, or from the popup menu (click the right mouse button with no elements selected). Or press the short cut key F10.

Update DDE Links...

This command enables you to update in one pass all data import links established in the variables scripts.

Choosing this command calls up a dialog box with a list of all DDE links established with the **Paste Link** procedure. You may update **All**, **None**, or only those selected by clicking on their description. After selecting the links to be updated, click on the **Update** button to import the new data into DS Lab.

The values for the Current Step are updated on exiting from this dialog box, unles the **Automatic Calculation** option in the **Model setup** dialog box is disabled.



The short cut for this command is the **DDE** tool . It may also be accessed from the popup menu which appears on clicking the right mouse button when no elements are selected.

Export to Excel

This command makes it extremely simple to transfer the results of a simulation to an Excel spreadsheet. Just select the variables whose values you want to export and then choose this command (also available by clicking the right mouse button), or click the **Export to Excel** tool or press the F11 short cut key. This will create a new spreadsheet with the names of the variables listed in the rows and the simulation steps in the columns.

If the **Chart** option in **Workspace Setup** was enabled, this command will also produce a chart in the default style. If the simulation is repeated with different parameters and/or formulas for the variables, the above operations must be repeated to export the new output data to Excel.

Note: For this command to function correctly, the version of Excel installed must be specified with the **Workspace...** command. For the transfer to be automatic, Excel must be either already running or in a directory included in the PATH defined in your systems AUTOEXEC.BAT file.

Report

DS Lab also allows you to display the results of a simulation in spreadsheet format through its own internal spreadsheet mode, without running an external application. Select the desired elements of the model and then choose this command (also available by clicking the right mouse button), or click the **Report** tool or press the F12 short cut key. With the current version of DS Lab, it is **not** possible to save reports created in this way.

Link with Excel...

This is another important feature of the program. DS Lab can automatically export data from simulations to an Excel spreadsheet in a previously defined format, without going through the DDE functions which can link most Windows applications.

DS Lab provides two modes of data exchange which work exclusively with Excel. The first, and simplest, is described in ***Export to Excel***. It must be repeated each time you want to export data to Excel.

The second mode is much more sophisticated. It creates a permanent link with Excel. You must specify beforehand the name of the worksheet to be used, the variables to be exported and their positions on the worksheet. The two applications are then actively linked *all the time*, so that the Excel spreadsheet is automatically updated with the results of each simulation carried out in DS Lab.

Choose the **Link with Excel** command from the **Simulation** menu and specify the following data in the dialog box that appears:

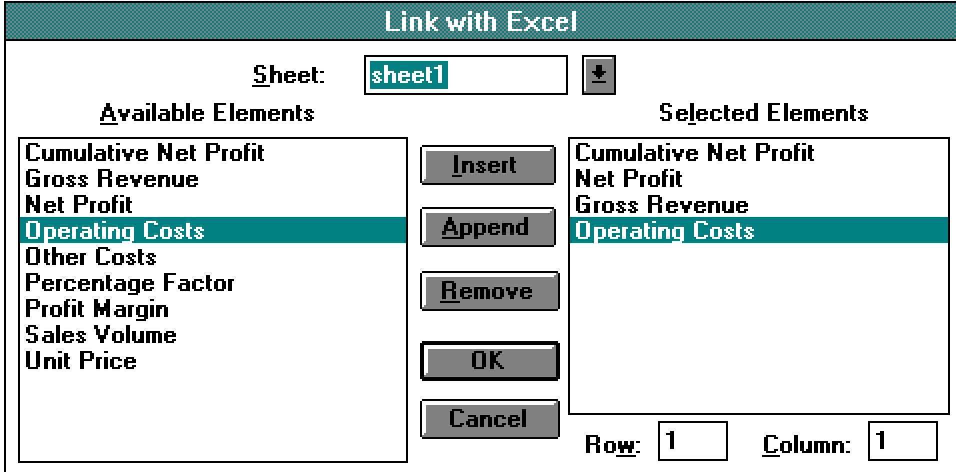
- The worksheet you intend to send data to. It must be chosen among those currently open.
- The elements of the model to be exported.

The dialog box shows two lists. The **Available Elements** list box contains all the elements which can be selected (all those in the model). The **Selected Elements** list box shows those elements which have been chosen for export to Excel. Between these two list boxes there are three buttons:

- **Insert** allows you to insert an element into the list of selected variables. Select the name of the element you want to export, then click on **Insert**. The element will be added to the **Selected Elements** list, above the highlighted element.

- **Append** allows you to add an element to the list of selected variables. Select the name of the element you want to export, then click on **Append**. The element will be added to the **Selected Elements** list, below the highlighted element.

- **Remove** an element from the **Selected Elements** list. Select the unwanted element then click on the **Remove** button.



In this dialog box you can also specify the area of the worksheet to be used. In the **Row** and **Column** boxes at the bottom right you enter the coordinates of the top left cell of the area in which to place the exported values.

After each simulation, the **Link with Excel** will send the results of the simulation to spreadsheets that have been prepared in advance as reports. The permanent link provides instantaneous updating of the reports.

The Window Menu

This menu contains two sections. The first offers commands affecting the placement of the open windows.

Cascade
Tile
Arrange Icons

The second part of the **Window** menu contains a list of all the models, text files and tables currently open. To go from one window to another, just click on the name of the file you wish to use.

Cascade

This is the standard mode for DS Lab, as for other Windows programs. Windows containing models are opened one on top of another. The user can resize them by dragging their frames to the desired height and width. To go from one model to another, click on any visible part of its window or select it from the list of currently open files in the lower section of the **Window** menu.

The short cut key for this command is SHIFT+F5.

Tile

This command arranges all the open windows side by side on the screen, allowing you to see them at the same time. Obviously the size of each window, and consequently the proportion of its contents displayed, diminishes as the number of windows increases. This mode is thus not the most suitable while working on a model, but can be useful when for example copying sections from one model to another, or for comparing two closely related models.

The short cut key for this command is SHIFT+F4.

Arrange Icons

This command arranges the icons in a row along the bottom of the screen. Windows can be reduced to icons by clicking on the down arrow in the top right corner of each window frame. This is different from closing the window. It is more like parking it out of the way while you do something else. This can be very useful, for example, when you have a lot of models open and you want to look at two of them side by side in Tiled mode. First, reduce all the other windows to icons; second, choose the **Tile** command to place the open windows side by side; and third, choose the **Arrange Icons** command to put the icons in a row.

The Excel! Menu

The presence of this menu is optional; it is enabled by the **Excel Menu** option in the **Workspace...** command of the **Options** menu.

The menu consists of the single command **Excel!**. Clicking on it takes you straight into Excel if it is in a directory included in the PATH statement of your computers AUTOEXEC.BAT file.

The Help Menu

This menu gives access to on-line help. It contains four commands:

Contents
How to Use Help
Product Support
About...

Contents

This command gives access to information about using DS Lab. Choose this command to access the list of contents, then select the subject you want to know about.

How to Use Help

Gives information on how to use Help.

Product Support

Gives information on how to contact technical support for DS Lab.

About...

Contains information on DS Lab products, including:

- the name of the program;
- the version number (remember to specify this when asking for upgrades);
- copyright information on this program, developed by DS Group s.r.l. and Prisma 2.0 s.r.l.

DS Lab Menus in Text Mode

When a text window is active, the menu bar changes. The commands for the management of models are replaced with specific text management commands. Thus when working with texts, the menu bar offers the following choices:

File, **Window**, **Excel!** and **Help** contain the same commands as in Model mode.

The following sections describe the commands in the remaining menus which are different from those available in Model mode.

The Edit Menu
The Find Menu

The Edit Menu

Word Wrap
Select All

Select All

This command selects the entire text. The other commands in the **Edit** menu (**Cut, Copy, Delete**) can then be applied to the whole text.

Word Wrap

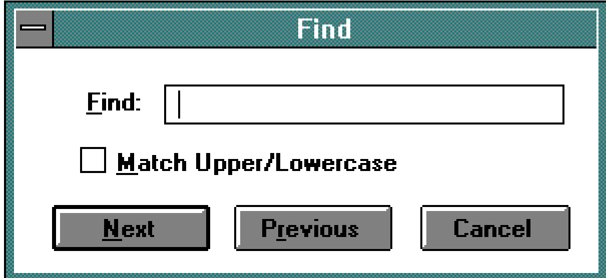
As its name implies, this command, when activated, causes words to wrap to the next line.

The Find Menu

This menu is unique to Text mode and consists of three commands:

- **Find...**
- **Next**
- **Previous**

The **Find...** command accesses a dialog box in which you are asked to specify the word or phrase to be searched for in the text. Subsequently, the **Next** and **Previous** commands find the same word or phrase further ahead and further back in the text.



The short cuts for the **Next** and **Previous** commands are **F3** and **F4**.

DS Lab Menus in Report Mode

When working with reports, there are only the following menus:

The **File** menu allows the same operations as in Model mode, with the exception that the **Save** and **Print** commands are absent. This is because DS Lab reports are provisional worksheets for evaluation of results only and cannot be saved to disk or printed.

The **Edit** menu has only one command   **Copy**   which allows results to be copied to the Clipboard for transfer to other Windows applications such as spreadsheets and word processors.

Window, **Excel!** and **Help** contain the same commands as in Model mode.

Editing the Scripts of the Variables

Once a model has been built graphically, it must be defined mathematically by entering the formulas and expressions defining the variables. For our purposes, these formulas and expressions will be called the *script* of a variable. A script can range from a simple formula to a complex program: the degree of complexity depends entirely on the user.

The Script Editing Dialog Box
Full Screen Editing Mode

The Script Editing Dialog Box

The Script Editing dialog box is accessed by double-clicking on the symbol of a variable. It is here that the selected variable is given its mathematical definition.

Variable

Connected Elements
Cumulative Net Profit
Net Profit

Functions/Instructions
Abs(number)
AddDay(internal_date , days)
AddMonth(internal_date , mon
AddWeek(internal_date , week
AddYear(internal_date , years
ADUE
AFVFutureValueIncreasingCash

Name
Cumulative Net Profit
Self-Reference
Starting Value
0
Always Calculate

([& <
)] | >
+ * ^ !
- / % =

Script
Cumulative Net Profit + Net Profit

DDE:
Paste Link
Update
Edit Script
OK
Cancel

The **Connected Elements** list box contains the names of all the elements connected to the variable currently being defined. The script of a variable may reference only the names of elements connected to it by an arrow. If at this stage it becomes obvious that other elements are needed in the script, you must return to the model and connect the additional elements. On returning to the Script Editing dialog box, you will find that the new names have been added to the list of elements available for selection.

The script defining the variable is composed in the **Script** text box. Choosing an element from the **Connected Elements** list box causes it to be copied into the **Script** box. The name can also be typed directly into the text box, but the first method is preferable as it eliminates the risk of typing errors.

The **Functions/Instructions** list box shows the functions and instructions which can be used in defining the script. Scrolling through this list should make it obvious that DS Lab is equipped to deal with even the most complex formula. There are numerous mathematical, financial and statistical functions; logical operators such as **And** and **Or**; and DS Lab script language programming instructions.

At the top right is an edit box containing the name of the variable, where it can be changed if desired.

Immediately below this is a *numerical input box* where a **Starting Value** is entered if required, and the **Self-Reference** check box. If this is selected, the variable can refer to itself in the script (that is, include a previous value of itself as an input to its formula). If **Self-Reference** is enabled, DS Lab automatically includes the name of the variable whose script is being edited in the **Connected Elements** list box.

On the right is a small calculator-style keypad for the benefit of those who prefer to use the mouse all the time, rather than pressing the corresponding keys on the keyboard.

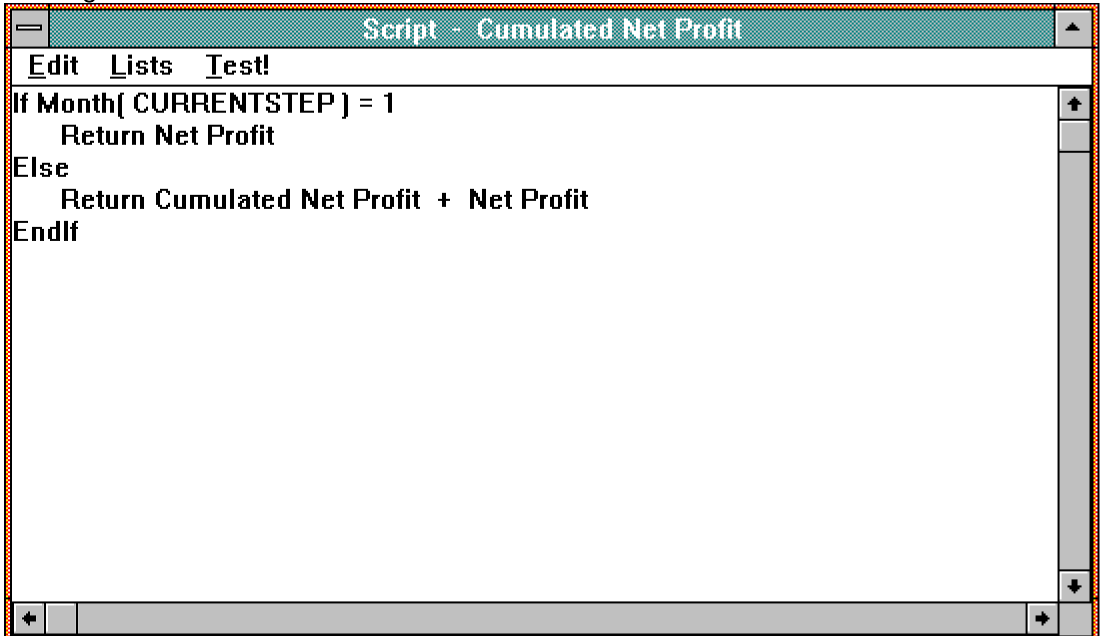
Further down is the **Always Calculate** check box.

At the bottom there is the **DDE** text box and two buttons, **Paste Link** and **Update**, for operations relating to the input of data through DDE.

When script editing is complete, choose **OK** to return to the model.

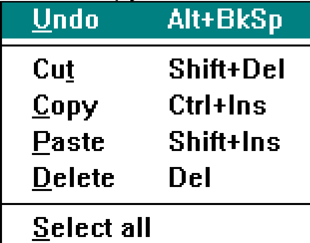
Full Screen Editing Mode

The Script Editing dialog box also contains a button marked **Edit Script**. Clicking this button expands the Script text box to full-screen mode for the convenience of those users who need to write sizable programs defining a variable.



The full screen *Script Editor* has three menus:

- **Edit** provides the usual **Cut, Copy** and **Paste** functions, which can be used not only within a script, but also to copy text from the script of one variable and paste it into another.



- **Lists** calls up the two list boxes containing the **Connected Elements** and the **Functions/Instructions** available for inclusion in the script.

- **Test!** consists of a single command which instantly checks the syntactic validity of the script that has been entered.

Double-clicking the control-menu box returns you to the standard **Script Editing** dialog box.

Rules for Building DS Lab Models

Building a model includes both setting it out graphically and defining it mathematically. During both of these stages, DS Lab carries out checks on its formal correctness.

Checks by the Graphic Model Editor
Checks by the Script Editor
Error Values

Checks by the Graphic Model Editor

- Only *variables* can have in-coming connections. *Series, constants, inputs* and *tables* are not defined from within the model. They can therefore only have out-going connections.

- *Shadow variables* cannot have in-coming connections, only out-going connections. The input connections must be defined in the primary variable.

- A model can contain more than one *shadow* of the same element, but only one of them may be connected to the same variable.

Checks by the Script Editor

The script editor carries out two kinds of checks:

- syntactical checks on the script;
- circular reference checks.

The first reviews the script of a variable to ensure that the syntax of the DS Lab language is respected. This language is fully described in ***The Script Language***, which contains details of the syntax and usage of all functions and instructions.

The second type of check, similar to that found in spreadsheet applications, prevents the insertion of undefined variables.

For example, the following scripts for the two variables VAR_A and VAR_B constitute an undefined system:

VAR_A = VAR_B + 10

VAR_B = 1000 - VAR_A

It is obvious that the value of VAR_A depends on that of VAR_B and vice versa.

DS Lab will accept the first formula, but will return a *circular reference* error message when the second is entered. The above example is very simple, but the principle remains valid for systems (models) containing a large number of elements.

It is important to observe, however, that the script of a variable may include a reference to a previous step. In the above example, if the script of VAR_B had been VAR_B = 1000-VAR_A[-1], there would have been no circular reference error.

Error Values

A variable will assume an error value in the following situations:

- The script is incorrectly defined;
- The calculation does not produce a meaningful value;
- Changes have taken place in the model which require a new simulation.

There are three possible types of error value:

Syntax Errors in the Script
Run-Time Errors
Use of a Variable with an Error Value (Cascade Error)

Syntax Errors in the Script

A question mark **[?]** below the name of a variable means that the script has not been defined or is syntactically incorrect.

Repeatedly pressing the function key F6 will cause DS Lab to display, one after another, all variables that return this error value.

Run-Time Errors

[#ADDDAY]	Error in the *internal_date* argument of the **AddDay** function.
[#ADDMONTH]	Error in the *internal_date* argument of the **AddMonth** function.
[#ADDWEEK]	Error in the *internal_date* argument of the **AddWeek** function.
[#ADDYEAR]	Error in the *internal_date* argument of the **AddYear** function.
[#AFV/*n*]	Error in an**Advanced Future Value Functions (AFV)**. The number identifies the type of error.
[#APV/*n*]	Error in an **Advanced Future Value Functions (AFV)**. The number identifies the type of error.
[#ARCCOS/-1,1]	Arc-cosine of a number less than -1 or greater than 1.
[#ARCSIN/-1,1]	Arcsine of a number less than -1 or greater than 1.
[#ARRAY/RC]	Invalid array notation referring to a table.
[#ARRAY]	Invalid array notation referring to a variable or series.
[#BOND/*n*]	Error in a **Bond Management** function. The number identifies the type of error.
[#DATE]	Error in a date passed to the **Date** function.
[#DAY]	Error in the *internal_date* argument of the **Day** function.
[#DAYWEEK]	Error in the *internal_date* argument of the **DayWeek** function.
[#DAYYEAR]	Error in the *internal_date* argument of the **DayYear** function.
[#DDEREQUEST]	Error in importing data from another Windows application with the **DDERequest** function.
[#DIV/0]	Division by zero.
[#DOMAIN]	Domain error in a mathematical function argument.
[#ERR]	Indicates a generic run-time error. All run-time errors not falling in any predefined category generate the value **#ERR**.
[#EVERY]	Errorin the arguments of the **Every** Function
[#EVERYDAY]	Errorin the arguments of the **EveryDay** Function
[#EVERYMONTH]	Errorin the arguments of the **EveryMonth** Function
[#EVERYWEEK]	Errorin the arguments of the **EveryWeek** Function
[#EVERYYEAR]	Errorin the arguments of the **EveryYear** Function
[#FACT/<=0]	Factorial of a negative number.
[#FACT/NOINT]	Factorial of a non-integer number.
[#FV/n]	Error in a **Future Value Functions (FV)**. The number identifies the type of error.
[#GETSTEP]	Error in the *user_defined_step* argument of the **GetStep** function.
[#INVENTORY/*n*]	Error in an **Inventory Management** function. The number identifies the type of error.
[#LOG/<=0]	Natural logarithm of a negative number.
[#LOG10/<=0]	Base 10 logarithm of a negative number.
[#MOD/0]	Division by 0 in a calculation in the model.
[#MONTH]	Error in the *internal_date* argument of the **Month** function.

[#NOMEMORY] Insufficient memory for the calculation.

[#NUMPERIODS/*n*] Error in the **NumberOfPeriods** function. The number identifies the type of error.

[#OVERFLOW] Number outside the allowed range.

[#PLOSS] Partial loss of significance.

[#POW/<0] Raising to a non-integer negative power with base less than zero.

[#POW/=0] Raising to a negative power with base equal to zero.

[#PV/*n*] Error in a **Present Value Functions (PV)**. The number identifies the type of error.

[#REQUEST] Error in importing data from another DS Lab model with the **Request** function.

[#SING] Error in a mathematical function argument.

[#SP/*n*] Error in a **Stock Portfolio Functions (SP)**. The number identifies the type of error.

[#SQRT/<0] Square root of a negative number.

[#ST/*n*] Error in a **Short Term Investment Functions (ST)**. The number identifies the type of error.

[#STAT/*n*] Error in a **Statistical Functions**. The number identifies the type of error.

[#TLOSS] Total loss of significance.

[#TREND1/*n*] Error in the **Trend1** function. The number identifies the type of error.

[#TREND2/*n*] Error in the **Trend2** function. The number identifies the type of error.

[#TREND3/*n*] Error in the **Trend3** function. The number identifies the type of error.

[#UNDERFLOW] Number outside the allowed range.

[#WEEK] Error in the *internal_date* argument of the **Week** function.

[#YEAR] Error in the *internal_date* argument of the **Year** function.

Use of a Variable with an Error Value (Cascade Errors)

[#*value*] Indicates a value that is no longer reliable. It is generated when another variable connected as an input assumes a value that is no longer reliable or when it is left undefined.

[#REF] The variable in question makes use of another variable which has a Run-Time error at the specified step.

These two error values are powerful debugging tools, enabling you to rapidly identify the root of a problem. By following back the chain of variables showing one of these error values, you quickly reach the variable which generated the error.

- In the case of a chain of **[#*value*]** errors, this variable will have an undefined or incorrectly defined script, and thus the error value **[?]**.

The Script Language

A *script* is used to define each variable in a model. The script is actually a program written in a simple programming language.

The purpose of the script is to give a numerical value to the variable. In most cases, it is very simple: a single expression that calculates a number. For example, the script of the *Total Costs* variable will be the sum of all the *Expense* variables connected to it.

In this respect, the DS Lab language is very similar to a spreadsheet formula language. The script can however become a small program including the *local variables* and control structures typical of all programming languages.

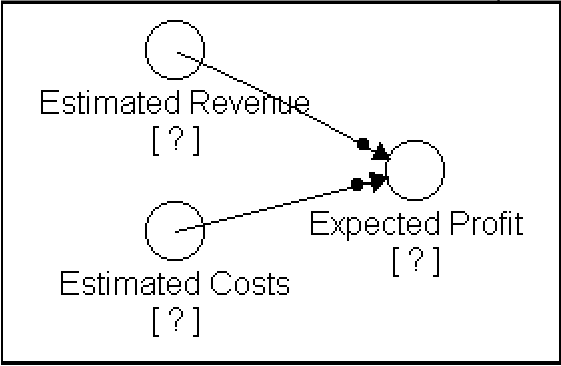
The Structure of a Script
Data Types and Formats
DS Lab Model Elements
Local or Temporary Variables
Step, RC and Array Notation
Script Programming Instructions
Operators
Functions
System Constants
System Variables

The Structure of a Script

The script is defined in connection with a variable: each variable has a script that calculates its value (even if this is a constant). Therefore a script must always return a single numerical value that will be the value of the variable at that step.

For this reason, the script is usually an expression which calculates a value, exactly like a spreadsheet formula. Before the final expression, however, there can be a short or long program that defines local variables, executes cycles, reads a table, etc.

For example, suppose we have two variables; one called *Estimated Revenue* and the other *Estimated Costs*. From these we want to calculate a profit forecast.



One possible script for the *Expected Profit* variable is simply the difference between revenue and costs:

```
estimated revenue - estimated costs
```

This is the simplest case: the script is a ***single expression*** like a spreadsheet formula.

Now, supposing we want to simulate a situation in which the profit must be no less than 10, the script could be developed like this:

```
minimum_profit = 10
local_estimated = estimated revenue - estimated costs
Return Max(minimum_profit, local_estimated)
```

This script no longer looks like a spreadsheet formula, more like a ***small program*** in a language such as Basic or dBase. The first two lines assign values to **Local Variables** and the third returns the greater of the two values.

The Script thus has this basic structure:

[preliminary program]

[final expression]

However, the *preliminary program* need not always exist; the script is then just an expression that calculates a value. The ***preliminary program*** is a sequence of instructions like any program in any programming language. A special instruction **Return** *expression* causes immediate termination of the programs execution. In this case the value returned by the script is the value of the *expression*, which must always be defined after the keyword **Return**.

The ***final expression*** need not exist either, because a script can return a value with the **Return** instruction. Most scripts, however, consist only of the *final expression*.

The *expression* is a formula like those in a spreadsheet. It can be a number, a variable or a simple or complex calculation containing operators, functions, constants and system variables. If no *final expression* is calculated in the Script, the returned value is 0.

There must be no other instruction after the *final expression*. This is checked when the script is entered.

Data Types and Formats

There are two data types in DS Lab: *numbers* and *dates*.

- *Numbers* can be integers, decimal numbers or expressed in exponential notation. Internally they are all saved in double precision floating point format (IEEE   Institute of Electrical and Electronics Engineers standard). This means that numbers can range approximately between 1.7E-308 and 1.7E+308.

- *Dates* are represented internally as an eight figure number obtained from the year, month and day in the form YYYYMMDD.

Single-figure months and days are preceded by a 0 (zero). For example, July 6th 1993 is represented internally by the number 19930706.

The system variables **CURRENTSTEP** and **TIME** contains the date of the current step unit in this format. The function **SystemDate** returns the date given by the computers internal clock in the same format.

DS Lab Model Elements

Variables
Constants
Series
Inputs
Tables

Variables

Variables are those elements whose value is calculated through a script. They are represented graphically by a circle.

For shadow variables, the symbol becomes a double circle.

A variable can be used in another variables script if it is connected graphically to the second variable, when it will appear in the list of **Connected Elements** in the second variables script editing box.

Constants

Constants represent a fixed value, which ordinarily does not change during the simulation and any subsequent simulations (though by using the **Poke** function, the value of a Constant can be changed in the course of a simulation).

Constants are represented by a square.

For shadow constants, the outline of the square is doubled.

To be used in the script of a variable, the constant must be connected to it graphically, when it will appear in the list of **Connected Elements** in the variables script editing box.

Series

Series represent data which is entered manually or copied from external sources, not calculated through a formula or script. Usually they are used to give starting data to a model. A series contains one value for each step in the simulation.

Series are represented by a crosshair symbol.

For shadow series, the outline of the symbol is doubled.

To be used in the script of a variable, the series must be connected to it graphically, when it will appear in the list of **Connected Elements** in the variables script editing box. A particular value in the series is referenced in the variables script by the use of *Step Notation.*

Inputs

Inputs, like constants, represent a fixed value. The only difference between the two types of element is that, when the **Recalculate** command is given, a dialog box appears listing all the Inputs in the model and prompting the user to confirm or change the value assigned to them. Inputs do not normally change their value during the simulation (though by using the **Poke** function, the value of an Input can be changed in the course of the simulation).

Inputs are represented by a triangle.

For shadow constants, the outline of the triangle is doubled.

To be used in the script of a variable, the input must be connected to it graphically, when it will appear in the list of **Connected Elements** in the variables script editing box.

Tables

Like series, *Tables* represent data which is entered manually or copied from external sources. The table format allows you to insert values and comments in a conveniently flexible form: in effect, each table is a small spreadsheet (100 rows x 16 columns).

Tables are represented by a grid symbol.

For shadow tables, the outline of the symbol is doubled.

To be used in the script of a variable, the table must be connected to it graphically, when it will appear in the list of **Connected Elements** in the variables script editing box.

In the script of a variable, you can refer to a particular cell in a table connected to it by means of the usual spreadsheet cell notation (*RC Notation*), e.g. R2C4 or R(*pointer*)C4, where *pointer* is the name of a local variable.

Local or Temporary Variables

A script may contain one or more local variables. The first time a local variable is used must be in an **assignment** statement which gives it an initial value. After that it can be used anywhere it is allowed. It can be reassigned with a new value later in the script. The local variable exists only during execution of the script in which it was defined: its value is not kept after the scripts value has been calculated. This is why it is also called temporary.

The name of a local variable must start with either a letter or an underscore character (_), and can include only alphanumeric and underscore characters.

Only the first 15 characters of a local variables name are significant. However, the name itself may be longer than 15 characters.

Step, RC and Array Notation

Step Notation
RC Notation
Array Notation

Step Notation

Step notation is used to refer to the value of an element at a specific step in the simulation. It is enclosed in square brackets after the name of an element. This notation can be used with *series* and *variables*.

Syntax

element **[** [+ | -] *expression* **]**

There are three types of step notation:

- **Backward Relative:** after the square bracket, the minus sign (**-**) must be inserted, indicating backward relative notation, followed by the number of steps.

For example, [-1] returns the value of *element* for the previous step. If this refers to a non-existent step, the value 0 is returned.

- **Forward Relative:** after the square bracket, the plus sign (**+**) must be inserted to indicate forward relative notation, followed by the number of steps.

For example, [+3] returns the value of *element* for three steps further forward. If this refers to an non-existent step, the value 0 is returned.

This notation cannot be used with variables because when the current step is calculated, the future steps have yet to be calculated.

- **Absolute:** after the square bracket, the step (1,2,...) or the period (January 1993, February 1993, ...) is specified for which the value is to be taken. If a non-existent step is referred to, the value 0 is returned.

In this case the model becomes dependent on the simulation parameters chosen: if the step unit is the day, the absolute reference must be expressed in days, etc.

Incorrect step notation is the most common cause of circular reference errors. Specifying a different step (normally a past step) in one of the variables will resolve most errors of this type.

RC Notation

RC notation is a means of indicating a particular cell in a table. This notation can only be applied to *tables*.

Syntax

table ***[R*** *[* ***(*** *] expression [* ***)*** *]* ***C*** *[* ***(*** *] expression [* ***)*** *]* ***]***

The parentheses are required when an *expression* other than a number is specified. For example, [R1C1] refers to the cell in row 1, column 1; whereas [R(SIMSTEP)C1] refers to the cell in column 1 of the row whose number is the same as that of the current step.

Array Notation

Array notation is used in DS Lab to specify a range or array of values. In the script language, array notation is used only in the arguments of functions that require a series of values.

For example, the **Sum** function calculates the sum of a series of values, and requires as its argument the name of an element followed by array notation. Array notation can be used with *variables, series* and *tables*; the syntax for the three cases differs, and is explained below for each case.

Any element referred to with array notation is included in the general term *array*.

Syntax 1 Variables

In the case of a *variable*, array notation consists of two *backward relative* or *absolute step notation* references indicating the first and last values to be included in the array, separated by a colon (:).

variable **[** [-] *expression* **:** [-] *expression* **]**

Note: In the case of absolute references, the value 0 (zero) is returned for any steps that have not yet been calculated (steps which are further forward in the simulation than the *current step*).

Syntax 2 Series

In the case of a *series*, array notation consists of two step notation references of any type (*backward relative*, *forward relative* or *absolute*) indicating the first and last values to be included, separated by a colon (:):

series **[** [+ | -] *expression* **:** [+ | -] *expression* **]**

The two elements may be specified in different ways. For example, to sum the last three steps, including the current one, of a series called *Values*, the following array notation could be used:

```
Sum (Values [-2 : SIMSTEP] )
```

where **SIMSTEP** is the system variable containing the number of the current step.

Syntax 3 Tables

In the case of a *table*, array notation consists of two references in *RC notation* indicating the first and last cells to be included in the array, separated by a colon (:):

table ***[R*** *[* ***(*** *] expression [* ***)*** *]* ***C*** *[* ***(*** *]expression [* ***)*** *]* ***:*** ***R*** *[* ***(*** *] expression [* ***)*** *]* ***C*** *[* ***(*** *]expression [* ***)*** *]* ***]***

The parentheses are required when an expression other than a number is specified. The array can be defined in a row or in a column. If it is defined in a column, the row must remain constant (e.g. [R1C1:R1C5]); if in a row, the column must remain constant (e.g. [R1C1:R5C1]).

Script Programming Instructions

The script programming instructions control the order in which operations are executed.

The available instructions are:

Assignment
Do Case ... EndCase
Do While ... EndDo
Exit
Return
For ... Next
If ... EndIf
Loop
LoopTime
Rem/Note/'/&&

Operators

The order of priority of the operators is shown in the following table. It is that followed by most programming languages:

-					
(sign)		**NOT**			
!					
^					
*****	**/**	**MOD**			
%					
+	**-**				
=	**<**	**>**	**<=**	**>=**	**<>**
&					
(string)					
AND					
&					
OR					
|					

The operators on the same line have the same priority. If two operations have the same priority they are executed from left to right. So, unless otherwise specified by the use of parentheses, mathematical operations are executed first, then relational operations and logical operations last.

For example, the expression

```
13 + 1 = 14
```

is true and gives as a result 1 (**TRUE**), because the mathematical operator + is evaluated before the relational one =.

If we had used parentheses, like this:

```
13 + (1 = 14)
```

we would have forced the evaluation of the relational operator = before the mathematical one. The expression within parentheses returns 0 (**FALSE**) which, when added to 13, returns 13 as the value of the whole expression.

Mathematical operators
Relational Operators
Logical Operators
String Operators

Mathematical Operators

The following mathematical operators (listed in order of priority) can be used in DS Lab:

With *one argument* and the syntax:

operator expression

-	**Change sign.**

With *two arguments* and the syntax:

expression1 operator expression2

^	**Raise to the power of**
*****	**Multiplication**
/	**Division**
% | MOD	**Remainder of an integer division**
+	**Addition**
-	**Subtraction**

Relational Operators

The following relational operators (all of the same priority) are available in DS Lab. They all take *two arguments* with the following syntax:

expression1 operator expression2

Operator	Description
=	**Equal to**
>	**Greater than**
>=	**Greater than or equal to**
<	**Less than**
<=	**Less than or equal to**
<> | !=	**Not equal to**

Logical Operators

The following logical operators (listed in order of priority) can be used in DS Lab:

With one argument:

NOT | !

with the syntax:

NOT *expression*

or

! *expression*

The **NOT** operator can be used in either of the above forms and returns the logical opposite of the *expression*. Any positive or negative value of the *expression* other than 0 has the logical value **TRUE**, so its **NOT** is **FALSE** (0).

With two arguments (equal priority):

AND | &

with the syntax:

expression1 **AND** *expression2*

or

expression1 **&** *expression2*

The **AND** operator can be used in either of the above forms and returns the value **TRUE** if both the expressions are true. Any positive or negative value of the expressions other than 0 has the logical value **TRUE**.

OR | |

with the following syntax:

expression1 **OR** *expression2*

or

expression1 **|** *expression2*

The **OR** operator can be used in either of the above forms and returns the value **TRUE** if at least one of the two expressions is **TRUE**. Any positive or negative value of the expressions other than 0 has the logical value **TRUE**.

String Operators

The following *String concatenating* operator is available in DS Lab:

&

with the following syntax:

string1 **&** *string2* **|** *mathematical_expression* **&** ...

This operator is used in functions that require strings as arguments (such as **DDEExecute, DDEPoke, DDERequest, Date**, etc.). For example, the following script can be used to export the value of the variable *Costs* to the Excel spreadsheet Sheet1, placing it in a cell in row 1 and the column corresponding to the step of the simulation (column 1 for the first step, column 2 for the second, etc.):

```
DDEPoke ("Excel", "Sheet1", "R1C"&SIMSTEP, Costs)
```

Functions

The functions that can be used in scripts are summarized here and listed by categories. In the DS Lab script language, it is conventional to capitalize the initial letter of each word making up the name of the function and to write the rest of the word in lower case. However, when you use a function in a script you can write it in either upper or lower case. The function will be saved internally with its name in the standard format and the next time you call up that script you will find it has the first letter of each word capitalized.

Financial Functions
Inventory Functions
Statistical Functions
Mathematical Functions
Time Related Functions
Input/Output and DDE Functions
Error Functions
Control Functions

Financial Functions

The financial functions are divided into seven groups of related functions. The names of all functions start with a group code. For example, the standard function in the Present Value group that calculates the Internal Rate of Return is named **PVRateOfReturn**. This allows immediate identification of the group to which a function belongs and puts all the functions of the same group together in alphabetical order.

The seven groups and their codes are:

Present Value Functions (PV)
Advanced Present Value Functions (APV)
Future Value Functions (FV)
Advanced Future Value Functions (AFV)
Bond Functions (Bond)
Short Term Investment Functions (ST)
Stock Portfolio Functions (SP)

In the financial functions, all percentages are written as decimal numbers. For example, 45% is written 0.45.

Present Value Functions (PV)

The Present Value functions are used to calculate the present value of a future amount. The future amount can be a single payment or a series of payments: for example, annuities, pensions and deferred payments. By calculating their present values, the PV functions enable you to compare cash flows taking place at different times. These functions are limited to cases involving fixed cash flows at regular intervals and constant interest rates, and those that can be reduced to this form. For cases not falling in this category, the **Advanced Present Value (APV)** functions should be used.

The functions in this group start with the code **PV (Present Value)** and are listed below:

PVNumberOfCashFlows
PVPresentValue
PVPresentValueDeferredCashFlows
PVPresentValueOfFutureValue
PVRateOfReturn
PVSizeOfCashFlows
PVSizeOfDeferredCashFlows.

Advanced Present Value Functions (APV)

The functions in the **Present Value (PV)** group are limited to cases involving fixed cash flows at regular intervals and constant interest rates. The **Advanced Present Value (APV)** functions are not subject to the these limitations and are suitable for cases in which the cash flows and/or interest rates vary.

To obtain this greater flexibility, these functions require more data. Series or tables must be used for all arguments having a different value at each step. For example, the function will read the variable cash flows listed in a table called *Cash Flows* using the notation
`Cash Flows [R1C1:R1C24].`

The functions of this group can be subdivided into three levels requiring increasing amounts of data:

In *first level* functions, the cash flow timings and interest rates are constant, but the value of the cash flow changes at a fixed rate. These functions can be used in cases where the cash flows are index-linked, and limit the amount of data required. An example is **APVPresentValueIncreasingCashFlows**.

In *second level* functions, the cash flow timings and interest rates are constant, but the value of the cash flows can vary for each step. This level is available in most spreadsheets, but without all the options provided by the functions of this group. An example is **APVPresentValueVariableCashFlows**.

In *third level* functions, both the interest rate and the amount of the cash flows can vary at each step. The only limitation is that the interest rate can change only when the end of a cash flow period coincides with the end of an interest compounding period. An example is **APVPresentValueVariableRates**.

Cash flows can be positive or negative. A *positive value* for a cash flow means a sum received by the account concerned: a borrowing or a bond coupon payment will therefore be positive. A *negative value* represents an outgoing sum: an interest payment or the repayment of a loan will be negative. In the second and third level functions, *interest rates* can also be *negative*.

The functions in this group start with the code **APV (Advanced Present Value)** and are listed below:

APVNumberOfCashFlows
APVPresentValueIncreasingCashFlows
APVPresentValueVariableCashFlows
APVPresentValueVariableRates
APVRateOfReturnIncreasingCashFlows
APVRateOfReturnVariableCashFlows
APVSizeOfCashFlows

Future Value Functions (FV)

The functions of this group estimate the growth of a fund or of any series of cash flows subject to compound interest. These functions are limited to cases involving constant cash flows at regular intervals with a constant interest rate, or which can be reduced to this form. For cases not falling in this category, the **Advanced Future Value Functions (AFV)** should be used.

These functions do not require series or tables; they use the given arguments to calculate a single value. The interest rate must be a number between 0 and 1, so the future value will always be greater than the present value. Both the number and the value of the cash flows must be greater than 0.

The functions in this group start with the code **FV (Future Value)** and are listed below:

FVFutureValue
FVFutureValueOfPresentValue
FVNumberOfCashFlows
FVNumberOfPeriods
FVRateOfReturn
FVSizeOfCashFlows

Advanced Future Value Functions (AFV)

The functions in the **Future Value (FV)** group are limited to cases involving fixed cash flows at regular intervals and constant interest rates. The **Advanced Future Value (AFV)** functions are not subject to the same limitations. They are suitable for cases in which the cash flows and/or interest rates vary.

To obtain this greater flexibility, these functions require more data. Series or tables must be used for all arguments having a different value at each step. For example, the function will read the variable cash flows listed in a table called *Cash Flows* using the notation
`Cash Flows [R1C1:R1C24].`

The functions of this group can be subdivided into three levels requiring increasing amounts of data:

In *first level* functions, the cash flow timings and interest rates are constant, but the value of the cash flows changes at a fixed rate. These functions can be used in cases where the cash flows are index-linked, and limit the amount of data required. An example is **AFVFutureValueIncreasingCashFlows**.

In *second level* functions, the cash flow timings and interest rates are constant, but the value of the cash flows can vary for each step. This level is available in most spreadsheets, but without all the options provided by the functions of this group. An example is **AFVFutureValueVariableCashFlows**.

In *third level* functions, both the interest rate and the amount of the cash flows can vary at each step. The only limitation is that the interest rate can change only when the end of a cash flow period coincides with the end of an interest compounding period. An example is **AFVFutureValueVariableRates**.

Cash flows can be positive or negative. A *positive value* for a cash flow means a sum received by the account concerned: a borrowing or a bond coupon payment will therefore be positive. A *negative value* represents an outgoing sum: an interest payment or the repayment of a loan will be negative. With the second and third level functions, *interest rates* can also be *negative*.

The functions of this class start with the code **AFV (Advanced Future Value)** and are listed below:

AFVNumberOfCashFlows
AFVFutureValueIncreasingCashFlows
AFVFutureValueVariableCashFlows
AFVFutureValueVariableRates
AFVRateOfReturnIncreasingCashFlows
AFVRateOfReturnVariableCashFlows
AFVSizeOfCashFlows

Bond Functions (Bond)

This group includes functions to evaluate individual bonds and to measure their yields and risk factors. A bond is a guarantee of receiving cash flows at a fixed interest rate (*coupons*) until the bond expires, at which time you receive the final interest payment and the repayment of the bonds *face value*. The term *Bonds* includes all categories of fixed-interest securities, including government or public authority bonds and private fixed-interest investments such as debentures. All percentages are expressed as decimal numbers. For example, a 12% coupon rate is expressed as 0.12.

Suppose at the end of 1991 we buy a 7% treasury bond expiring in 1996 with a face value of $1000. This means that each year for the next 5 years we will receive 7% interest on $1,000 (the *coupon rate*), that is, $70. For simplicity, let us assume that interest is paid only once a year (though in practice payments are usually made twice yearly, or even more frequently in the case of private securities).

Our bond guarantees the following cash flows:

1992	1993	1994	1995	1996
70	70	70	70	1070

The 1996 cash flow is the sum of the final coupon and the face value of the bond.

The *price of a bond* is always expressed as a percentage of its face value. In this case, let us suppose its price is 91.49%, that is we pay $914.90 for it (the *issue price*).

We will have the following series of cash flows:

1991	1992	1993	1994	1995	1996
-914	70	70	70	70	+1070

The internal rate of return (IRR) of the bonds total cash flows is the discount rate which makes the value of the bonds future cash flows equal to its purchase price. It is also known as *yield to maturity*.

The functions of this group start with the word **Bond** and are listed below:

BondAccruedInterest
BondCost
BondCostEstimate
BondDuration
BondPrice
BondPriceEstimate
BondSensitivity
BondYieldToMaturity

Short Term Investment Functions (ST)

The meaning of **Short Term** is not rigidly defined in financial jargon, but it usually means less than a year or less than one interest period. Certificates of Deposit, treasury bills and bank deposit accounts are typical short term investments. Short term instruments are usually calculated using the 360/30 day convention, meaning that there are 30 days in a month and 360 days in a year.

When not otherwise specified, the yields in these functions are compounded annually. The discount rate is annual but is not compounded; thus, a rate of 9% for two years would total 18%.

The functions that use **Note** (in this case, short term investment) are divided into three types according to the information they require:

- **Type 1:** those which require as arguments the discount rate and the face value of the security;
- **Type 2:** those which require as arguments the price and the face value of the security;
- **Type 3:** those which require as an argument the internal rate of return of the security.

The functions of this group start with the code **ST** and end with the number of the group they belong to. They are listed below:

STEndingBalance
STNoteDiscount2
STNoteDiscount3
STNotePrice1
STNotePrice3
STNoteYield1
STNoteYield2

Stock Portfolio Functions (SP)

This group comprises investment portfolio management functions. The functions of this group start with the code **SP (Stock Portfolio)** and are listed below:

SPEquityCallOption
SPEquityRateOfReturn
SPPortfolioAveragePeriodicReturn
SPPortfolioRateOfReturn
SPPortfolioStandardDeviation1
SPPortfolioStandardDeviation2
SPPortfolioTimeWeightedRateOfReturn

Inventory Functions (Inventory)

This group comprises inventory management functions. These are standard functions found in most inventory management packages. Their inclusion in DS Lab allows you to build models employing the same criteria   reorder, minimum stock, etc.   used in management systems.

The names of these functions all start with the word **Inventory** and are listed below:

InventoryEconomicOrderQuantity
InventoryReorderPoint
InventorySafetyStock
InventoryServiceLevel

Statistical Functions

This section includes some basic statistical functions. These are essential tools in many areas of financial analysis. In particular, some of these functions are the basis for stock market analysis and portfolio management. The functions are listed below:

Alpha
Beta
CorrelationCoefficient
Mean
StandardDeviation
StandardErrorOfBeta
StandardErrorOfRegression
Trend1
Trend2
Trend3
Variance
WeightedAverage

Mathematical Functions

This section lists all the mathematical functions available in DS Lab.

Abs
ArcCos
ArcSin
ArcTan
Cos
CosH
Exp
Fact
Int
Log
Log10
Max
Min
Rand
Round
Sin
SinH
Sqrt
Sum
SumAll
Tan
TanH

Time Related Functions

This section lists the functions used to manage the time element of simulations. These functions are listed below:

AddDay
AddMonth
AddWeek
AddYear
Date
Day
DayWeek
DayYear
Every
EveryDay
EveryMonth
EveryWeek
EveryYear
Month
NumberOfPeriods
SystemDate
Week
Year

Input/Output and DDE Functions

This section lists the functions used to manage input and output of data and DDE links. The functions in this category are listed below:

Request: to import data from other DS Lab models.

Poke: to export data to other DS Lab models.

Execute: to execute DS Lab commands.

DDERequest: to import data from other Windows applications.

DDEPoke: to export data to other Windows applications.

DDEExecute: to make other Windows applications execute commands.

Error Functions

This group contains a single function which, because of its peculiar characteristics, cannot be included in any of the other groups: the **Error** function. This allows a Script to be interrupted at any time, returning an error value (**#ERR**). It can be used as an alarm signal to indicate an overflow or out of range value or to show that something has gone wrong (for example in the transmission of data with the DDE functions).

Control Functions

This group contains two functions used to control the execution of the simulation. The functions are:

GetStep, which returns the number of the current user defined step;

CalculateFrom, which allows a simulation to be interrupted and restarted from a different step. In a script containing a test to see whether a value satisfies a given condition, it can be used to restart the simulation if the condition is not satisfied.

System Constants

Numerical Constants
Financial Constants
Logical Constants
Environment Constants

Numerical Constants

- **PI** This constant has the value of π (3.14159...).
- **E** This constant corresponds to the value of Napiers natural number (2.7182818285...).

Financial Constants

Periods Per Year
Year and Month Length
Cash Flow Timing
Type of Interest with Fractional or Short Periods

Periods Per Year

All time periods used in the financial functions (in particular, cash flow timings and interest compounding periods) are expressed as **Periods per year**. The following constants allow you to specify the required unit. The third column contains the real annual interest rate corresponding to a nominal rate of 12%.

	Value	*Effective interest rate*
CONTINUOUS	0	.1275
ANNUAL	1	.12
SEMIANNUAL	2	.1236
QUARTERLY	4	.12551
BIMONTHLY	6	.12616
MONTHLY	12	.12683
SEMIMONTHLY	24	.12716
BIWEEKLY	26	
WEEKLY	52	
DAILY	365	.12747

BIWEEKLY and **WEEKLY** are not valid interest compounding periods. They have been included to accommodate particular cash flow situations. **CONTINUOUS** is not a valid cash flow timing period.

Notice that using these *Periods per year* may introduce a variance in the results. For example, if you are working on a problem with monthly deposits (**MONTHLY**) and daily compounding (**DAILY**), the deposits are assumed to be equally spaced, that is, all the months are taken to be the same length. Thus if $1,000 is deposited on the last day of every month starting from December 31st, with 12% annual interest per 365 day year, the exact value at the next December 31st, before the deposit, will be $12,816.74, while the result calculated by **FVFutureValue** would be $12,813.40.

Year and Month Length

In some situations, DS Lab offers the choice of two calendar options. One option dictates the standard 365-day year calendar. With the other, known as the 360/30 day convention, years have 360 days and all months have 30 days. The effect of this option is a slight increase in long term yields, little difference in short term yields, and some confusion.

While available in the **Bond** and **Short Term** functions, the 360/30 day convention cannot be used in the cash flow functions. All the **Present Value**, **Future Value**, **Advanced Present Value**and **Advanced Future Value** functions assume a year of 365 days.

Some functions in these groups have an extra argument to which is assigned one of the two constants:

- **C365** for a 365 day year;
- **C360** to use the 360/30 day convention.

Cash Flow Timing

In the analysis of cash flows, there are two possible cash flow timings:

- **ORDNRY** is at the end of the period.
- **ADUE** is at the beginning of the period.

These two options give significantly different results. For example, depositing $100 per month for 5 years at a rate of 12% compounded monthly yields $8,167 if interest is paid at the end of the month (**ORDNRY**) and $8,249 if it is paid at the beginning (**ADUE**). The difference between the two is exactly one compounding period for each cash flow.

Type of Interest with Fractional or Short Periods

Two constants allow you to define whether interest is simple or compound for fractional periods:

- **CMPND** compound interest
- **SIMPLE** simple interest

This distinction concerns the calculation of interest for less than a whole compounding period. In this case, contrary to what might be expected, simple interest gives a higher return than compound. For example, $1,000 invested for six months at 12% compounded annually will return $60 with simple interest and only $58.30 with compound interest.

This difference also appears in situations where the cash flow period is more frequent than the deposit period. In such cases, each deposit can earn significant interest before compounding. In this case, if $1000 were deposited every three months, the accumulated value after two years would be $8,861.60 with simple interest and $8,852.59 with compound interest.

In real-life applications, simple interest is more common for fractional periods, but compound interest is sometimes used to simplify calculation. If simple interest is used, the compounding period should be an integer multiple of the cash flow period.

If the compounding period is the same as or more frequent than the cash flow period, compound interest is always used. This means that for DS Labs financial functions, compound interest is used even if simple interest has been specified in the functions arguments.

Logical Constants

A logical expression returns the result True or False. There are two logical constants that can be used in a DS Lab script:

- **TRUE** value of 1
- **FALSE** value of 0

Environment Constants

- **UNIT**	represents the step unit **Unit** and has the value 0.
- **DAYUNIT**	represents the step unit **Day** and has the value 1.
- **WEEKUNIT**	represents the step unit **Week** and has the value 2.
- **MONTHUNIT**	represents the step unit **Month** and has the value 3.
- **YEARUNIT**	represents the step unit **Year** and has the value 4.
- **USERDEFINEDUNIT**	represents the step unit **User Defined** and has the value 5.

System Variables

Calculation Status Variables
Simulation Parameter Variables

Calculation Status Variables

- PERIOD/SIMSTEP

These are equivalent and represent the number of the step currently being calculated, starting from the value 1 which refers to the **First Step** of the simulation.

These variables are used to vary script execution at different steps, independently of the **Step Unit** being used. For example, to execute certain calculations at the first step only, the following script fragment may be used:

```
If SIMSTEP = 1 Then
    ...
    ...
EndIf
```

- SIMULATION

The system variable **SIMULATION** has the value 1 (**TRUE**) during simulation, and 0 (**FALSE**) during the test calculation of a script on exiting from the script editing dialog box. It is used to execute certain operations only during the simulation, and not while the model is being built.

For example, if a script is to open another model, cause it to recalculate and close it again, you will probably want to carry out these operations only during actual simulations, not every time the script containing these instructions is modified. In such a case, the script can include an **If** instruction as follows:

```
If SIMULATION
    Execute("SYSTEM"; "OPEN(MODEL1.LAB)")
    Execute("SYSTEM"; "RECALCULATE(MODEL1.LAB)")
    Execute("SYSTEM"; "CLOSE(MODEL1.LAB)")
EndIf
```

- STARTLOOP

The system variable **STARTLOOP**, together with the programming instruction **LoopTime** and the function **CalculateFrom**, enable *Goal Seeking* capabilities in DS Lab. The **STARTLOOP** variable is **TRUE** the first time a step is calculated, **FALSE** at each recalculation of the same step. The **LoopTime** instruction allows you to force recalculation of a step, while the function **CalculateFrom** allows recalculation to be started from a different step.

For further information and a detailed example of the use of the **STARTLOOP** variable.

- TIME/CURRENTSTEP

These variables are equivalent and represent the date of the current step in the internal format YYYYMMDD. There are, however, the following exceptions:

- When the Step unit is **Unit** or **User Defined**, **TIME**/**CURRENTSTEP** variables contain the number of the current step expressed as an integer.
- When the Step unit is **Year**, the last four figures (the month and day) are zero.
- When the Step unit is **Month**, the last two figures (representing the day) are zero.
- When the Step unit is **Week**, **TIME**/**CURRENTSTEP** variables assume the date of the first day of the current week.

Simulation Parameter Variables

- **FIRSTSTEP** contains the value assigned to the simulation parameter **First Step**.
- **LASTSTEP** contains the value assigned to the simulation parameter **Last Step**.
- **MULTIPLEUNIT** contains the value assigned to the simulation parameter **Multiple Unit**.
- **NUMBEROFSTEPS** contains the value assigned to the simulation parameter **N° of Steps** and represents the number of steps between the **First Step** and the **Last Step** inclusive.
- **SIMULATIONLENGTH** contains the value assigned to the simulation parameter **Simulation Length** and represents the number of steps that will be calculated in the next simulation starting from the current step.
- **SIMULATIONSTART** contains the value assigned to the simulation parameter **Simulation Start** and represents the first step to be calculated.
- **STEPUNIT** contains a number identifying the simulation parameter **Step Unit**:

  0 (the constant **UNIT**) if the Step Unit is **Unit**;

  1 (the constant **DAYUNIT**) if the Step Unit is **Day**;

  2 (the constant **WEEKUNIT**) if the Step Unit is **Week**;

  3 (the constant **MONTHUNIT**) if the Step Unit is **Month**;

  4 (the constant **YEARUNIT**) if the Step Unit is **Year**.

  5 (the constant **USERDEFINEDUNIT**) if the Step Unit is **User Defined**.

Functions and Instructions

This chapter contains a description of all the functions and instructions for DS Lab in alphabetical order.

Each entry is divided into the following subsections:

- **Description**
- **Syntax**
- **Returns**
- **Comments**
- **See Also**
- **Example**

Note: In the **Syntax** and **Example** subsections of those functions which require more than one argument, the *list separator character* is shown as a comma (,). This character is determined by the settings established in the International dialog box of the Windows Control Panel. If on exiting from the Script Editing dialog box DS Lab returns the error message Incorrect List Separator Character, either replace the comma(s) in the script with the *list separator character* in use in your copy of Windows, or use Control Panel to change the list separator character to a comma.

Important: The examples shown in this manual can be found in the examples files installed along with DS Lab.

Abs
AddDay
AddMonth
AddWeek
AddYear
AFVFutureValueIncreasingCashFlows
AFVFutureValueVariableCashFlows
AFVFutureValueVariableRates
AFVNumberOfCashFlows
AFVRateOfReturnIncreasingCashFlows
AFVRateOfReturnVariableCashFlows
AFVSizeOfCashFlows
Alpha
APVNumberOfCashFlows
APVPresentValueIncreasingCashFlows
APVPresentValueVariableCashFlows
APVPresentValueVariableRates
APVRateOfReturnIncreasingCashFlows
APVRateOfReturnVariableCashFlows
APVSizeOfCashFlows
ArcCos
ArcSin
ArcTan
Assignment
Beta
BondAccruedInterest
BondCost
BondCostEstimate
BondDuration
BondPrice
BondPriceEstimate
BondSensitivity

BondYieldToMaturity
CalculateFrom
CorrelationCoefficient
Cos
CosH
Date
Day
DayWeek
DayYear
Do Case ... EndCase
Do While ... EndDo
DDEExecute
DDEPoke
DDERequest
Error
Every
EveryDay
EveryMonth
EveryWeek
EveryYear
Execute
Exit
Exp
Fact
For... Next
FVFutureValue
FVFutureValueOfPresentValue
FVNumberOfCashFlows
FVNumberOfPeriods
FVRateOfReturn
FVSizeOfCashFlows
GetStep
If ... EndIf
Int
InventoryEconomicOrderQuantity
InventoryReorderPoint
InventorySafetyStock
InventoryServiceLevel
Log
Log10
Loop
LoopTime
Max
Mean
Min
Month
NumberOfPeriods
Poke
PVNumberOfCashFlows
PVPresentValue
PVPresentValueDeferredCashFlows
PVPresentValueOfFutureValue
PVRateOfReturn
PVSizeOfCashFlows
PVSizeOfDeferredCashFlows
Rand

Rem / Note / ' / &&
Request
Return
Round
Sin
SinH
SPEquityCallOption
SPEquityRateOfReturn
SPPortfolioAveragePeriodicReturn
SPPortfolioRateOfReturn
SPPortfolioStandardDeviation1
SPPortfolioStandardDeviation2
SPPortfolioTimeWeightedRateOfReturn
Sqrt
StandardDeviation
StandardErrorOfBeta
StandardErrorOfRegression
STEndingBalance
STNoteDiscount2
STNoteDiscount3
STNotePrice1
STNotePrice3
STNoteYield1
STNoteYield2
Sum
SumAll
SystemDate
Tan
TanH
Trend1
Trend2
Trend3
Variance
Week
WeightedAverage
Year

Abs - Function

Description

Finds the absolute value of a number.

Syntax

absolute_number=**Abs***(number)*

Returns

The number if positive; its opposite if negative.

Example

The following script:

```
Abs (-3)
```

will return the value 3.

AddDay - Function

Description

Adds a given number of days to a date in internal format.

Syntax

internal_date=**AddDay**(*internal_date, days*)

Returns

The date in internal format.

If the *internal_date* argument does not represent a valid date, the error value **#ADDDAY** is returned.

Comments

The argument *days* can be negative. In this case the days will be subtracted from the *internal_date* argument.

See Also

AddWeek, **AddMonth**, **AddYear**

Example

In a model in which the step unit is **Day**, the function call:

```
AddDay(TIME,7)
```

will add 7 days to the current date.

AddMonth - Function

Description

Adds a given number of months to a date in internal format.

Syntax

*internal_date=***AddMonth***(internal_date, months)*

Returns

The date in internal format.

If the *internal_date* argument does not represent a valid date, the error value **#ADDMONTH** is returned.

Comments

The argument *months* can be negative. In this case the months will be subtracted from the *internal_date* argument.

See Also

AddDay, **AddWeek**, **AddYear**

Example

In a model in which the step unit is **Month**, the function call:

```
AddMonth(TIME,3)
```

will add 3 months to the current date.

AddWeek - Function

Description

Adds a given number of weeks to a date in internal format.

Syntax

internal_date = **AddWeek***(internal_date, weeks)*

Returns

The date in internal format.

If the *internal_date* argument does not represent a valid date, the error value **#ADDWEEK** is returned.

Comments

The argument *weeks* can be negative. In this case the weeks will be subtracted from the *internal_date* argument.

See Also

AddDay, **AddMonth**, **AddYear**

Example

In a model in which the step unit is **Week**, the function call:

```
AddWeek(TIME,5)
```

will add 5 weeks to the current date.

AddYear - Function

Description

Adds a given number of years to a date in internal format.

Syntax

internal_date =**AddYear***(internal_date, years)*

Returns

The date in internal format.

If the *internal_date* argument does not represent a valid date, the error value **#ADDYEAR** is returned.

Comments

The argument *years* can be negative. In this case the years will be subtracted from the *internal_date* argument.

See Also

AddDay, **AddWeek**, **AddMonth**

Example

In a model in which the step unit is **Year**, the function call:

```
AddYear (TIME,2)
```

will add 2 years to the current date.

AFVFutureValueIncreasingCashFlows - Function

Description

Calculates the Future Value of a series of cash deposits made at regular intervals, with a constant interest rate and deposits increasing at each step by a given percentage.

Syntax

future_value = **AFVFutureValueIncreasingCashFlows**(
number_of_deposits,
deposit_period,
annual_interest_rate,
deposit_amount,
deposit_timing,
short_period_option,
compounding_period,
rate_of_deposit_increase)

Returns

The future value of the series of deposits.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions **(#AFV**) and one of the following numbers:

/1 number of deposits, interest rate, rate of increase or deposit amount is not valid;
/2 timing constant is not valid;
/3 short period option or deposit amount is not valid;
/4 deposit or compounding period is out of range;
/9 the iteration process gave rise to an out of range value.

Comments

This function calculates the value of a series of deposits with all interest reinvested at each interest period. Unlike the standard function **FVFutureValue**, it requires an argument indicating the rate of increase in the deposit amount, so as to represent cash flows which are index-linked or vary by a fixed percentage.

The increase in the deposits takes place annually, so that all the cash flows in a given year are the same. The rate of increase can be negative to allow representation of payments decreasing by a constant percentage.

The deposit amount after y years is given by the following formula:

$$\text{deposit}(y) = \text{deposit}(0) * (1+\text{rate_of_increase})^{y}$$

where deposit(0) is the *deposit amount* variable, representing the amount for the first year.

See Also

AFVFutureValueVariableCashFlows, **AFVFutureValueVariableRates**, **AFVNumberOfCashFlows**, **AFVRateOfReturnIncreasingCashFlows**, **AFVRateOfReturnVariableCashFlows**, **AFVSizeOfCashFlows**, **FVFutureValue**, **FVFutureValueOfPresentValue**, **FVNumberOfCashFlows**, **FVNumberOfPeriods**, **FVRateOfReturn**, **FVSizeOfCashFlows**

Example

We want to know whether it is advantageous to sign a life insurance contract which, maturing in 15 years, will yield $130,000. The payments are monthly, indexed to the annual inflation rate, and start at $200.

Let us suppose that the alternative is annual compound interest at 10% and that the average inflation rate is 6%.

```
number_of_deposits = 15 * 12
deposit_period = MONTHLY
annual_interest_rate = 0.10
deposit_amount = 200
deposit_timing = ADUE
short_period_option = SIMPLE
compounding_period = ANNUAL
rate_of_deposit_increase = 0.06
AFVFutureValueIncreasingCashFlows(
    number_of_deposits,
    deposit_period,
    annual_interest_rate,
    deposit_amount,
    deposit_timing,
    short_period_option,
    compounding_period,
    rate_of_deposit_increase)
```

The future value returned is $112,628.64, so on this basis the contract gives a better rate of return and should be signed.

AFVFutureValueVariableCashFlows - Function

Description

Calculates the Future Value of a series of deposits of varying amounts, made at regular intervals with a constant interest rate.

Syntax

future_value = **AFVFutureValueVariableCashFlows**(
deposit_period,
annual_interest_rate,
connected element containing cash flows **[***array notation***]***,*
deposit_timing,
short_period_option,
compounding_period)

The brackets after the *connected element containing cash flows* argument are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The future value of the series of deposits.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions **(#AFV)** and one of the following numbers:
/1 number of deposits or interest rate is not valid;
/2 timing constant is not valid;
/3 short period option is not valid;
/4 deposit or compounding period is out of range;
/9 the iteration process gave rise to an out of range value.

Comments

This function calculates the future value of a series of deposits with all interest reinvested at each interest period. Unlike the standard function **FVFutureValue**, this function allows a different value cash flow for each deposit period.

The cash flows are passed to the function by specifying the name of the connected element. The array notation must be in brackets.

For example, if the *connected element* is a table called *Cash Flows*, the array can be written like this:

```
Cash Flows [R1C1:R1C24]
```

See Also

AFVFutureValueIncreasingCashFlows, AFVFutureValueVariableRates, AFVNumberOfCashFlows, AFVRateOfReturnIncreasingCashFlows, AFVRateOfReturnVariableCashFlows, AFVSizeOfCashFlows, FVFutureValue, FVFutureValueOfPresentValue, FVNumberOfCashFlows, FVNumberOfPeriods, FVRateOfReturn, FVSizeOfCashFlows

Example

We want to find the amount accumulated in a bank account by depositing a variable sum every two months for two years (12 steps), given an interest rate of 8% (cash flows may be either deposits or withdrawals).

To do this, an element containing the array of twelve cash flows, in which negative values represent withdrawals and positive values deposits, must be connected to the variable containing this function.

For example, we can use the first 12 values of the series called *Cash Flows 1* containing the following values:

Step1	4000	Step7	-2600
Step2	800	Step8	1000
Step3	-1300 1200		Step9
Step4	-500	Step10	-1600
Step5	1000	Step11	-900
Step6	700	Step12	800

The script will be as follows:

```
deposit_period = BIMONTHLY
annual_interest_rate = 0.08
deposit_timing = ADUE
short_period_option = CMPND
compounding_period = SEMIANNUAL
AFVFutureValueVariableCashFlows(
    deposit_period,
    annual_interest_rate,
    Cash Flows 1 [1:12],
    deposit_timing,
    short_period_option,
    compounding_period)
```

At the end of the chosen period the bank account will contain $3,179.

AFVFutureValueVariableRates - Function

Calculates the Future Value of a series of deposits of varying amounts, made at regular intervals with a variable interest rate.

Syntax

future_value = **AFVFutureValueVariableRates**(
deposit_period,
connected element containing interest rates **[***array notation***]***,*
connected element containing cash flows **[***array notation***]***,*
deposit_timing,
compounding_period)

The brackets after the *connected element containing interest rate* and *connected element containing cash flows* arguments are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The future value of the series of deposits.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#AFV**) and one of the following numbers:

/1 number of deposits is not valid;
/2 timing constant is not valid;
/4 deposit or compounding period is out of range;
/9 the iteration process gave rise to an out of range value;
/11 interest rate is not valid.

Comments

This function calculates the future value of a series of deposits with all interest reinvested at each interest period. Unlike the standard function **FVFutureValue**, both the cash flows and the interest rate can have different values at each deposit period.

The cash flow and interest rate values are passed to the function by specifying two arrays of values contained in two connected elements (or two parts of the same connected element, e.g. a table). The array notation must be in brackets. The two arrays must correspond: for example, the third payment in the *Deposits* array earns a rate of interest given by the third value of the *Interest Rates* array.

The only limitation on variability is that interest rates can change only when a deposit period and an interest calculation period finish at the same time. For example, if the interest calculation is QUARTERLY while the deposit period is MONTHLY, the interest array will have to repeat the same rate for three deposit periods, since the payment period and the interest calculation period finish at the same time only at the end of every quarter.

For example, suppose that a table named *Deposits and Interest Rates*, containing the deposits in the first row and the corresponding interest rates in the second, is connected to the variable containing this function. In this case both the arrays are contained in the same connected element.

The *connected element containing interest rates [array notation]* argument could contain the following reference:

```
Deposits and Interest Rates [R1C1:R1C24]
```

while the *connected element containing cash flows [array notation]* argument could contain:

```
Deposits and Interest Rates [R2C1:R2C24]
```

See Also

AFVFutureValueIncreasingCashFlows, **AFVFutureValueVariableCashFlows**, **AFVNumberOfCashFlows**, **AFVRateOfReturnIncreasingCashFlows**, **AFVRateOfReturnVariableCashFlows**, **AFVSizeOfCashFlows**, **FVFutureValue**, **FVFutureValueOfPresentValue**, **FVNumberOfCashFlows**, **FVNumberOfPeriods**, **FVRateOfReturn**, **FVSizeOfCashFlows**

Example

We want to know the future value of a bank account in which various deposits and withdrawals are made and which is subject to at least two changes in the interest rate. The cash flows and corresponding interest rates remain the same for the first five years, then change for two years, and change again for the last three years. Bimonthly cash flows are expected for ten years, for a total of 60 steps.

One or two elements containing input values are connected to the variable containing this function.

For example, we can connect two series named *Cash Flows 2* and *Interest Rates* containing the following values:

$2,000 at 0.11 interest repeated for the first 30 steps;
-$1,000 at 0.085 interest repeated for the next 12 steps;
$1,500 at 0.09 interest repeated for last 18 steps.

The script will be as follows:

```
deposit_period = BIMONTHLY
deposit_timing = ADUE
compounding_period = SEMIANNUAL
AFVFutureValueVariableRates(
    deposit_period,
    interest rates [1:60],
    cash flows 2 [1:60],
    deposit_timing,
    compounding_period)
```

At the end of the period the bank account will contain $137,164.16.

AFVNumberOfCashFlows - Function

Description

Calculates the Number of Cash Flows needed to obtain a given future value by means of a series of deposits at regular intervals, with a constant interest rate and a fixed rate of increase of the deposits at each step.

Syntax

number_of_deposits = **AFVNumberOfCashFlows***(*
future_value,
deposit_period,
deposit_amount,
annual_interest_rate,
compounding_period,
deposit_timing,
short_period_option,
rate_of_deposit_increase)

Returns

The number of deposits.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#AFV**) and one of the following numbers:

/1 number of deposits, interest rate or deposit amount is not valid;
/2 timing constant is not valid;
/3 short period option is not valid;
/4 deposit or compounding period is out of range;
/9 the iteration process gave rise to an out of range value;
/10 the maximum number of iterations was exceeded without converging on a value for the number of deposits.

Comments

This function is used to answer questions such as: How many deposits do I have to make to accumulate a certain capital sum?

Unlike the standard function **FVSizeOfCashFlows**, a rate of increase in the deposit amount may be specified so as to represent cash flows which are index- linked or vary by a fixed percentage.

The increase in the deposits takes place annually, so that all the cash flows in a given year are the same.

The deposit amount after y years is given by the following formula:

deposit(y) = deposit(0) * (1+rate_of_increase) ^{y}

where deposit(0) is the *deposit amount* argument, representing the amount for the first year.

This function is iterative. The iteration process terminates when a value with an error of less than one-millionth is obtained or when more than 100 iterations have been made. In the latter case the error value **#AFV/10** is returned.

See Also

AFVFutureValueIncreasingCashFlows, **AFVFutureValueVariableCashFlows**, **AFVFutureValueVariableRates**, **AFVRateOfReturnIncreasingCashFlows**, **AFVRateOfReturnVariableCashFlows**, **AFVSizeOfCashFlows**, **FVFutureValue**, **FVFutureValueOfPresentValue**, **FVNumberOfCashFlows**, **FVNumberOfPeriods**, **FVRateOfReturn**, **FVSizeOfCashFlows**

Example

A client wants to know from an insurer how many index-linked deposits he has to make to accumulate a capital sum of $100,000, given a 10% interest rate. The first year the cash flow amount is $2,100, payable twice a year. In the following years the inflation index will increase the deposits by 6.5% annually.

```
future_value = 100000
deposit_period = SEMIANNUAL
deposit_amount = 2100
annual_interest_rate = 0.10
compounding_period = ANNUAL
deposit_timing = ORDNRY
short_period_option = SIMPLE
rate_of_deposit_increase = 0.065
AFVNumberOfCashFlows(
    future_value,
    deposit_period,
    deposit_amount,
    annual_interest_rate,
    compounding_period,
    deposit_timing,
    short_period_option,
    rate_of_deposit_increase)
```

The smallest number of deposits to exceed $100,000 is found to be 22.

AFVRateOfReturnIncreasingCashFlows - Function

Description

Calculates the Internal Rate of Return needed to obtain a given future value by means of a series of deposits at regular intervals, with a constant interest rate and a percentage increase of the deposits at each step.

Syntax

rate_of_return = **AFVRateOfReturnIncreasingCashFlows***(*
future_value,
deposit_period,
deposit_amount,
number_of_deposits,
compounding_period,
short_period_option,
deposit_timing,
rate_of_deposit_increase)

Returns

The internal rate of return of a series of deposits.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#AFV**) and one of the following numbers:

/1 number of deposits, deposit amount or rate of increase is not valid;
/2 timing constant is not valid;
/3 short period option is not valid;
/4 deposit or compounding period is out of range;
/9 the iteration process gave rise to an out of range value;
/10 the maximum number of iterations was exceeded without converging on a value for the rate of return.

Comments

This function can be used to determine the feasibility of an investment plan or to evaluate the risk involved in reaching given objectives.

As compared to the standard function **FVRateOfReturn**, it allows a fixed rate of increase in the cash flows.

The increase in the deposits takes place annually, so that all the cash flows in a given year are the same.

The deposit amount after y years is given by the formula:

$$deposit(y) = deposit(0) * (1 + rate_of_increase)^{y}$$

where deposit(0) is the deposit amount variable, representing the amount for the first year.

This function is iterative. The iteration process terminates when a value with an error of less than one-millionth is obtained or when more than 100 iterations have been performed. In the latter case the error value **#AFV/10** is returned.

See Also

AFVFutureValueIncreasingCashFlows, **AFVFutureValueVariableCashFlows**, **AFVFutureValueVariableRates**, **AFVNumberOfCashFlows**, **AFVRateOfReturnVariableCashFlows**, **AFVSizeOfCashFlows**, **FVFutureValue**, **FVFutureValueOfPresentValue**, **FVNumberOfCashFlows**, **FVNumberOfPeriods**, **FVRateOfReturn**, **FVSizeOfCashFlows**

Example

An investor wants to know what rate of return he should ask his broker to obtain in order to accumulate $300,000 in 20 years, depositing $300 monthly for the first year and increasing the deposit amount by 7% each year.

```
future_value = 300000
deposit_period = MONTHLY
deposit_amount = 300
number_of_payments = 20 * 12
compounding_period = ANNUAL
short_period_option = CMPND
deposit_timing = ADUE
rate_of_deposit_increase = 0.07
AFVRateOfReturnIncreasingCashFlows(
    future_value,
    deposit_period,
    deposit_amount,
    number_of_payments,
    compounding_period,
    short_period_option,
    deposit_timing,
    rate_of_deposit_increase)
```

The rate of return the investor should ask for is 8.10%.

AFVRateOfReturnVariableCashFlows - Function

Description

Calculates the Internal Rate of Return needed to accumulate a given future value by means of a series of variable cash flows at regular intervals.

Syntax

rate_of_return = **AFVRateOfReturnVariableCashFlows***(*
future_value,
deposit_period,
connected element containing cash flows **[***array notation***]***,*
starting_rate,
deposit_timing,
short_period_option,
compounding_period)

The brackets after the *connected element containing cash flows* argument are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The internal rate of return of a series of payments.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#AFV**) and one of the following numbers:

/1 future value is not valid;
/2 timing constant is not valid;
/3 short period option is not valid;
/4 deposit or compounding period is out of range;
/9 the iteration process gave rise to an out of range value;
/10 the maximum number of iterations was exceeded without converging on a value for the number of cash flows.

Comments

This function can be used to determine the viability of an investment plan or to evaluate the risk involved in reaching given objectives.

Unlike the standard function **FVRateOfReturn**, the cash flows can vary for each deposit period.

This function is iterative. The iteration process terminates when a value with an error of less than one-millionth is obtained or when more than 100 iterations have been performed. In the latter case the error value **#AFV/10** is returned.

To facilitate calculation of the correct rate of return, one argument contains the starting value for the iteration. If no value is specified, this argument defaults to 0.10 (10%).

The calculated rate of return is the lowest needed to obtain at least the required future value; a smaller value would not reach the given sum.

The cash flows are passed to the function by specifying the name of the connected element. The array notation must be in brackets. For example, if the connected element is a table called *Cash Flows*, the array could be written like this:

```
Cash Flows [R1C1:R1C24]
```

See Also

AFVFutureValueIncreasingCashFlows, **AFVFutureValueVariableCashFlows**, **AFVFutureValueVariableRates**, **AFVNumberOfCashFlows**, **AFVRateOfReturnIncreasingCashFlows**, **AFVSizeOfCashFlows**, **FVFutureValue**, **FVFutureValueOfPresentValue**, **FVNumberOfCashFlows**, **FVNumberOfPeriods**, **FVRateOfReturn**, **FVSizeOfCashFlows**

Example

We want to know the rate of return which will accumulate $5,000 by means of variable quarterly cash flows for a total of 12 steps.

To do this, an element containing the values of the twelve cash flows must be connected to the variable containing this function.

For example, we could use the first 12 values of a series called *Cash Flows 3* in which the following array is entered:

```
Step1     4000      Step7     -2600
Step2      800      Step8      1000
Step3    -1300      Step9      1200
Step4     -500      Step10    -1600
Step5     1000      Step11     -900
Step6      700      Step12      800
```

The script will be as follows:

```
future_value = 5000
deposit_period = QUARTERLY
starting_rate_of_return = 0.1
deposit_timing = ADUE
short_period_option = CMPND
compounding_period = ANNUAL
AFVRateOfReturnVariableCashFlows(
    future_value,
    deposit_period,
    cash flows 3 [1:12],
    starting_rate_of_return,
    deposit_timing,
    short_period_option,
    compounding_period)
```

The required rate of return is 19.16%.

AFVSizeOfCashFlows - Function

Description

Calculates the initial Size of Cash Flow needed to accumulate a specified future value by means of a series of regular deposits, given a constant interest rate and a regular fixed percentage increase in the deposit amount.

Syntax

deposit_amount = **AFVSizeOfCashFlows***(*
number_of_deposits,
deposit_period,
annual_interest_rate,
future_value,
deposit_timing,
short_period_option,
compounding_period,
rate_of_deposit_increase)

Returns

The amount of the initial deposit.

If the deposit increase is 0 (zero), this will be the value for all cash flows; otherwise the amount remains constant during each year and changes from year to year according to the following rule:

- for the first year the returned value applies;

- for all subsequent years, the amount can be calculated by the following formula, which returns the amount after y years (deposit(y)):

deposit(y) = deposit(0) * (1 + rate_of_increase) y

where deposit(0) is the returned value.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions **(#AFV**) and one of the following numbers:

/1 number of deposits, rate of increase, future value or interest rate is not valid;
/2 timing constant is not valid;
/3 short period option is not valid;
/4 deposit or compounding period is out of range;
/7 number of deposits insufficient;
/9 the iteration process gave rise to an out of range value;
/10 the maximum number of iterations was exceeded without converging on a value for the amount of the initial deposit.

Comments

This function can be used to find the weekly, monthly, etc. amount needed to accumulate a certain capital in a given time.

Unlike the standard function **FVSizeOfCashFlows**, a rate of increase in the deposit amount may be specified so as to represent cash flows which are index- linked or vary by a fixed percentage.

This function is iterative. The iteration process terminates when a value with an error of less than one-millionth is obtained or when more than 100 iterations have been performed. In the latter case the error value **#AFV/10** is returned.

See Also

AFVFutureValueIncreasingCashFlows, **AFVFutureValueVariableCashFlows**, **AFVFutureValueVariableRates**, **AFVNumberOfCashFlows**, **AFVRateOfReturnIncreasingCashFlows**, **AFVRateOfReturnVariableCashFlows**, **FVFutureValue**, **FVFutureValueOfPresentValue**, **FVNumberOfCashFlows**, **FVNumberOfPeriods**, **FVRateOfReturn**, **FVSizeOfCashFlows**

Example

To find the starting deposit amount needed to accumulate $100,000 in 10 years at an annual interest rate of 11% compounded daily, with deposits increasing by 7% each year:

```
number_of_deposits = 40
deposit_period = QUARTERLY
annual_interest_rate = 0.095
future_value = 100000
deposit_timing = ADUE
short_period_option = CMPND
compounding_period = DAILY
rate_of_deposit_increase = 0.07
AFVSizeOfCashFlows(
    number_of_deposits,
    deposit_period,
    annual_interest_rate,
    future_value,
    deposit_timing,
    short_period_option,
    compounding_period,
    rate_of_deposit_increase)
```

The initial deposit will have to be $1,129.28.

Alpha - Function

Description

Calculates the Alpha coefficient (α) of two arrays of values, that is the intercept of the regression line of the two arrays .

Syntax

alpha = **Alpha***(*
connected element containing dependent array **[***array notation***]**
connected element containing independent array **[***array notation***]***)*

The brackets after the *connected element containing array* arguments are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The alpha index.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#STAT**) and one of the following numbers:
/1 number of observations less than 3;
/9 calculation has given rise to an out of range value;
/18 division by zero.

Comments

This function constitutes the first part of *Linear Regression Analysis*, which aims at obtaining a linear relation between two arrays of values (one dependent, the other independent), given these values as the only available data. The relation obtained is of the type:

dependent value = α + β * independent value

The second part of the relation, the slope coefficient Beta (= β) is calculated by the DS Lab function **Beta**.

The **Alpha** function is a basic tool of stock market analysis. Given a securitys quotations as a dependent array and a market index representing the sector or market segment of the security as an independent array, the function calculates the Alpha index of the security (Equity Alpha).

The alpha index represents the intersection of the regression line with the y axis. In the case of a security, the security price and index values values are passed to the function by specifying two arrays of values contained in two connected elements (or two parts of the same connected element, e.g. a table). The array notation must be in brackets.

The array must contain a number of values equal to that specified in the *number of observations* argument; these values can be assigned a zero value. The two arrays must correspond; for example, the third value of the Bond array represents the quotation for the day on which the index value is represented by the third value of the Index array.

Thus the argument connected element containing dependent array [array notation] might contain:

```
Bond [1:24]
```

while the argument *connected element containing independent array [array notation]* could contain the following reference with the same number of values:

```
Index [1:24]
```

See Also

Beta, **CorrelationCoefficient**, **StandardErrorOfBeta**, **StandardErrorOfRegression**, **Trend1**, **Trend2**, **Trend3**

Example

These are the latest closing prices of a share and the values of a hypothetical index for the same period:

Share A (dep.)	Index (indep.)
311.850	37.125
312.600	37.000
309.140	35.500
307.570	35.875
310.490	36.750

We want to know the linear relation between the two arrays of values. This can be done by the following script:

```
Alpha(
   values 1[R2C2:R6C2],
   values 1[R2C3:R6C3])
```

The Alpha constant has the value 222.06189.

APVNumberOfCashFlows - Function

Description

Calculates the Number of Cash Flows occurring at regular intervals with a fixed interest rate and increasing from period to period by a fixed percentage factor, starting from a given present value.

Syntax

number_of_payments = **APVNumberOfCashFlows***(*
present_value,
payment_period,
payment_amount,
annual_interest_rate,
compounding_period,
payment_timing,
short_period_option,
payment_growth_rate)

Returns

The number of payments.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#APV**) and one of the following numbers:

/1 payment amount, present value or interest rate is not valid;
/2 timing constant is not valid;
/3 short period option is not valid;
/4 payment or compounding period is out of range;
/9 the iteration process gave rise to an out of range value;
/10 the maximum number of iterations was exceeded without converging on a value for the number of payments.

Comments

This function calculates the number of cash flows necessary to compensate in the future for a present value at a fixed interest rate (a typical example would be the number of payments needed to pay off a loan).

Unlike the standard function **PVNumberOfCashFlows**, a percentage increase in the payment amount may be specified, so as to represent cash flows which are index-linked or vary by a fixed percentage.

The percentage increase in the deposits takes place annually, so that all the cash flows in a given year are the same.

The payment amount after y years is given by the following formula:

payment(y) = payment(0) * (1+rate_of_increase) y

where payment(0) is the *payment amount* argument, representing the amount for the first year.

The number of payments returned is always a whole number. In practice this number will be the largest number of periods which can be sustained with the specified amount: another period would require greater capital (present value).

This function is iterative. The iteration process terminates when a value with an error of less than one-millionth is obtained or when more than 100 iterations have been performed. In the latter case the error value #APV/10 is returned.

See Also

APVPresentValueIncreasingCashFlows, **APVPresentValueVariableCashFlows**, **APVPresentValueVariableRates**, **APVRateOfReturnIncreasingCashFlows**, **APVRateOfReturnVariableCashFlows**, **APVSizeOfCashFlows**, **PVNumberOfCashFlows**, **PVPresentValue**, **PVPresentValueDeferredCashFlows**, **PVPresentValueOfFutureValue**, **PVSizeOfCashFlows**, **PVSizeOfDeferredCashFlows**

Example

We want to find the number of increasing withdrawals which can be made, given a sum of $250,000 invested at a rate of 9%. The withdrawals are monthly and for the first year amount to $1,800 each. In subsequent years they increase by 6% per year. The interest is compounded twice a year.

```
present_value = 250000
payment_period = MONTHLY
payment_amount = 1800
annual_interest_rate = 0.09
compounding_period = SEMIANNUAL
payment_timing = ORDNRY
short_period_option = CMPND
payment_growth_rate = 0.06
APVNumberOfCashFlows(
    present_value,
    payment_period,
    payment_amount,
    annual_interest_rate,
    compounding_period,
    payment_timing,
    short_period_option,
    payment_growth_rate)
```

The number of withdrawals which can be made is 177.

APVPresentValueIncreasingCashFlows - Function

Description

Calculates the Present Value of a series of payments occurring at fixed intervals, with a constant interest rate and payments increasing at each step by a given percentage.

Syntax

present_value = **APVPresentValueIncreasingCashFlows***(*
number_of_payments,
payment_period,
annual_interest_rate,
payment_amount,
payment_timing,
short_period_option,
compounding_period,
payment_growth_rate)

Returns

The function returns the present value of the series of payments.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#APV**) and one of the following numbers:
/1 number of payments, interest rate, rate of increase or payment amount is not valid;
/2 timing constant is not valid;
/3 short period option is not valid;
/4 payment or compounding period is out of range;
/9 the calculation gave rise to an out of range value.

Comments

This function calculates the present value of a number of cash flows which take place in the future. Unlike the standard function **PVPresentValue**, it is possible to specify a rate of increase for the series of payments, so as to represent cash flows which are index-linked or vary by a fixed percentage.

The increase in the deposits takes place annually, so that all the cash flows in a given year are the same. The rate of increase can be negative to allow representation of payments decreasing by a constant percentage.

The payment amount after y years is given by the following formula:

payment(y) = payment(0) * (1+growth_rate) y

where payment(0) is the *payment amount* argument, representing the amount for the first year.

See Also

APVNumberOfCashFlows, **APVPresentValueVariableCashFlows**, **APVPresentValueVariableRates**, **APVRateOfReturnIncreasingCashFlows**, **APVRateOfReturnVariableCashFlows**, **APVSizeOfCashFlows**, **PVNumberOfCashFlows**, **PVPresentValue**, **PVPresentValueDeferredCashFlows**, **PVPresentValueOfFutureValue**, **PVSizeOfCashFlows**, **PVSizeOfDeferredCashFlows**

Example

We want to calculate the value of a business by calculating the present value of a series of cash flows, which it is assumed will take place monthly for the next 10 years. They amount to $34,000 monthly in the first year with an annual growth rate of 7% for the following years. The interest rate used in this calculation is the prime rate, assumed to be 14%.

```
number_of_payments = 10 * 12
payment_period = MONTHLY
annual_interest_rate = 0.14
payment_amount = 34000
payment_timing = ADUE
short_period_option = CMPND
compounding_period = MONTHLY
payment_growth_rate = 0.07
APVPresentValueIncreasingCashFlows(
    number_of_payments,
    payment_period,
    annual_interest_rate,
    payment_amount,
    payment_timing,
    short_period_option,
    compounding_period,
    payment_growth_rate)
```

The present value of the business is $2,835,533.13.

APVPresentValueVariableCashFlows - Function

Description

Calculates the Present Value of a series of payments of variable amounts occurring at regular intervals with a constant interest rate.

Syntax

present_value = **APVPresentValueVariableCashFlows***(*
payment_period,
annual_interest_rate,
connected element containing cash flows **[***array notation***]***,*
payment_timing,
short_period_option,
compounding_period)

The brackets after the *connected element containing cash flows* argument are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The present value of the series of payments.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions **(#APV)** and one of the following numbers:

/1 number of payments or interest rate is not valid;
/2 timing constant is not valid;
/3 short period option is not valid;
/4 payment or compounding period is out of range;
/9 the calculation gave rise to an out of range value.

Comments

This function calculates the present value of a series of cash flows which take place in the future. Unlike the standard function **PVPresentValue**, this function allows a different value cash flow for each payment period.

The sequence of cash flows is passed to the function by specifying the name of the connected element. The array notation must be in brackets. For example, if the connected element is a table called *Cash Flows*, the array can be written like this:

```
Cash Flows [R1C1:R1C24]
```

See Also

APVNumberOfCashFlows, APVPresentValueIncreasingCashFlows, APVPresentValueVariableRates, APVRateOfReturnIncreasingCashFlows, APVRateOfReturnVariableCashFlows, APVSizeOfCashFlows, PVNumberOfCashFlows, PVPresentValue, PVPresentValueDeferredCashFlows, PVPresentValueOfFutureValue, PVSizeOfCashFlows, PVSizeOfDeferredCashFlows

Example

We wish to determine the Net Present Value (NPV) of an investment which is expected to yield 13 cash flows at 6-monthly intervals. The discount rate is assumed to be 12%.

An element containing the series of the 13 cash flows must be connected to the variable containing this function.

For example, we can use the first 13 values of a series called *Cash Flows* containing the following values:

```
Period 1       -400
Period 2       -400
Period 3       -275
Period 4        150
Period 5        150
Period 6        150
Period 7        150
Period 8        150
Period 9        150
Period 10       150
Period 11       150
Period 12       150
Period 13       300
```

The script will be as follows:

```
payment_period = QUARTERLY
annual_interest_rate = 0.12
payment_timing = ADUE
short_period_option = CMPND
compounding_period = ANNUAL
APVPresentValueVariableCashFlows(
    payment_period,
    annual_interest_rate,
    cash flows 1 [1:13],
    payment_timing,
    short_period_option,
    compounding_period)
```

The Net Present Value is $274.96.

APVPresentValueVariableRates - Function

Description

Calculates the Present Value of a series of payments of varying amounts occurring at fixed intervals with a variable interest rate.

Syntax

present_value = **APVPresentValueVariableRates***(*
payment_period,
connected element containing interest rates **[***array notation***]***,*
connected element containing cash flows **[***array notation***]***,*
payment_timing,
compounding_period)

The brackets after the *connected element containing interest rates* and *connected element containing cash flows* arguments are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The function returns the present value of the series of payments.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions **(#APV)** and one of the following numbers:

/1 number of payments is not valid;
/2 timing constant is not valid;
/3 short period option is not valid;
/4 payment or compounding period is out of range;
/9 the calculation gave rise to an out of range value;
/11 interest rate not valid.

Comments

This function calculates the present value of a series of cash flows which take place in the future. Unlike the standard function **PVPresentValue**, this function allows changes in the value of the cash flows and in the interest rate for each payment period.

The cash flow and interest rate values are passed to the function by specifying two arrays of values contained in two connected elements (or two parts of the same connected element, e.g. a table). The array notation must be in brackets. The two arrays must correspond: for example, the third payment is subject to the interest rate given by the third value of the *Interest Rates* array.

The only limitation on variability is that interest rates can change only when a deposit period and an interest calculation period finish at the same time. For example, if the interest calculation is **QUARTERLY** while the payment period is **MONTHLY**, the interest array will have to repeat the same rate for three payment periods, since the payment period and the interest calculation period finish at the same time only at the end of every quarter.

For example, let us suppose that a table named Payments and Interest Rates, containing the payments in the first row and the corresponding interest rates in the second, is connected to the function. In this case both the arrays are contained in the same connected element.

The *connected element containing cash flows [array notation]* argument could contain the following reference:

```
Payments and Interest Rates [R1C1:R1C24]
```

while the *connected element containing interest rates [array notation]* argument could contain:

```
Payments and Interest Rates [R2C1:R2C24]
```

See Also

APVNumberOfCashFlows, APVPresentValueIncreasingCashFlows, APVPresentValueVariableCashFlows, APVRateOfReturnIncreasingCashFlows, APVRateOfReturnVariableCashFlows, APVSizeOfCashFlows, PVNumberOfCashFlows, PVPresentValue, PVPresentValueDeferredCashFlows, PVPresentValueOfFutureValue, PVSizeOfCashFlows, PVSizeOfDeferredCashFlows

Example

We want to find the present value of a series of varying cash flows at a variable interest rate. An initial investment is foreseen which involves cash expenditure in the initial period; then a constant flow of income, and at the end the repayment of the initial capital. The total duration is 5 years, or 20 quarterly periods, plus the initial period, giving a total of 21 periods.

We need to use array notation with one or two connected elements connected to the variable containing this function.

For example, two series named *Cash Flows* and *Interest Rates* can be connected, containing the following values:

	CASH FLOWS	INTEREST RATES
First Period	-1,250,000	0.12
Periods 2 to 20	+25,000	0.12
Period 21	+1,500,000	0.12

The script will be as follows:

```
payment_period = QUARTERLY
payment_timing = ADUE
compounding_period = ANNUAL

APVPresentValueVariableRates(
    payment_period,
    Interest Rates[1:21],
    Cash Flows 2 [1:21],
    payment_timing,
    compounding_period)
```

The Present Value is negative, amounting to $-22,543.51.

APVRateOfReturnIncreasingCashFlows - Function

Description

Calculates the Internal Rate of Return required to obtain a cash flow which increases from period to period by a constant percentage factor, starting from a given present value.

Syntax

rate_of_return = **APVRateOfReturnIncreasingCashFlows***(*
present_value,
payment_period,
payment_amount,
number_of_payments,
compounding_period,
short_period_option,
payment_timing,
payment_growth_rate)

Returns

The function returns the internal rate of return needed to produce the specified present value from a series of payments.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions **(#APV)** and one of the following numbers:

/1 number of payments, payment amount or growth rate is not valid;
/2 timing constant is not valid;
/3 short period option is not valid;
/4 payment or compounding period is out of range;
/7 number of payments is not sufficient;
/9 the iteration process gave rise to an out of range value;
/10 the maximum number of iterations was exceeded without converging on a value for the rate of return.

Comments

This function can be used to establish the viability of projects or to evaluate the risk involved in reaching given objectives.

As compared to the standard function **PVRateOfReturn**, it allows a fixed rate of increase in the cash flows.

The increase in the payments takes place annually, so that all the cash flows in a given year are the same.

The payment amount after y years is given by the following formula:

payment(y) = payment(0) * (1+growth_rate) ^{y}

where payment(0) is the *Payment Amount* argument, representing the amount for the first year.

This function is iterative. The iteration process terminates when a value with an error of less than one-millionth is obtained or when more than 100 iterations have been performed. In the latter case the error value #APV/10 is returned.

To obtain the standard Internal Rate of Return (IRR), write the script assigning the value 0 (zero) to the *present value* argument.

The IRR is the discount rate which makes the Net Present Value (NPV) of the cash flows equal to zero. This is a widely used tool, but it must be used with caution, as it has theoretical limitations which need to be taken into account.

For example, two quite different cash flows can have the same IRR:

-100, +150 IRR = .50 (50%)
+100, -150 IRR = .50 (50%)

Alternatively, the same cash flow can have different IRRs:

-400, +2500, +2500 IRR = .25 (25%) and IRR = 4 (400%).

The function only returns values below 1 (100%).

See Also

APVNumberOfCashFlows, APVPresentValueIncreasingCashFlows, APVPresentValueVariableCashFlows, APVPresentValueVariableRates, APVRateOfReturnVariableCashFlows, APVSizeOfCashFlows, PVNumberOfCashFlows, PVPresentValue, PVPresentValueDeferredCashFlows, PVPresentValueOfFutureValue, PVSizeOfCashFlows, PVSizeOfDeferredCashFlows

Example

We want to find the internal rate of return of a series of cash flows starting from $750 in the first year, increasing each year by 5.5%. The present value of the series of payments is calculated at $100,000. The payments take place monthly for a total of 20 years.

The script will be as follows:

```
present_value = 100000
payment_period = MONTHLY
payment_amount = 750
number_of_payments = 20*12
compounding_period = DAILY
short_period_option = CMPND
payment_timing = ORDNRY
payment_growth_rate = 0.055
APVRateOfReturnIncreasingCashFlows(
    present_value,
    payment_period,
    payment_amount,
    number_of_payments,
    compounding_period,
    short_period_option,
    payment_timing,
    payment_growth_rate)
```

The internal rate of return is 11.55%.

APVRateOfReturnVariableCashFlows - Function

Description

Calculates the Internal Rate of Return required to obtain a cash flow of the specified size, starting from a given present value.

Syntax

interest_rate = **APVRateOfReturnVariableCashFlows***(*
present_value,
payment_period,
connected element containing cash flows **[***array notation***]***,*
starting_rate,
payment_timing,
short_period_option;
compounding_period)

The brackets after the *connected element containing cash flows* argument are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The function returns the internal rate of return which brings the series of payments to the specified present value.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions **(#APV)** and one of the following numbers:

/1 present value is not valid;
/2 timing constant is not valid;
/3 short period option is not valid;
/4 payment or compounding period is out of range;
/7 number of values in the series of cash flows is not sufficient;
/9 the iteration process gave rise to an out of range value;
/10 the maximum number of iterations was exceeded without converging on a value for the rate of return.

Comments

This function can be used to establish the viability of projects or to evaluate the risk involved in reaching given objectives.

Unlike the standard function **PVRateOfReturn**, the cash flows can vary for each payment period.

This function is iterative. The iteration process terminates when a value with an error of less than one-millionth is obtained or when more than 100 iterations have been performed. In the latter case the error value #APV/10 is returned.

To facilitate calculation of the correct rate of return, one argument contains the starting value for the iteration. If no value is specified, this argument defaults to 0.10 (10%).

To obtain the standard Internal Rate of Return (IRR), write the script assigning the value 0 (zero) to the *present value* argument.

The IRR is the discount rate which makes the Net Present Value (NPV) of the cash flows equal to zero.

The IRR is a widely used tool, but it must be used with caution, as it has theoretical limitations which need

to be taken into account.

For example, two quite different cash flows can have the same IRR:

-100, +150	IRR = .50 (50%)
+100, -150	IRR = .50 (50%)

Alternatively, the same cash flow can have different IRRs:

-400, +2500, +2500   IRR = .25 (25%) and IRR = 4 (400%).

The function only returns values below 1 (100%).

The sequence of cash flows must be passed to the function by specifying the name of the connected element. The array notation must be in brackets.

For example, if the connected element is a table named *Cash Flows*, the array can be written like this:

```
Cash Flows [R1C1:R1C24]
```

See Also

APVNumberOfCashFlows, APVPresentValueIncreasingCashFlows, APVPresentValueVariableCashFlows, APVPresentValueVariableRates, APVRateOfReturnIncreasingCashFlows, APVSizeOfCashFlows, PVNumberOfCashFlows, PVPresentValue, PVPresentValueDeferredCashFlows, PVPresentValueOfFutureValue, PVSizeOfCashFlows, PVSizeOfDeferredCashFlows

Example

We want to find the internal rate of return (the rate that will yield zero present value) of a cash flow which occurs monthly for a total of 13 periods. The interest rate is expected to be 12%.

An element containing the values of the 13 cash flows must be connected to the variable containing this function.

For example, we can use the first 13 values of a series called *Cash Flows 3* containing the following values:

```
Period 1      -400
Period 2      -400
Period 3      -275
Period 4       150
Period 5       150
Period 6       150
Period 7       150
Period 8       150
Period 9       150
Period 10      150
Period 11      150
Period 12      150
Period 13      200
```

The script will be as follows:

```
present_value = 0
payment_period = QUARTERLY
starting_rate = 0.10
payment_timing = ADUE
short_period_option = CMPND
compounding_period = ANNUAL
APVRateOfReturnVariableCashFlows(
    present_value,
    payment_period,
    Cash Flows 3 [1:13],
    starting_rate,
    payment_timing,
    short_period_option,
    compounding_period)
```

The rate found is 25.09%.

APVSizeOfCashFlows - Function

Description

Calculates the initial Size of Cash Flow needed to obtain a given present value by means of a series of cash flows at regular intervals, given a constant interest rate and a regular fixed percentage increase in the cash flow amount.

Syntax

size_of_first_payment = **APVSizeOfCashFlows***(*
number_of_payments,
payment_period,
annual_interest_rate,
present_value,
payment_timing,
short_period_option,
compounding_period,
rate_of_increase_in_payments)

Returns

The size of the first cash flow.

If the rate of increase in payments is 0 (zero), this will be the value of all cash flows; otherwise, the size of payments will remain constant during each year, and will vary from year to year according to the following rule:

- for the first year the returned value applies;

- for all subsequent years, the amount can be calculated by the following formula, which returns the amount after y years (payment(y)):

payment(y) = payment(0) * (1 + rate_of_increase) y

where payment(0) is the returned value.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions **(#APV)** and one of the following numbers:

/1 present value, interest rate, rate of increase or number of payments is not valid;
/2 timing constant is not valid;
/3 short period option is not valid;
/4 payment or compounding period is out of range;
/7 number of payments insufficient;
/9 the iteration process gave rise to an out of range value;
/10 the maximum number of iterations was exceeded without converging on a value for the number of deposits.

Comments

This function is used to find what annuity a client can afford, given the sum available for investment. It can also be used to calculate the payment plan on a mortgage, once the value of the mortgage and the repayment period are known. Unlike the standard **PVSizeOfCashFlows** function, a rate of increase in the deposit amount may be specified so as to represent cash flows which are index- linked or vary by a fixed percentage.

This function is iterative. The iteration process terminates when a value with an error of less than one-

millionth is obtained or when more than 100 iterations have been made. In the latter case the error value **#AFV/10** is returned.

See Also

APVNumberOfCashFlows, **APVPresentValueIncreasingCashFlows**, **APVPresentValueVariableCashFlows**, **APVPresentValueVariableRates**, **APVRateOfReturnIncreasingCashFlows**, **APVRateOfReturnVariableCashFlows**, **PVNumberOfCashFlows**, **PVPresentValue**, **PVPresentValueDeferredCashFlows**, **PVPresentValueOfFutureValue**, **PVSizeOfCashFlows**, **PVSizeOfDeferredCashFlows**

Example

We wish to find the size of the index-linked payments needed to repay a mortgage of $100,000 at an interest rate of 14.75% compounded twice yearly. The payments take place twice yearly over 10 years and increase each year by 6.5%.

```
number_of_payments = 10 * 2
payment_period = SEMIANNUAL
annual_interest_rate = 0.1475
present_value = 100000
payment_timing = ADUE
short_period_option = CMPND
compounding_period = SEMIANNUAL
rate_of_increase = 0.065
APVSizeOfCashFlows(
    number_of_payments,
    payment_period,
    annual_interest_rate,
    present_value,
    payment_timing,
    short_period_option,
    compounding_period,
    rate_of_increase)
```

Each of the payments in the first year will be $7,210.82.

ArcCos - Function

Description

Calculates the arc-cosine.

Syntax

arc-cosine = **ArcCos***(expression)*

Returns

The arc-cosine in radians.

If the argument is less than -1 or greater than 1, the function returns the error value **#ARCCOS/-1,1**.

See Also

Sin, **Tan**, **Cos**, **ArcSin**, **ArcTan**, **SinH**, **CosH**, **TanH**

Example

The following script:

```
ArcCos(0)
```

returns the value 1.5708 (radians).

ArcSin - Function

Description

Calculates the arcsine.

Syntax

arcsine = **ArcSin***(expression)*

Returns

The arcsine in radians.

If the argument is less than -1 or greater than 1, the function returns the error value **#ARCSIN/-1,1**.

See Also

Sin, **Tan**, **Cos**, **ArcCos**, **ArcTan**, **SinH**, **CosH**, **TanH**

Example

The following script:

```
ArcSin(0)
```

returns the value 0 (radians).

ArcTan - Function

Description

Calculates the arctangent.

Syntax

arctangent = **ArcTan***(expression)*

Returns

The arctangent in radians.

See Also

Sin, **Tan**, **Cos**, **ArcSin**, **ArcCos**, **SinH**, **CosH**, **TanH**

Example

The following script:

```
ArcTan(0)
```

returns the value 0 (radians).

Assignment - Instruction

Description

Assigns a value to a local variable. If it is the first time that the variable appears in the script, it is also defined.

Syntax

local_variable = expression

Returns

The assignment instruction has no return value. An assignment at the end of a script is meaningless, thus the variable is assigned the value 0.

Comments

The name of a local variable follows the normal programming language syntax. It must thus begin with an alphabetical character and contain only alphanumeric characters or the _(underscore) character. Only the first 15 characters are significant in distinguishing one local variable from another.

Example

To assign to a variable named *Total_Credit* the sum of three credits:

```
total_credit = credit1 + credit2 + credit3
```

Beta - Function

Description

Calculates the Beta coefficient (β) of two arrays of values, that is the slope of the regression line of the two array.

Syntax

beta = **Beta***(*
connected element containing dependent array **[***array notation***]***,*
connected element containing independent array **[***array notation***]***)*

The brackets after the *connected element containing array* arguments are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The beta index.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#STAT**) and one of the following numbers:

/1 number of observations less than 3;
/9 calculation has given rise to an out of range value;
/18 division by zero.

Comments

This function constitutes the second part of *Linear Regression Analysis* aimed at obtaining a linear relation between two arrays of values (one dependent, the other independent), given these values as the only available data. The relation is as follows:

dependent value = α + β * independent value

The **Beta** function is a basic tool of stock market analysis. Given a securitys quotations as a dependent array and a market index representing the sector or market segment of the security as an independent array, the function calculates the Beta index of the security (Equity Beta).

The beta index represents the slope of the regression line: it expresses the change in the security price for a unit change in the index. Beta values greater than 1 or less than -1 represent securities tending to react to factors affecting the index with proportionately greater volatility.

In the case of a security, the values for the security price and the index are passed to the function by specifying two arrays of values contained in two connected elements (or two parts of the same connected element, e.g. a table). The array notation must be in brackets. The two arrays must correspond; for example, the third value of the *Bond* array represents the quotation for the day on which the index value is represented by the third value of the *Index* array.

Thus the *connected element containing dependent array [array notation]* argument might contain the following reference:

```
Bond [1:24]
```

while the *connected element containing independent series [array notation]* argument could contain the following reference with the same number of values:

```
Index [1:24]
```

See Also

Alpha, **CorrelationCoefficient**, **StandardErrorOfRegression**, **Trend1**, **Trend2**, **Trend3**

Example

Given the latest closing values of a share and the value of a hypothetical index for the same period:

Share A(dep.)	Index (indep.)
311.850	37.125
312.600	37.000
309.140	35.500
307.570	35.875
310.490	36.750

we want to know the linear relation between the two arrays of values.

```
Beta(
    values 1[R2C2:R6C2],
    values 1[R2C3:R6C3])
```

The Beta slope coefficient is 2.42162.

BondAccruedInterest - Function

Description

Calculates the Accrued Interest of a bond with reference to a specified purchase date.

Syntax

accrued_interest = **BondAccruedInterest***(*
face_value,
coupon_rate,
coupon_payment_period,
maturity_date,
purchase_date,
calendar_option)

Returns

The accrued interest of the bond.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#BOND**) and one of the following numbers:
/4 compounding period is out of range;
/14 either the maturity or purchase date is invalid;
/15 the coupon rate is less than 0 (zero) or greater than 1 (one);
/16 the coupon payment period is more frequent than MONTHLY or less frequent than ANNUAL.

Comments

This function calculates the accrued interest of a bond with reference to a specified purchase date. The accrued interest is one of the pricing components for a bond, as can be seen from the following formula:

bond cost = price * face value + accrued interest

The *calendar option* argument must take one of the following two values:

- **C360** to use the 360/30-day convention, which stipulates a 360-day year and a 30-day month;
- **C365** for exact year and month length.

This function assumes that the maturity date is also the coupon payment date.

See Also

BondDuration, **BondPrice**, **BondCost**, **BondPriceEstimate**, **BondCostEstimate**, **BondSensitivity**, **BondYieldToMaturity**

Example

To find the accrued interest of a bond maturing on December 9, 1999 with the following characteristics:

```
face_value = 1000
coupon_rate = 0.0625
coupon_payment_period = SEMIANNUAL
```

```
maturity_date = Date("December 9 1999")
purchase_date = Date("July 4 1992")
calendar_option = C360
BondAccruedInterest(
    face_value,
    coupon_rate,
    coupon_payment_period,
    maturity_date,
    purchase_date,
    calendar_option)
```

The accrued interest amounts to $4.34.

BondCost - Function

Description

Calculates the cost of a bond with reference to specified maturity and purchase dates.

Syntax

bond_cost = **BondCost***(*
face_value,
yield_to_maturity,
compounding_period,
coupon_rate,
coupon_payment_period,
maturity_date,
purchase_date,
calendar_option)

Returns

The cost of the bond.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#BOND**) and one of the following numbers:

/1 yield is not valid;
/4 compounding period is out of range;
/14 either the maturity or purchase date is invalid;
/15 the coupon rate is less than 0 (zero) or greater than 1 (one);
/16 the coupon payment period is more frequent than MONTHLY or less frequent than ANNUAL.

Comments

This function calculates the cost of a bond with reference to a specific purchase date. The cost summarizes all the pricing components for a bond as shown in the following formula:

bond cost = price * face value + accrued interest

The *calendar option* argument must take one of the following two values:

- **C360** to use the 360/30-day convention, which stipulates a 360-day year and a 30-day month;
- **C365** for exact year and month length.

This function assumes that the maturity date is also the coupon payment date.

See Also

BondDuration, BondPrice, BondAccruedInterest, BondPriceEstimate, BondCostEstimate, BondSensitivity, BondYieldToMaturity

Example

To find the cost of a bond maturing on December 9, 1999 with the following characteristics:

```
face_value = 1000
yield_to_maturity = 0.12
```

```
compounding_period = SEMIANNUAL
coupon_rate = 0.0625
coupon_payment_period = SEMIANNUAL
maturity_date = Date("December 9 1992")
purchase_date = Date("July 4 1992")
calendar_option = C360
BondCost(
    face_value,
    yield_to_maturity,
    compounding_period,
    coupon_rate,
    coupon_payment_period,
    maturity_date,
    purchase_date,
    calendar_option)
```

The bond cost with reference to the given date is $980.46.

BondCostEstimate - Function

Description

Calculates the estimated cost of a bond with reference to a duration of a specified number of periods.

Syntax

bond_cost = **BondCostEstimate***(*
face_value,
yield_to_maturity,
compounding_period,
coupon_rate,
coupon_payment_period,
number_of_periods)

Returns

The cost of the bond.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#BOND**) and one of the following numbers:
/1 yield is not valid;
/4 compounding period is out of range;
/14 either the maturity or purchase date is invalid;
/15 the coupon rate is less than 0 (zero) or greater than 1 (one);
/16 the coupon payment period is more frequent than MONTHLY or less frequent than ANNUAL.

Comments

This function calculates an estimate of the bond cost based on commonly available information. The compounding period and coupon payment period will usually be set to SEMIANNUAL.

The margin of error of the estimate will be greater for short term bonds.

See Also

BondDuration, **BondPrice**, **BondCost**, **BondAccruedInterest**, **BondPriceEstimate**, **BondSensitivity**, **BondYieldToMaturity**

Example

To estimate the cost of a bond which matures in 1999 with the following characteristics:

```
face_value = 1000
yield_to_maturity = 0.12
compounding_period = SEMIANNUAL
coupon_rate = 0.0625
coupon_payment_period = SEMIANNUAL
number_of_periods = (1999 - 1992) * 2
BondCostEstimate(
    face_value,
    yield_to_maturity,
    compounding_period,
```

```
    coupon_rate,
    coupon_payment_period,
    number_of_periods)
```

The estimated bond cost is $732.77.

BondDuration - Function

Description

Calculates the duration of a bond.

Syntax

bond_duration = **BondDuration***(*
yield_to_maturity,
compounding_period,
coupon_rate,
coupon_payment_period,
number_of_payments)

Returns

The duration of the bond in years.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#BOND**) and one of the following numbers:

/1 yield is not valid;
/4 compounding period is out of range;
/9 impossible to price the bond with the specified arguments;
/14 number of payments is less than 1;
/15 the coupon rate is less than 0 (zero) or greater than 1 (one);
/16 the coupon payment period is more frequent than MONTHLY or less frequent than ANNUAL.

Comments

This function calculates the duration of a bond assuming the discount rate and the yield to maturity are equal (Macaulays Duration).

See Also

BondPrice, **BondCost**, **BondAccruedInterest**, **BondPriceEstimate**, **BondCostEstimate**, **BondSensitivity**, **BondYieldToMaturity**

Example

To find the duration of a bond which matures in 1999 with the following characteristics:

```
yield_to_maturity = 0.12
compounding_period = SEMIANNUAL
coupon_rate = 0.0625
coupon_payment_period = SEMIANNUAL
number_of_payments = (1999 - 1992) * 2
BondDuration(
    yield_to_maturity,
    compounding_period,
    coupon_rate,
    coupon_payment_period,
    number_of_payments)
```

The duration of the bond is 5.53 years.

BondPrice - Function

Description

Calculates the price of a bond with reference to a specified maturity and purchase date.

Syntax

bond_price = **BondPrice***(*
yield_to_maturity,
compounding_period,
coupon_rate,
coupon_payment_period,
maturity_date,
purchase_date,
calendar_option)

Returns

The price of the bond as a percentage of its face value.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#BOND**) and one of the following numbers:
/1 yield is not valid;
/4 compounding period is out of range;
/14 either maturity or purchase date is invalid;
/15 the coupon rate is less than 0 (zero) or greater than 1 (one);
/16 the coupon payment period is more frequent than MONTHLY or less frequent than ANNUAL.

Comments

This function calculates the price of a bond as a percentage of its face value. The price is only one of the pricing components for a bond, as is shown by the following formula:

bond cost = price * face value + accrued interest

The *calendar option* argument must take one of the following two values:

- **C360** to use the 360/30-day convention, which stipulates a 360-day year and a 30-day month;
- **C365** for exact year and month length.

This function assumes that the maturity date is also the coupon payment date.

See Also

BondDuration, **BondCost**, **BondAccruedInterest**, **BondPriceEstimate**, **BondCostEstimate**, **BondSensitivity**, **BondYieldToMaturity**

Example

To find the price of a bond which matures on 9 December 1999 with the following characteristics:

```
yield_to_maturity = 0.12
compounding_period = SEMIANNUAL
```

```
coupon_rate = 0.0625
coupon_payment_period = SEMIANNUAL
maturity_date = Date("December 9 1999")
purchase_date = Date("July 4 1992")
calendar_option = C360
BondPrice(
    yield_to_maturity,
    compounding_period,
    coupon_rate,
    coupon_payment_period,
    maturity_date,
    purchase_date,
    calendar_option)
```

The bond price is 72.22% of its face value.

BondPriceEstimate - Function

Description

Calculates an estimate of the price of a bond with reference to a specified number of periods.

Syntax

bondprice = **BondPriceEstimate***(*
yield_to_maturity,
compounding_period,
coupon_rate,
coupon_payment_period,
number_of_periods)

Returns

The price of the bond in standard format.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#BOND**) and one of the following numbers:
/1 yield is not valid;
/4 compounding period is out of range;
/14 either maturity or purchase date is invalid;
/15 the coupon rate is less than 0 (zero) or greater than 1 (one);
/16 the coupon payment period is more frequent than MONTHLY or less frequent than ANNUAL.

Comments

This function calculates an estimate of the price of a bond based on commonly available information. The compounding period and coupon payment period will normally be set to SEMIANNUAL.

The bond price is measured as a percentage of the face value. The inaccuracy of the estimate will be greater for short term bonds.

See Also

BondDuration, **BondPrice**, **BondCost**, **BondAccruedInterest**, **BondCostEstimate**, **BondSensitivity**, **BondYieldToMaturity**

Example

To estimate the price of a bond maturing on December 9, 1999 with the following characteristics:

```
yield_to_maturity = 0.12
compounding_period = SEMIANNUAL
coupon_rate = 0.0625
coupon_payment_period = SEMIANNUAL
number_of_periods = (1999 - 1992) * 2
BondPriceEstimate(
    yield_to_maturity,
    compounding_period,
    coupon_rate,
    coupon_payment_period,
```

```
    number_of_periods)
```

The estimated bond price is $73.28% of its face value.

BondSensitivity - Function

Description

Calculates the sensitivity of a bonds price to changes in its yield.

Syntax

bond_sensitivity = **BondSensitivity***(*
yield_to_maturity,
compounding_period,
coupon_rate,
change_in_yield,
coupon_payment_period,
maturity_date,
purchase_date,
calendar_option)

Returns

The percentage change in the price of the bond.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#BOND**) and one of the following numbers:

/1 yield is not valid or the change is greater than 1 or less than -1;
/4 compounding period is out of range;
/14 either maturity or purchase date is invalid;
/15 the coupon rate is less than 0 (zero) or greater than 1 (one);
/16 the coupon payment period is more frequent than MONTHLY or less frequent than ANNUAL.

Comments

For this function, the yield to maturity (YTM) is assumed to be the relative rate at which the bond is a competitive investment.

The function can be used for risk analysis by calculating the effect of a change in the yield to maturity on the bond price.

The *calendar option* argument must take one of the following two values:

- **C360** to use the 360/30-day convention, which stipulates a 360-day year and a 30-day month;
- **C365** for exact year and month length.

This function assumes that the maturity date is also the coupon payment date.

See Also

BondDuration, **BondPrice**, **BondCost**, **BondAccruedInterest**, **BondPriceEstimate**, **BondCostEstimate**, **BondYieldToMaturity**

Example

To find the bond sensitivity given a hypothetical increase in the yield to maturity of 1.75%:

```
yield_to_maturity = 0.12
```

```
compounding_period = SEMIANNUAL
coupon_rate = 0.0625
change_in_yield = 0.0175/0.12
coupon_payment_period = SEMIANNUAL
maturity_date = Date("June 15 2005")
purchase_date = Date("September 9 1990")
calendar_option = C360
BondSensitivity(
    yield_to_maturity,
    compounding_period,
    coupon_rate,
    change_in_yield,
    coupon_payment_period,
    maturity_date,
    purchase_date,
    calendar_option)
```

This script returns a value of -0.125: that is, if the yield to maturity changes from 11 to 12%, the bond price decreases by 12.5%.

BondYieldToMaturity - Function

Description

Calculates the Yield To Maturity (YTM) of a bond.

Syntax

yield_to_maturity = **BondYieldToMaturity***(*
coupon_rate,
price,
compounding_period,
coupon_payment_period,
maturity_date,
purchase_date,
calendar_option)

Returns

The yield to maturity of a bond.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#BOND**) and one of the following numbers:
/4 compounding period is out of range;
/9 values out of range during approximations;
/10 maximum number of approximations without reaching a yield to maturity value has been exceeded;
/14 either maturity or purchase date is invalid;
/15 the coupon rate is less than 0 (zero) or greater than 1 (one);
/16 the coupon payment period is more frequent than MONTHLY or less frequent than ANNUAL.

Comments

This function calculates the yield to maturity (YTM) of a bond.

This is the standard method for evaluating and pricing bonds, enabling comparison of a bond with other investments.

The *calendar option* argument must take one of the following two values:

- **C360** to use the 360/30-day convention, which stipulates a 360-day year and a 30-day month;
- **C365** for exact year and month length.

This function assumes that the maturity date is also the coupon payment date.

See Also

BondDuration, **BondPrice**, **BondCost**, **BondAccruedInterest**, **BondPriceEstimate**, **BondCostEstimate**, **BondSensitivity**

Example

To calculate the yield to maturity of a bond with the following characteristics:

```
coupon_rate = 0.0625
price = 98.50
```

```
compounding_period = DAILY
coupon_payment_period = SEMIANNUAL
maturity_date = Date("December 9 1999")
purchase_date = Date("July 4 1992")
calendar_option = C360
BondYieldToMaturity(
    coupon_rate,
    price,
    compounding_period,
    coupon_payment_period,
    maturity_date,
    purchase_date,
    calendar_option)
```

The yield to maturity of the bond under these conditions is 6.4%.

CalculateFrom - Function

Description

Interrupts the simulation and restarts calculation from the specified step.

Syntax

CalculateFrom *(step)*

Returns

This function has no return value since it suspends the simulation, leaving the value of the variable for the current step undefined (red traffic light).

Comments

The function **CalculateFrom** is used to suspend the simulation and restart it from a different step.

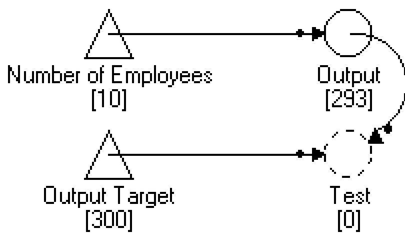
See Also

LoopTime

Example

We want to find the Number of Employees needed to produce a given annual Output. A model is built containing as Input elements *Output Target* and *Number of Employees*.

In this example, Output is calculated by a logarithmic function in the script of the *Output* variable; this variable could be replaced by a complex sub-model.



The variable called *Test* checks whether the calculated Output meets the Target. If not, the Number of Employees is incremented by 1 and the simulation is run again from the start. The script of this variable is as follows:

```
If SIMULATION AND TIME = LASTSTEP AND Output Target > Output
    ' increment number of employees and try again
    Personnel = Request("CALCFROM.LAB" , "Number of Employees")
    Poke ("CALCFROM.LAB" , "Number of Employees" , Personnel + 1)
    CalculateFrom(1)
EndIf
```

The test is executed only during simulation (SIMULATION system variable) and only when the last step is reached (TIME = LASTSTEP).

CorrelationCoefficient - Function

Description

Calculates the Correlation Coefficient of two arrays of values.

Syntax

correlation_coefficient = **CorrelationCoefficient***(*
connected element containing dependent array **[***array notation***]**
connected element containing independent array **[***array notation***]***)*

The brackets after the *connected element containing array* arguments are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The correlation coefficient.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#STAT**) and one of the following numbers:
/1 number of inputs less than 3;
/9 calculation has given rise to an out of range value;
/18 division by zero.

Comments

The correlation coefficient is used to assess the reliability of a regression. The closer this number is to +1 (direct correlation) or 1 (inverse correlation), the closer the relation between the two arrays of values is considered to be and the greater the reliability of the regression for the prediction of future values.

This coefficient can be used in stock market analysis by passing it the quotations of a security as a dependent array and a market index representing the sector or market segment of the security as an independent array.

In the case of a security, the values for the security price and the index are passed to the function by specifying two arrays of values contained in two connected elements (or two parts of the same connected element, e.g. a table). The array notation must be in brackets. The two arrays must correspond; for example, the third value of the *Bond* array represents its price on the day for which the index value is represented by the third value of the *Index* array.

Thus the *connected element containing dependent array [array notation]* argument might contain the following reference:

```
Bond [1:24]
```

while the *connected element containing independent array [array notation]* argument could contain the following reference with the same number of values:

```
Index [1:24]
```

See Also

Alpha, **Beta**, **StandardErrorOfBeta**, **StandardErrorOfRegression**

Example

These are the latest closing prices of a share and the value of a hypothetical index for the same period:

Share A (dep.)	Index (indep.)
311.850	37.125
312.600	37.000
309.140	35.500
307.570	35.875
310.490	36.750

To find the degree of correlation between the two arrays of values:

```
CorrelationCoefficient(
   values 1[R2C2:R6C2],
   values 1[R2C3:R6C3])
```

The Correlation Coefficient is 0.85947. Attempts to predict values of one array from the other using regression analysis will therefore give reasonably accurate results.

Cos - Function

Description

Calculates the cosine of an angle in radians.

Syntax

cosine = **Cos***(expression)*

Returns

The cosine of the angle.

Attempts to find the cosine of very large values may generate the error value **#PLOSS** representing a partial loss of significance, or the error value **#TLOSS** representing a total loss of significance.

See Also

Sin, **Tan**, **ArcSin**, **ArcCos**, **ArcTan**, **SinH**, **CosH**, **TanH**

Example

The following script:

```
cos (PI)
```

returns the value -1, that is the cosine of the angle that measures radians.

CosH - Function

Description

Calculates the hyperbolic cosine.

Syntax

hyperbolic_cosine = **CosH***(expression)*

Returns

The hyperbolic cosine.

See Also

Sin, **Cos**, **Tan**, **ArcSin**, **ArcCos**, **ArcTan**, **SinH**, **TanH**

Example

The following script:

```
cosH (0)
```

returns the value 1.

Date - Function

Description

Changes a date to internal format.

Syntax

internal_date = ***Date***(**"**external_date**"**)

The quotation marks inside the parentheses are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The date in internal format.

If the argument is not a valid date, the function returns the error value **#DATE**.

Comments

The year in the *external_date* argument must be written in full (1992, not 92).
The month must be written in letters, not figures (July, not 7 or 07), but may be abbreviated to as many characters as are required to identify the month unambiguously.

Within these constraints, the date will be correctly interpreted whether it is expressed in US form (*month GG YYYY*) or European form (*GG month YYYY*).

See Also

SystemDate

Example

The scripts:

```
Date("September 15 1993")
Date("15 September 1993")
```
and:
```
Date("Sep 15 1993")
```

all return the value 19930915.

Day - Function

Description

Finds the day (1-31) of a date in internal format.

Syntax

day = **Day***(internal_date)*

Returns

The number of the day.

If the *internal_date* argument does not represent a valid date, the error value **#DAY** is returned.

See Also

Week, **Month**, **Year**, **DayWeek**, **DayYear**

Example

If the simulation step is **DAY**, the system variable **TIME** contains the current day in internal format. To find out which day of the month it is, the **Day** function can be used as follows:

```
Day(TIME)
```

DayWeek - Function

Description

Finds the day of the week (0 = Sunday, ...6 = Saturday) of a date in internal format.

Syntax

day = **DayWeek***(internal_date)*

Returns

The number of the day of the week between 0 and 6 inclusive.

If the *internal_date* argument does not represent a valid date, the error value **#DAYWEEK** is returned.

See Also

Day, **Week**, **Month**, **Year**, **DayYear**

Example

If the simulation step is **Day** or **Week**, the system variable **TIME** contains the current day in internal format. To find out which day of the week it is the **DayWeek** function can be used as follows:

```
DayWeek(TIME)
```

DayYear - Function

Description

Finds the day of the year (1-366) of a date in internal format.

Syntax

day = **DayYear***(internal_date)*

Returns

The number of the day of the year.

If the *internal_date* argument does not represent a valid date, the error value **#DAYYEAR** is returned.

See Also

Day, **Week**, **Month**, **Year**, **DayWeek**

Example

If the simulation step is **Day** or **Week**, the system variable **TIME** contains the current day in internal format. To find out which day of the year it is, the **DayYear** function can be used as follows:

```
DayYear(TIME)
```

Do Case ... EndCase - Instruction

Description

The instruction **Do Case** is used to determine which of several available blocks of instructions will be executed.

Syntax

Do Case
[**case** *condition1*
statement block 1]
[**case** *condition2*
statement block 2]
[**otherwise/default**
default statement block]
EndCase

The keyword **default** is equivalent to the word **otherwise**.

Returns

The value returned by the block of instructions executed.

Example

The script:

```
Do Case
    case Month(TIME) = 1
       costs = 1000
    case Month(TIME) = 2
       costs = 2000
    Otherwise
       costs = 3000
EndCase

Return costs
```

returns the value 1000 for each January in the simulation, the value 2000 for each February and 3000 for all other months.

Do While ... EndDo - Instruction

Description

The instruction **Do While** is used to repeat a block of instructions on the basis of a condition.

Syntax

Do While *requirement*
block of instructions
EndDo

Returns

The **Do While** instruction has no return value.

Example

The script:

```
cycle = 0
max_cycles = 10
total = 0

Do While cycle < max_cycles
   total = total + 10
   cycle = cycle + 1
EndDo

Return total
```

makes the cycle repeat 10 times, thus returning the value 100.

DDEExecute - Function

Description

Causes a command to be executed by another Windows application that supports DDE.

Syntax

send_command = **DDEExecute***(application,topic,item)*

Returns

The function returns 1 if successful, 0 if unsuccessful.

Comments

The *application* argument specifies the name of the application with which to communicate; the *topic* argument, the name of what responds to the command (for example, in Excel, SYSTEM should be specified), while the *item* argument specifies the command to be executed. The application referred to must be open at the time of execution.

The syntax of *item* (in this case, the command to be executed by the other application) depends on the messages which a particular application responds to. For example, in the case of Excel it is possible to send any function of the macro language.

See Also

DDEPoke, DDERequest, Execute, Poke, Request, DDE Messages to which DS Lab Responds

Example

The following script:

```
DDEExecute("Excel","System","[NEW(1)]")
```

opens a new Excel worksheet.

DDEPoke - Function

Description

Exports data to another Windows application that supports DDE.

Syntax

Send_value = **DDEPoke***(application,topic,item,value)*

Returns

The value 0 if successful, 1 if unsuccessful.

Comments

The *application* argument specifies the name of the application with which to communicate; the *topic* argument, the name of the document or sheet (as it appears in the title bar of the window concerned, complete with the extension DOC, XLS, WKS, etc.); while the *item* argument specifies the exact place (generally the cell) to which the data is to be exported. The application and the topic (that is, the sheet or document) referred to must be open at the time of execution.

See Also

DDEExecute, **DDERequest**, **Execute**, **Poke**, **Request**, **DDE Messages to which DS Lab Responds**

Example

The script:

```
DDEPoke("Excel","Sheet1.xls","R1C1",354)
```

sends the value 354 to the first cell in the upper left-hand corner of Excel Sheet1.

DDERequest - Function

Description

Allows data to be imported from another Windows application that supports DDE.

Syntax

requested_value = **DDERequest***(application,topic,item)*

Returns

The requested value if successful, otherwise the error value **#DDEREQUEST**.

Comments

The *application* argument specifies the name of the application with which to communicate; the *topic* argument, the name of the document or sheet (as it appears in the title bar of the window concerned, complete with the extension DOC, XLS, WKS, etc.); while the *item* argument specifies the exact place (generally the cell) from which the data is to be imported. The application and the topic (that is, the sheet or document) referred to must be open at the time of execution.

See Also

DDEExecute, **DDEPoke**, **Execute**, **Poke**, **Request**, **DDE Messages to which DS Lab Responds**

Example

The script:

```
DDERequest("Excel","Sheet1.xls","R1C"&PERIOD)
```

returns the values present in the first line of Excel Sheet1; more precisely, the value contained in cell R1C1 for the first period, the value contained in cell R1C2 for the second period and so on.

Error - Function

Description

This function allows you to interrupt the Script at any moment and to assign the value **#ERR** to the variable that contains it.

Syntax

Error()

Returns

The function causes the variable to assume the value **#ERR**.

Comments

This function, unlike others, can be used in place of an instruction since it is not obligatory to use it inside an assignment or after the **Return** instruction.

In practice it is possible to use this function as a warning that a limit value has been exceeded or as a message of incorrect transmission of data for I/O and DDE functions.

Example

The following script assumes an error value if there is insufficient liquidity:

```
investment = fixed_costs + 1st_year_costs
available_funds = 3500000
If initial_investment > available_funds
   Error()
Else
   Return investment
EndIf
```

Every - Function

Description

Assumes the value 1 (TRUE) for steps separated by increment intervals, starting from a given step (start_step). Counts steps based on the CURRENTSTEP value if the Step Unit is Unit, counts steps based on the SIMSTEP if the Step Unit is anything other than Unit.

Syntax

return = **Every***(start_step, increment)*

Returns

1 (TRUE) or 0 (FALSE).

The presence of erros in the arguments will cause the function to return the error value **#EVERY**.

Comments

Used to determine steps separated by fixed intervals. It is important to understand the distinction between CURRENTSTEP (the label for the current step) and SIMSTEP (the sequence of the current) to use this function effectively.

When the parameter Step Unit is set to Unit, the function refers to the number of the Current Step (TIME or CURRENTSTEP variables), and counts starting with the function argument start_step and the given interval.

For example, when the Step Unit is Unit the script Every(1, 3) has the value 1 (TRUE) when the variable CURRENTSTEP (that is the label displayed in the simulation step window) assumes the values 1, 4, 7, 10, etc. Which step this refers to in the sequence of steps will depend on the parameter Starting step. If the parameter Starting step is 5 (CURRENTSTEP 1,4 will not show), the third step SIMSTEP 3 (CURRENTSTEP value 7), will be the first step for which the condition is true and for each third step thereafter.

When the parameter Step Unit is set to a measure other than Unit, the function refers to the period number (PERIOD or SIMSTEP variables). It cannot use CURRENTSTEP to count because this is now contains a label (a month for example) rather than a number.
For example, when the Step Unit is set to Month, the script Every(1, 3) has the value 1(TRUE) when the variable SIMSTEP assumes the values 1, 4, 7, 10, etc. That is, beginning at the first step in the sequence, regardless of which month of the year it may be, and each third step thereafter. If the parameter Starting Step is February, Every(1,3) will be true for February, May, August etc.

See Also

EveryDay, **EveryWeek**, **EveryMonth**, **EveryYear**

Example

If the simulation step is Unit, this script is true every three simulation steps from the **First Step** of the model:

```
Every(FIRSTSTEP, 3)
```

EveryDay - Function

Description

Assumes the value 1 (TRUE) for steps separated by increment intervals, starting from a given date (*start_date*).

Syntax

return = **EveryDay***(start_date, increment)*

Returns

1 (TRUE) or 0 (FALSE).

The presence of erros in the arguments will cause the function to return the error value **#EVERYDAY**.

Comments

Used to determine steps separated by fixed intervals.

See Also

Every, **EveryWeek**, **EveryMonth**, **EveryYear**

Example

If the simulation step is Day, this script is true every three days starting from the first day of the simulation:

```
EveryDay(FIRSTSTEP, 3)
```

EveryMonth - Function

Description

Assumes the value 1 (TRUE) for steps separated by increment intervals, starting from a given date (*start_date*).

Syntax

return = **EveryMonth***(start_date, increment)*

Returns

1 (TRUE) or 0 (FALSE)

The presence of erros in the arguments will cause the function to return the error value **#EVERYMONTH**.

Comments

Used to determine steps separated by fixed intervals.

See Also

Every, **EveryDay**, **EveryWeek**, **EveryYear**

Example

If the simulation step is Month, this script is true every three months from the start of the simulation:

```
EveryMonth(SIMULATIONSTART, 3)
```

EveryWeek - Function

Description

Assumes the value 1 (TRUE) for steps separated by increment intervals, starting from a given date (*start_date*).

Syntax

return = **EveryWeek***(start_date, increment)*

Returns

1 (TRUE) or 0 (FALSE)

The presence of erros in the arguments will cause the function to return the error value **#EVERYWEEK**.

Comments

Used to determine steps separated by fixed intervals.

See Also

Every, **EveryDay**, **EveryMonth**, **EveryYear**

Example

If the simulation step is Week, this script is true every three weeks from the initial week of the model:

```
EveryWeek(FIRSTSTEP, 3)
```

EveryYear - Function

Description

Assumes the value 1 (TRUE) for steps separated by increment intervals, starting from a given date (*start_date*).

Syntax

return = **EveryYear***(start_date, increment)*

Returns

1 (TRUE) or 0 (FALSE)

The presence of erros in the arguments will cause the function to return the error value **#EVERYYEAR**.

Comments

Used to determine steps separated by fixed intervals.

See Also

Every, **EveryDay**, **EveryWeek**, **EveryMonth**

Example

If the simulation step is Year, this script is true every three years from the initial year of the model:

```
EveryYear(FIRSTSTEP, 3)
```

Execute - Function

Description

Allows execution of the commands to which DS Lab responds as DDE messages.

Syntax

command = **Execute***(topic,item)*

Returns

The value 0 if successful, 1 if unsuccessful.

Comments

The keyword SYSTEM must be specified as the *topic* argument, and the DS Lab command to be executed as the *item* argument. The commands that can be executed are those to which DS Lab responds in DDE. They are as follows:

- **OPEN** (*name of model*)
- **RECALCULATE** (*name of model*)
- **CALCULATE** (*name of model*)
- **CLOSE** (*name of model, save_option*)
- **UPDATEDDELINKS** (*name of model*)

Notes: The *name of model* argument cannot specify the same model containing the **Execute** function.

With **CLOSE**, the *save_value* argument must be:

0 to close the model without saving it;

1 to save the model before closing.

See Also

DDEExecute, **DDEPoke**, **DDERequest**, **Poke**, **Request**

Example

The script:

```
Execute("SYSTEM","RECALCULATE(STOCKS.LAB)")
```

executes recalculation of the model STOCKS.LAB.

Exit - Instruction

Description

The Exit instruction is used to interrupt a Do While or a For cycle.

Syntax

Exit

Returns

The Exit instruction has no return value.

Example

The script:

```
cycle = 1
max_cycles = 10
total = 0
max_total = 80
Do While cycle = max_cycles
    total = total + 10
    If total > max_total
    Exit
    EndIf
    cycle = cycle + 1
EndDo
Return total
```

interrupts the cycle either when CYCLE is greater than MAX_CYCLES or when TOTAL is greater than MAX_TOTAL.

Exp - Function

Description

Calculates the exponential of an expression.

Syntax

exponential = **Exp***(expression)*

Returns

The value of E (Napiers number), the base of natural logarithms, raised to an expression. If the number is too large, the error value **#OVERFLOW** is returned.

See Also

Log

Example

The script:

```
Exp(3)
```

returns the value 20.08554.

Fact - Function

Description

Calculates the factorial of an expression.

Syntax

factorial = **Fact***(expression)*

Returns

The factorial of the argument.

If the number is too large, the error value **#OVERFLOW** is returned.

In the case of an argument less than zero, the error value **#FACT/<0** is returned.

If the argument is incomplete, the error value **#FACT/NOINT** is returned.

Example

The script:

```
Fact(10)
```

returns the value 3,628,800.

For... Next - Instruction

Description

The instruction **For** is used to execute a block of instructions a predetermined number of times.

Syntax

For *counter_variable* = *expression* **To** *expression* *[***Step** *expression]*
block of instructions
Next

Returns

The **For** instruction has no return value.

Comments

The keyword **Step** may be omitted, in which case the *counter variable* is raised by 1 every cycle. If the keyword **Step** is included, it must be followed by an expression giving the increase in the *counter variable* at each cycle.

Example

The script:

```
max_cycles = 10
total = 0
For cycle = 1 To max_cycles
    total = total + 10
Next
Return total
```

makes the cycle repeat 10 times, thus returning the value 100.

FVFutureValue - Function

Description

Calculates the Future Value of a series of fixed payments made at regular intervals.

Syntax

future_value = **FVFutureValue***(*
deposit_period,
number_of_deposits,
deposit_timing,
deposit_amount,
annual_interest_rate,
short_period_option,
compounding_period)

Returns

The Future Value of a series of deposits.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#FV**) and one of the following numbers:
/1 number of deposits, interest rate or deposit amount is not valid;
/2 timing constant is not valid;
/3 short period option is not valid;
/4 deposit or compounding period is out of range.

Comments

This function calculates the value of a series of deposits with all interest reinvested at each interest period.

See Also

AFVFutureValueIncreasingCashFlows, **AFVFutureValueVariableCashFlows**, **AFVFutureValueVariableRates**, **AFVNumberOfCashFlows**, **AFVRateOfReturnIncreasingCashFlows**, **AFVRateOfReturnVariableCashFlows**, **AFVSizeOfCashFlows**, **FVFutureValueOfPresentValue**, **FVNumberOfCashFlows**, **FVNumberOfPeriods**, **FVRateOfReturn**, **FVSizeOfCashFlows**

Example

A client intends to invest $350 per month for 15 years. He can get 12.5% interest, compounded semiannually.

The script would be as follows:

```
deposit_period = MONTHLY
number_of_deposits = 15 * 12
deposit_timing = ORDNRY
deposit_amount = 350
annual_interest_rate = 0.125
short_period_option = CMPND
compounding_period = SEMIANNUAL
```

```
FVFutureValue(
    deposit_period,
    number_of_deposits,
    deposit_timing,
    deposit_amount,
    annual_interest_rate,
    short_period_option,
    compounding_period)
```

The function calculates that, on the basis of the above assumptions, after 15 years the client will receive $177,978.34.

In this particular case, the choice of the **CMPND** option for the short term makes no difference.

If the client intended to make an initial payment immediately, the *deposit_timing* argument should be given the value **ADUE**.

FVFutureValueOfPresentValue - Function

Description

Calculates the Future Value of a sum of money left in deposit for a certain number of periods. This function does not deal with a series of cash flows but the return of the capital plus interest.

Syntax

future_value = **FVFutureValueOfPresentValue***(*
present_value,
annual_interest_rate,
number_of_periods,
short_period_option,
compounding_period)

Returns

The function returns the Future Value.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#FV**) and one of the following numbers:
/1 number of periods, actual value or interest rate is not valid;
/3 short period option is not valid;
/4 compounding period is out of range.

Comments

This function returns the total value of a deposit after a certain number of periods in which the interest is reinvested.

The number of periods can include decimal fractions; it always refers to the compound interest calculation period. If the **CONTINUOUS** option is chosen for the compounding period, the number of periods is in years.

See Also

AFVFutureValueIncreasingCashFlows, **AFVFutureValueVariableCashFlows**, **AFVFutureValueVariableRates**, **AFVNumberOfCashFlows**, **AFVRateOfReturnIncreasingCashFlows**, **AFVRateOfReturnVariableCashFlows**, **AFVSizeOfCashFlows**, **FVFutureValue**, **FVNumberOfCashFlows**, **FVNumberOfPeriods**, **FVRateOfReturn**, **FVSizeOfCashFlows**, **STEndingBalance**

Example

A client buys a $50,000 three-year bond without coupons, bearing compound interest at 12% annually.

```
present_value = 50000
annual_interest_rate = 0.12
number_of_periods = 3
short_period_option = CMPND
compounding_period = CONTINUOUS
FVFutureValueOfPresentValue(
    present_value,
```

```
    annual_interest_rate,
    number_of_periods,
    short_period_option,
    compounding_period)
```

At the end of three years he will receive $71,666.47.

FVNumberOfCashFlows - Function

Description

Calculates the Number of Cash Flows of a given amount at regular intervals necessary to obtain a specific future value.

Syntax

number_of_deposits = **FVNumberOfCashFlows***(*
future_value,
deposit_period,
deposit_amount,
annual_interest_rate,
compounding_period,
short_period_option,
deposit_timing)

Returns

The number of deposits.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#FV**) and one of the following numbers:
/1 number of deposits, interest rate or deposit amount is not valid;
/2 timing constant is not valid;
/3 short period option is not valid;
/4 deposit or compounding period is out of range;
/9 impossible to capitalize the given future value.

Comments

This function is used to answer questions like: How many deposits do I have to make to accumulate a certain capital?

The number of deposits returned is always a whole number. This number will be the minimum number of deposits of the given amount which will have to be made to accumulate the future value: one fewer deposit would return a future value smaller than that desired.

See Also

AFVFutureValueIncreasingCashFlows, **AFVFutureValueVariableCashFlows**, **AFVFutureValueVariableRates**, **AFVNumberOfCashFlows**, **AFVRateOfReturnIncreasingCashFlows**, **AFVRateOfReturnVariableCashFlows**, **AFVSizeOfCashFlows**, **FVFutureValue**, **FVFutureValueOfPresentValue**, **FVNumberOfPeriods**, **FVRateOfReturn**, **FVSizeOfCashFlows**

Example

A client wants to accumulate $100,000 by depositing $500 per month in a fund bearing 9.75% monthly compound interest. This script determines how many deposits are necessary.

```
future_value = 100000
deposit_period = MONTHLY
deposit_amount = 500
```

```
annual_interest_rate = 0.0975
compounding_period = MONTHLY
short_period_option = CMPND
deposit_timing = ADUE
FVNumberOfCashFlows(
    future_value,
    deposit_period,
    deposit_amount,
    annual_interest_rate,
    compounding_period,
    short_period_option,
    deposit_timing)
```

To reach $100,000, 119 deposits are needed.

FVNumberOfPeriods - Function

Description

Calculates the Number of Periods needed to obtain a specific future value, given a present value and a specific interest rate. This function does not deal with a series of cash flows but the return of the capital plus interest.

Syntax

number_of_periods = **FVNumberOfPeriods***(*
future_value,
annual_interest_rate,
present_value,
short_period_option,
compounding_period)

Returns

The number of compounding periods. This can be a decimal number.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#FV**) and one of the following numbers:
/1 future value, present value or interest rate is not valid;
/3 short period option is not valid;
/4 compounding period is out of range.

Comments

This function is used to find how long is needed to reach a fixed sum, given a known interest rate.

The number of periods returned refers to the compounding period. If the compounding period is specified as **CONTINUOUS**, the number of periods is in years.

See Also

AFVFutureValueIncreasingCashFlows, **AFVFutureValueVariableCashFlows**, **AFVFutureValueVariableRates**, **AFVNumberOfCashFlows**, **AFVRateOfReturnIncreasingCashFlows**, **AFVRateOfReturnVariableCashFlows**, **AFVSizeOfCashFlows**, **FVFutureValue**, **FVFutureValueOfPresentValue**, **FVNumberOfCashFlows**, **FVRateOfReturn**, **FVSizeOfCashFlows**

Example

To find out in how many weeks a capital sum doubles at 11% weekly compound interest:

```
present_value = 100
future_value = present_value * 2
annual_interest_rate = 0.11
short_period_option = CMPND
compounding_period = WEEKLY
FVNumberOfPeriods(
    future_value,
    annual_interest_rate,
    present_value,
```

```
    short_period_option,
    compounding_period)
```

The capital doubles in 328.92 weeks, that is a little more than 6 years.

FVRateOfReturn - Function

Description

Calculates the Internal Rate of Return needed to accumulate a specific future value, given a series of equal deposits made at fixed intervals.

Syntax

annual_interest_rate = **FVRateOfReturn***(*
future_value,
deposit_period,
deposit_amount,
number_of_payments,
compounding_period,
short_period_option,
deposit_timing)

Returns

The annual interest rate.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#FV**) and one of the following numbers:

/1 number of deposits, future value or deposit amount is not valid;
/2 timing constant is not valid;
/3 short period option is not valid;
/4 deposit or compounding period is out of range;
/7 calculation not possible with only one deposit and deposit timing option set to ORDNRY;
/8 future value smaller than total deposits;
/9 the iteration process gave rise to an out of range value;
/10 the maximum number of iterations was exceeded without converging on a value for the interest rate.

Comments

This function can be used to determine the viability of a project or to evaluate the risk factor in pursuing a certain goal.

The function is iterative. The iteration process stops when a value is found with an error of less than one millionth or when more than 100 iterations have been made. In the latter case the error value **#FV/10** to return an error value.

See Also

AFVFutureValueIncreasingCashFlows, **AFVFutureValueVariableCashFlows**, **AFVFutureValueVariableRates**, **AFVNumberOfCashFlows**, **AFVRateOfReturnIncreasingCashFlows**, **AFVRateOfReturnVariableCashFlows**, **AFVSizeOfCashFlows**, **FVFutureValue**, **FVFutureValueOfPresentValue**, **FVNumberOfCashFlows**, **FVNumberOfPeriods**, **FVSizeOfCashFlows**

Example

A client wants to know the interest rate at which he would have to invest to obtain $300,000 in 15 years

time, depositing $350 each month.

The **FVRateOfReturn** function could be used as follows:

```
future_value = 300000
deposit_period = MONTHLY
deposit_amount = 350
number_of_payments = 15 * 12
compounding_period = SEMIANNUAL
short_period_option = CMPND
deposit_timing = ORDNRY
FVRateOfReturn(
    future_value,
    deposit_period,
    deposit_amount,
    number_of_payments,
    compounding_period,
    short_period_option,
    deposit_timing)
```

The calculation returns an annual interest rate of 18.11%.

The values for the *compounding period*, *short period option* and *deposit timing* arguments were chosen at random because nothing was specified in the example.

FVSizeOfCashFlows - Function

Description

Calculates the Size of Cash Flows necessary to accumulate a specified future value, given a series of equal deposits made at fixed intervals.

Syntax

deposit_amount = **FVSizeOfCashFlows***(*
future_value,
deposit_period,
number_of_deposits,
annual_interest_rate,
compounding_period,
short_period_option,
deposit_timing)

Returns

The amount of the deposits.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#FV**) and one of the following numbers:

/1 number of deposits, interest rate or future value is not valid;
/2 timing constant is not valid;
/3 short period option is not valid;
/4 deposit or compounding period is out of range.

Comments

This function can be used to find the weekly, monthly, etc. deposit amount necessary to accumulate a certain sum after a specified period.

See Also

AFVFutureValueIncreasingCashFlows, **AFVFutureValueVariableCashFlows**, **AFVFutureValueVariableRates**, **AFVNumberOfCashFlows**, **AFVRateOfReturnIncreasingCashFlows**, **AFVRateOfReturnVariableCashFlows**, **AFVSizeOfCashFlows**, **FVFutureValue**, **FVFutureValueOfPresentValue**, **FVNumberOfCashFlows**, **FVNumberOfPeriods**, **FVRateOfReturn**

Example

A young couple want to accumulate $70,000 in 5 years. To do this they intend to use a fund which guarantees a semiannual compound interest rate of 10%. They reckon to deposit once a month.

The function can be used as follows:

```
future_value = 70000
deposit_period = MONTHLY
number_of_deposits = 5 * 12
annual_interest_rate = 0.1
compounding_period = SEMIANNUAL
short_period_option = CMPND
deposit_timing = ORDNRY
FVSizeOfCashFlows(
```

```
    future_value,
    deposit_period,
    number_of_deposits,
    annual_interest_rate,
    compounding_period,
    short_period_option,
    deposit_timing)
```

Given these values, the couple need to deposit $908.80 per month.

GetStep - Function

Description

Finds the simulation step corresponding to the named user defined step.

Syntax

step_number = **GetStep***(***"***name_of_step***"***)*

The quotation marks inside the parentheses are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The number of the step.

If the *name_of_step* argument is not the name of a user defined step in the simulation, the error value **#GETSTEP** is returned.

Comments

This function is used where a different part of the script is to be executed for different user defined steps. It eliminates the need to modify the script when additional steps are added to the model, changing the position of the existing ones.

Example

A variable called *Market Index* is to have a different value for each step in a model containing user defined steps representing three different markets:

1. Europe

2. America

3. Asia.

The script of the variable containing this function could read as follows:

```
Do Case
   Case SIMSTEP = GetStep("Europe")
      ' if step is Europe
      Local_Market_Index = 25
   Case SIMSTEP = GetStep("America")
      ' if step is America
      Local_Market_Index = 33
   Case SIMSTEP = GetStep("Asia")
      ' if step is Asia
      Local_Market_Index = 20
EndCase

Return Local_Market_Index
```

If ... EndIf - Instruction

Description

The instruction **If** determines which block of instructions will be executed on the basis of a condition.

Syntax

> **If** *condition 1 [***Then***]*
>   *instruction block 1 ...*
> *[***Else If** *condition n[***Then***]*
>   *instruction block n*
> *[***Else**
>   *instruction block else]*
> **EndIf**

Returns

The **If** instruction returns a meaningful value only if the instruction block which is executed ends with an expression. In this case the returned value is the value of the expression.

Comments

The key word **Then** is optional: it can be used or not according to the users preferred programming style.

Example

The script:

```
expenses = 0

If Month(TIME) = 12
   expenses = 1000
Else
   expenses = 500
EndIf

Return expenses
```

returns the value 1000 every December of the simulation and the value 500 in all other cases.

Int - Function

Description

Calculates the integer part of an expression.

Syntax

integer= **Int***(expression)*

Returns

The integer part of the argument.

See Also

Round

Example

The script:

```
Int(3.5)
```

returns the value 3.

InventoryEconomicOrderQuantity - Function

Description

Calculates Inventory Economic Order Quantity for an inventory item, taking account of the order cost and inventory carrying cost per unit.

Syntax

order_size = **InventoryEconomicOrderQuantity***(*
annual_sales,
order_cost,
inventory_carrying_cost_per_unit)

Returns

The Inventory Economic Order Quantity.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#INVENTORY**) and the following number:
/1 one of the arguments is less than zero

Comments

The *order cost* argument represents administration, transport and production startup costs.

The *inventory carrying cost per unit* represents the costs of keeping inventory, mainly financial costs due to the immobilization of capital, equipment costs, insurance, taxes, etc.

This function does not consider variable costs such as discounts.

See Also

InventoryServiceLevel, **InventorySafetyStock**, **InventoryReorderPoint**

Example

A manufacturer makes 6 sizes of doors. Sales of the smallest model average 560 units each year. Costs associated with setting up production for the door are $1300. Each door costs $78 to produce. Inventory carrying costs per unit are 32% of value annually.

```
annual_sales = 560
order_cost = 1300
inventory_carrying_cost_per_unit = 78 * 0.32
InventoryEconomicOrderQuantity(
    annual_sales,
    order_cost,
    inventory_carrying_cost_per_unit)
```

The Inventory Economic Order Quantity is 242 doors.

InventoryReorderPoint - Function

Description

Calculates the most efficient Inventory Reorder Point.

Syntax

inventory_reorder_point = **InventoryReorderPoint***(*
average_sales,
standard_deviation_of_sales,
risk_of_running_out_of_item)

Returns

The Inventory Reorder Point.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#INVENTORY**) and one of the following numbers:
/1 one of the arguments is out of range;
/2 the risk of running out of the item is not valid.

Comments

This function calculates the Inventory Reorder Point, that is the level at which the item should be reordered.

Average sales should be calculated for the period required for the reorder to be delivered.

Maximum demand is set at 3.5 times standard deviation above average. The risk of running out of the item is written as a decimal number: 0.10 represents the risk that stock will reach zero in 10 reorder periods out of 100.

See Also

InventoryEconomicOrderQuantity, **InventoryServiceLevel**, **InventorySafetyStock**

Example

Consider an inventory item with the following characteristics:

```
average_sales = 10.8
standard_deviation_of_sales = 2.7
risk_of_running_out_of_item = 0.05
InventoryReorderPoint(
    average_sales,
    standard_deviation_of_sales,
    risk_of_running_out_of_item)
```

The reorder point is a stock level of 15 units.

InventorySafetyStock - Function

Description

Calculates the Inventory Safety Stock.

Syntax

safety_level = **InventorySafetyStock***(*
average_sales,
standard_deviation_of_sales,
risk_of_running_out_of_item)

Returns

The Inventory Safety Stock.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#INVENTORY**) and one of the following numbers:
/1 one of the arguments is out of range;
/2 the risk of running out of the item is not valid.

Comments

This function calculates the Inventory Safety Stock, that is the stock that should be on hand when a new order is made so as not to run out of the item.

Average sales should be calculated for the period required for the reorder to be delivered.

Maximum demand is set at 3.5 times standard deviation above average. The risk of running out of the item is written as a decimal number: 0.10 represents the risk that stock will reach zero in 10 reorder periods out of 100.

See Also

InventoryEconomicOrderQuantity, **InventoryServiceLevel**, **InventoryReorderPoint**

Example

Consider an inventory item with the following characteristics:

```
average_sales = 10.8
standard_deviation_of_sales = 2.7
risk_of_running_out_of_item = 0.05
InventorySafetyStock(
    average_sales,
    standard_deviation_of_sales,
    risk_of_running_out_of_item)
```

The safety stock level will be 4 units.

InventoryServiceLevel - Function

Description

Calculates the Inventory Service Level.

Syntax

inventory_service_level = **InventoryServiceLevel***(*
order_quantity,
average_sales,
standard_deviation_of_sales,
inventory_reorder_point)

Returns

The Inventory Service Level.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#INVENTORY**) and one of the following numbers:
/1 one of the arguments is out of range;
/2 the calculated Inventory Service Level is over 100%.

Comments

This function calculates the Inventory Service Level as the percentage of orders that will be satisfied from inventory.

Average sales should be calculated for the period required for the new order to be delivered.

Maximum demand is set at 3.5 times standard deviation above average.

See Also

InventoryEconomicOrderQuantity, **InventorySafetyStock**, **InventoryReorderPoint**

Example

Consider an inventory item with the following characteristics:

```
order_size = 242
average_sales = 10.8
standard_deviation_of_sales = 2.7
inventory_reorder_point = 11
InventoryServiceLevel(
    order_size,
    average_sales,
    standard_deviation_of_sales,
    inventory_reorder_point)
```

The Inventory Service Level is 99.60%.

Log - Function

Description

Calculates the natural logarithm of an expression.

Syntax

Logarithm= **Log***(expression)*

Returns

The logarithm of the argument.

In the case of a number outside the allowed range, the error value **#OVERFLOW** will be returned.

In the case of an argument smaller than or equal to zero, the error value **#LOG/<=0** will be returned.

See Also

Exp,Log10

Example

The script:

```
Log(1)
```

returns the value 0.

Log10 - Function

Description

Calculates the base 10 logarithm of an expression.

Syntax

Logarithm= **Log10***(expression)*

Returns

The base 10 logarithm of the argument.

A number out of the allowed range will cause the function to return the error value **#OVERFLOW**.

An argument less than or equal to zero will cause the function to return the error value **#LOG10/<=0**.

See Also

Exp, **Log**

Example

The script:

```
Log10(1)
```

returns the value 0.

Loop - Instruction

Description

The **Loop** instruction returns to the test of a **Do While** or **For** cycle.

Syntax

Loop

Returns

The **Loop** instruction has no return value.

Example

The script:

```
cycle = 0
max_cycles = 10
total = 0
Do While cycle < max_cycles
    cycle = cycle + 1
    If cycle MOD 2 = 0
    Loop
    EndIf
    total = total + cycle
EndDo
Return total
```

calculates the sum of the even numbers (MOD 2 = divisible by 2 without remainder) from 1 to 10.

LoopTime - Instruction

Description

The **LoopTime** instruction causes a simulation not to advance to the next step after calculating values for all the variables, but to restart calculation from the same step.

Syntax

LoopTime

Returns

The **LoopTime** instruction has no return value.

See Also

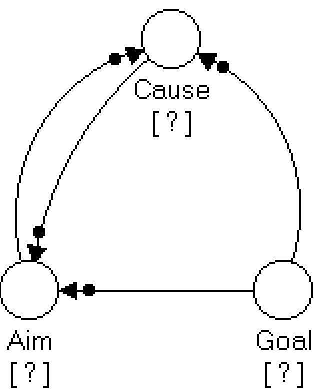
<u>**CalculateFrom**</u>

Comments

The **LoopTime** instruction, in conjunction with the **STARTLOOP** system variable, allows *Goal Seeking* capability. In practice, this is a different kind of tool from traditional Goal Seeking environments. There is no parallel environment in which a Goal Seek may be executed on an existing model: the model itself has to be built in view of the Goal Seek. This can be limiting in certain cases but is very powerful where there are sophisticated requirements.

The **LoopTime** instruction causes the <u>current step</u>   the one which is currently being calculated   to be recalculated. The **STARTLOOP** variable contains the value **TRUE** the first time it is calculated, **FALSE** all the other times the same period is recalculated. With these basic tools, sophisticated Goal Seeking algorithms can easily be created, because attention is focused on each variable, leaving to DS Lab the task of evaluating all the connections.

The technique is to organize part of a model like this:



In the variable *Aim,* a test is carried out: if its value satisfies the goal given by the *Goal* variable, the simulation proceeds to the next step without executing the **LoopTime** instruction; otherwise (that is, if *Aim* is not satisfied), **LoopTime** is executed. This means that all the variables for the <u>current step</u> will be recalculated, particularly the *Cause* variable which represents an input to the *Aim* variable (usually not

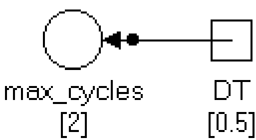
directly but through other intermediate variables, as will be illustrated in the following example). The optimization algorithm is put in the variable *Cause*, which has *Aim* and *Goals* as its inputs and will therefore determine its own value.

As a particular case of the technique described above, it is worth noting the possibility of carrying out an *intermediate step calculation* a fixed number of times for each step. This can be very useful when greater precision is needed in the calculations without saving intermediate data. For example, the step unit could be MONTH and every month be made to recalculate four times. To obtain this only requires one disconnected variable which invokes the **LoopTime** instruction.

The script of this variable could be structured like this:

```
max_cycles = 4
If STARTLOOP     REM if this is the first time
    If max_cycles > 1
    LoopTime
    EndIf
    Return 1
Else
    If (cycles[-1] + 1) < max_cycles
    LoopTime()
    EndIf
    Return cycles [-1] + 1
EndIf
```

This last example shows how to use DS Lab to execute a dynamic simulation with *Forrester methodology*, or in terms comprehensible only to the experts, how to introduce the DT.



In this case the maximum number of cycles (the local variable *max_cycles* in the preceding example) can be found on the basis of DT as follows:

```
max_cycles = 1 / DT
```

As with other languages for dynamic simulation of systems, DT must be chosen so that *max_cycles* will be an integer.

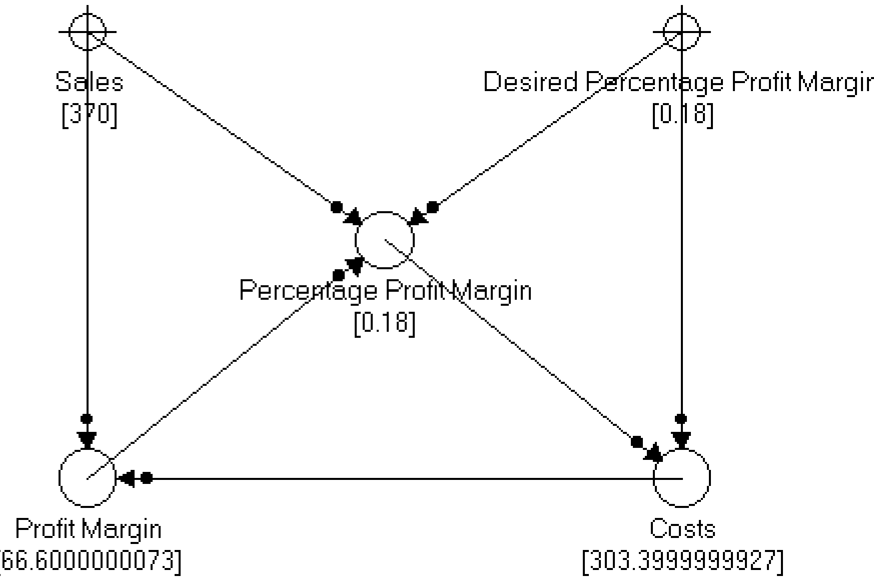
The *level equations* (a technical term used in *Dynamic Systems Analysis*) of the model must be presented in the usual way with DT.

For example, in a Material flow simulations model, the script for the *Inventory* variable will look like this:

```
Inventory + (Incoming Material - Outgoing Material) * DT
```

Example

Suppose you want to find the cost level (variable *Costs*)which, with a given *Sales* amount, yields a certain *Percentage Profit Margin*. In this case the *Desired Percentage Profit Margin* is a series containing the desired percentage in each step.



The **LoopTime** instruction is used in the variable *Percentage Profit Margin* as follows: If the percentage profit margin is different from the goal (by any chosen margin of error), the **LoopTime** instruction is executed, that is it is decided not to go to the next step but to recalculate the current step. The script of the variable *Costs* has the function of increasing or decreasing costs as required so as to obtain a profit margin, and thus a percentage, nearer to the desired goal.

Script of the variable *Percentage Profit Margin*:

```
percentage = Profit Margin  /  Sales
If (Abs(percentage - Desired Percentage Profit Margin) > 0.00000000000001)
LoopTime
EndIf
Return percentage
```

Script of the variable *Costs*:

```
If STARTLOOP
   Return  Costs [-1]
Else
   Return Costs[-1] - Costs[-1]* (Desired Percentage Profit Margin -
Percentage Profit Margin[-1])
EndIf
```

Max - Function

Description

Finds the greater of two expressions.

Syntax

greater = **Max***(expression1, expression2)*

Returns

The greater of the two expressions.

See Also

Min

Example

The script:

```
Max(100, 150)
```

returns the greater of the two values.

Mean - Function

Description

Calculates the mean of an array of numbers.

Syntax

mean = **Mean***(connected element containing array of values* **[***array notation***]***)*

The brackets after the *connected element containing array* argument are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The mean of an array of values.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#STAT**) and the following number:
/9 calculated value out of range.

Comments

This function calculates the mean of an array of values.

The values must be passed to the function by specifying the name of the connected element. The array notation must be in brackets.

Consider a hypothetical connected series called *Values*. An example for the *connected element containing array [array notation]* could be:

Values [1:24]

See Also

StandardDeviation, **Variance**

Example

To find the mean yield of the following portfolio:

	YIELDS	WEIGHTS
Bonds	0.085	0.25
Shares	0.132	0.5
Short Term Securities	0.07	0.25

```
Mean(Values 2 [R2C2 : R4C2])
```

The Mean is 0.09567.

Min - Function

Description

Finds the lesser of two expressions.

Syntax

lesser = **Min***(expression1, expression2)*

Returns

The lesser of the two expressions.

See Also

Max

Example

The script:

```
Min (100, 150)
```

returns the lesser of the two values.

Month - Function

Description

Find the month (1-12) of a given date in internal format.

Syntax

month=**Month***(internal_date)*

Returns

The number of the month.
If the *internal_date* argument does not represent a valid date, the error value **#MONTH** is returned.

See Also

Day, **Week**, **Year**, **DayWeek**, **DayYear**

Example

The script:

```
Month(TIME)
```

will return the number of the current month in the simulation.

NumberOfPeriods - Function

Description

Gives the number of periods between two dates, in the selected time units.

Syntax

N_Periods=**NumberOfPeriods***(StartingDate, FinishDate, TimeUnit)*

Returns

The number of periods.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#NUMPERIODS**) and one of the following numbers:

/1 Invalid time unit. The range is 0 to 366;

/2 Starting date later than finish date;

/4 Time unit not valid;

/5 Day or month in the dates out of range.

Example

To find the number of weeks between two dates:

```
NumberOfPeriods(
    Date("June 15 1992"),
    Date("September 15 1992"),
    WEEKLY)
```

The answer is 13.14 (equivalent to 13 weeks and one day).

Poke - Function

Description

This function is used to pass data to elements in the same or other DS Lab models. It can be used as an automatic communication channel between two or more DS Lab models.

Syntax

Send_Data=**Poke***(***"***topic***"***,* **"***item[***[***step notation***]***]***"***, value)*

The quotation marks inside the parentheses are in **bold type** to emphasize that their use is compulsory with this function. The brackets enclosing the *step notation* argument are in bold type to emphasize that they are compulsory if a step notation reference is specified.

Returns

0 if successful (the data was passed), otherwise 1.

Comments

The *topic* argument is the name of a DS Lab model (as it appears in the model window, complete with the extension .LAB); the *item* argument is the name of the element which is to receive the data. The *step notation* argument is optional. Its default value is the current step in the model receiving the data. The model to which the data is sent must be open at the time of execution.

The destination of the **Poke** function cannot be a table.

See Also

DDEExecute, **DDEPoke**, **DDERequest**, **Execute**, **Request**.

Example

The script:

```
Poke ("MODEL1.LAB","Revenues[2]", 500)
```

changes the value of the *Revenues* variable at the second step in the model MODEL1 to 500.

PVNumberOfCashFlows - Function

Description

Calculates the Number of Cash Flows of a given fixed amount obtainable from a given present value.

Syntax

number_of_payments = **PVNumberOfCashFlows***(*
present_value,
payment_period,
payment_amount,
annual_interest_rate,
compounding_period,
short_period_option,
payment_timing)

Returns

The number of payments.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#PV**) and one of the following numbers:

/1 present value, interest rate or payment amount is not valid;
/2 timing constant is not valid;
/3 short period option is not valid;
/4 payment or compounding period is out of range;
/9 present value is incompatible with this choice of arguments.

Comments

This function addresses questions such as: If I have a given amount of capital, how long will it support me at a desired level?

The returned value is always an integer, representing the maximum number of periods for which payments of the desired amount are possible. One more period would require a bigger initial deposit (present value).

See Also

APVNumberOfCashFlows, **APVPresentValueIncreasingCashFlows**, **APVPresentValueVariableCashFlows**, **APVPresentValueVariableRates**, **APVRateOfReturnIncreasingCashFlows**, **APVRateOfReturnVariableCashFlows**, **APVSizeOfCashFlows**, **PVPresentValue**, **PVPresentValueDeferredCashFlows**, **PVPresentValueOfFutureValue**, **PVRateOfReturn**, **PVSizeOfCashFlows**, **PVSizeOfDeferredCashFlows**

Example

A client wishes to make monthly withdrawals of $1000 from an investment with an initial value of $50,000 which pays interest at an annual rate of 9.5%, compounded monthly. He wants to know how many withdrawals he will be able to make.

```
current_balance = 50000
```

```
payment_period = MONTHLY
payment_amount = 1000
annual_interest_rate = 0.095
compounding_period = MONTHLY
short_period_option = CMPND
payment_timing = ORDNRY
PVNumberOfCashFlows(
    current_balance,
    payment_period,
    payment_amount,
    annual_interest_rate,
    compounding_period,
    short_period_option,
    payment_timing)
```

He will be able to make 63 withdrawals.

PVPresentValue - Function

Description

Calculates the Present Value of a series of equal payments made at regular intervals.

Syntax

present_value = **PVPresentValue***(*
compounding_period,
payment_period,
number_of_payments,
annual_interest_rate,
payment_amount,
short_period_option,
payment_timing)

Returns

The present value of the series of payments.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#PV**) and one of the following numbers:
/1 number of payments, interest rate or payment amount is not valid;
/2 timing constant is not valid;
/3 short period option is not valid;
/4 payment or compounding period is out of range.

Comments

This function calculates the present value of a series of future cash flows. It is a standard tool of financial analysis, typically used to compare different cash flows. It answers questions like: How much can a client borrow, given the level of payment he can afford and the interest and duration of loans available? a typical mortgage question or: How much must the client invest to obtain a given fixed income? (annuity).

See Also

APVNumberOfCashFlows, **APVPresentValueIncreasingCashFlows**, **APVPresentValueVariableCashFlows**, **APVPresentValueVariableRates**, **APVRateOfReturnIncreasingCashFlows**, **APVRateOfReturnVariableCashFlows**, **APVSizeOfCashFlows**, **PVNumberOfCashFlows**, **PVPresentValueDeferredCashFlows**, **PVPresentValueOfFutureValue**, **PVRateOfReturn**, **PVSizeOfCashFlows**, **PVSizeOfDeferredCashFlows**

Example

A client would like to draw $500 per month for 48 months. He can get 14% interest compounded monthly.

```
compounding_period = MONTHLY
payment_period = MONTHLY
number_of_payments = 48
annual_interest_rate = 0.14
payment_amount = 500
short_period_option = CMPND
```

```
payment_timing = ORDNRY
PVPresentValue(
    compounding_period,
    payment_period,
    number_of_payments,
    annual_interest_rate,
    payment_amount,
    short_period_option,
    payment_timing)
```

He would have to invest $18,297.27.

PVPresentValueDeferredCashFlows - Function

Description

Calculates the net Present Value of a series of payments of a fixed amount starting after a certain number of periods.

Syntax

present_value = **PVPresentValueDeferredCashFlows***(*
annual_interest_rate,
number_of_payments,
payment_period,
number_of_deferment_periods,
payment_amount,
short_period_option,
compounding_period)

Returns

The present value of the deferred cash flows.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#PV**) and one of the following numbers:
/1 number of payments, number of deferment periods, interest rate or payment amount not valid;
/3 short period option is not valid;
/4 payment period or compounding period out of range.

Comments

This function calculates the present value of a series of cash flows starting at a given number of periods from now. It is more flexible than the standard function **PVPresentValue** as it can deal with cash flows starting at a future time.

The period of deferment is calculated in the same units as the payment period. Thus if payments will be **MONTHLY**, the time until payments start must also be expressed in months.

There is no *payment_timing* argument: payments are assumed to be at the start of each period. If this is not the case, adding a period to the number of deferment periods will give the correct answer.

See Also

APVNumberOfCashFlows, **APVPresentValueIncreasingCashFlows**, **APVPresentValueVariableCashFlows**, **APVPresentValueVariableRates**, **APVRateOfReturnIncreasingCashFlows**, **APVRateOfReturnVariableCashFlows**, **APVSizeOfCashFlows**, **PVNumberOfCashFlows**, **PVPresentValue**, **PVPresentValueOfFutureValue**, **PVRateOfReturn**, **PVSizeOfCashFlows**, **PVSizeOfDeferredCashFlows**

Example

We want to know the present value of a $2,700 monthly cash flow starting in 8 months time and lasting for two years. The interest rate is 12.75% compounded monthly.

```
annual_interest_rate = 0.1275
```

```
number_of_payments = 24
payment_period = MONTHLY
number_of_deferment_periods = 8
payment_amount = 2700
short_period_option = CMPND
compounding_period = MONTHLY
PVPresentValueDeferredCashFlows(
    annual_interest_rate,
    number_of_payments,
    payment_period,
    number_of_deferment_periods,
    payment_amount,
    short_period_option,
    compounding_period)
```

The present value is $52,872.63.

PVPresentValueOfFutureValue - Function

Description

Calculates the Present Value of a lump sum payment due after a given number of periods. This function does not deal with cash flows but with repayment at maturity of the investment.

Syntax

present_value = **PVPresentValueOfFutureValue***(*
future_value,
annual_interest_rate,
number_of_periods_before_payment,
short_period_option,
compounding_period)

Returns

The present value of the future lump sum.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#PV**) and one of the following numbers:
/1 number of periods, future value or interest rate is not valid;
/3 short period option is not valid;
/4 compounding period is out of range.

Comments

This function can be used to calculate the present value of an investment maturing in the future or to find what deposit is required now to achieve a desired amount later under given conditions.

The *number of periods* may include a decimal fraction. It must however be expressed in the same units as the interest compounding period. If compounding is **CONTINUOUS**, it is assumed to be expressed in years.

See Also

APVNumberOfCashFlows, **APVPresentValueIncreasingCashFlows**, **APVPresentValueVariableCashFlows**, **APVPresentValueVariableRates**, **APVRateOfReturnIncreasingCashFlows**, **APVRateOfReturnVariableCashFlows**, **APVSizeOfCashFlows**, **PVNumberOfCashFlows**, **PVPresentValue**, **PVPresentValueDeferredCashFlows**, **PVRateOfReturn**, **PVSizeOfCashFlows**, **PVSizeOfDeferredCashFlows**

Example

To calculate the present value of a $2000 note at 13.5% compounded daily, payable in 60 days time:

```
future_value = 2000
annual_interest_rate = 0.135
number_of_periods_before_payment = 60
short_period_option = CMPND
compounding_period = DAILY
PVPresentValueOfFutureValue(
   future_value,
```

```
    annual_interest_rate,
    number_of_periods_before_payment,
    short_period_option,
    compounding_period)
```

The present value is $1,956.11.

PVRateOfReturn - Function

Description

Calculates the Internal Rate of Return required to provide a fixed cash flow of a given size from a specified initial amount.

Syntax

rate_of_return = **PVRateOfReturn***(*
present_value,
payment_period,
payment_amount,
number_of_payments,
compounding_period,
short_period_option,
payment_timing)

Returns

The required rate of return.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#PV**) and one of the following numbers:

/1 number of payments, present value or payment amount is not valid;
/2 timing constant is not valid;
/3 short period option is not valid;
/4 payment or compounding period out of range;
/7 calculation is not possible with a single payment and ADUE payment timing;
/8 future value is less than sum of payments;
/9 the iteration process gave rise to an out of range value;
/10 the maximum number of iterations was exceeded without converging on a value for the rate of return.

Comments

This function may be used in determining the viability of an investment plan or in calculating the risks associated with new ventures.

The function is iterative. It terminates after the 6th decimal fraction (millionths) of the value or at the 100th loop (in which case it returns the error value **#PV/10**).

See Also

APVNumberOfCashFlows, **APVPresentValueIncreasingCashFlows**, **APVPresentValueVariableCashFlows**, **APVPresentValueVariableRates**, **APVRateOfReturnIncreasingCashFlows**, **APVRateOfReturnVariableCashFlows**, **APVSizeOfCashFlows**, **PVNumberOfCashFlows**, **PVPresentValue**, **PVPresentValueDeferredCashFlows**, **PVPresentValueOfFutureValue**, **PVSizeOfCashFlows**, **PVSizeOfDeferredCashFlows**

Example

A client has a $40,000 mortgage that he is repaying in 30 semiannual payments of $3,200. He wants to

know the real interest rate.

```
present_value = 40000
payment_period = SEMIANNUAL
payment_amount = 3200
number_of_payments = 30
compounding_period = SEMIANNUAL
short_period_option = CMPND
payment_timing = ORDNRY
PVRateOfReturn(
    present_value,
    payment_period,
    payment_amount,
    number_of_payments,
    compounding_period,
    short_period_option,
    payment_timing)
```

The interest rate is 13.85%.

PVSizeOfCashFlows - Function

Description

Calculates the Size of Cash Flows of fixed size, made at regular intervals, corresponding to a given present value.

Syntax

payment_amount = **PVSizeOfCashFlows***(*
compounding_period,
payment_period,
number_of_payments,
annual_interest_rate,
present_value,
short_period_option,
payment_timing)

Returns

The size of payment required.

Any errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#PV**) and one of these numbers:

/1 present value, interest rate or number of payments is not valid;

/2 timing constant is not valid;

/3 short period option is not valid;

/4 payment or compounding period is out of range.

Comments

This function is used to find what annuity a client can afford, given the sum available to invest.

It also answers the common question: What is the combined principal and interest payment on a mortgage or loan of a certain amount, given the length of time available for repayment?

See Also

APVNumberOfCashFlows, **APVPresentValueIncreasingCashFlows**, **APVPresentValueVariableCashFlows**, **APVPresentValueVariableRates**, **APVRateOfReturnIncreasingCashFlows**, **APVRateOfReturnVariableCashFlows**, **APVSizeOfCashFlows**, **PVNumberOfCashFlows**, **PVPresentValue**, **PVPresentValueDeferredCashFlows**, **PVPresentValueOfFutureValue**, **PVRateOfReturn**, **PVSizeOfDeferredCashFlows**

Example

A client needs a $100,000 mortgage. He wants to repay it over 15 years at 14%. What is his monthly payment?

```
compounding_period = MONTHLY
payment_period = MONTHLY
number_of_payments = 15 * 12
```

```
annual_interest_rate = 0.14
present_value = 100000
short_period_option = CMPND
payment_timing = ORDNRY
PVSizeOfCashFlows(
    compounding_period,
    payment_period,
    number_of_payments,
    annual_interest_rate,
    present_value,
    short_period_option,
    payment_timing)
```

He will pay $1,331.74 monthly.

PVSizeOfDeferredCashFlows - Function

Description

Calculates the Size of Cash Flows of fixed size, made at regular intervals, corresponding to a given present value, starting after a given number of periods.

Syntax

payment_amount = **PVSizeOfDeferredCashFlows***(*
annual_interest_rate,
number_of_payments,
payment_period,
number_of_deferment_periods,
present_value,
short_period_option,
compounding_period)

Returns

The size of the payments.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#PV**) and one of the following numbers:

/1 number of payments, number of deferment periods, present value or interest rate is not valid;
/3 short period option is not valid;
/4 payment or compounding period out of range;

Comments

Given the sum available to invest, this function will calculate the amount obtainable as a deferred annuity.

It can be used in any case involving constant cash flows, as it is possible to establish the interval between now and when payments are expected to start (which may be set to zero).

The period of deferment is calculated in the same units as the payment period. Thus if payments will be **MONTHLY**, the time until payments start must also be expressed in months.

There is no *payment_timing* argument: payments are assumed to be at the start of each period. If this is not the case, adding a period to the number of deferment periods will give the correct result.

See Also

APVNumberOfCashFlows, APVPresentValueIncreasingCashFlows, APVPresentValueVariableCashFlows, APVPresentValueVariableRates, APVRateOfReturnIncreasingCashFlows, APVRateOfReturnVariableCashFlows, APVSizeOfCashFlows, PVNumberOfCashFlows, PVPresentValue, PVPresentValueDeferredCashFlows, PVPresentValueOfFutureValue, PVRateOfReturn, PVSizeOfCashFlows

Example

We want to know the amount of the monthly annuity payments that can be derived from the investment of $50,000 at 9% interest. Payments are to begin in 4 years time and last for 20 years.

```
interest_rate = 0.09
number_of_payments = 20 * 12
payment_period = MONTHLY
deferment_periods = 4 * 12
present_value = 50000
short_period_option = CMPND
compounding_period = MONTHLY
PVSizeOfDeferredCashFlows(
    interest_rate,
    number_of_payments,
    payment_period,
    deferment_periods,
    present_value,
    short_period_option,
    compounding_period)
```

We can expect monthly payments of $639.14.

Rand - Function

Description

Returns a random number between 0 and 1.

Syntax

number = ***Rand()***

Returns

The random number.

Example

To obtain a number between 0 and n, multiply n by the number returned by Rand():

```
number = Rand()
Return number * n
```

while to obtain a number between a and b with a < b, we can use the formula:

```
a + ( b - a ) * Rand()
```

Rem / Note / ' / && - Instruction

Description

To insert comments on a line of the script. The line will be considered as a comment up to the end of the line.

Syntax

Rem *comment line*
or
Note *comment line*
or
' *comment line*
or
&& *comment line*

Returns

No return value.

Comments

The four comment versions are equivalent. They have been included to allow use of the programming style preferred.

Example

```
If value > maximum
   Return 0    Rem finishes if too large
EndIf
```

Request - Function

Description

Imports data from an element in the same or another DS Lab model, allowing automatic linking between models.

Syntax

Requested_value=**Request** *(***"***topic***"***,***"***item***[***step notation***]"***)*

The quotation marks inside the parentheses are in **bold type** to emphasize that their use is compulsory with this function. The brackets enclosing the *step notation* argument are in **bold type** to emphasize that they are compulsory if a step notation reference is specified.

Returns

The value requested if the data transmission was successful, otherwise the error value **#REQUEST**.

Comments

The *topic* argument is the name of a DS Lab model (as it appears in the model window, complete with the extension .LAB); the *item* argument is the name of the element from which the data is to be taken. The *step notation* argument is optional. Its default value is the current step in the model receiving the data. The model from which data is requested must be open at the time of execution.

The **Request** function cannot be used with tables.

See Also

DDEExecute, **DDEPoke**, **DDERequest**, **Execute**, **Poke**

Example

The script:

```
Request ("Model1.lab","Income[2]")
```

returns the value of the *Income* variable in the model MODEL1 at the second step.

Return - Instruction

Description

Interrupts the script and returns the value of the specified expression.

Syntax

Return *expression*

Returns

The value of the specified expression.

Comments

The instructions **If ... Else**, **Do Case ... EndCase**, **Do While ... EndDo** and **For ... Next** must always use the instruction **Return** to finish the script.

Example

```
Return 5+7
```

Round - Function

Description

Rounds an expression to the nearest integer.

Syntax

integer=**Round***(expression)*

Returns

The integer closest to the argument. If the decimal part of the argument is exactly .5, it is rounded up.

See Also

Int

Example

The script:

```
Round (3.5)
```

returns the value 4.

Sin - Function

Description

Calculates the sine of an angle in radians.

Syntax

sine=**Sin***(expression)*

Returns

The sine of the angle.

Attempts to find the sine of very large values may generate the error values **#PLOSS,** representing a partial loss of significance, or **#TLOSS,** representing a total loss of significance.

See Also

Tan, **Cos**, **ArcSin**, **ArcCos**, **ArcTan**, **SinH**, **CosH**, **TanH**

Example

The script:

```
Sin (PI / 4)
```

returns the value 0.70711, that is the sine of the angle measuring radians.

SinH - Function

Description

Calculates the hyperbolic sine.

Syntax

hyperbolic_sine=**SinH***(expression)*

Returns

The hyperbolic sine.

See Also

Tan, **Cos**, **ArcSin**, **ArcCos**, **ArcTan**, **Sin**, **CosH**, **TanH**

Example

The script:

```
SinH(PI)
```

returns the value 11.54874.

SPEquityCallOption - Function

Description

Calculates the price of an Equity Call Option.

Syntax

call_option = **SPEquityCallOption***(*
current_share_price,
call_exercise_price,
time_to_expiration,
internal_rate_of_return,
variance_of_stock_price)

Returns

The call option value.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#SP**) and one of the following numbers:
/1 current price or call price not valid;
/2 internal rate of return not valid;
/18 division by zero in calculation.

Comments

This function uses the Black-Scholes model (F. Black and M. Scholes, *The Pricing of Options and Corporate Liabilities*, Journal of Political Economy, 1973.) It makes several assumptions:

- No taxes or transaction costs;
- No dividends paid on the stock;
- No short sales restrictions;
- Riskless borrowing and lending available at the discount rate;
- Instantaneous portfolio adjustment;
- Option exercised at maturity.

Returns on treasury bills are typically used for the risk free return estimate.
Time to expiration, variance and rate of return must all refer to the same time period.

See Also

SPEquityRateOfReturn

Example

The current price of a security is $62.00. Historically, the standard deviation of returns for a six month holding period has been 0.26. The strike price of the call is $66.00, expiring in 45 days. The yield on six monthly treasury bills is 6.25%.

```
current_price = 62
```

```
call_price = 66
time_to_expiry = 45 / (6*30)
internal_rate_of_return = 0.0625/2
price_variance = 0.262
SPEquityCallOption(
    current_price,
    call_price,
    time_to_expiry,
    internal_rate_of_return,
    price_variance)
```

The theoretical value of the option is $4.89.

SPEquityRateOfReturn - Function

Description

Calculates the Internal Rate of Return of an equity, based on dividend projections.

Syntax

rate_of_return = **SPEquityRateOfReturn***(*
latest_year_dividend,
current_stock_price,
years_in_projection,
annual_dividend_growth)

Returns

The Equity Internal Rate of Return.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#SP**) and one of the following numbers:
/1 current price or percentage dividend growth is not valid;
/14 number of years is less than 1.

Comments

This is a typical technique for pricing shares. The function simulates a quarterly dividend cash flow. The rate of return calculated is calculated on the basis of this cash flow and the shares current market price. Transaction costs are not included. The dividend remains constant for four quarters and then increases by the growth factor specified.

See Also

SPEquityCallOption

Example

We want to calculate the internal rate of return of a share currently priced at $62.00. The last dividend paid was $3.50 and a growth of 22% per year is expected.

```
latest_year_dividend = 3.50
current_stock_price = 62.00
years_in_projection = 3
annual_dividend_growth = 0.22
SPEquityRateOfReturn(
    latest_year_dividend,
    current_stock_price,
    years_in_projection,
    annual_dividend_growth)
```

The Internal Rate of Return on the stock under these conditions is 19.22%.

SPPortfolioAveragePeriodicReturn - Function

Description

Calculates the Portfolio Average Periodic Return from a series of cash flows and periodic valuations of the portfolio.

Syntax

average_return = **SPPortfolioAveragePeriodicReturn***(*
valuation_reference_period,
connected element containing cash flows **[***array notation***]***,*
connected element containing periodic valuations **[***array notation***]***)*

The brackets after the *connected element containing array* arguments are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The portfolios Average Periodic Return.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#SP**) and one of the following numbers:
/2 a valuation is less than zero;
/4 the reference period is not valid;
/7 insufficient number of values in the array;
/9 calculated value out of range;
/10 the maximum number of iterations was exceeded without converging on a value for the average periodic return.

Comments

The Average Periodic Return of the portfolio.

The cash flows are assumed to take place at the start of each period. They will have positive or negative values as money is invested or withdrawn, deposits being positive and withdrawals negative.

The values for the cash flows and periodic valuations of the portfolio are passed to the function by specifying two arrays of values contained in two connected elements (or two parts of the same connected element, e.g. a table). The array notation must be in brackets. The two arrays must correspond; for example, after the cash flow given in the third value of the *Cash Flows* array, the portfolio valuation will be given by the third value of the *Valuations* array.

For example, suppose that a table named *Cash Flows and Valuations* is connected, containing the cash flows in the first row and the periodic valuations in the second. In this case, therefore, both the arrays are contained in the same connected element.

The *connected element containing cash flows [array notation]* argument might contain the following reference:

```
Cash Flows and Valuations [R1C1:R1C6]
```

while the *connected element containing periodic valuations [array notation]* argument could contain the following reference with the same number of values:

```
Cash Flows and Valuations [R2C1:R2C6]
```

See Also

SPPortfolioStandardDeviation1, **SPPortfolioStandardDeviation2**, **SPPortfolioRateOfReturn**, **SPPortfolioTimeWeightedRateOfReturn**

Example

Consider a portfolio with the following characteristics:

PERIOD	VALUATIONS	CASH FLOWS
1	$ 1000	$ 980
2	$ 2000	$ 1050
3	$ 3000	$ 950
4	$ 3000	$ -100

We want to know its average return.

```
valuation_reference_period = ANNUAL
SPPortfolioAveragePeriodicReturn(
    valuation_reference_period,
    Cash Flows and Valuations [R2C3:R5C3],
    Cash Flows and Valuations [R2C2:R5C2])
```

The Portfolio Average Periodic Return is 1,19%.

SPPortfolioRateOfReturn - Function

Description

Calculates the Internal Rate of Return of a stock portfolio from an array of cash flows and periodic valuations.

Syntax

rate_of_return = **SPPortfolioRateOfReturn***(*
valuation_reference_period,
connected element containing cash flows **[***array notation***]***,*
connected element containing periodic valuations **[***array notation***]***)*

The brackets after the *connected element containing array* arguments are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The Portfolios Internal Rate of Return.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#SP**) and one of the following numbers:
/2 a valuation is less than zero;
/4 the reference period is not valid;
/7 insufficient number of values in the array;
/9 calculated value out of range;
/10 the maximum number of iterations was exceeded without converging on a value for the rate of return.
/19 insufficient memory.

Comments

This function calculates a portfolios Internal Rate of Return.

This index weights each period with the value of the portfolio at the start of the period.

The cash flows are assumed to take place at the start of each period. They will have positive or negative values as money is invested or withdrawn, deposits being positive and withdrawals negative.

The values for the cash flows and periodic valuations of the portfolio are passed to the function by specifying two arrays of values contained in two connected elements (or two parts of the same connected element, e.g. a table). The array notation must be in brackets. The two arrays must correspond; for example, after the cash flow given in the third value of the *Cash Flows* array, the portfolio valuation will be given by the third value of the *Valuations* array.

For example, suppose that a table named *Cash Flows and Valuations* is connected, containing the cash flows in the first row and the periodic valuations in the second. In this case, therefore, both the arrays are contained in the same connected element.

The *connected element containing cash flows [array notation]* argument might contain the following reference:

```
Cash Flows and Valuations [R1C1:R1C6]
```

while the *connected element containing periodic valuations [array notation] argument* could contain the following reference with the same number of values:

```
Cash Flows and Valuations [R2C1:R2C6]
```

See Also

SPPortfolioStandardDeviation1, **SPPortfolioStandardDeviation2**, **SPPortfolioAveragePeriodicReturn**, **SPPortfolioTimeWeightedRateOfReturn**

Example

Consider a portfolio with the following characteristics:

PERIOD	VALUATIONS	CASH FLOWS
1	$ 1000	$ 980
2	$ 2000	$ 1050
3	$ 3000	$ 950
4	$ 3000	$ -100

We want to know its Internal Rate of Return.

```
valuation_reference_period = ANNUAL
SPPortfolioRateOfReturn(
    valuation_reference_period,
    Cash Flows and Valuations [R2C3:R5C3],
    Cash Flows and Valuations [R2C2:R5C2])
```

The Internal Rate of Return is 1.33%.

SPPortfolioStandardDeviation1 - Function

Description

Calculates the Standard Deviation of a portfolios average return.

Syntax

standard_deviation = **SPPortfolioStandardDeviation1***(*
connected element containing expected returns of components **[***array notation***]***,*
connected element containing proportions of the portfolio per component **[***array notation***]***,*
connected element containing standard deviation for each component **[***array notation***]***,*
connected element containing correlation matrix of the components **[***array notation***]***)*

The brackets after the *connected element containing array* arguments are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The Standard Deviation of the portfolios average return.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#SP**) and one of the following numbers:

/2 a component has a negative weight (portfolio percentage);
/3 a component has an average return greater, in absolute value, than 100%;
/4 the weights total more than 1;
/5 the absolute value of a correlation value is greater than 100%;
/6 calculated value out of range (perhaps because one of the components or correlation values is not initialized)

Comments

This function calculates the Standard Deviation of a portfolios average return.

The Standard Deviation is calculated from the weighted average of the deviations of the components, taking account of their correlations. This adjustment is made to reflect the fact that the returns of the different components are interrelated.

Together with the Portfolio Average Return, which is the weighted average of the returns of the components, the Standard Deviation measures the portfolios efficiency. If the Average Return of two portfolios is the same, the one with the smaller Standard Deviation is the more efficient.

In all the arguments, percentages are expressed as decimal fractions: for example 0.55, not 55%.

The values are passed to the function by specifying four arrays of values contained in four connected elements (or different parts of the same connected element, e.g. a table). The array notation must be in brackets.

The first three arguments containing array notation correspond; for example, the third value of the *Expected Returns* array represents the return of the component occupying a percentage of the portfolio equal to the third value of the *Portfolio Percentage* array, and its standard deviation is the third value of the corresponding array. The fourth argument, the *correlation matrix*, must be passed as an array in the following order, where (R(i,j) is the correlation between components i and j. Assuming a series containing *n* components:

R(1,2), R(1,3), R(1,4), ..., R(1,n),

R(2,3), R(2,4), ..., R(2,n),

R(3,4), ..., R(3,n),

...

R(n-1,n)

For example, suppose that a table named *Component Values* is connected to the function and that the values of the first three arrays are in the first three rows and the correlation matrix in the fourth. In this case all the arrays therefore refer to the same connected element. Supposing a portfolio with four components, the following four arrays give the values of the four arguments:

connected element containing expected returns of components [array notation]:
```
Component Values [R1C1:R1C4]
```

connected element containing proportions of the portfolio per component [array notation]:
```
Component Values [R2C1:R2C4]
```

connected element containing standard deviation for each component [array notation]:
```
Component Values [R3C1:R3C4]
```

connected element containing correlation matrix of the components [array notation]:
```
Component Values [R4C1:R4C6]
```

The matrix array contains six values, the number of different pairs among the four components.

See Also

SPPortfolioStandardDeviation2

Example

A portfolio is composed of 50% shares, 25% bonds and the remaining 25% cash. Their characteristics are respectively:

	EXPECTED RETURN	STANDARD DEVIATION
SHARES	12%	17%
BONDS	9%	9%
CASH	6%	3.5%

	CORRELATIONS
SHARES/BONDS	0.12
SHARES/CASH	-0.23
BONDS/CASH	0.2

We want to know the Standard Deviation of the portfolios average return.

```
SPPortfolioStandardDeviation1(
    portfolio1 [R2C2:R4C2],
    portfolio1 [R2C4:R4C4],
    portfolio1 [R2C3:R4C3],
    portfolio1 [R7C2:R9C2])
```

The standard deviation of this portfolio is 8.95%.

SPPortfolioStandardDeviation2 - Function

Description

Calculates the Standard Deviation of a portfolios Average Return from a series of cash flows and periodic valuation of the portfolio.

Syntax

standard_deviation = **SPPortfolioStandardDeviation2***(*
valuation_reference_period,
connected element containing cash flows **[** *array notation* **]***,*
connected element containing periodic valuations **[** *array notation* **]***)*

The brackets after the *connected element containing array* arguments are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The Standard Deviation of the portfolios average return.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#SP**) and one of the following numbers:

- /2 a valuation is less than zero;
- /4 the reference period is not valid;
- /7 insufficient number of values in the array;
- /9 calculated value out of range;
- /10 the maximum number of iterations was exceeded without converging on a value for the standard deviation.
- /19 insufficient memory.

Comments

This function calculates the Standard Deviation of the portfolios average return.

Together with the Portfolio Average Return, the Standard Deviation is the tool which measures the efficiency of a portfolio: if the Average Return of two portfolios is the same, the one with the smaller Standard Deviation is the more efficient.

The cash flows are assumed to take place at the start of each period. They will have positive or negative values as money is invested or withdrawn, deposits being positive and withdrawals negative.

The values for the cash flows and periodic valuations of the portfolio are passed to the function by specifying two arrays of values contained in two connected elements (or two parts of the same connected element, e.g. a table). The array notation must be in brackets. The two arrays must correspond; for example, after the third value of the *Cash Flows* array the portfolio valuation will be represented by the third value of the *Valuations* array.

For example, suppose that a table named *Cash Flows and Valuations* is connected, containing the cash flows in the first row and the periodic valuations in the second. In this case, therefore, both the arrays are contained in the same connected element.

The *connected element containing cash flows [array notation]* argument might contain the following reference:

```
Cash Flows and Valuations [R1C1:R1C6]
```

while the *connected element* *containing periodic valuations [array notation]* argument could contain the following reference with the same number of values:

```
Cash Flows and Valuations [R2C1:R2C6]
```

See Also

SPPortfolioStandardDeviation1, SPPortfolioRateOfReturn, SPPortfolioTimeWeightedRateOfReturn, SPPortfolioAveragePeriodicReturn

Example

Consider a portfolio with the following characteristics:

PERIOD	VALUATIONS	CASH FLOWS
1	$ 1000	$ 980
2	$ 2000	$ 1050
3	$ 3000	$ 950
4	$ 3000	$ -100

We want to know its Standard Deviation.

```
valuation_reference_period = ANNUAL
SPPortfolioStandardDeviation2(
    valuation_reference_period,
    Cash Flows and Valuations [R2C3:R5C3],
    Cash Flows and Valuations [R2C2:R5C2])
```

The Standard Deviation of the portfolio is 2.53%.

SPPortfolioTimeWeightedRateOfReturn - Function

Description

Calculates the Time Weighted Rate of Return (all periods given the same weight) of a portfolio from a series of cash flows and periodic valuations of the portfolio.

Syntax

time_weighted_rate_of_return =
SPPortfolioTimeWeightedRateOfReturn*(*
valuation_reference_period,
connected element containing cash flows **[** *array notation* **]***,*
connected element containing the periodic valuation **[** *array notation* **]***)*

The brackets after the *connected element containing array* arguments are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The Time Weighted Rate of Return.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#SP**) and one of the following numbers:

/2 a valuation is less than zero;
/4 the reference period is not valid;
/7 insufficient number of values in the array;
/9 calculated value out of range;
/10 the maximum number of iterations was exceeded without converging on a value for the rate of return.
/19 insufficient memory.

Comments

This function calculates the geometric average rate of return with each period weighted equally. This indicator can be useful in comparing a portfolios performance with other investments because it is independent of cash flow timings.

The cash flows are assumed to take place at the start of each period. They will have positive or negative values as money is invested or withdrawn: deposits are positive and withdrawals negative.

The values for the cash flows and periodic valuations of the portfolio are passed to the function by specifying two arrays of values contained in two connected elements (or two parts of the same connected element, e.g. a table). The array notation must be in brackets. The two arrays must correspond; for example, after the third dividend in the *Cash Flows* array, the portfolio valuation will be represented by the third value in the *Valuations* array.

For example, suppose that a table named *Cash Flows and Valuations* is connected, containing the cash flows in the first row and the periodic valuations in the second. In this case, therefore, both the arrays are contained in the same connected element.

The *connected element containing cash flows [array notation]* argument might contain the following reference:

```
Cash Flows and Valuations [R1C1:R1C6]
```

while the *connected element* *containing periodic valuations [array notation]* argument could contain the following reference with the same number of values:

```
Cash Flows and Valuations [R2C1:R2C6]
```

See Also

SPPortfolioStandardDeviation1, **SPPortfolioStandardDeviation2**, **SPPortfolioRateOfReturn**, **SPPortfolioAveragePeriodicReturn**

Example

Consider a portfolio with the following characteristics:

PERIOD	VALUATIONS	CASH FLOWS	
	1	$ 1000	$ 980
	2	$ 2000	$ 1050
	3	$ 3000	$ 950
	4	$ 3000	$ -100

We want to know its rate of return, regardless of the timing of cash flows.

```
valuation_reference_period = ANNUAL
SPPortfolioRateOfReturn(
    valuation_reference_period,
    Cash Flows and Valuations [ R2C3 : R5C3 ],
    Cash Flows and Valuations [ R2C2 : R5C2 ])
```

The rate of return, independent of the timing of the cash flows, is 1.16%.

Sqrt - Function

Description

Calculates the square root of an expression.

Syntax

square_root = **Sqrt***(expression)*

Returns

The square root of the argument.

If the argument is less than zero, the error value **#SQRT/<0** is returned.

See Also

Exp

Example

The script:

```
Sqrt(9)
```

returns the value 3.

StandardDeviation - Function

Description

Calculates the Standard Deviation for an array of values.

Syntax

standard_deviation = **StandardDeviation***(*
connected element containing array of values **[***array notation***]***)*

The brackets after the *connected element containing array of values* argument are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The standard deviation of the array.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#STAT**) and one of the following numbers:
/9 calculated value out of range.

Comments

This function calculates the standard deviation of the array of values.

The array must be passed to the function by specifying the name of the connected element. The array notation must be in brackets.

Consider a hypothetical connected series called *Values*. The *connected element containing array [array notation]* might be:

```
Values [1:24]
```

See Also

Mean, **Variance**

Example

To find the standard deviation yield of the following portfolio:

	YIELDS	WEIGHTS
Bonds	0.085	0.25
Shares	0.132	0.5
Short Term Securities	0.07	0.25

```
StandardDeviation( Values 2 [R2C2 : R4C2] )
```

The Standard Deviation is 0.02641.

StandardErrorOfBeta - Function

Description

Calculates the Standard Error of Beta of two arrays of values.

Syntax

standard_error = **StandardErrorOfBeta***(*
connected element containing dependent array **[***array notation***]**
connected element containing independent array **[***array notation***]***)*

The brackets after the *connected element containing array* arguments are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The standard error of the Beta index.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#STAT**) followed by one of the following numbers:
/1 number of observations less than 3;
/9 calculated value out of range;
/18 division by zero.

Comments

This function gives the error in the estimation of Beta; it is a basic tool of stock market analysis. Given a securitys quotations as a dependent array and a market index representing the sector or market segment of the security as an independent array, the function calculates the Standard Error of Beta of the security (Equity Standard Error of Beta).

The beta index represents the slope of the regression line: it expresses the change in the security price for a unit change in the index. Beta values greater than 1 or less than -1 represent securities tending to react to factors affecting the index with proportionately greater volatility.

In the case of a security, the values for the security price and the index are passed to the function by specifying two arrays of values contained in two connected elements (or two parts of the same connected element, e.g. a table). The array notation must be in brackets. The two arrays must correspond; for example, the third value of the *Bond* array represents the quotation for the day on which the index value is represented by the third value of the *Index* array.

Thus the connected element containing dependent array [array notation] argument might contain the following reference:

```
Bond [1:24]
```

while the *connected element containing independent array [array notation]* argument could contain the following reference with the same number of values:

```
Index[1:24]
```

See Also

Alpha, **Beta**, **CorrelationCoefficient**, **StandardErrorOfRegression**

Example

These are the latest closing values of a share and the value of a hypothetical index for the same period:

Share A (dep.)	Index (indep.)
311.850	37.125
312.600	37.000
309.140	35.500
307.570	35.875
310.490	36.750

To find the error in the Beta coefficient of the two arrays of values:

```
StandardErrorOfBeta(
    values 1[R2C2:R6C2],
    values 1[R2C3:R6C3])
```

The average error in the estimation of Beta is 0.83157.

StandardErrorOfRegression - Function

Description

Calculates the Standard Error of Regression for two arrays of values.

Syntax

standard_error = **StandardErrorOfRegression***(*
connected element containing dependent array **[***array notation***]**
connected element containing independent array **[***array notation***]***)*

The brackets after the *connected element containing array* arguments are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The Standard Error of Regression.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#STAT**) and one of the following numbers:
/1 number of observations less than 3;
/9 calculated value out of range;
/18 division by zero.

Comments

This function calculates the Standard Error of Regression of two arrays.

It can be used in stock market analysis by passing it the quotations of a security as a dependent array and a market index representing the sector or market segment as an independent array.

The two arrays of prices and index values are passed to the function by specifying two arrays of values contained in two connected elements (or two parts of the same connected element, e.g. a table). The array notation must be in brackets. The two arrays must correspond: for example, the third value of the *Bond* array represents the quotation for the day on which the index value is represented by the third value of the *Index* array.

For example, the *connected element containing dependent array [array notation]* argument might contain the following reference:

```
Bond [1:27]
```

while the *connected element containing independent array [array notation]* argument could contain the following reference with the same number of values:

```
Index [1:27]
```

See Also

Alpha, **Beta**, **CorrelationCoefficient**, **StandardErrorOfBeta**

Example

These are the latest closing values of a share and the value of a hypothetical index for the same period,

which are contained in a table called *Values 1*:

Share A (dep.)	Index (indep.)
311.850	37.125
312.600	37.000
309.140	35.500
307.570	35.875
310.490	36.750

To find the error of the regression line of the two arrays of values:

```
StandardErrorOfRegression(
    values 1 [R2C2:R6C2],
    values 1 [R2C3:R6C3])
```

The average error in estimating the dependent values from the regression line is 1.19967.

STEndingBalance - Function

Description

Calculates the ending balance for an interest bearing short term deposit, using the 360/30-day convention.

Syntax

end_balance = **STEndingBalance***(*
starting_balance,
period_starting_date,
maturity_date,
compounding_period,
short_period_option,
annual_yield)

Returns

The balance at maturity.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#ST**) and one of the following numbers:
/1 starting value or yield is not valid;
/4 compounding period is out of range (WEEKLY and BIWEEKLY are not accepted);
/14 either maturity date or purchase date is invalid.

Comments

This routine should only be used for an account paying interest on a 360/30-day basis. It is the short period counterpart of **FVFutureValueOfPresentValue**.

For daily compounding (*compounding_period* is **DAILY**), the periods are normal length days, otherwise the number of periods is based on the 360/30 convention, under which a year is 360-days and all months are 30-day.

See Also

FVFutureValueOfPresentValue, **STNotePrice1**, **STNotePrice3**, **STNoteYield1**, **STNoteYield2**, **STNoteDiscount2**, **STNoteDiscount3**.

Example

An 18 month deposit pays 8.5% interest compounded daily on a 360-day year. The period begins on June 15 1990.

```
deposit = 100000
starting_date = Date("June 15 1990")
maturity_date = Date("December 15 1991")
compounding_period = DAILY
short_period_option = CMPND
yield_rate = 0.085
STEndingBalance(
    deposit,
```

```
    starting_date,
    maturity_date,
    compounding_period,
    short_period_option,
    yield_rate)
```

The deposit will grow to $113,811.53.

STNoteDiscount2 - Function

Description

Calculates the discount of a note (short term security).

Syntax

> *discount* = **STNoteDiscount2***(*
> *cost,*
> *days_to_maturity,*
> *face_value,*
> *calendar_option)*

Returns

The discount of the note.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#ST**) and the following number:
/1 face value or number of days is not valid.

Comments

This function calculates the discount rate of a note from readily accessed information such as its price and maturity.

The *calendar option* argument must take one of the following two values:

- **C360** to use the 360/30-day convention, which stipulates a 360-day year and a 30-day month;
- **C365** for exact year and month length.

If using a quoted price as the cost (93.25 indicates that the note is selling for 93.25% of its face value), use 100 for the *face_value* argument.

See Also

STEndingBalance, **STNotePrice1**, **STNotePrice3**, **STNoteYield1**, **STNoteYield2**, **STNoteDiscount3**.

Example

To calculate the discount rate applied to a note with the following characteristics:

```
price = 99.63
days_to_maturity = 15
face_value = 100
calendar_opt = C360
STNoteDiscount2(
    price,
    days_to_maturity,
    face_value,
    calendar_opt)
```

The discount will be 8.88% (on a 360-day basis).

STNoteDiscount3 - Function

Description

Calculates the discount of a note (short term security).

Syntax

discount = **STNoteDiscount3***(*
days_to_maturity,
expected_return_rate,
calendar_option)

Returns

The discount of the note.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#ST**) and the following number:
/1 number of days is not valid.

Comments

This function calculates the discount rate of a note from its expected rate of return.

The *calendar option* argument must take one of the following two values:

- **C360** to use the 360/30-day convention, which stipulates a 360-day year and a 30-day month;
- **C365** for exact year and month length.

See Also

STEndingBalance, **STNotePrice1**, **STNotePrice3**, **STNoteYield1**, **STNoteYield2**, **STNoteDiscount2**.

Example

To calculate the discount rate applied to a note with the following characteristics:

```
days_to_maturity = 91
expected_return = 0.07125
calendar_opt = C360
STNoteDiscount3(
    days_to_maturity,
    expected_return,
    calendar_opt)
```

The discount will be 7% (on 360-day basis).

STNotePrice1 - Function

Description

Calculates the price of a note (short term security).

Syntax

price = **STNotePrice1***(*
days_to_maturity,
discount_rate,
calendar_option)

Returns

The current price of a discounted note as a percentage of face value.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#ST**) followed by one of the following numbers:
/1 number of days is not valid;
/9 cumulative discount exceeds 100%.

Comments

This function calculates the current cost of a discounted note as a percentage of its face value on the basis of its characteristics.

The *calendar option* argument must take one of the following two values:

- **C360** to use the 360/30-day convention, which stipulates a 360-day year and a 30-day month;
- **C365** for exact year and month length.

See Also

STNotePrice3, **STEndingBalance**, **STNoteYield1**, **STNoteYield2**, **STNoteDiscount2**, **STNoteDiscount3**

Example

To calculate the price of a note with the following characteristics:

```
days_to_maturity = 182
discount = 0.0779
calendar_opt = C360
STNotePrice1(
    days_to_maturity,
    discount,
    calendar_opt)
```

The note is priced at 96.06.

STNotePrice3 - Function

Description

Calculates the price of a note (short term security).

Syntax

price = **STNotePrice3***(*
days_to_maturity,
expected_return_rate,
calendar_option)

Returns

The current price of the note as a percentage of face value.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#ST**) and the following number:
/1 number of days is not valid;

Comments

This function calculates the cost of note as a percentage of its face value, using the expected return rate.

The *calendar option* argument must take one of the following two values:

- **C360** to use the 360/30-day convention, which stipulates a 360-day year and a 30-day month;
- **C365** for exact year and month length.

See Also

STNotePrice1, **STEndingBalance**, **STNoteYield1**, **STNoteYield2**, **STNoteDiscount2**, **STNoteDiscount3**

Example

To calculate the price of a note with the following characteristics:

```
days_to_maturity = 91
expected_return = 0.07125
calendar_opt = C360
STNotePrice3(
    days_to_maturity,
    expected_return,
    calendar_opt)
```

The price will be 98.23.

STNoteYield1 - Function

Description

Calculates the yield of a note (short term security).

Syntax

yield = **STNoteYield1***(*
days_to_maturity,
discount_rate,
calendar_option)

Returns

The yield (rate of return) of the note.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#ST**) and one of the following numbers:
/1 face value or number of days is not valid;
/9 cumulative discount exceeds 100%.

Comments

This function calculates the yield of a note on the basis of its characteristics.

The *calendar option* argument must take one of the following two values:

- **C360** to use the 360/30-day convention, which stipulates a 360-day year and a 30-day month;
- **C365** for exact year and month length.

See Also

STNoteYield2, **STEndingBalance**, **STNoteDiscount2**, **STNoteDiscount3**,**STNotePrice1**, **STNotePrice3**

Example

To find the yield of a twenty-six week treasury bill sold at a discount of 7.79%.

```
days_to_maturity = 182
discount = 0.0779
calendar_opt = C360
STNoteYield1(
    days_to_maturity,
    discount,
    calendar_opt)
```

The yield is 8.11%.

STNoteYield2 - Function

Description

Calculates the yield of a note (short term security).

Syntax

yield = **STNoteYield2***(*
price,
days_to_maturity,
face_value,
calendar_option)

Returns

The yield of a note.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#ST**) and the following number:
/1 face value or number of days is not valid.

Comments

This function calculates the yield rate of a note from readily accessed information such as its price and maturity.

The *calendar option* argument must take one of the following two values:

- **C360** to use the 360/30-day convention, which stipulates a 360-day year and a 30-day month;
- **C365** for exact year and month length.

If using a quoted price (93.25 indicates that the note is selling for 93.25% of its face value) as the *price* argument, use 100 for the *face_value* argument.

See Also

STNoteYield1, STEndingBalance, STNotePrice1, STNotePrice3, STNoteDiscount2, STNoteDiscount3.

Example

To calculate the yield of a note with the following characteristics:

```
price = 99.63
days_to_maturity = 15
face_value = 100
calendar_opt = C360
STNoteYield2(
    price,
    days_to_maturity,
    face_value,
    calendar_opt)
```

The yield will be 8.91%.

Sum - Function

Description

Returns the sum of the values in an array.

Syntax

sum_value = **Sum***(connected element [array notation])*

Returns

The sum of the values of the array.

See Also

SumAll

Example

After connecting a series called *Array*, the **Sum** function could be used as follows to obtain the sum of the first twelve values:

```
Sum(array[1:12])
```

SumAll - Function

Description

Returns the sum of the values for the current step of all the elements connected to a variable.

Syntax

Total = **SumAll***()*

Returns

The sum of the values for the current step of all the elements connected to the variable.

Comments

In the case of a variable representing the sum of all the elements connected to it, this function eliminates the need to specify in the script the names of all the connected elements. If additional elements are connected to the variable, their values will be automatically added to the value of the variable without the user having to add their names to the script.

See Also

Sum

Example

A variable has three elements connected to it, whose values for the current step are 3, 58 and 24. The script:

```
SumAll()
```

will return the value 85 for the current step.

SystemDate - Function

Description

Returns the system date in internal format.

Syntax

internal_date = **SystemDate()**

Returns

The date in internal format.

See Also

Date

Example

The following script, executed on September 20 1993:

```
SystemDate()
```

returns the value 19930920

Tan - Function

Description

Calculates the tangent of an angle in radians.

Syntax

Tangent = **Tan***(expression)*

Returns

The tangent of the angle.

In the case of an angle near PI/2, an **#OVERFLOW** error may be generated.

Attempts to find the tangent of very large values may generate the error values **#PLOSS**, representing a partial loss of significance, or **#TLOSS**, representing a total loss of significance.

See Also

Sin, **Cos**, **ArcSin**, **ArcTan**, **SinH**, **CosH**, **TanH**

Example

The script:

```
Tan(PI)
```

returns the value 0.

TanH - Function

Description

Calculates the hyperbolic tangent.

Syntax

Hyperbolic_tangent = **TanH***(expression)*

Returns

The hyperbolic tangent.

See Also

Sin, **Cos**, **Tan**, **ArcSin**, **ArcTan**, **SinH**, **CosH**

Example

The script:

```
TanH(PI)
```

returns the value 0.99627.

Trend1 - Function

Description

Calculates the next occurring value on a trend line. The trend line is based on a given array and is calculated using the least squares procedure.

Syntax

trend = **Trend1***(element* **[***array notation***]***)*

The brackets after the *element* argument are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The next number in the specified series.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this function (**#TREND1**) and one of the following numbers:
/1 less than 3 values specified in the array;
/9 calculated value out of range;
/18 division by zero.

Comments

This function uses the least squares procedure to estimate the parameters of a *regression line,* which is then used to calculate the number following the last one given in the series.

The independent variable in this case consists of the series of natural numbers (1, 2, 3...).

Obviously, in many situations the linear trend given by this function will not give any degree of predictive accuracy.

See Also

Alpha, **Beta**, **Trend2**, **Trend3**

Example

Suppose we want to forecast the average dollar-lire exchange rate for April based on the actual rates for the three months prior, January through March. First, place a series element on the model page, name it *Actual Rates* and enter the following values for the first three steps:

```
1320        1420        1457
```

Create a variable element named *Forecasted Rate* and connect to it the first element. Enter as the script for the variable the following:

```
Trend1 ( Actual Rates [ 1 : 3] )
```

This will generate the forecasted average dollar-lire exchange rate for April of 1536.

Trend2 - Function

Description

Calculates the value at the specified point on a trend line. The trend line is based on a given array and is calculated using the least squares procedure.

Syntax

trend = **Trend2***(element* **[***array notation***]***, position)*

The brackets after the *element* argument are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The number corresponding to the specified position in the specified array.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this function (**#TREND2**) and one of the following numbers:
/1 less than 3 values specified in the array;
/9 calculated value out of range;
/18 division by zero.

Comments

This function uses the Least Squares procedure to estimate the parameters of a *regression line,* which is then used to calculate the number corresponding to a specified position in the series.

The independent variable in this case consists of the series of natural numbers (1, 2, 3...).

Obviously, in many situations the linear trend given by this function will not give any degree of predictive accuracy.

See Also

Alpha, **Beta**, **Trend1**, **Trend3**

Example

In this example we will forecast the average dollar-lire exchage rate for the current month based on the actual rates of three months, January through March. First, place a series element on the model page, name it *Actual Rates* and enter the following values for the first three steps:

```
1320        1420        1457
```

Create a variable element named *Forecasted Rates* and connect to it the first element. Enter as the script for the variable the following:

```
Trend2( Actual Rates [ 1 : 3] ; SIMSTEP )
```

This will generate the forecasted average dollar-lire exchange rate for the current month. For example, in December the value will be 2084.

Trend3 - Function

Description

Calculates the dependent value associated with a given independent value based on a trend line. This trend line is derived from two arrays (one independent and one dependent) using the least squares procedure.

Syntax

trend = **Trend3***(*
connected element containing dependent array **[***array notation***]***,*
connected element containing independent array **[***array notation***]***,*
new value of independent array)

The brackets after the *connected element containing array* arguments are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The estimated values of the dependent array.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this function (**#TREND3**) and one of the following numbers:
/1 less than 3 values specified in the array;
/9 calculated value out of range;
/18 division by zero.

Comments

This function uses the Least Squares procedure to estimate the parameters of a *regression line* which is then used to calculate the values of the dependent array, given the values of the independent array, by the formula:

Dependent array = $\alpha + \beta$ * independent array

The two arrays are passed to the function by specifying two arrays of values contained in two connected elements (or two parts of the same connected element, e.g. a table). The array notation must be in brackets. The two arrays must correspond: for example, the third value of the *Security* array represents the quotation for the day on which the index value is represented by the third value of the *Index* array.

Suppose, for example, that two series called *Security* and *Index* are connected.

The *connected element containing dependent array [array notation]* argument could contain the reference:

```
Security [1:24]
```

while the *connected element containing independent array [array notation]* argument could contain the following reference with the same number of values:

```
Index [1:24]
```

The **Trend3** function will calculate a predicted value for the Security if the index has a value of 37,800 (assuming a linear relationship).

Obviously, in many situations the linear trend given by this function will not give any degree of predictive accuracy.

See Also

Alpha, Beta, StandardErrorOfBeta, CorrelationCoefficient, StandardErrorOfRegression, Trend1, Trend2

Example

Given the following recent quotations of a bond and the values of a market index for the same period, contained in a table called *Values 1*:

BOND A (dependent)	INDEX (independent)
311.85	37.125
312.65	37
309.14	35.5
307.57	35.875
310.49	36.75

we want a price estimate for the bond if the index goes to 37.8.

We therefore write the following script:

```
Trend3(
    values 1[R2C2:R6C2],
    values 1[R2C3:R6C3],
    37.8)
```

The price of the security may be expected to rise to $313.63.

Variance - Function

Description

Calculates the variance of an array of values.

Syntax

variance = **Variance***(connected element containing array* **[***array notation***]***)*

The brackets after the *connected element containing array* argument are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The variance of the array.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#STAT**) and the following number:
/9 calculated value out of range.

Comments

This function calculates the variance of the array of values. The values must be passed to the function by specifying the name of the connected element. The array notation must be in brackets.

Consider a hypothetical connected series called *Values*. An example for the *connected element containing array [array notation]* could be:

```
Values [1:36]
```

See Also

Mean, **StandardDeviation**

Example

To find the variance yield of the following portfolio:

	YIELDS	WEIGHTS
Bonds	0.085	0.25
Shares	0.132	0.5
Short Term Securities	0.07	0.25

```
Variance( Values 2 [R2C2 : R4C2] )
```

The Variance is 0.00070.

Week - Function

Description

Finds the week (1-52) of a given date in internal format.

Syntax

This_week=**Week***(internal_date)*

Returns

The number of the week.

If the *internal_date* argument does not represent a valid date, the error value **#WEEK** is returned.

See Also

Day, **Month**, **Year**, **DayWeek**, **DayYear**

Example

If the step unit is **DAY** or **WEEK**, the system variable **TIME** contains the current day in internal format. The script:

```
Week(TIME)
```

will return the number of the current week in the simulation.

WeightedAverage - Function

Description

Calculates the weighted average of a number of values.

Syntax

weighted_average = **WeightedAverage***(*
connected element containing values **[***array notation***]***,*
connected element containing weights **[***array notation***]***)*

The brackets after the *connected element containing array* arguments are in **bold type** to emphasize that their use is compulsory with this function.

Returns

The Weighted Average.

The presence of errors in the arguments will cause the function to return an error value consisting of the error variable for this class of functions (**#STAT**) and one of the following numbers:
/1 number of observations less than 3;
/9 calculated value out of range.

Comments

This function calculates the Weighted Average of a number of values.

The values and weights of the inputs are passed to the function by specifying two arrays of values contained in two connected elements (or two parts of the same connected element, e.g. a table). The array notation must be in brackets. The two arrays must correspond; for example, the third value of the *Values* array corresponds to the element whose weight is in the third position in the *Weights* array.

For example, the *connected element containing values [array notation]* might contain the following reference:

```
Values [1:24]
```

while the *connected element containing weights [array notation]* argument could contain the following reference with the same number of values:

```
Weights [R2C1:R2C24]
```

See Also

Mean, **SPPortfolioStandardDeviation1**

Example

To find the weighted average yield of the following portfolio:

	YIELDS	WEIGHTS
Bonds	0.085	0.25
Shares	0.132	0.5
Short Term Securities	0.07	0.25

```
WeightedAverage(
    values 2 [R2C2 : R4C2],
    values 2 [R2C3 : R4C3])
```

The Weighted Average is 0.10475.

Year - Function

Description

Finds the year of a given date in internal format.

Syntax

This_year=**Year***(expression)*

Returns

The number of the year.
If the *internal_date* argument does not represent a valid date, the error value **#YEAR** is returned.

See Also

Day, **Week**, **Month**, **DayWeek**, **DayYear**

Example

If the step unit is **YEAR**, the system variable **TIME** contains the current year in internal format. The script:

```
Year(TIME)
```

will return the number of the current year in the simulation.

Accrued Interest

If a *bond* is traded between interest payments, the sum paid for it includes the interest due for the partial period during which the seller has held the bond. The buyer then keeps the entire coupon payment when it falls due.

Annuity

A series of regular payments from an investment. As with a mortgage, each payment of an annuity includes a portion of the principal invested as well as the interest earned.

Array

A series of values used as data in the script of a DS Lab *variable* and referenced by means of *array notation.*

Bond

For DS Labs purposes, this term is used to include all categories of fixed-interest securities, including government or public authority bonds and private fixed-interest investments such as debentures.

Bond Cost

The cost of a bond at any given moment is given by its actual market price   that is, its face value multiplied by its *discount* plus the accrued interest to the date of purchase:

Bond Cost = Discount * Face Value + Accrued Interest

Bond price

The price of a *bond* is expressed as a percentage of its face value. In DS Lab, it is represented as a decimal fraction, e.g. 0.925 (equivalent to 92.5%)

Compound Interest

When money is invested at compound interest, the interest earned is automatically added to the principal and itself earns interest for all subsequent periods.

Compounding Period

The calculating period for compound interest, that is, the intervals at which interest is calculated and added to the principal. Also called Conversion Period.

The compounding period is represented in DS Lab as the number of periods in a year, and it is generally advisable to define it by means of the programs predefined financial constants.

A special case is compound interest that is calculated continuously. DS Labs financial functions are able to cope with this situation by defining the compounding period as CONTINUOUS (value 0).

Connected Element

An element which is graphically connected to a *variable* by a connection arrow becomes an *input* to that variable and is available to be referenced in its script.

Coupon Payment Period

The number of times a year when bond coupons are payable. Depending on the bond, interest payments may take place annually, twice a year (SEMIANNUAL) or more frequently.

Coupon Rate

The *simple interest* or nominal interest rate used to calculate the periodic coupons payable on a *bond*.

Current Step

The step of a DS Lab simulation shown in the **Step units** list box on the horizontal toolbar, and for which the values of the elements are shown below each of them in the model window.

Deferred Annuity

An *annuity* whose payments start at a future date. Until then, compound interest is calculated and is added to the principal at each *compounding period*.

Discount

The percentage by which a future value is reduced at the present time. It is calculated on the basis of an appropriate interest rate, which for this reason is known as the *discount rate*.

Discount Rate

Used with the Present Value (**PV**) functions, this is the rate by which the future value is reduced to give the present value. It may be the inflation rate or the return given by an available alternative investment such as government bonds.

Effective Rate

The annual interest rate, taking account of the effect of compounding. If the *compounding period* is the year, the effective rate is the same as the nominal rate. If it is shorter than a year, the effective rate will be higher. The more frequently interest is compounded, the greater the difference: the greatest difference is given by continuous compounding.

Face Value

The nominal value of a bond, which may be sold initially at a lower *issue price;* its market price is expressed as a percentage of the face value.

Internal Rate of Return

The *discount rate* which makes the value of the bonds future cash flows equal to its purchase price, or, expressed differently, which makes the Net Present Value (NPV) of the cash flows equal to zero.

Local Variable

A script can contain *local variables*. The first time a local variable is used must be in an *assignment*; after that it can be used anywhere it is allowed. It can be reassigned with a new value later in the script. The local variable exists only during execution of the script in which it was defined: its value is not kept after the scripts value has been calculated.

Nominal Yield

The *nominal yield* of a bond is its market price as a percentage of its face value.

Note

In the Short Term **(ST)** financial functions, this term indicates any kind of short term investment.

Present Value

The calculated value at the present time of cash flows or payments which will take place in the future.

Short Period

A fraction of the interest *compounding period*. For short periods, simple rather than compound interest is normally used.

Short Term

The meaning of this term is not rigidly defined in financial jargon, but it usually means less than a year or less than one interest period. Certificates of Deposit, treasury bills and bank deposit accounts are typical short term investments. Short term interest is usually calculated using the 360/ 30 day convention, meaning that for calculation purposes all months are considered as having 30 days and the year 360 days.

Simple Interest

Also known as *annual rate* or *nominal interest.* Interest is calculated only on the principal invested and not on any interest previously earned.

DDE Messages to which DS Lab Responds

You can send the messages listed below from any Windows application to DS Lab by specifying the following DDE arguments:

Application = "DS Lab"

Topic = "SYSTEM"

Item = one of the following messages:

OPEN *(model_name)*

RECALCULATE *(model_name)*

CALCULATE *(model_name)*

CLOSE (*model_name*, *save_option*)

UPDATEDDELINKS *(model_name)*

With **CLOSE,** the second argument must be:

0 to close the model without saving it;

1 to save the model before closing.

How to...

Startup Operations

Starting DS Lab
Creating a New Model
Opening a Previously Saved Model
Moving from One Model to Another
Protecting the Model

Graphic Building of the Model

Inserting Elements in the Model

Selecting Elements in the Model

- Selecting a Single Element
- Selecting a Rectangular Area of the Model
- If the Elements to be Selected are Not Adjacent

Moving Elements in the Model
Deleting Elements from the Model
Copying Part of a Model Within the Same Model
Copying Part of One Model to Another
Inserting a Shadow Element
Connecting Another Element to a Variable
Deleting or Changing the Shape of a Connection Arrow
Selecting Overlapping Connection Arrows
Associating a Document with the Model

Defining the Numerical Values of the Model

Defining Variables
Entering Values in a Series
Speeding Up Simulations

Model Formatting Options

Coloring an Element
Defining the Format of Numbers
Converting One Element to Another
Setting the Print Size

Moving Around in the Model

Defining a New Zone of the Model
Moving From One Defined Zone to Another
Searching for an Element
Searching for Undefined Variables
Searching for Variables with Run-Time Errors

Exporting and Importing Data

Exporting Values to Excel
Making Links with Excel
Establishing Input Links from Other DS Lab Models
Establishing Output Links to Other DS Lab Models
Executing a Sub-Model
Importing Data with DDE (DDE Input Procedures)
Exporting Data with DDE (DDE Output Procedures)
Updating DDE Input Links

Starting DS Lab

1. In the Program Manager window, double-click the DS Lab group icon if the window is not already open.

2. Double-click the DS Lab program icon .

Creating a New Model

1. From the **File** menu, choose **New...**
2. In the dialog box which appears, choose **Model** to start a new model, or **Text** to open a text file, by clicking on the desired option button.
3. Choose the **OK** button to confirm your choice, or **Cancel** to abandon the creation of a new model/text file.

Opening a Previously Saved Model

1. From the **File** menu, choose **Open...**. A dialog box appears.
2. In the list box on the right, select the directory in which the model was previously saved by clicking on its name.
3. Select the model in the list box on the left by clicking on its name.
4. Choose the **OK** button to confirm the opening of the model or **Cancel** to abandon the operation.

Note: The last four models used are listed at the bottom of the **File** menu and can be reopened simply by choosing them from the menu.

Moving from One Model to Another

You can have more than one model open at a time. To move from one model to another:

- Choose the name of the model on which you wish to work from the **Window** menu.

OR:

- Click on a visible part of the model on which you wish to work.

Protecting the Model

1. Choose the command **Password...** from the **Options** menu.
2. Enter the password you have chosen (no more than 8 characters) and press **Enter** or click on the **OK** button.
3. Type the password a second time in the edit box and press **Enter** or click on the **OK** button again. If the two versions are different, you will be asked to start again from scratch.

WARNING: **The password can be modified only from within the model.** If you subsequently forget it, the model will be irrevocably lost!

Inserting Elements in the Model

1. Click the tool representing the element you wish to insert in the model.
   OR: Choose the desired element from the **Model** menu.
   OR: Press the corresponding short cut key (ALT + one of the function keys F6 to F10).
2. Move the pointer to the part of the model where the element is to be inserted.
3. Click the mouse button.

Selecting a Single Element

1. Select the **Selection Arrow** tool .
2. Click on the desired element. It is now shown with a broken outline.

Selecting a Rectangular Area of the Model

1. Select the **Selection Arrow** tool .
2. Move the pointer to the top left corner of the desired area.
3. While keeping the mouse button pressed, drag to the bottom right corner of the area.
4. Release the mouse button. The selected elements are now shown with broken outlines.
5. Any of the selected elements may now be deselected by clicking on it while holding down the SHIFT key.

If the Elements to be Selected are Not Adjacent

1. Select the **Selection Arrow** tool .
2. Select the first element by clicking on it.
3. While holding down the SHIFT key, click on the other element(s) to be selected. As each element is selected, it is shown with a broken outline.

Note: Clicking a second time on a selected element will deselect it.

Moving Elements in the Model

1. If several elements are to moved together, select them.
2. Move the pointer to one of the elements to be moved, depress the mouse button and, keeping it pressed, drag the element(s) to the new position.
3. Release the mouse button.

Note: The connection arrows automatically move to the new position together with the elements. However, it may be necessary to move them or change their curvature to improve the appearance of the model: see ***Deleting or Changing the Shape of a Connection Arrow***.

Deleting Elements from the Model

1. Select the element(s) to be deleted.
2. Open the **Edit** menu, or click the right mouse button, and choose **Delete**.
   OR: Press the DEL key.
3. A dialog box will appear, asking you to confirm the deletion of the selected elements. Choose **OK** to confirm the deletion of the selected element(s), or **Cancel** to abandon the operation.

Note: When an element is deleted from the model, all its connections with other elements are automatically deleted as well. If the deleted element was referenced in any formula of the model, this formula will become indeterminate. A **?**, instead of a value, will appear beneath the name of all the elements whose script refers to the deleted element. Therefore, the user will have to remove the name of the deleted element from all formulas in the model.

Copying Part of a Model Within the Same Model

1. Select the desired part of the model.
2. Open the **Edit** menu, or click the right mouse button, and choose **Copy** (or press CTRL+INS**)**.
3. Move the pointer to the place where you want the copied elements.
4. Open the **Edit** menu, or click the right mouse button, and choose **Paste** (or press SHIFT + INS).

A copy of the selected part of the model is inserted in the new position. The new elements will have the same names as the originals with the addition of a final 2. All the connections between the copied elements are preserved. The scripts of the new variables and the values of the other copied elements will be the same as the originals and can be edited in the usual way.

However, if a variable is copied without one or more of the elements connected to it, that element will be missing from the **Connected Elements** list box in its script editing box and its value will assume the error value **[?]**.

The copied section remains selected after copying and can be moved by clicking on one of its elements and dragging to a new position.

Copying Part of One Model to Another

1. Select the desired part of the model.
2. Open the **Edit** menu, or click the right mouse button, and choose **Copy** (or press CTRL+INS**)**.
3. If the destination model is already open, click on a visible part of its window or select its name from the **Window** menu. Otherwise, open it from the **File** menu.
4. Open the **Edit** menu, or click the right mouse button, and choose **Paste** (or press SHIFT + INS).

The part of the model that was previously copied is inserted in the new model. To move it, click on one of the elements in the copied section which is still selected and drag it to the area where you want it. Once the desired position has been reached, release the mouse button.

Inserting a Shadow Element

1. Select the Shadow tool .
   OR: Choose the **Shadow** command from the **Model** menu.
   OR: Press ALT+F5.
2. Click on the element that is to be shadowed.
3. Move the pointer to the position where the shadow is to be inserted and click.

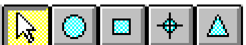
Notes: If the shadowing operation is done while holding down the SHIFT key, the element and its shadow change places: that is, the shadow is left in the original position while the primary element is moved.

A shadow can itself be shadowed. Any number of shadows of the same element can be present in a model.

Remember that a shadow variable cannot be the destination of connection arrows. The shadow assumes all the values of the primary element.

While a shadow cannot be placed in a different model from the primary element, the same result can be obtained by using the **Poke** and **Request** funtions in a variables script.

Connecting Another Element to a Variable

1. Select the Connect tool .

OR: Choose the **Connect** command from the **Model** menu.

2. Move the pointer to the element from which the connection is to start.

3. Press the mouse button and, keeping it pressed, drag the pointer to the second element.

4. Release the button once you have reached the second element.

Note: The destination of a connection must be a variable.

Deleting or Changing the Shape of a Connection Arrow

To select a connection arrow, click on the dot just behind the head of the arrow.

- Once the arrow has been selected, it can be deleted by pressing the DEL key.
- If it is to be moved, the procedure is as follows:

1. Position the pointer on the black dot behind the head of the arrow.
2. Press the left mouse button.
3. Keeping it pressed, move the mouse until the required curvature is obtained.
4. Release the mouse button.

Selecting Overlapping Connection Arrows

When a variable has a large number of incoming connections, several connection arrows may be superimposed. To select one of them:

1. Position the pointer on the black dot behind the head of the arrow.
2. Click the left mouse button repeatedly until the desired arrow is selected.

Associating a Document with the Model

1. Click the **Text** tool .
   OR: Choose the **Open Text** command from the **File** menu.

A text window will be opened containing a file with the same name as the currently open model but with the suffix .TXT.

2. Enter the text to accompany the model.
3. To save the text file, choose **Save** from the **File** menu.

Defining Variables

Once the model has been drawn, each variable element must have a script assigned to it, which defines it in mathematical terms.

The procedure is as follows:

1. Move the pointer to the variable for which you want to define the script.
2. Double-click. The dialog box for variables appears.
3. If the variable is to refer to itself in the script, select the **Self-Reference** check box. The name of the variable whose script is being edited is added to those available for inclusion in the script.
4. Define the initial value, if required, by clicking on the **Starting Value** text box and entering the value.
5. Move the cursor onto the **Script** text box and click.
6. Compose the script. You can do this by selecting the components (connected elements, functions and keywords) from the appropriate list box.
7. Choose the **OK** button when you have finished. DS Lab now checks that the syntax of the script is correct.

Another way of defining a variable is to use the **Edit Variables** command from the **Model** menu.

Entering Values in a Series

The values in a series can be entered in two ways:

- through the dialog box which is displayed by double-clicking on the element, or
- through the **Edit Series** command from the **Model** menu (or clicking the **Edit Series** tool).

The first method allows you to edit only one series at a time, while the second enables you to work on all the series at one time.

In both cases it is possible, by using the **Copy**, **Cut** and **Paste** commands, to use a series of values already contained in a DS Lab model or another Windows application such as a spreadsheet.

The values can also be imported from another Windows application and automatically updated every time they are changed by establishing a DDE link. This can only be done for one series at a time, by double-clicking on it and then clicking the **Paste Link** button in the dialog box. For further details, see ***Establishing Input Links from Other DS Lab Models*** and ***Setting Up a Permanent Import Link from Another Application***.

Speeding Up Simulations

When enabled, the **Update Values during Simulation** option updates the value of each element on screen at each step of the simulation. This slows down the simulation, since the screen is redrawn at each step.

To disable this option:

1. Choose the **Model setup...** command from the **Options** menu (or from the popup menu accessed by clicking the right mouse button with no elements selected).
2. Disable the **Update Values during Simulation** option by clicking its check box (leaving it empty, without an X in it).
3. Choose **OK** to exit from the dialog box.

The updating of the name of the current step in the drop-down list box on the horizontal toolbar also slows down the simulation. To disable this:

1. Choose the **Workspace...** command from the **Options** menu or the popup menu.
2. Disable the **Horizontal Toolbar** option by clicking its check box (leaving it empty, without an X in it).
3. Choose **OK** to exit from the dialog box.

Coloring an Element

1. Select the element to be colored.

2. Click the **Color** tool .

OR: From the **Options** menu or the popup menu, choose **Color...**

3. A dialog box appears. Click on the desired color.

4. Choose the **OK** button to confirm, or **Cancel** to abandon the operation.

Note: If you would like all new elements added to the model to have the same color, select the **Set as Default** option in the dialog box.

Defining the Format of Numbers

1. Select the element whose numerical value is to be expressed in the new format.
2. Select the Number Format tool .
   OR: From the **Options** menu or the popup menu, choose **Number Format...**
3. A dialog box will appear. Select the desired format.
4. Choose the **OK** button to confirm, or **Cancel** to abandon the operation.

Note: If you would like all new elements added to the model to use the same format, select the **Set as Default** option in the dialog box.

Converting One Element to Another

1. Select the element to be converted.
2. From the **Model** menu, choose **Convert**.
3. From the cascading menu, choose the type of element to which to convert the selected element.

The program warns you that the value(s) of the element (or the script, in the case of a variable) may be lost as a result of the conversion.

Note: The **Convert** command works with only one element at a time. It is not possible to use the **Convert** command if more than one element has been selected. Nor is it possible to convert a variable which has other elements connecting to it.

Setting the Print Size

In DS Lab you can define a fixed zoom level for printing. The procedure is as follows:

1. From the **Options** menu, choose **Model setup...**.
2. In the dialog box which appears, enable the **Show Page Breaks** option.

3. Click the **Zoom Out** or **Zoom In** tool until the desired scale is obtained.

4. Click the **Set Print Size** tool .

OR: From the **Model** menu, choose **Set Print Size**.

The current zoom level is now set as that to be used for printing.

5. Click the **All Pages** tool .

OR: From the **View** menu, choose **All Pages**.

The screen now displays a *print preview*. The vertical and horizontal lines represent page breaks.

To print, select **Print** from the **File** menu.

Defining a New Zone of the Model

1. Select the part of the model to be memorized as a view zone.
2. Click the center rectangle of the **Zones** tool .
   OR: From the **View** menu, choose **Zones...**.
   OR: Press **F5.**
3. A dialog box appears. Click on the **Defined Zones** text box and type the name of the new zone.
4. Choose the **Add** button. The name of the new zone is added to the list of defined zones.
5. Choose the **Close** button to return to the model.

Moving From One Defined Zone to Another

1. Click the center rectangle of the **Zones** tool .
   OR: From the **View** menu, choose **Zones...**, OR press F5.
2. A dialog box appears. From the **Defined Zones** list box, select the zone to which you wish to go .
3. Click the **Go** button.

Searching for an Element

1. From the **View** menu, choose **Find...**

    OR: Press F3, **OR** click the **Find** tool.
2. In the dialog box which appears, enter the name of the element you want to find.
3. If you want DS Lab to find matches to the whole name of the element click on the **Match Whole Element Name** option in the dialog box. When this option is disabled, DS Lab will return element names which contain the search text. For example, it will find the text *cost* in both the elements *Total Cost* and *Marketing Costs*.
4. Choose the **Find** button to start the search.

Matching elements will be presented one at a time and in the order in which they were created.

-To find the next occurrence of the selected name, choose **Next** from the **View** menu, or press the **F4** key.

Searching for *Undefined Variables*

DS Lab has a feature which facilitates finding variables with a non-existent or invalid script.

-From the **View** menu, choose **Undefined Variables**
OR: Press **F6**.

The first variable fulfilling the specified criterion will be selected and displayed in the center of the screen.

Choosing the same menu command (or pressing **F6**) again will display the next variable. When no more **Undefined Variables** are found, the following message is displayed: There are no Undefined Variables.

Searching for Variables with Run-Time Errors

The following command locates any variables whose script has returned errors during calculation.

- From the View menu, choose **Run-Time Errors**
  OR: Press **F7**.

The first variable with a Run-Time error will be selected and displayed in the center of the screen. Choosing the menu command again (or pressing **F7**) will display the next variable. When no more run-time errors are found, the following message is displayed: There are no Run-Time Errors in the current step.

Exporting Values to Excel

The simplest mode of interaction with Excel, is achieved by selecting the elements to be transferred to the spreadsheet, after the simulation has been completed, and then clicking the **Export to Excel** tool , choosing **Export to Excel** from the **Simulation** menu or the popup menu, or pressing F11.

This creates a new spreadsheet, in which the elements are listed by name in the rows and the simulation steps in the columns. Data is simply transferred from DS Lab to Excel.

If the simulation is repeated with different parameters and/or formulas for the variables, the same operations must be repeated to export the new data to Excel.

To summarize:

1. Perform the simulation.
2. Select the first element to be exported to Excel. If there is more than one, move to the second and click on it while holding down the SHIFT key. Repeat the procedure until all the desired elements have been selected.
   OR: If the elements are adjacent, they may be selected by holding down the left mouse button and dragging the pointer to the opposite corner of the rectangular area containing the elements.
3. Click the **Export to Excel** tool, press F11 or choose **Export to Excel** from the **Simulation** menu or the popup menu..

Note 1: In order for DS Lab to properly send the data to Excel, it must know what version of Excel you are running. To verify this, choose **Workspace...** from the **Options** menu and check that the version shown in the **Excel** section of the dialog box is correct.

Note 2: It is possible to have Excel automatically create a chart of the values exported. Follow these steps before exporting to Excel:

1. From the **Options** menu, choose **Workspace...**.
2. In the dialog box which appears, enable the **Create Chart** option.

Making Links with Excel

A more sophisticated mode of sending data to Excel, is to create a *permanent link with an Excel spreadsheet*. Once you have defined the destination worksheet, the elements to be exported and the cells in which to position them, the Excel worksheet will be updated automatically every time a new DS Lab simulation is carried out.

1. Run the simulation.
2. From the **Simulation** menu, choose **Link with Excel...** A dialog box appears.
3. Enter the name of the destination sheet in the **Worksheet** drop-down list box.
4. From the **Available Elements** list box, select the first element to be linked.
5. Choose **Append** to add it to the **Selected Elements** list below the high-lighted element.
OR: Choose **Insert** to add the element above the highlighted element.
Repeat steps 3 and 4 for each of the elements to be included in the link.
6. In the **Row** and **Column** boxes, specify the coordinates of the top left cell of the zone in which to place the exported values.
7. Choose the **OK** button to exit from the dialog box.

Note: To remove an element from the **Selected Elements** list box, select it then click the **Remove** button.

Establishing Input Links from Other DS Lab Models

1. Make sure the model from which data is to be imported is open.
2. Access the dialog box of the variable which is to import a value from another model.
3. Place the cursor in the Script box.
4. Choose **Request** from the Functions list box.
5. Specify the name of the model as the first argument and the name of the element whose value is to be imported as the second, with step notation if required.

If step notation is not used, the value is read from the current step in the source model.

Establishing Output Links to Other DS Lab Models

1. Make sure the model to which data is to be exported is open.
2. Access the dialog box of the variable which is to export a value to another model.
3. Place the cursor in the Script box.
4. Choose **Poke** from the Functions list box.
5. Specify the name of the model as the first argument and the name of the element to which to export the value as the second, with step notation if required.

If step notation is not used, the value is sent to the current step of the destination model.

In accordance with the logic of DS Lab, the value can be exported to any step of a series, or to the starting value (step 0) or past steps of a variable. (Here past step means a step earlier than **Simulation Start** for variables where the **Always Calculate** option is not selected).

Executing a Sub-Model

1. Make sure the model to be executed is open.
2. Access the script from which you want to run the model.
3. Choose **Execute** from the Functions list box.
4. Specify SYSTEM as the first argument, and CALCULATE (*model_name*) or RECALCULATE (*model_name*) as the second.

Importing Data with DDE (DDE Input Procedures)

There are three ways of importing data into DS Lab through DDE:

1. Exporting the data from a server application (in the Excel macro language, for example, by a **Poke** function); this makes a Cold Link from the server application to DS Lab.
2. Requesting the data from DS Lab with the **DDERequest** function; this too makes a Cold Link from the server application to DS Lab.
3. Using the **Paste Link** button within a variables script. This establishes a permanent Hot Link with the server application, though data is not updated automatically as soon as it changes, but only when the user decides to do so (see ***Updating DDE Input Links***).

N.B. In the case of variables and series, the elements **Starting Value** must be included in the array of linked values. For example, if a DDE link is established from the first three cells of a spreadsheet to a DS Lab series, the first cell will contain the **Starting Value**, the second the value for the first step and the third the value for the second step.

Exporting Data from a Server Application to a DS Lab Element
Requesting Data from DS Lab with the DDERequest Function
Setting Up a Permanent Import Link from Another Application

Exporting Data from a Server Application to a DS Lab Element

You can export a value from any Windows application to a DS Lab element by specifying the following DDE arguments:

- *Application* = "DS Lab"
- *Topic* = **"***name of an open model***"**
- *Item* = **"***name of the element in the model, with step notation if required***"**

With this technique, only one value at a time can be sent, and only to places where calculation is not involved; in other words, to series and to the starting (step 0) value or a past step of a variable. (Here past step means a step earlier than **Simulation Start** for variables where the **Always Calculate** option is not selected).

For example, to send the value of the A1 cell of the Excel worksheet SHEET1.XLS to the third step of the COSTS variable in the DS Lab model MODEL1, the following Excel macro can be used:

```
INITIATE("DSLab";"MODEL1.LAB")
POKE(Channel;"COSTS[3]";'SHEET1.XLS'!A1)
TERMINATE(Channel)
RETURN()
```

where Channel is the name given to the cell containing the first instruction.

Requesting Data from DS Lab with the DDERequest Function

This technique is the opposite of the previous one: DS Lab requests a value from another Windows application. The **DDERequest** function can only be used in the script of a variable, so values can be imported only by variables.

For example, to import the value of the top left cell of the Excel worksheet SHEET1.XLS to a DS Lab variable, the following script could be used:

```
DDERequest("Excel";"SHEET1.XLS";"R1C1")
```

Setting Up a Permanent Import Link from Another Application

When we speak of DDE links between applications, we usually mean this kind of link. In DS Lab, DDE links can be set up from the dialog boxes of both series and variables. All steps of a series can be linked. In the case of a variable, the starting (step 0) value and all past steps can be linked. (Here past step means a step earlier than **Simulation Start** for variables where the **Always Calculate** option is not selected).

The simplest procedure is to use the **Paste Link** button, but the link can also be typed in manually in the **DDE** edit box.

- To use the **Paste Link** button:

1. Run the application from which the data is to be imported.
2. Copy the required zone (a series of values).
3. Enter DS Lab.
4. Double-click on the variable to which the values are to be linked.
5. Click the **Paste Link** button.
6. Click the **Update** button to import the data.

As an example, if a DDE link with the first twelve cells of the Excel worksheet SHEET1.XLS is set up, the following link will be obtained:

Excel|SHEET1.XLS!R1C1:R1C12

Exporting Data with DDE (DDE Output Procedures)

There are three ways of exporting data from DS Lab to other applications using DDE:

1. Requesting data from a client application by a **Request** function (in Excel, for example, Request); this makes a Cold Link from the client application to DS Lab.
2. Exporting the data from DS Lab with the **DDEPoke** function. This too makes a *Cold Link* from the client application to DS Lab.
3. Using the **Copy**, **Copy Link Total** and **Copy Link Current** commands from the **Edit** menu. In this case a permanent Hot Link is established with the client application.

Requesting Data from DS Lab from Within a Client Application
Exporting Data from DS Lab Using the DDEPoke Function
Setting Up a Permanent Export Link to Another Application

Requesting Data from DS Lab from Within a Client Application

Any Windows application can import the value of a DS Lab element by specifying the following DDE arguments:

Application = "DSLab";

Topic = **"***name of an open modell***"**

Item = **"***name of an element in the model, with step notation if required***"**.

With this technique, only one value of a variable or a series (including the starting value of a variable) can be imported at one time from DS Lab. For example, to import the value of the Revenue variable for the current step in the MODEL1 model into Visual Basic, the following program could be used:

```
Sub Form_Clic ()
'LinkRequest Method Example
    Const NONE = 0, Hot = 1, Cold = 2
    If Text1.LinkMode = NONE Then
       Text1.LinkTopic = "dslab|model1"
       Text1.LinkItem = "Revenue"
       Text1.LinkMode = COLD
       Text1.LinkRequest
    Else
       Text1.LinkRequest
End If
End Sub
```

Exporting Data from DS Lab Using the DDEPoke Function

This technique is the opposite of the previous one: DS Lab sends a value to another Windows application. Since the **DDEPoke** function can only be used in the script of a variable, only the values of variables can be exported in this way. For example, to export the value of the *Revenue* variable at the current step from a DS Lab model to the first cell in the top left corner of the Excel worksheet SHEET1.XLS, the following script might be used:

```
DDEPoke("Excel";"SHEET1.XLS";"R1C1";REVENUE)
```

Setting Up a Permanent Export Link to Another Application

When we speak of DDE links between applications, we usually mean this kind of link. As well as the standard Windows DDE functions, DS Lab includes two additional options **Copy Link Total** and **Copy Link Current** in the **Edit** menu which provide a very simple way to make DDE links for all the step values of one or more elements (**Link Total**), or for a single step value of one or more elements (**Link Current**).

There are, therefore, two possible procedures. The first is the standard Windows method using the **Copy** command. This can export values from only one element at a time. The procedure is as follows:

1. Select the desired element (one only).
2. Open the **Edit** menu, or click the right mouse button, and choose **Copy**.
3. Enter the application with which you want to make the link.
4. Place the cursor where the data is to appear.
5. From the **Edit** menu, choose **Paste Link**.

The second procedure, using the **Copy Link Total** and **Copy Link Current** commands, is as follows:

1. From the **Options** menu, choose **Copy Link Format...** to define the format of the link (this will be different for each Windows application). The default format is that used by Excel.
2. Select the element(s) to link.
3. Open the **Edit** menu, or click the right mouse button, and choose **Copy Link Total** or **Copy Link Current**, depending on whether you wish to export the values of all the steps, or only those of the current step.
4. Enter the application with which you want to make the link.
5. Move to the zone where the data is to appear.
6. Open the **Edit** menu, or click the right mouse button, and choose **Paste** (or press SHIFT+INS).

Updating DDE Input Links

To update the data input via DDE Hot Links from other applications, there are two possible procedures:

- Enter the dialog box of each element you want to update by double-clicking on it, and choose the **Update** button.
  OR:
- Click the **DDE** tool , or from the **Simulation** menu, choose **Update DDE Links...** . This operation accesses a dialog box in which you can select one or more elements to be updated simultaneously.

Common Questions and Answers

When I ask for a report or a listing of the current step values, how do I *change the orientation of the steps* and elements so that the steps run along the top or down the side?

Choose the **Workspace...** command in the **Options** menu. At the bottom of the dialog box there is a section entitled **Default Settings for Copy** where you may select the orientation, the current step values listing and the format for copying to other Windows applications.

There are so many ways to *import and export data* in DS Lab; what is the best way?

A. Importing: To import data you will use only once and never update, use the **Copy** command in the **Edit** menu in the source application and the **Paste** button in the dialog box of the receiving DS Lab element.

If the data will be updated, use the **Copy** command in the **Edit** menu of the source application and the Paste Link button in the dialog box of the receiving DS Lab element. Remember to use the Update button to update the link from within the element dialog box, or the **Update DDE Links...** command in the **Simulation** menu to update all or selected links.

Please note that when using the **Paste Link** command to import a series of values into a DS Lab Series element, the first value in the series is placed in the **Starting Value** field, not the **First Step** field. This means that the user must include a zero as the first value in the series to be pasted if the **Starting Value** is to be zero.

To import data under script control (for example, when the import depends upon the result of a calculation), use the DS Lab **Request** function from within a script.

B. Exporting: To export calculated value(s) which you will not be updating, use the ***Excel*** tool when Excel is the destination application. If Excel is not the destination application, use **Copy** from the **Edit** menu in DS Lab and **Paste** from the **Edit** menu in the destination Windows application. (The actual command in the destination application may not be **Edit|Paste**; please refer to the documentation of that application for instructions on how to paste from the Windows clipboard.)

To export calculated value(s) which you will frequently update, use the DS Lab **Link with Excel...** command in the **Simulation** menu when the destination application is Excel. If Excel is not the destination application, use **Edit|Copy** from DS Lab and **Edit|Paste Special|Link** or **Edit|Paste Link** in the destination Windows application. (The actual command in the destination application may not be **Edit|Paste Special|Link** or **Edit|Paste Link**; please refer to the documentation of that application for instructions on how to paste from the Windows clipboard.) Be aware that not all Windows programs support DDE links from other applications

To establish Hot Links to another application (so that the values are updated automatically), use the **Copy Link Current** or **Copy Link Total** commands from the **Edit** menu to copy the values from DS Lab. In the destination application, use the **Edit|Paste** (not **Paste Link**) command to paste the DDE links.

Note: DS Lab must be instructed as to the format of the Hot Link required by the destination application. Each application requires its own format. The default format is set to Excel. If the destination application is not Excel, you must specify the link format in the **Options|Copy Link Format...** dialog box. Please refer to the documentation of that application for specifications on the format to use.

The difference between **Copy Link xxx/Paste** and **Copy/Paste Link** is that **Copy Link Current** and **Copy Link Total** create an individual DDE link for each step of each element. This is beneficial when the data will be split up and moved around within the destination application. Using **Edit|Copy** from DS Lab and **Edit|Paste Link** in the destination application creates a single DDE link for an array, which can not be broken up.

If you want to update a value in another application under program control of a script, use the DS Lab **Poke** function.

Remember to use the **Workspace** command in the **Options** menu to choose the orientation of the steps and elements so that the steps run along the top, or down the side. At the bottom of the dialog box there is a section entitled **Default Settings for Copy**: click on the option button corresponding to the orientation desired. You may also elect to copy the current step values, step headings and element names to the destination Windows application.

Please refer to the DS Lab manual for more details on all the functions described above.

In applications other than DS Lab, the actual commands for Copy, Paste, or Paste Link might differ slightly. Please refer to the applications manual for more details.

I have copied data into another application and the *step headings are different from what I saw in DS Lab*.

If step headings are time based (months for example) they will be copied over in the format defined by the International section of the Windows Control Panel.

I tried to copy some elements in DS Lab and paste them into another application without success, why?

The DS Lab Edit/Copy and Edit/Paste link in another application will not work if the first selected element is a comment. Re-select the DS Lab elements axcluding any comment elements, then copy and paste.

Why can I not copy link into Windows write?

Windows Write accepts links as graphical images. DS Lab supports only alphanumeric links to other applications.

Why does the *Excel tool* not work?

The Excel directory must be included in the PATH statement of your computers AUTOEXEC.BAT file. The Excel tool will not work if Excel is iconized; use Alt-tab to switch between applications. The Excel tool will not work if the user has an autoexec sheet in the excel statup directory or an autoexec macro in Sheet1. This is because Excel ignores the command sent by DS Lab and executes the macros. Remove the autoexec macro and/or remove the autoexec sheet from the startup directory when working with both applications.

When I use the *Export to Excel tool* Excel opens but nothing else happens, Why?

Answer: In Excel, use the Options|Workspace... command to open a dialog box. At the bottom of this dialog box there is a set of check boxes; one of them says "Ignore remote requests." Make sure that this box is NOT selected. It should NOT have an "X" in it.

I have used the Simulation/Link with Excel command but nothing happens in Excel, why?

You must first recalculate the model in order to cause the values to be exported to Excel. Simulation Link with Excel will not work if Excel is iconized.

Why does the *Link with Excel* tool not display any Excel worksheets?

You need to open Excel and have the destination sheets open.

Why does the *Link with Excel* Elements list box not display any tables?

DS Lab cannot link a table to Excel since it does not know which of the values in the table you want to link to Excel. If you want to link a value from the table to Excel, create a Variable which selects one of the values from the table based on any criteria you choose, and then link that variable to Excel.

When I try to *copy large models* I see an error message in the destination application about the whole file not being copied. Why?

There can be one of two reasons. The clipboard is limited to 64K of data; this is a Windows limitation. For Excel, you can circumvent this limit by using the **Simulation|Link with Excel...** command.

You may also have a problem in the destination application if you place the step headings in columns and the number of steps is greater than the number of columns available in the receiving application. Most spreadsheets are limited to 256 columns. A thirty year mortgage has 360 monthly payments, so it is easy to develop DS Lab models which have more steps than spreadsheets have columns. Try putting the step headings in rows, rather than columns.

Sometimes it seems to take for ever to *paste into another application*. Why?

Be patient, sometimes users dont realize that they are moving huge amounts of data between two applications. If it really does take forever, the problem may be in the receiving application. Many applications can handle only a limited number ofDDE links for example. This limit varies from application to application. You may need to contact the Technical Support department of the application in question to find out what the limit is, since most manuals do not document this.

I created *DDE links* into a spreadsheet and included the step heading and element names. When I changed the Element names in DS Lab they did not change in the spreadsheet, nor did the values change. Why?

Only the values have a live DDE link, if the titles or the step units are changed these changes cannot be carried over by DDE links (these must remain as a fixed reference for the DDE links to work). If you change the Element names the values cannot be updated because the DDE links refer to an element name which no longer exists.

How do I *print a report* of Values in DS Lab if I dont have a spreadsheet to export to?

You can copy values from DS Lab to Windows word processors as well as spreadsheets. We will give you an example of how to export to the Windows Write word processor which comes bundled with every Windows package. It can usually be found in your Accessories program group. If you cannot find it on your system refer to your Windows manual for loading and using this word processor. Most of the instructions given will apply to any Windows word processor such as Word for Windows, with modifications appropriate to the commands actually available. Make sure Windows Write (or your word processor) is open before starting.

In DS Lab, use the **Workspace...** command in the **Options** menu to open the **Workspace Setup** dialog box. At the bottom of the box you will find a section titled **Default Settings for Copy**. Select whether you want the step headings to be in columns or in rows. Be sure to consider the page orientation of the word processor and how the data will fit when you make your choices. Generally speaking, if you have few elements, and many steps, place the step headings in rows; if you have many elements and few steps, place the step headings in columns. You may also choose to include step headings, element names and initial values by clicking the option. Choose the **OK** button after you have made your selection.

Now you have two options. You can copy directly to the word processor or you can first preview what you copy by using the DS Lab report function. To preview and then export the values for the current step, click the Current Step report tool, use the mouse to select the cells you wish to copy, then use the **Edit|Copy** command. Switch to Windows Write and use the **Edit|Paste** command.

To preview then export selected elements for multiple steps, use the Selection arrow to select the elements you wish to copy, click the Report tool, use the mouse to highlight the cells you wish to copy, then use the **Edit|Copy** command. Switch to Windows Write and use the **Edit|Paste** command. In Windows Write, click on the **Document|Ruler On** command if the ruler is not displayed and use the tabs to align the data you have copied. In Word for Windows, you can also use tabs or highlight the values you have copied over and use the **Table|Convert Text to Table...** command to very quickly arrange the data

in a Word for Windows table.

In both these cases, using the report function first will preserve the formatting used in DS Lab. If step headings are time based (months for example) they will be copied over in the format defined by the International section of the Windows control panel.

Another way to copy data is to do so without previewing through the Report facility of DS Lab. This has the advantage of allowing you to set up hot links using the Paste Link command of the receiving word processor. However, formatting is not preserved.

I want to use some of the more complex functions. What is the best way to use *local variables* in a script?

Opening some of the sample files will show you how local variables are used in scripts. In general, the best way to write a script using multi-argument functions is to choose the **Script Edit** button from the Variable Dialog box. This will enlarge the Script edit box. From the **List** menu, choose the **Functions/Instructions** command. From the list box presented, choose the function you want to use. One click will show you the complete function with its syntax and an explanation of what it does. A double click will copy that function into the script where the cursor is located.

Once the function is in the Script edit box, use the **Edit|Copy** and **Edit|Paste** commands to copy each argument to a line (one argument per line) above the function itself. Place an equal sign after each argument then select an element or a keyword from the **List|Functions...** or **List|Connected Elements...** commands. This will turn each argument name into a local variable and assign a value from an element or a keyword to that local variable. Though slightly more time consuming at first, this technique reduces the risk of syntax errors within the function, makes it very easy to read the script at a later date, and provides for the use of the same local variable in a multi-function script.

Arguments which require either a step notation or an array notation reference cannot be represented by a local variable, and a local variable cannot be used in array or step notation.

My model uses Step Unit Day, Month, ... and during a *Paste Link* operation in Excel 4 my Step Headings appear as numbers. Why?

This is due to the fact that Excel 4 does not distinguish between dates and numbers in Paste Link. You have to manually enter the Format Number dialog box in Excel and assign the correct date format.

Functions that reference the *value of a variable at a future step* do not give the correct answer. Why?

This is because the values of the variable in future steps have not yet been calculated. The value will be correct for the last step of the model, once the model has been calculated. It is particularly important to bear this in mind when using functions such as the **PresentValue** and **FutureValue** financial functions, when the values are to be calculated as part of a variable. Such scripts can still be useful, for example when calculating the future value of payments made to date, since the value for future steps is assumed to be zero.

When *zooming out*, lines and text do not shrink proportionately, resulting in lines overlapping text. Why, and will this affect the printed output?

This is a limitation caused by Windows, Windows display drivers, and the way they handle text on screen. At very small magnifications what you see on screen will not correspond exactly to what will be printed. Experiment a few times to see how your particular screen driver and your particular print driver correspond.

I want to *select multiple elements* that are not contiguous. How do I do this?

Hold down the SHIFT key while you select the elements with the mouse.

I have many connections coming into one variable. How do I *select a particular arrow* to move it or delete it when they overlap?

Without moving the mouse, keep clicking until the connection you want changes from a solid line to a dotted line.

Why does a *Variable display NC* in front of the value for certain steps?

NC means Not Calculated. This shows up in Variables with the **Always Calculate** option disabled for those steps prior to the **Simulation Start** step. To edit the values in those steps, use the **Edit Past Steps...** command in the **Model** menu or the corresponding tool. This is an excellent way to place actual data in a forecast model for past periods, for which the actual value is known.

Why are the *columns available in a Table* element not the same number as the number of steps?

Table elements are not the equivalent of stacked series elements. Tables are meant to be used when a reference table is required which is disconnected from the steps in a model.

Why can I not *paste a selected range of cells into a series* from my spreadsheet when I am in Edit| Series... mode?

This happens when the range selected is a column of values instead of a row of values. Copy and transpose the range of values from a column into a row and then redo the copy paste command.

Can I do something to *speed up recalculations*?

The simplest thing to do is to use the **Model Setup...** command in the **Options** menu and make sure that the **Show Values during Simulation** check box is disabled. This will prevent DS Lab from updating the values on screen after every step is calculated and significantly speed the recalculation. The recalculation speed can be even further accelerated up by using the **Toolbar** check box in the **Options|Workspace...** dialog box to eliminate the horizontal tool bar from the display and thus prevent DS Lab from updating the current step box after every step is calculated.

Purchasing a math co-processor if your system does not have one will have a varying effect, depending on the complexity of the scripts you have and the functions you use. Generally speaking, the complex functions that are slowest to recalculate are the ones that would benefit most from a math co-processor.