## Contents

Entering data ... 6
Editing, selecting and filling cells ... 6
Entering numbers, text strings, date/time strings and error codes ... 10
Entering Unicode characters ... 13
Using cell references ... 13
Merging and splitting cells ... 15
Selecting cells, navigation, scrolling, key shortcuts ... 16
Entering formulas ... 18
Entering data: Formula Composer ... 21
Using array formulas ... 24
Inserting series ... 25
Inserting sequences/series using formulas ... 34
Drop-down lists ... 35
Using the Increment, Decrement and Operations commands ... 38
Spell-checking and adding Hunspell dictionaries ... 39
Generating passwords ... 44
Worksheet views ... 45
Splitting worksheet views ... 45
Row and column synchronization options ... 48
Pivot Tables ... 49
Using Pivot Tables ... 49
Creating Pivot Table Reports ... 55
Pivot Table Functions ... 57
Monte Carlo Simulations ... 59
Using Monte Carlo Simulations ... 59
Lookup functions ... 62
hLookUp() function ... 62
vLookUp() function ... 66
Match() function ... 70
Unique() function ... 75

Sort() function ... 77
Regular Expressions ... 80
Using regular expressions in formulas ... 80
Find & Replace - full text searches and replacing ... 82
Using the FILTER() function ... 88
Filtering tables ... 88
Filtering tables ... 92
Inspecting cells and finding certain cell types ... 94
Inspecting cells and finding certain cell types: Using the "Inspect Cell" pane ... 94
Formatting ... 94
Using data styles and numeric formats ... 94
Using Cell Style Palettes ... 96
Using Custom Styles ... 97
Hyperlinks ... 97
Images in Cells ... 98
Charts and images ... 100
Creating Charts ... 100
Charts in Cells ... 104
Password protection ... 106
Protecting and encrypting workbooks ... 106
Protecting workbook structure ... 106
Protecting and encrypting workbooks ... 107
Opening and saving files ... 107
Content and statistics ... 107
GS-Calc and ODF files ... 108
Opening and saving GS-Base databases (saved as *.xls workbooks) ... 109
Opening and saving Excel 2003 XML files ... 111
Saving files: PDF files ... 111
Text files ... 113
Text *.zip archives ... 116
xBase files ... 119
Printing workbooks ... 119
Printing and print previewing ... 119
Settings ... 120

Settings ... 120
Legacy compatibility ... 126
Custom DLL libraries ... 127
Adding functions in DLL libraries ... 127
gsclib.h ... 132
dllmain.cpp ... 134
gsclib.cpp ... 135
Scripting ... 143
Interfaces and methods ... 143
XBaseParams ... 144
TextParams ... 147
MergeParams ... 151
FormatParams ... 152
PrintSettings ... 167
Worksheet ... 176
Workbook ... 192
Application ... 210
Samples ... 219
COM Programming ... 226
Interfaces and methods ... 226
Samples ... 242
Support ... 258
Support ... 258
Copyrights ... 258
Copyrights ... 258
Mathematical Functions ... 259
Text Functions ... 276
Date/Time Functions ... 283
Financial Functions ... 298
Lookup & Reference Functions ... 304
Logical Functions ... 321
Informational Functions ... 325
Engineering Functions ... 330
Statistical Functions ... 339

Optimization/Solver Functions ..... 385
Matrix, Eq. Systems Functions ..... 393

# Citadel5

## GS-Calc

- A versatile and easy-to-use portable spreadsheet with advanced data processing capabilities.
- 12 million rows x 16,384 columns. Optimized for large data sets.
- ODF spreadsheet format or exceptionally fast and compact binary format as the default file format.
- Fast pivot tables supporting up to 12 million rows, implemented as pivot table formulas.
- Extensive match/look-up functionality for more precise and fastest possible searching.
- Multi-core calculations taking full advantage of multiple processor cores.
- Around 390 built-in formulas including specialized numerical functions: matrix decompositions; linear equation sets with improving iterations; least squares (weighted, constrained), regression with orthogonal polynomials; time series analysis; minimization; linear programming, integer programming and quadratic programming.
- Array formulas and 3D-formulas; pivot tables formulas; UDF functions accepting and returning arrays.
- Workbooks containing any number of hierarchically organized worksheets.
- Multi-pane worksheet windows with up to 100 freely sync views; advanced sync options.
- 2D/XY and 3D charts capable of handling very large amounts of data.
- Strong password protection and encryption; password-protecting workbook structures and individual cells.
- Hidden formulas and cells excluded from printing. Printing watermarks.
- Dual COM interfaces enabling you to use GS-Calc functionality in your own applications.
- Standard editing and formatting tools and options; optional Excel-, OpenFormula- and Google-Docs-compatible formula syntax, predefined numeric styles and user-defined styles.
- Clean installation (no registry entries are required); optional fully portable setup.
- Saving workbooks to PDF: saving entire workbooks (with the worksheet tree structure preserved), single worksheets, ranges and single charts; very small output files.
- Importing, exporting and editing text/CSV, OpenDocument *.ods, dBase III-IV, Clipper, FoxPro 2.x and Excel xml files. It can be also useful as a csv viewer/converter or dbf viewer/converter.
- Splitting and merging text and Excel xlsx/xls workbooks.

# Entering data

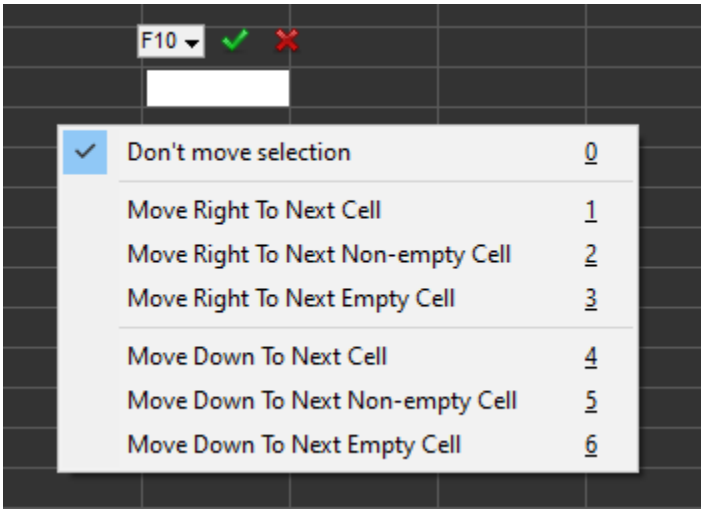
## Editing, selecting and filling cells

**To enter new cell contents**

Do one of the following:

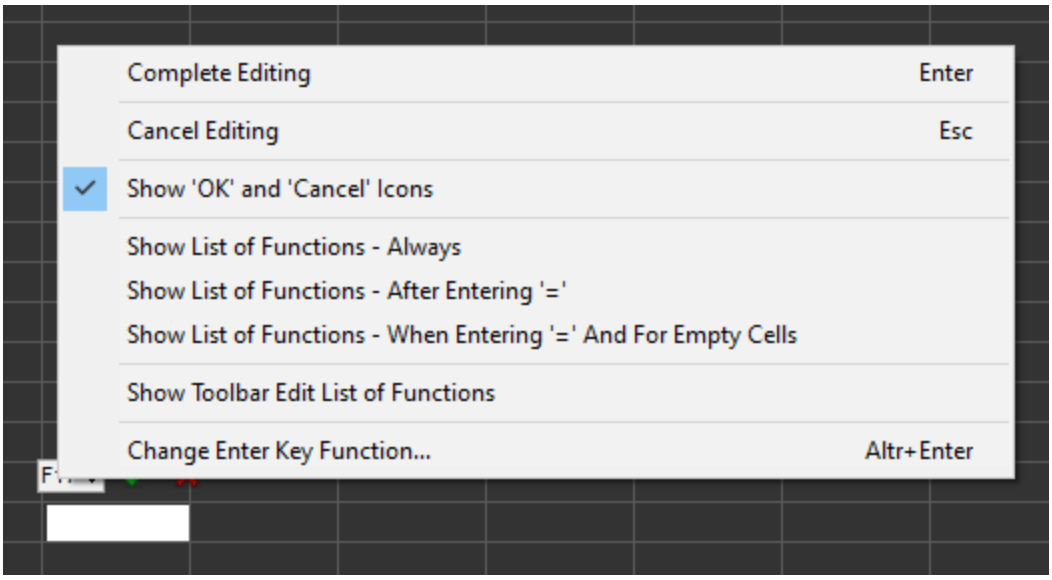
- Press **Enter**, **F2**, any letter/digit key or double-click a given cell.

  Note: The **Settings > Options > Editing > First Enter starts editing** can override the Enter key behavior globally. Similarly, the **AutoScroll Range** table toolbar button can override the Enter key behavior for individual cell ranges. The shortcut is **Alt+Enter** - it displays the corresponding context menu which changes the global settings or local AutoScroll settings if it's active in the current location:

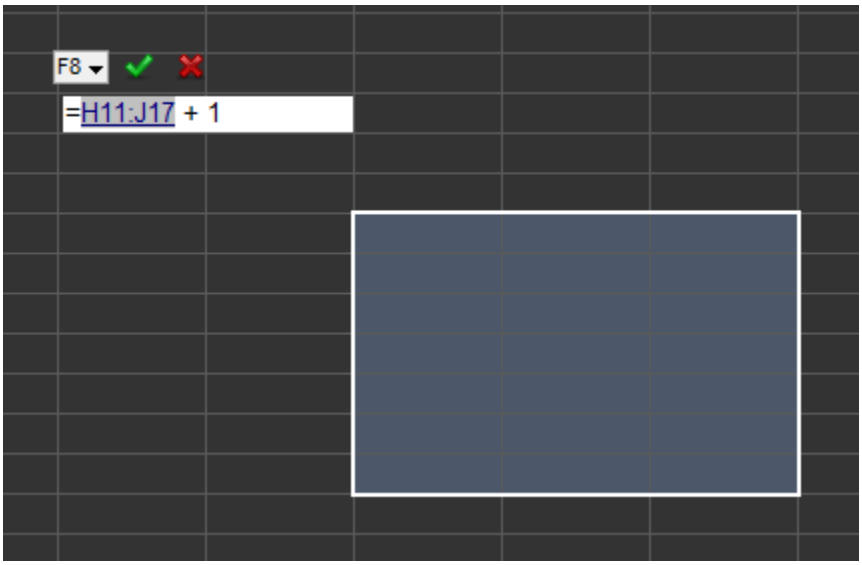



  You can enter up to 1024 characters. To accept changes, press **Enter** or any of the cursor keys or click another cell.
  To enter a new line in the edited cell, press **Ctrl+Enter**.
  To cancel changes, press **Esc**.
  If a given cell has the "Date" format, the "Date and Time Picker" control is activated instead of the usual plain edit control. For more information, see the "Date and Time Picker" section below.

  GS-Calc displays an additional button above the edited cell which:
  - shows the current location,
  - enables you to display a menu with options that control how the List of Functions dialog box is displayed:

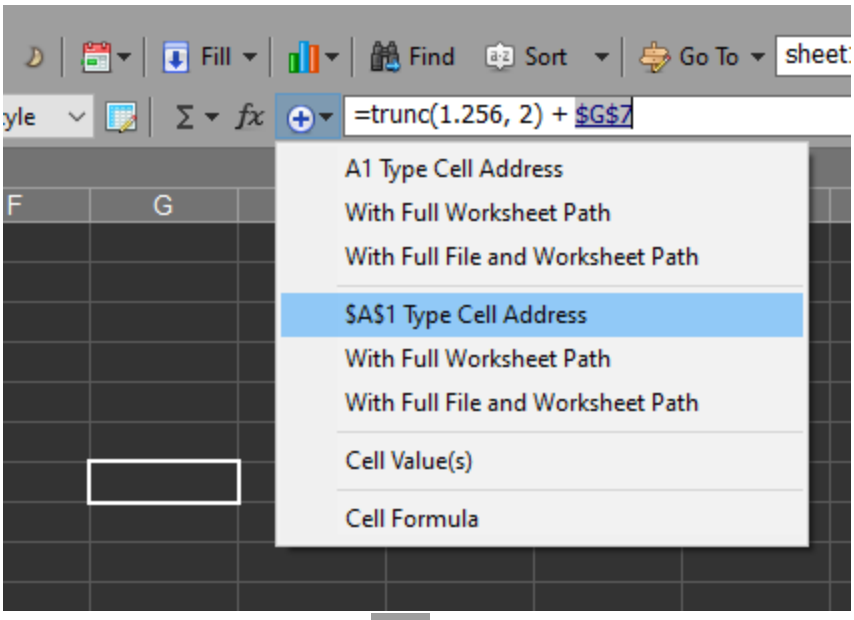



If the edited cell contains the leading "=" indicating a formula, clicking or selecting other cells causes adding their addresses to the edited text or replacing the address pointed by the current cursor/selection position within the edited cell.




To hide the List of Functions permanently, clearing all the menu check marks.

To compose a formula, you can use the **fx** edit field on the toolbar. The **+** and **fx** toolbar buttons enable inserting functions, various types of cell ranges, existing formulas, single values or values as arrays.




Clicking the "Enter" button  inserts the created formula in the current cell(s).

To add or replace cell references directly in the edited worksheet cell, select a given range using the left mouse button. Selected ranges or cells will be added to the edited formula or will replace the reference pointed by the edit cursor.

- Edit the cell contents in the **Formula** edit field on the toolbar. The cell contents is loaded by clicking the "Sigma" button and the new value is accepted by clicking the "Enter" button (or by pressing Enter). Previously entered values are available via the attached menu. To clear the history of the entered values use the **Settings > Clear Lists of Used Files and Data** command.

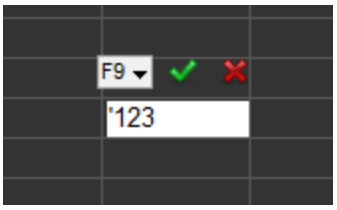
If you set some numeric or date/time format for a given cell, GS-Calc will try to interpret the data according to that format. If this fails, all the remaining formats will be tried out.

If a given cell is unformatted, GS-Calc will try out all subsequent standard formats. The date/time format has precedence over the fractional format.

If the **Auto-formatting** option is turned on, the matching format will become the new format of the edited cell unless the cell was formatted using a user-defined style.

To enter numeric values as text labels use one of the following:

- Press and hold down the **Shift** key before pressing **Enter**.
- Enter the apostrophe ' as the first character in a cell:




If a given text string actually contains ' as the first character, you need to double it. Alternatively you can also use enclosing ".

**To select a range of cells**

Use the SHIFT key with cursor keys or the left mouse button. You can also enter a given range in the "Go To" edit field on the main toolbar and press the Enter key or button. The list of the recently entered selections is attached to that Enter toolbar button.

**To delete cell contents and/or attributes**

Select a given cell or a range of cells and press Del. By default, GS-Calc displays a dialog box enabling you to choose which data should be deleted: numeric data, textual data, formulas, formatting, drop-down lists or comments. You can you turn it off and display that dialog only when Ctrl+Del is pressed.
Alternatively you can press Backspace to delete both the content and any other data at once.

**To move/copy cell contents or a range**

Place the mouse cursor on the top border of the selection and drag the entire cell/range to the desirable location.
If you want to copy the selection, press and hold down the **Ctrl** key.

**To paste cells choosing data from a list of recent Copy command results**

Use the **Settings > Options** command and specify the **Copy/Paste Level**. The valid range is 1-16. Subsequent Copy command results will be retained in memory up to that level in a cycling manner. The list of data available for pasting is attached to the **Paste** button.
To paste only textual data, press and hold down the **Shift** key when choosing a given item to paste.

**To fill a given range with some values, use one of the following:**

1. Copy the source cell or range, select the destination range and paste the copied data.
2. Select the destination range and enter the value in the toolbar combo box.
3. Select the destination range and use the **Insert > Series** command. See Inserting Series

**To use a drop-down list to enter data**

Use the Tools > List command and specify which list should be used for a given cell(s). The same list can be shared by any number of cells in any number of worksheets in one document. One workbook can contain up to 255 lists. The number of list items is unlimited.

**To use a "Date and Time Picker" control to enter data**

The control is displayed by default for each cell that is formatted as "Date" and doesn't have any drop-down list attached to it at the same time. This feature can be turned off by clearing the **Settings > Options > Use Date Picker** checkbox. The following shortcuts key be used with that control:

| | |
|---|---|
| Arrows keys | Changing the active control field and changing control field values |
| End and Home | Setting the maximum and minimum values for the current field of the control |
| Plus and Minus | Incrementing/decrementing the value of the current control field |
| Alt+Down | Displays the "Month Calendar" control. Same as clicking the "drop-down" arrow |
| PageUp, PageDown | If the "Month Calendar" control is active, moves to the previous/next month |

## Entering numbers, text strings, date/time strings and error codes

GS-Calc recognizes and uses the following basic data/cell types:

| Type | Examples | Comments |
|---|---|---|
| Numbers | 123<br>-123<br>123.09<br>1.23e+02 | The actual decimal and thousand separators depend on your system regional settings and the current **Settings > Locales** menu selection.<br><br>Max. positive value: 1.7976931348623158e+308<br>Min. positive value: 2.2250738585072014e−308<br>Precision: up to 15 digits |
| Text strings | abc<br>abc def<br>123 | Use the Edit > Convert commands to convert numbers to text strings, and vice versa.<br><br>When editing a cell you can force GS-Calc to treat the entered data as a text label by pressing and holding down the **SHIFT** key before pressing **ENTER**. |

| | | |
|---|---|---|
| | | GS-Calc automatically converts text strings to numbers in formulas that require numeric arguments, however such text strings must represent unformatted numbers with a period (**.**) as a decimal separator. For example:<br>="1.5" + 1<br>is a valid formula but<br>="1,500.00" + 1<br>will return the **#NUM!** error. |
| Date/Time strings<br>Period strings | Date/Time:<br>2006-01-31<br>2006-01-31T13:10:55<br>2006-01-31T13:10:55.123<br>2006-01-31T23:30:00+02:00<br>13:10:00<br>13:10:55.123<br>13:10:55-00:30<br>Periods:<br>P1Y2M3DT10H30M60S.000<br>(one year, 2 months, 3 days, 10 hours 30 minutes, 60 seconds, 0 milliseconds)<br>P2M4D<br>-P120D<br>PT23H34M | Date/time/period values can be specified either as date/time serial numbers or as generic date/time/period strings.<br><br>A date/time serial number is a floating point value representing a date between January 1, 100 and December 31, 9999. The integer part of that value represents days and the fractional part represents the time of the day. The value 1.0 represents December 31, 1899. Negative numbers represent dates prior to December 30, 1899.<br><br>A generic date/time string is a string that conforms to the www.w3.org specification for date/time/period data types.<br><br>For more information about using date/time/period strings as an alternative for serial numbers please see the **Formula Composer** window and the included samples.<br><br>**Note:** To maintain maximum portability, you should date serial numbers as currently most of other spreadsheet applications use that method of date/time encoding. |
| Arrays | A four-element 2x2 arrays:<br><br>{1, 2; "abc", "def"}<br><br>{1.00e+10; 2.00e+10; #N/A!; true} | Array elements can be unformatted numbers, text strings, Boolean values or error codes.<br><br>Generic (system-independent) GS-Calc settings activated with the **Settings > Locales > Generic** command require commas to separate columns (**,**) and |

| | | |
|---|---|---|
| | | semicolons (**;**) to separate rows.<br>If you choose to use the system regional settings and if your system requires commas to be used as decimal separators, use semicolons (**;**) to separate columns and backslashes (****) to separate rows. |
| Boolean values | true ( = 1 )<br>false ( = 0 ) | |
| Error codes | **#DIV/0!** Division by 0<br><br>**#NAME?** An invalid name was found in the specified formula<br><br>**#NULL!** The returned result is an empty intersection or an empty array<br><br>**#N/A!** No data/result available<br><br>**#NUM!** Invalid number, overflow or underflow<br><br>**#REF!** Invalid cell/range reference:<br>(+) a column or row number out of range,<br>(+) an array (result) exceeding worksheet dimensions,<br>(+) a non-existing worksheet or workbook,<br>(+) an inaccessible file/workbook | Error codes can be returned by formulas or can be entered 'as is'. |

| | | |
|---|---|---|
| | **#VALUE!** Invalid argument type or too long text string<br><br>**#IMPL!** The specified function is not yet implemented but the name is reserved for future use<br><br>**#SYNTAX!** Syntax error: incorrect number of arguments, missing parenthesis etc. | |

## Entering Unicode characters

Unicode characters can be edited/entered using the following methods:

1. Enter a sequence of 1-6 hexadecimal digits and immediately press **Alt+X**. Note that any preceding and not separated valid hexadecimal digits will be converted as well. This method can be used only when editing cells; it's not available in dialog boxes.

2. Press **Ctrl+Shift+U** then enter a sequence of 1-6 hexadecimal digits and immediately press **Space**.

3. Press and hold down **Alt**, then on the numeric pad press **+** and a desirable hexadecimal code, then release **Alt**. This method may require adding the following key in the Windows registry:
   HKEY_CURRENT_USER/Control Panel/Input Method/EnableHexNumpad
   The value of the above key should be set to "1" (entered as REG_SZ).

## Using cell references

| A1 notation | RC notation | Refers To |
|---|---|---|
| $A$1 | R1C1 | A cell in the 1st row and the 1st column.<br>This is the **absolute reference**: |

| | | |
|---|---|---|
| | | Inserting/deleting rows/columns and copying/pasting the cell don't change such references. |
| A1 | R[0]C[0]<br>R[1]C[2]<br>R[-1]C[-2] | A cell in the 1st row and the 1st column.<br>This is the **relative reference**: inserting/deleting rows/columns and copying/pasting the cell may change references in formulas to retain the proper links between cells.<br>For example:<br><br>• If you move/copy the '=B2' formula from the A1 cell to the C3 cell, it'll be replaced by '=D4'.<br>• If you move/copy the '=B$2' formula from the A1 cell to the C3 cell, it'll be replaced by '=D$2'.<br>• If you move/copy the '=$B2' formula from the A1 cell to the C3 cell, it'll be replaced by '=$B4'. |
| $1:$1<br>1:1 | R1 | The entire 1st row. |
| $A:$A<br>A:A | C1 | The entire 1st column. |
| $1:$5 | R1:R5 | Rows 1 to 5. |
| $A:$E | C1:C5 | Columns 1 to 5. |
| $A$1:$E$5 | R1C1:R5C5 | A range of cells specified by the top-left A1 cell and the bottom-right E5 cell. |
| Sheet1!$A$1<br>'Sheet1'!$A$1 | Sheet1!R1C1<br>'Sheet1'!R1C1 | The 'A1' cell from the "Sheet1" worksheet in the same document.<br>**Note:**<br><br>• Quotation marks delimiting the worksheet path/name can be skipped if that path doesn't contain any spaces or non-alphanumeric characters.<br>• If the name itself contains quotation mark(s), it must be enclosed in quotation marks and the inner marks must be doubled. |
| 'Folder1\Sheet1'!$A$1:$E$5 | "Folder1\Sheet1"!R1C1:R5C5 | The A1:E5 range from the "Folder1\Sheet1" worksheet in the same workbook. |
| 'Folder1'!$A$1 | "Folder1"!R1C1 | The A1 cells (returned as one column) from all worksheets in the "Folder1" folder in the same workbook. |

| | | |
|---|---|---|
| 'Folder1'!$A$1:$E$5 | "Folder1"!R1C1:R5C5 | All cells from the A1:E5 ranges (returned as one column) from all worksheets in the "Folder1" folder in the same workbook. |
| 'd:\documents\[sample.gsc]Sheet1'!$A$1<br>'d:\documents\[sample.gsc]Sheet1'!A1 | "d:\documents\[sample.gsc]Sheet1"!R1C1 | The A1 cell from the "Sheet1" worksheet in the d:\documents\sample.ods workbook.<br>**Note:**<br>• If the reference includes a file path, the file name must be delimited by square brackets. The file path can be either absolute or relative. Relative paths (which is also true for empty/not specified paths) are resolved based on the path of the document where that reference is placed or - if the document is unsaved - the path of the last open document or the default document folder. |
| "d:\[sample.gsc]Folder1"!$A$1:$E$5 | "d:\[sample.gsc]Folder1"!R1C1:R5C5 | All cells from the A1:E5 ranges (returned as one column) from all worksheets in the "Folder1" folder in the d:\sample.ods workbook. |

## Merging and splitting cells

You can use the **Format > Merge Cells...** command to &quotmerge" adjoining cells. When a group of cells is merged, the contents of the top-left cell of that group is displayed within the entire merged area. The remaining merged cells are treated as covered and they are not displayed.

Merging doesn't change how the merged cells are referenced in formulas, how they are copied, edited or formatted. For example, to edit the contents displayed within the merged group or format it, you must apply that action the top-left left cell of that group.

Merged ranges can be of any size and they cannot overlap.

The **Merge Cells** dialog box enables you to specify the following additional options:

**Merge selected cell contents and place it the top-left cell**

The contents of all cells within the merged group will be concatenated and copied to the top-left cell. Merged cells are separated by a single space and lines of cells are separated by new-line characters. Merged and copied cells are deleted.

**Alignment options**

You can adjust any of the current alignment options for the top-left cell. To modify them later, select the top-left merged cell and use any of the style editing command.

**Save As Defaults / Reset Defaults**

Click the "Save As Defaults" button to save the specified alignment options for subsequent cell merging operations. If there are already some saved/predefined alignment settings, the title of this button changes to "Reset Defaults", which causes this dialog box to use the current format of the top-left cell of the merged selection.

To split (merged) cells, select a given group of merged cells (or at least the top-left cell) and use the **Format > Split** command.

To find all merged cell groups, use **View > Inspect Cells** command to open the **Inspect Cells** pane, click the **Find & Add** button and choose the **Merged Cells** command.

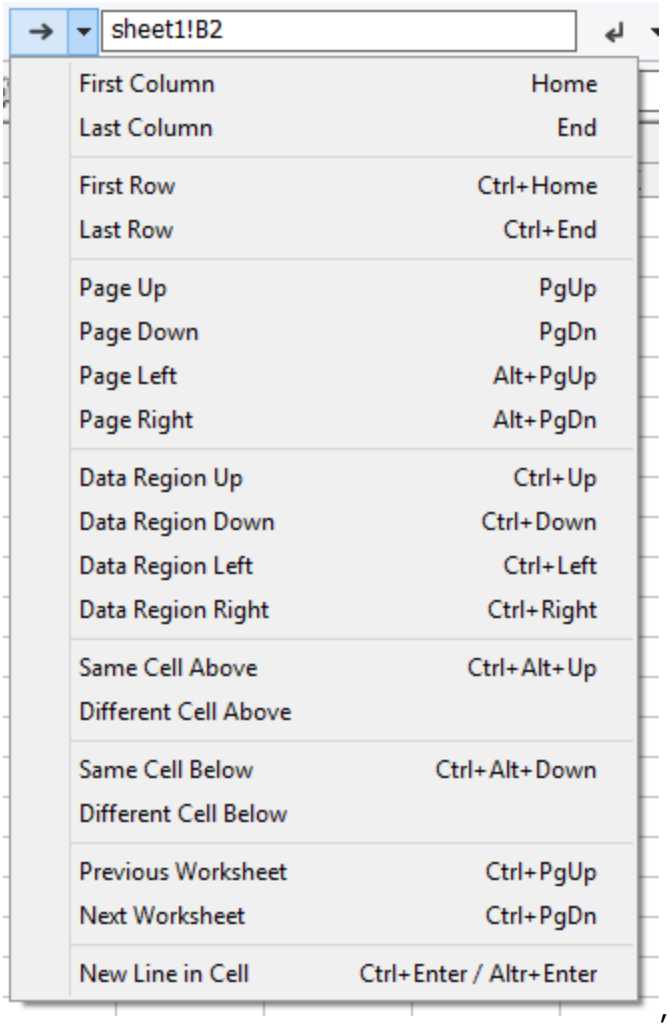
## Selecting cells, navigation, scrolling, key shortcuts

**To select a range of cells**

Do one of the following:

- Use the mouse cursor.
- Press **Shift** and use any of the scrolling keys/shortcuts.
- Enter the desirable cell or range in the **Location** field on the toolbar and click the "Enter" button or press **Enter**.
  You can also bookmark any location. To add a new bookmark click the "*" button. To jump to the next/previous bookmarked location, use the "<*" and "*>" toolbar buttons or manage the bookmarks via the **View > Bookmarks** pane.
- Press **Ctrl+Space** to select the entire current column
- Press **Shirt+Space** to select the entire current row

**Workbook window navigation keys/shortcuts**




**Print preview navigation keys/shortcuts**

Use the standard cursor keys (Up, Down, Left, Right, PgUp, PgDn, PgUp+Alt, PgDn+Alt). The Ctrl+PgUp and Ctrl+PgDn shortcuts cause jumping to the previous/next preview page.

**Other shortcuts**

| | |
|---|---|
| **Ctrl+B** | Bold font |
| **Ctrl+I** | Italic font |
| **Ctrl+U** | Underline font |
| **Ctrl+T** | Strikeout font |
| **F6** | Next pane in the following order:<br>1. Worksheet tree (if it's visible)<br>2. Worksheet window<br>4. Inspect cells (if it's visible) |
| **Ctrl+F6** | Opening/closing worksheet tree pane |

**To jump to the desirable cell/range**

- Use the **Location** edit field as explained above.

**To use the 'auto-scroll' function**

Click the worksheet window using the mouse wheel button and choose the desirable scrolling speed and direction by moving the mouse cursor around the clicked point.

## Entering formulas

**To enter/edit a formula**

Enter a desirable expression including the leading "=".
For example, to sum up A1, A2 and A3 and place the result in A4, enter in the A4 cell:

=A1 + A2 + A3
or
=sum(A1:A3)

to count numbers greater than 2 in the range A1:A100 and place the result in B4, enter in the B4 cell:

=countIf(A1:A100, ">2")

Other examples:

="abc"
={1, 2, 3; "abc", " 'def' ", ' "ghi" '}
=1 + 2
="a" & 'b' & 3
=fv(0.75%, 36, -500, -5500, 0)
=LProg({2,2;1,2;4,0}, {1;1;1}, {14;8;16}, {2; 4},0,,)
=sum(a1:b2, c3, 5:5, 10, sum(20, 30, 40))
=sum(A1:B2, C3, E:E, 10, sum(20, 30, 40)) =index(timeSeries($A$1:$A$100, 3, 10, 0.9), , 1)

Numbers used as arguments cannot be formatted. For example, the expression =$1,000.00 + 1 is invalid. Text strings should be delimited either by single or double quotation marks.

Formulas can contain circular references. For example, if the A1 cell contains '=B2 + 1', B2 contains '=C3 + 1' and C3 contains '=A1', then each time you update the workbook, GS-Calc will be performing recursive/circular recalculation for those three cells. The number of such circular updates depends on the **Recursion level** parameter specified in the **Options** dialog box. Possible values are from 1 to 255.
Circular cell markers are displayed if that level >= 2. To list all formulas with circular references, use the **Find All > Circular Formulas command on the Find toolbar.**

To browse all available formula categories and examples, use the **List Of Formulas** dialog box that is displayed when editing formulas or after clicking the **Formula** edit fields on the toolbar.

**Note: Square brackets [,] in parameter lists denote parameters that are optional and can be omitted. For example:**
**LProg(A, s, b, c, vector, [epsilon], [m])**

If your system (setting) uses commas as decimal separators, use semicolons (**;**) to separate function arguments or - if you prefer to use system independent (US) settings - select the generic locales via the **Settings > Locales > Generic** command.

**Available operators:**

| Operator | Operation | Comments | Precedence |
|---|---|---|---|
| **=** | Equal | Compares numbers or text strings (the comparison is not case-sensitive).<br>Example: A1=4 , B2="abc" | 9 |
| **<** | Less than | Compares numbers or text strings (the comparison is not case-sensitive).<br>Example: A1<4 , B2<"abc" | 9 |
| **>** | Greater than | Compares numbers or text strings (the comparison is not case-sensitive).<br>Example: A1>4 , B2>"abc" | 9 |
| **<=** | Less than or equal | Compares numbers or text strings (the comparison is not case-sensitive).<br>Example: A1<=4 , B2<="abc" | 9 |
| **>=** | Greater than or equal | Compares numbers or text strings (the comparison is not case-sensitive).<br>Example: A1>=4 , B2>="abc" | 9 |
| **<>** | Not equal | Compares numbers or text strings (the comparison is not case-sensitive).<br>Example: A1<>4 , B2<>"abc" | 9 |
| **+** | Addition | Adds numbers. | 8 |
| **-** | Subtraction | | 8 |
| **&** | String concatenation | Merges text strings.<br>Example: A1 & "abc" , "a" & "b" | 8 |

| | | | |
|---|---|---|---|
| ***** | Multiplication | Multiplies numbers. | 7 |
| **/** | Division | Divides numbers. | 7 |
| **^** | To the power of | Calculates the power of. | 6 |
| **-** | Negative | Example: -A1 | 5 |
| **%** | Percent | Specifies a number entered as a percentage.<br>Example: 12% | 4 |
| **_** | Intersection | Specifies an (reference to) intersection of two ranges.<br>Example: 5:5_b:b , a1:a5_a5:a8 | 3 |
| **!** | Worksheet reference | Specifies a reference to another worksheet/folder.<br>Example: sheet2!(5:5_b:b) | 2 |
| **:** | Range of cells | Creates a range.<br>Example: r1c1:r5c5 , A1:B5 | 1 |

To recalculate formulas

By default, changing any cell contents updates all formulas and charts in all worksheets. Use the Tools > Automatic Updating command to turn on/off automatic updating.

To count formulas entered within the current selection, in the current worksheet or in the entire document

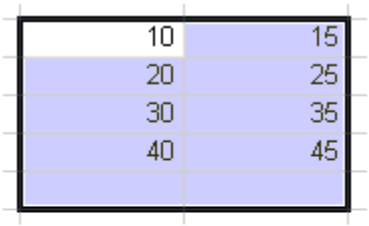
Use the File > Statistics command.

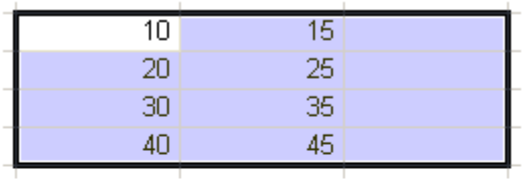
To find a sum, min/max/mean value etc. of a given range of cells

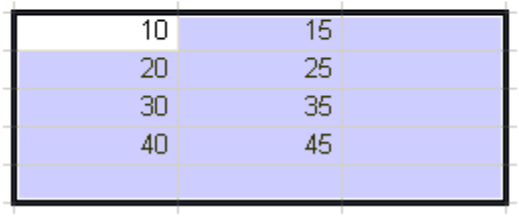
Select that range. The sum, minimum, maximum and mean values are displayed in the 2nd status bar pane.

or

1. **Select a cell range as shown below:**



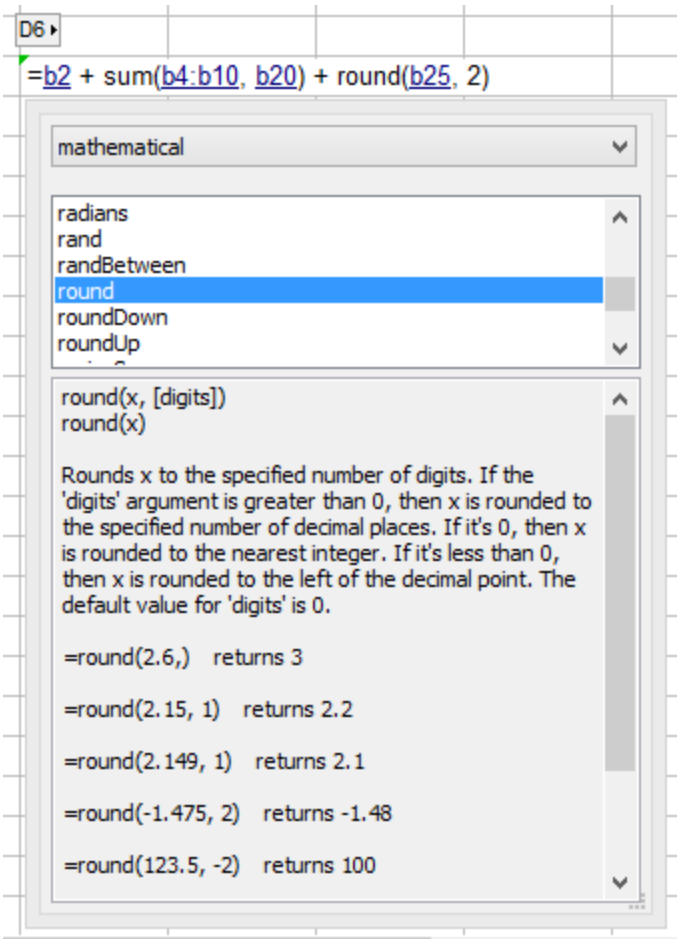

2. **Click the Formula button on the toolbar and choose the desirable function.**

## Entering data: Formula Composer

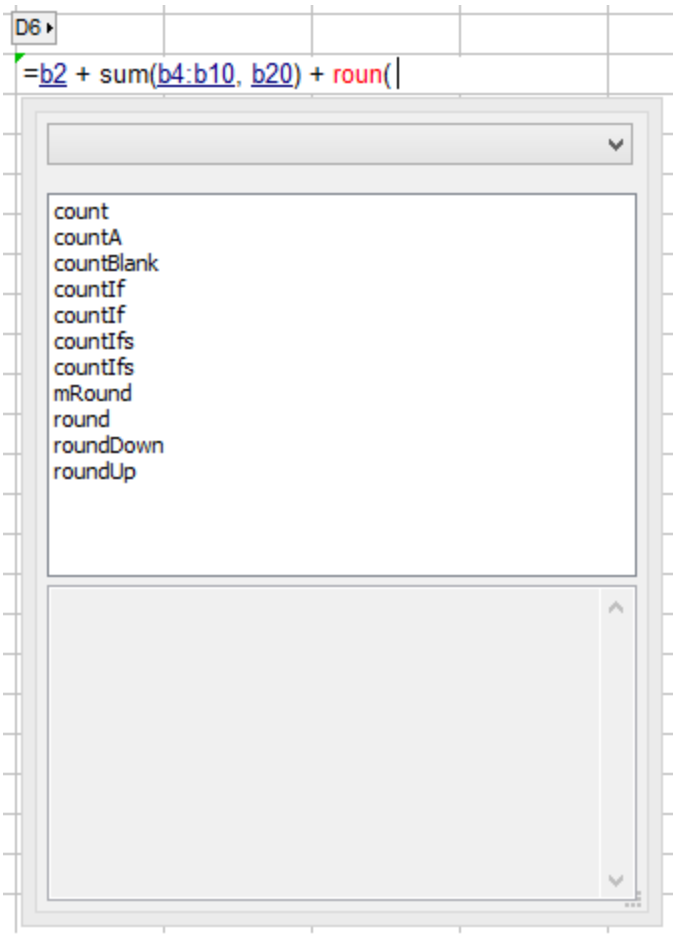
Use the List Of Functions dialog box to make it easier to enter the built-in functions.

The List Of Functions dialog box is displayed by default when editing a cell and after activating the Formula edit field on the toolbar. The **Insert > Formula** command simply starts editing of the current cell.

The List Of Functions dialog box can be resized. To resize the field containing descriptions and the list of functions, drag the splitter line separating both parts of that dialog box.

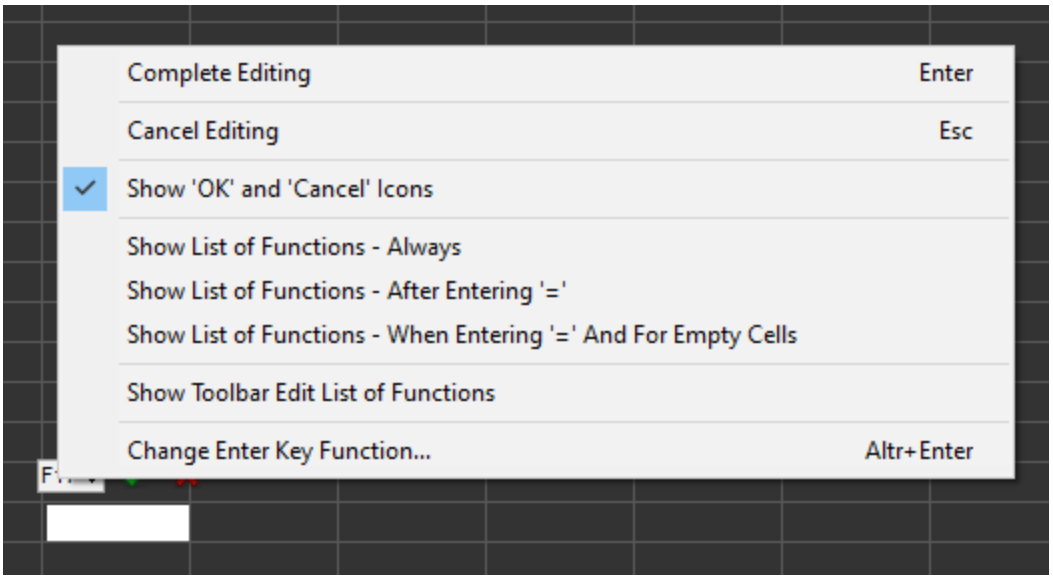



If you enter an incorrect name of a function including the starting '(', the dialog box will list possible correct functions.




To insert a given name, double-click it or press Enter. The name is inserted at the cursor position within the edited cell. If there any selection, the selection is replaced.

The above List Of Functions dialog box is displayed by default if a given cell contains the initial '=' indicating a formula. You can also select other options:




## Using array formulas

An array formula is a formula returning an array of values (of various types) instead of a single number, text string or error value. Any non-array formula that requires at least one non-array argument can be transformed into an array formula by specifying some range/array as that argument. For example:

```
=A1            ->  =A1:A5

=sqrt(E5) + 1  ->  =sqrt(E5) + {1; 2; 3; 4; 5}

=sqrt(E5) + 1  ->  =sqrt(E5:J10) + 1
```

If there are more such arguments, all of them must represent ranges with the same number of columns and rows. For example:

=sqrt(E5:F6) + {1, 2; 3, 4}

is correct, but

=sqrt(E5:F6) + {1, 2}

will return the #VALUE! error. The dimensions of the returned array match those of the formula arguments.

You can use array formulas whenever you need to generate a series of numbers/labels (as when creating chart data series) or just to simplify some calculations. For example:

to count numbers from the range A1:B100 that are either greater than 100 and smaller than 200 or greater than 250 and smaller than 300, use the following formula:

=sum(((A1:B100 > 100)*(A1:B100 < 200) + (A1:B100 > 250)*(A1:B100 < 300)))

and to sum such values

=sum(((A1:B100 > 100)*(A1:B100 < 200) + (A1:B100 > 250)*(A1:B100 < 300))*A1:B100)

to convert and display errors as empty strings:

=if(isError(b1:c5), if(errorType(b1:c5)=0, "", ""), b1:c5)

(Note: If ranges are used, all **if()** function arguments must represent ranges of the same size.)

to sum values from a range, ignoring errors:

=sum(if(isError(b1:c5), errorType(b1:c5)=0, b1:c5))

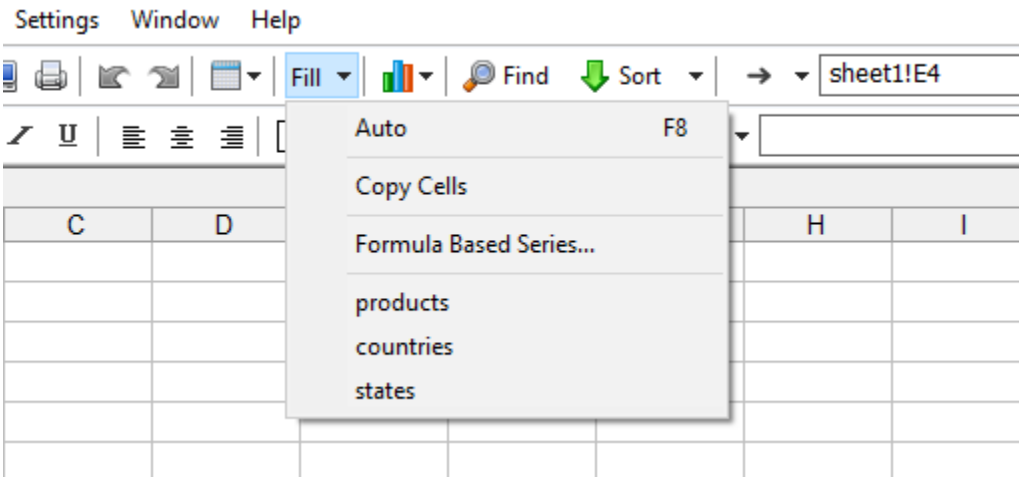
## Inserting series

### Inserting an automatic series (F8)

This will fill a given range of cells with values that can be numbers/dates, formulas, built-in lists (day/month names), custom lists.
The type of the series is determined automatically and it depends on the entered initial one or two cells of that series.

If the first one or two cells don't match any of the above series types, a simple "copy" operation is performed.
If the first two cells are separated by some gap of empty cells, this gap will be duplicated within the rest of the inserted series.

- Built-in list (day/month names) items can be substrings (words) within cell strings. For example, if the first cell contains "departure on Monday", the rest of the inserted series will be in the same form: "departure on Tuesday" etc.
- Custom list items can also be substrings (words) within cell strings, but to be recognized, instead of the "F8" command such series must be used explicitly by its name, clicking the "Fill" button.




**Built-in numeric, date/time and formula series**

If you specify only one element of the series, GS-Calc will use the following predefined default increments to calculate subsequent elements:
1 for numbers, 1 hours for time values and 1 day for date values, 1 column/row references in formulas.

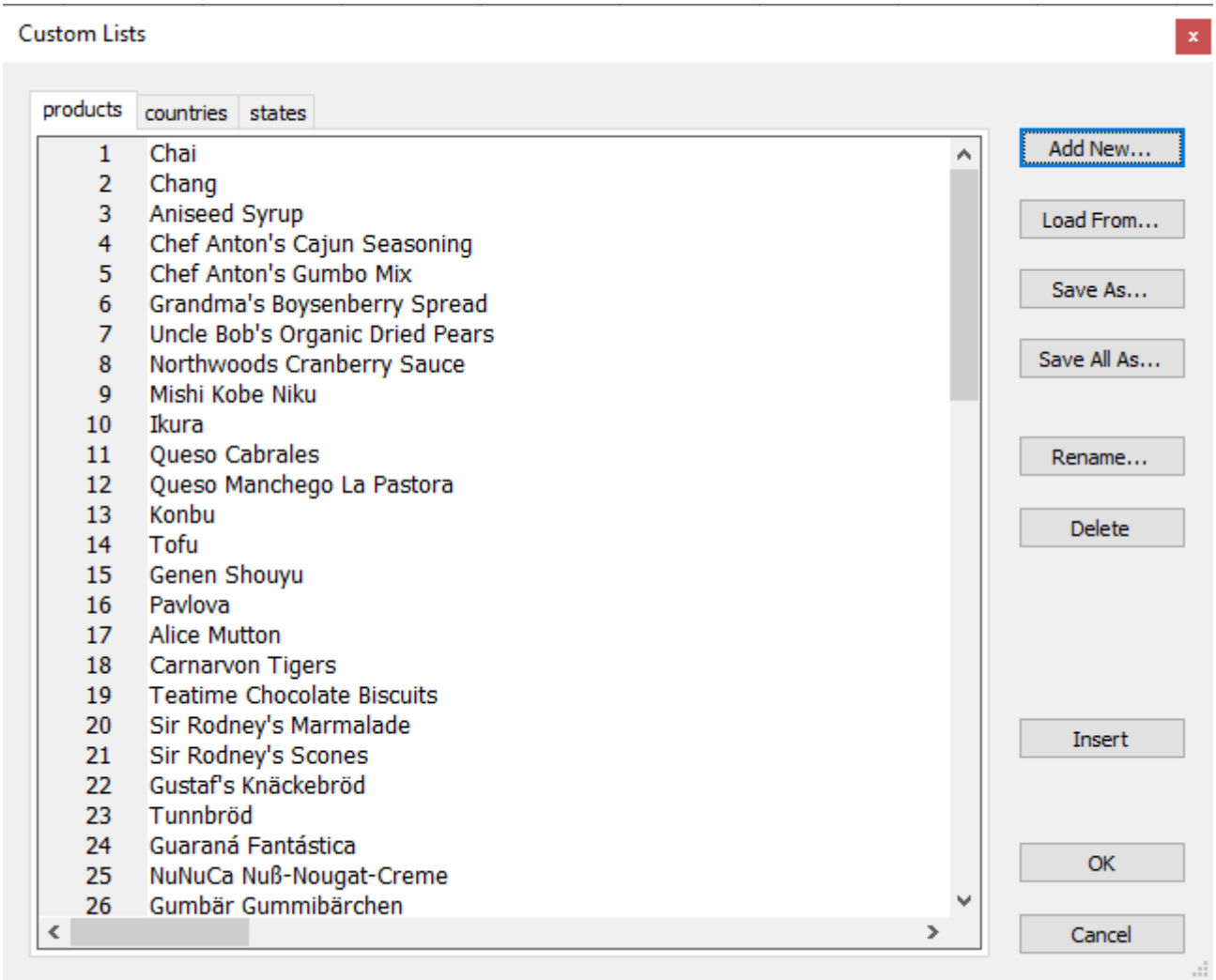
Both relative and absolute cell references can be modified depending on the current settings in the **Settings > Options** window.

**Choosing a specific custom list to insert series.**

Click the "Fill" toolbar button and choose the desirable list or use the "Manage" dialog box and use the "insert" button.

If you choose a specific custom list by name:

- Custom list items can also be substrings (words) within cell strings, but to be recognized, instead of the "F8" command such series must be used explicitly by its name, clicking the "Fill" button.
- The target cell range doesn't have to contain the initial 1-2 series elements. If it's empty, the series will simply start from the 1st list item.

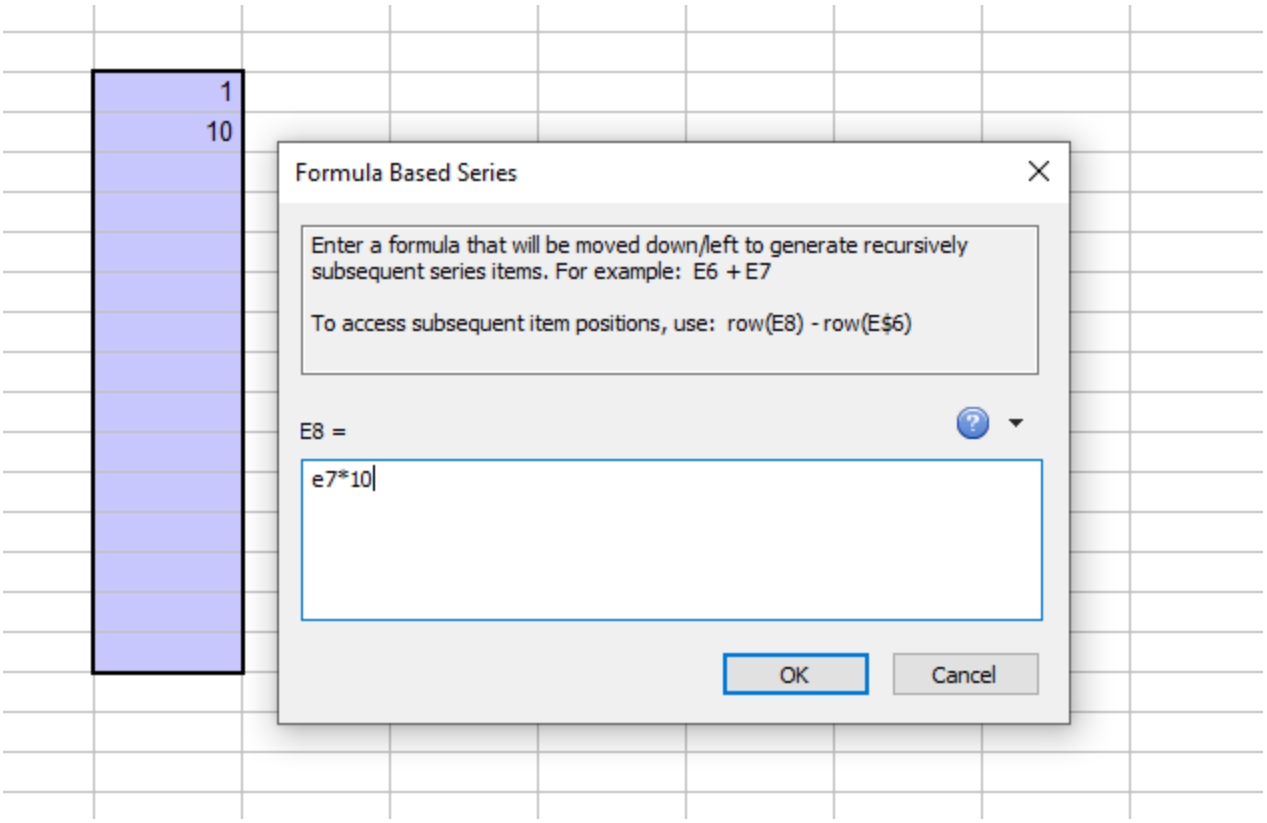



**Inserting formula-based series**

This enables you to define any type of data series including those referring to resources elsewhere in your workbooks.

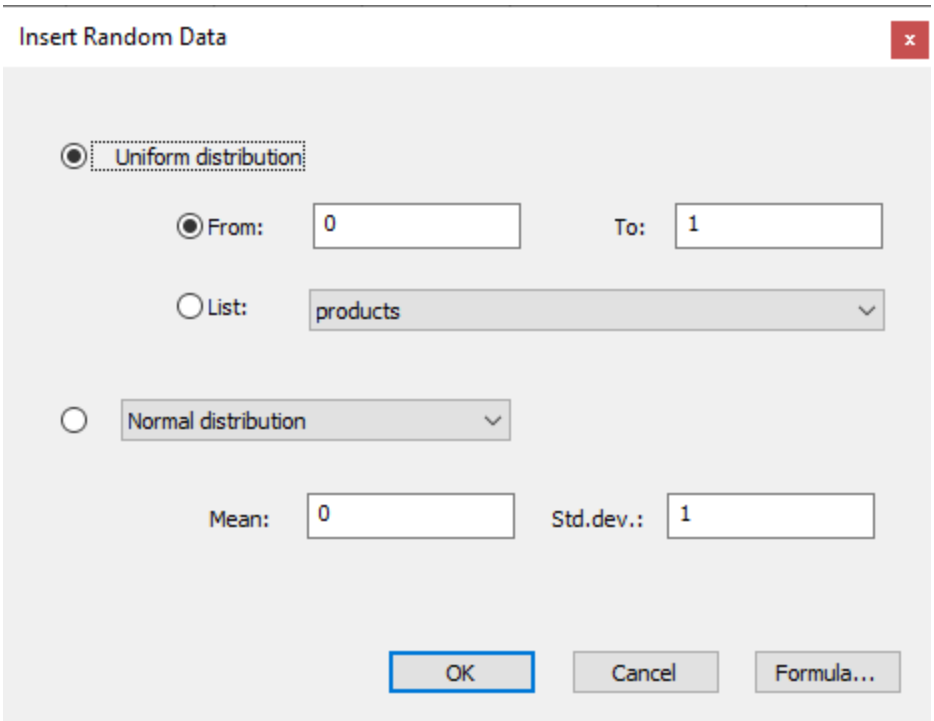
For this series type you need to define a formula that determines the first item of the series. Then GS-Calc will accordingly update references in this formula to generate values in subsequent cells of the target cell range. Such a formula can either use the preceding value to generate the next one or each value can be calculated directly by using the item position within the series.

Used formulas are saved as a "recently used" list and references are updated automatically when applied to another worksheet location.



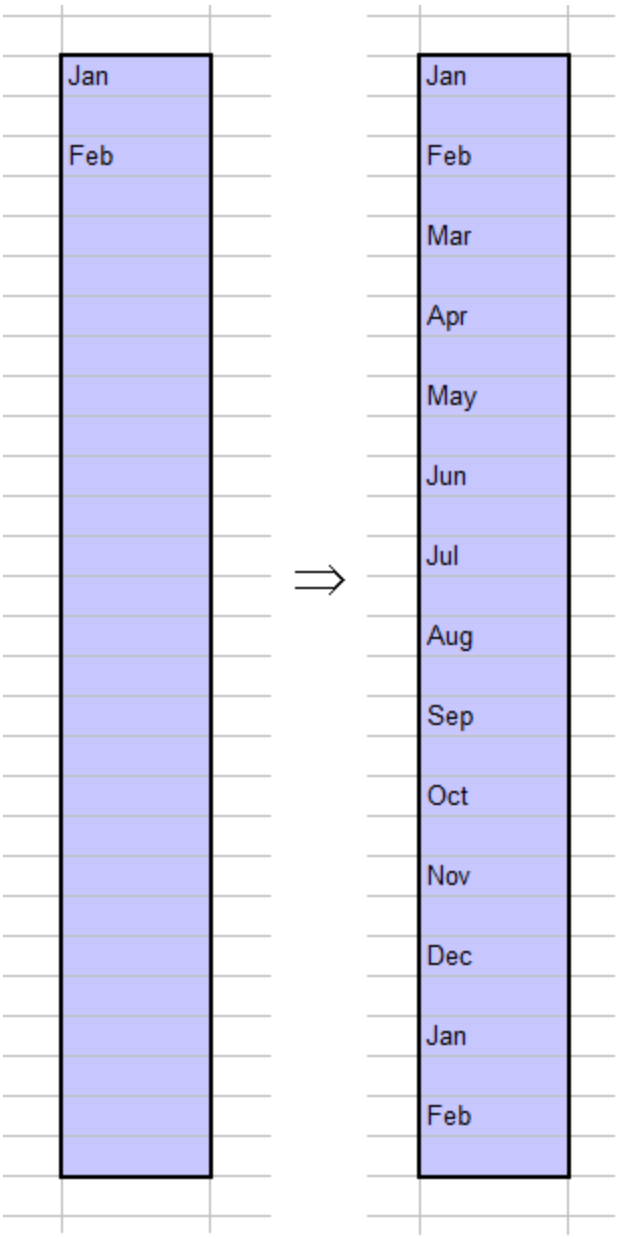

**Inserting random series**

You can use several popular distribution types. Additionally, for the uniform distribution you can choose to create a series of randomly selected custom list items.

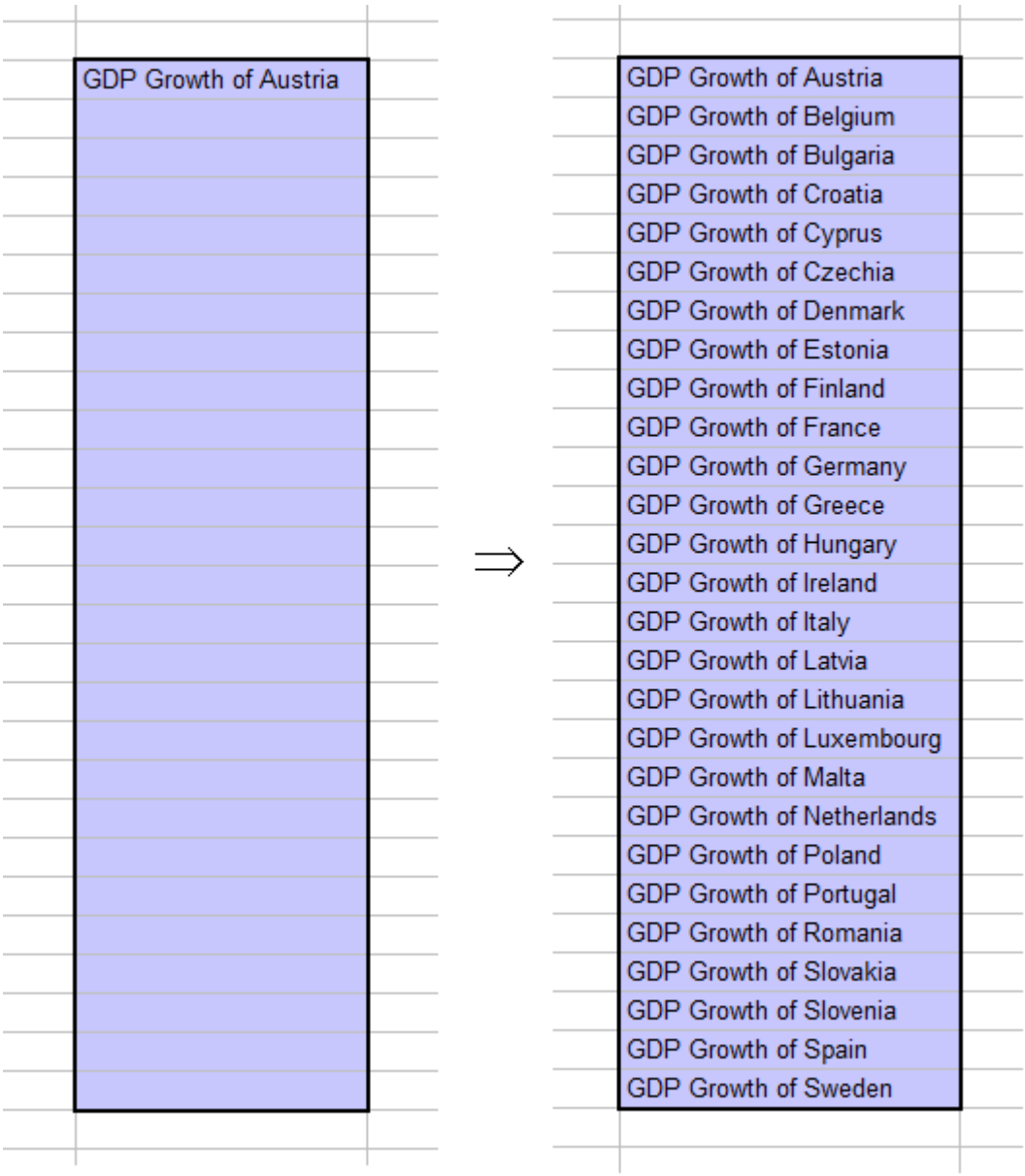



**Examples**

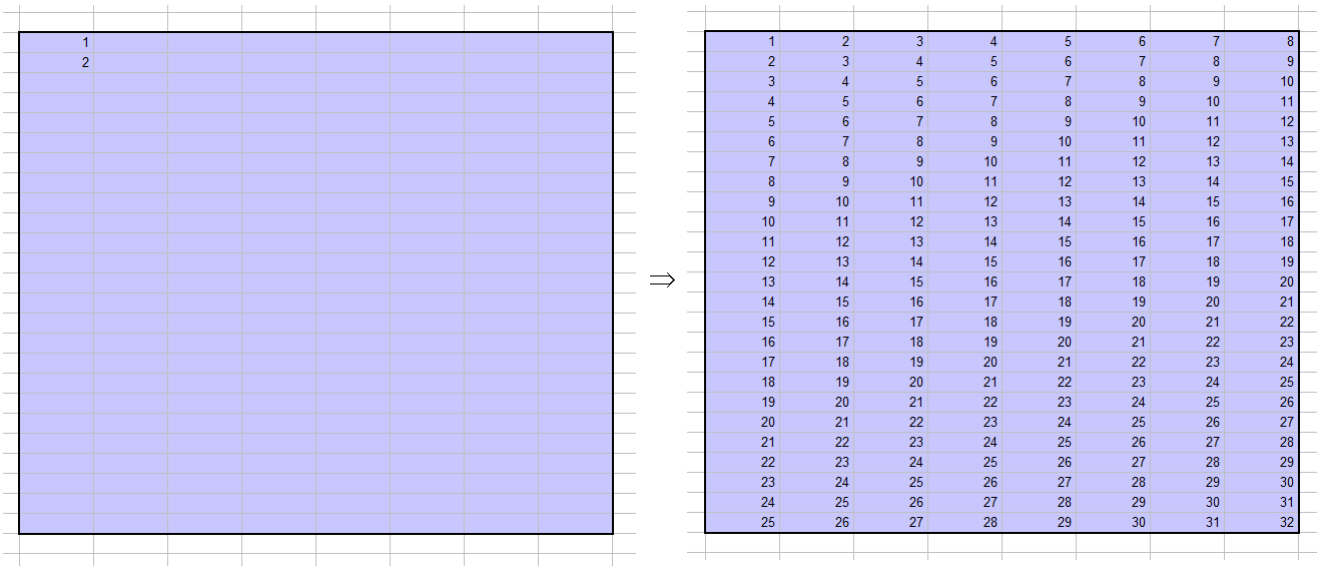
**Inserting a built-in series of month (system-dependent) names:**



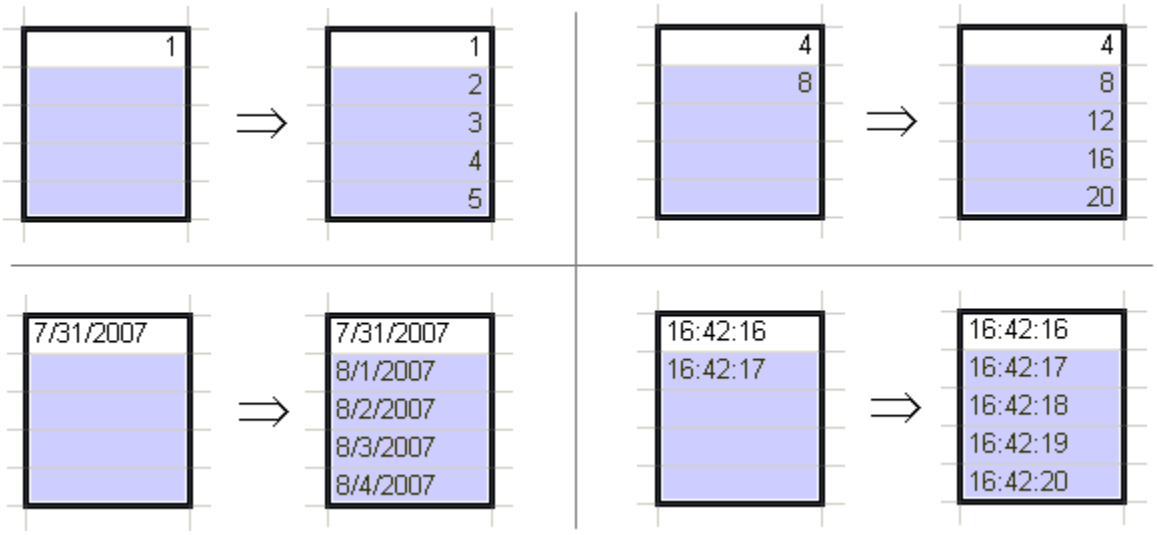

**Inserting a series with embedded custom list items:**



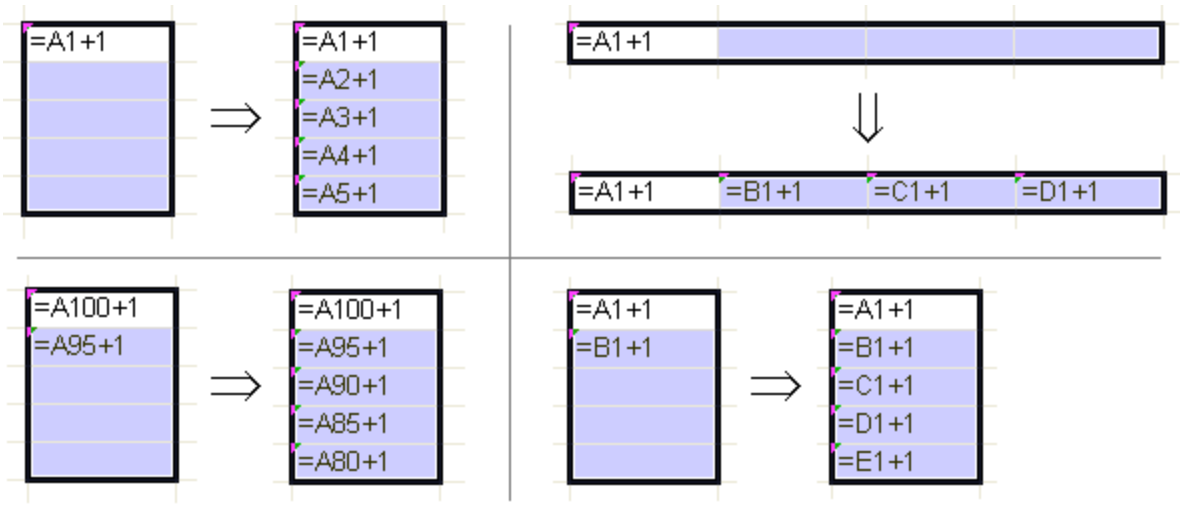

**Inserting a built-in "2d" series of numbers:**



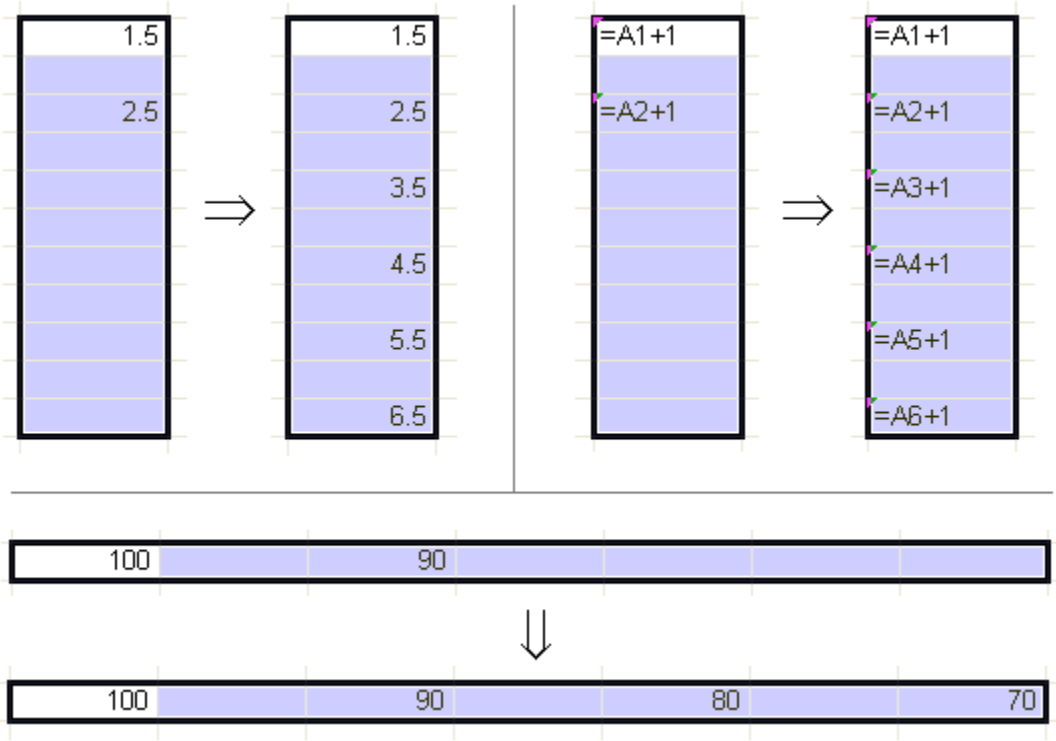
**Inserting series of numbers and dates:**




**Inserting series of formulas:**




**Inserting series with elements placed every n-th cell:**




**Inserting many series at once:**

| | | | | |
|---|---|---|---|---|
| 5 | | =a1+1 | | 6/5/2012 |
| | | | | |
| | | =a10+1 | | |
| | | | | |
| | | | | |
| | | | | |
| | | | | |

⇓

| | | | | |
|---|---|---|---|---|
| 5 | | =a1+1 | | 6/5/2012 |
| 6 | | | | 6/6/2012 |
| 7 | | =A10+1 | | 6/7/2012 |
| 8 | | | | 6/8/2012 |
| 9 | | =A19+1 | | 6/9/2012 |
| 10 | | | | 6/10/2012 |
| 11 | | =A28+1 | | 6/11/2012 |

## Inserting sequences/series using formulas

**Inserting a sequences of values**

mtxSeries(n, x, [step])

Generates a column vector (a one-column matrix) of 'n' subsequent numbers or generic date/time strings.

The 'x' argument specifies the first element of the series. This can be a number or date/time string.

The 'step' argument specifies the value by which the subsequent numbers/dates are incremented. If it's omitted, it's assumed to be 1 for numeric series and "P1D" (one day) for date series. The general form of the date/time period is:
[+|-]PnYnMnDTnHnMnS.nnn
For example:
P1Y2M10DT11H5M4S.355 represents a period of 1 year, 2 months, 10 days, 11 hours, 5 minutes, 4 seconds, 355 thousandths.
PT12H7M represents a period of twelve hours and seven minutes

=mtxSeries(5, 1, 0.1) returns {1; 1.1; 1.2; 1.3; 1.4}

=mtxSeries(5, "2005-12-01", "P1D") returns {"2005-12-01"; "2005-12-02"; "2005-12-03"; "2005-12-04"; "2005-12-05"}

=mtxSeries(5, "12:50:00", "PT2M") returns {"12:50:00"; "12:50:02"; "12:50:04"; "12:50:06"; "12:50:08"}

**Inserting a random series**

mtxRand2(n, [seed1], [seed2], [type], [v1], [v2])

Generates a column vector (a one-column matrix) of random floating point numbers for the specified distribution type.

The 'n' argument specifies the size of the returned vector.

The 'seed1' and 'seed2' arguments determine the starting numbers for the first and the second MLCG generator. If both those values are omitted, the first call to 'mtxRand2' will always starts from the same hard-coded values of 'seed1' and 'seed2' and subsequent calls will use the previously generated values returning partial series occurring one after another. If both 'seed1' and 'seed2' are specified, 'mtxRand2' will always be generating one and the same partial series.
To generate two or more independent partial series that do not occur one after another, specify a fixed 'seed1' value and skip the 'seed2' argument.

The 'type' argument specifies the distribution type:

- 0 - uniform (0, 1)
- 1 - normal with the 'v1' mean and the 'v2' standard deviation
- 2 - exponential with the 'v1' mean
- 3 - Poisson with the 'v1' mean
- 4 - Bernoulli with the 'v1' probability
- 5 - geometric with the 'v1' probability

If 'type' is omitted, it's assumed to be 0. If a given distribution type doesn't require the 'v1' and/or 'v2' arguments, they should be omitted.

=mtxRand2(4, 10000, 25000,,,) returns {0.71261246808435; 0.28457538373252; 0.42105025462307; 0.93072516522912}

## Drop-down lists

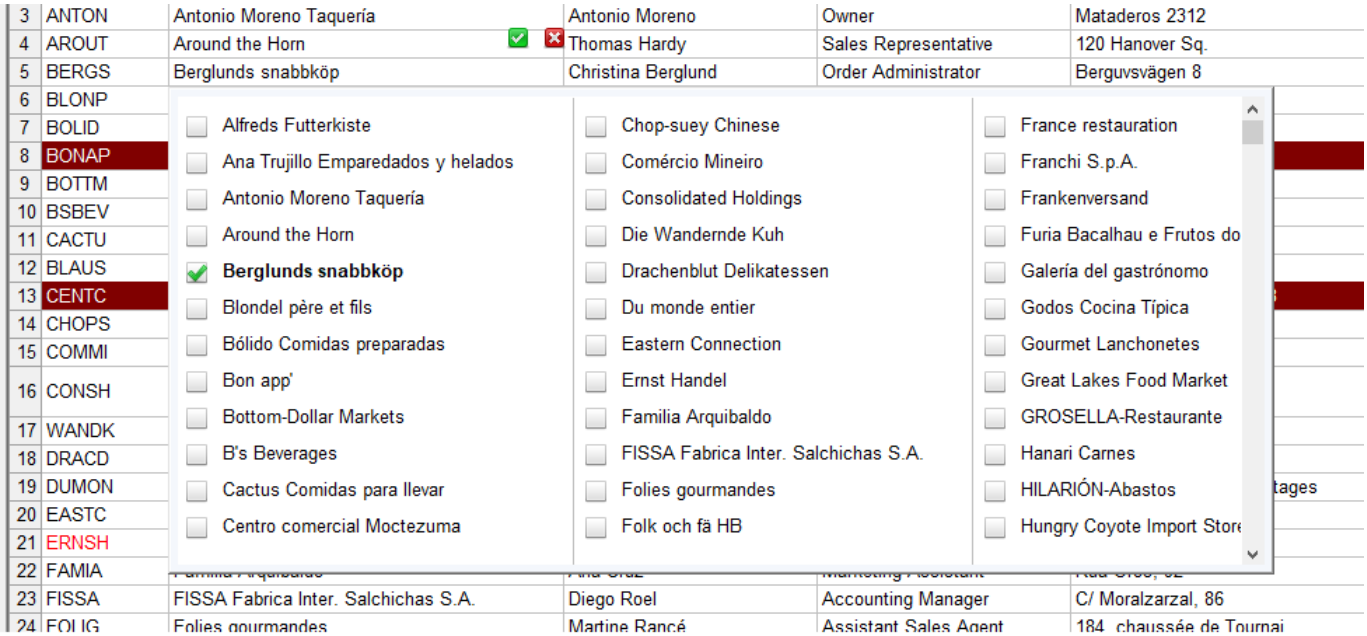
You can use the **Insert > Drop-Down Lists** dialog box to enable displaying drop-down lists for the current field(s) and/or to manage (create, edit, delete) lists in the current workbook.

When editing a cell with a drop-down list attached to it, GS-Calc opens a windows with a list of values with checkboxes and you can quickly choose predefined field values instead of re-typing them.
Pressing the cursor keys **Up** (in the first line of that edited field) or **Down** (in the last line of that edited field) switches the "focus" to the displayed list window. To switch back to the edited field, scroll the list to its first item and press **Up** or simply click the edited field.

To scroll the list, use the standard cursor keys: **Up**, **Down**, **PgUp**, **PgDn**, **Home**, **End**. If a list has multiple columns you can also use **Left**, **Right** to jump between columns.

To (un)check a given list item, press **Space** or click the check-box. If the "multi-selection" option is disabled for a list, the check-box button of the current list item is always "on" and checking another one always automatically unchecks the previous one.
To accept selected list item(s), press Enter, double click (one of) the selected list item(s) or click the "OK" button displayed above the edited field.

You can display list items in **1 to 99 columns**. Dragging the vertical grid-lines in the list window resizes all these list columns. To resize the list window drag its bottom-right corner or its edges.



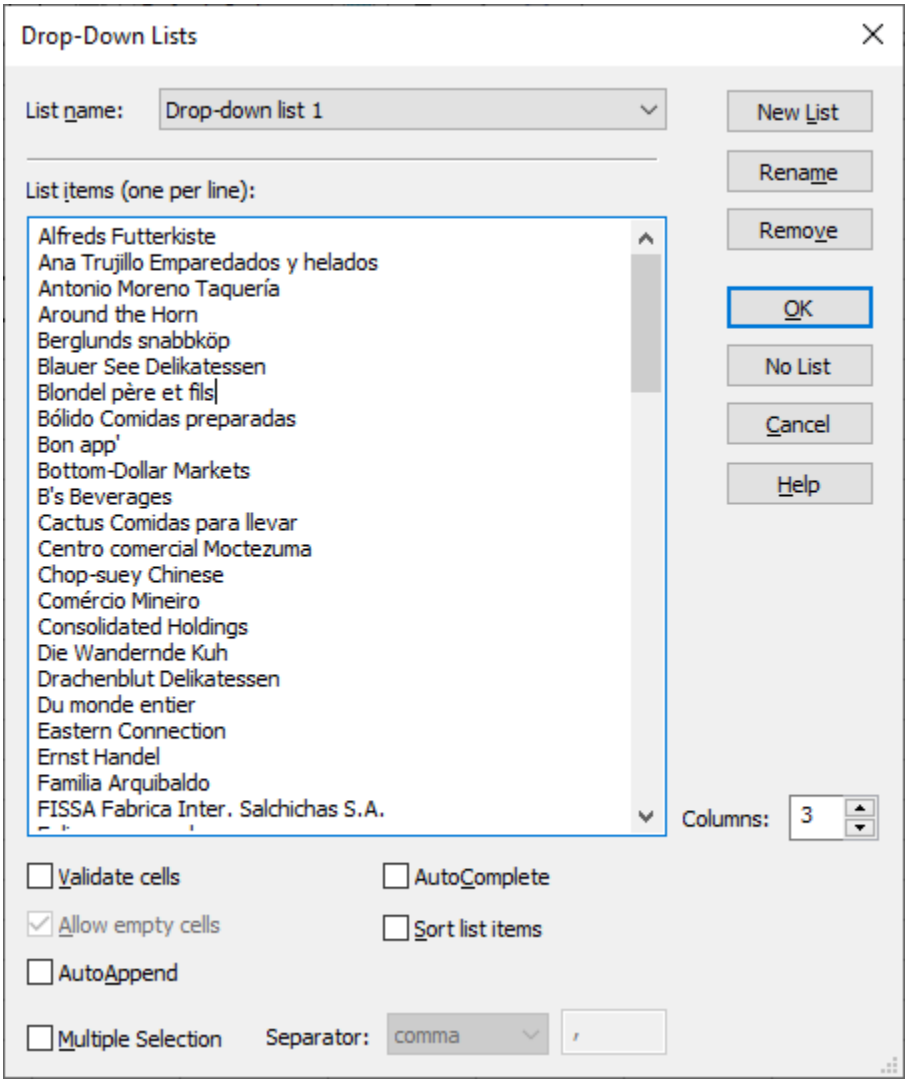

To create a new list click the **New List** button and to add new list items, enter or paste them in the **List Items** edit field: one item in one line.

The maximum number of list items is the same as the maximum number of rows: 12 millions.

If you select the **AutoAppend** option, each new unique value entered in the corresponding field(s) will be added to the list automatically.

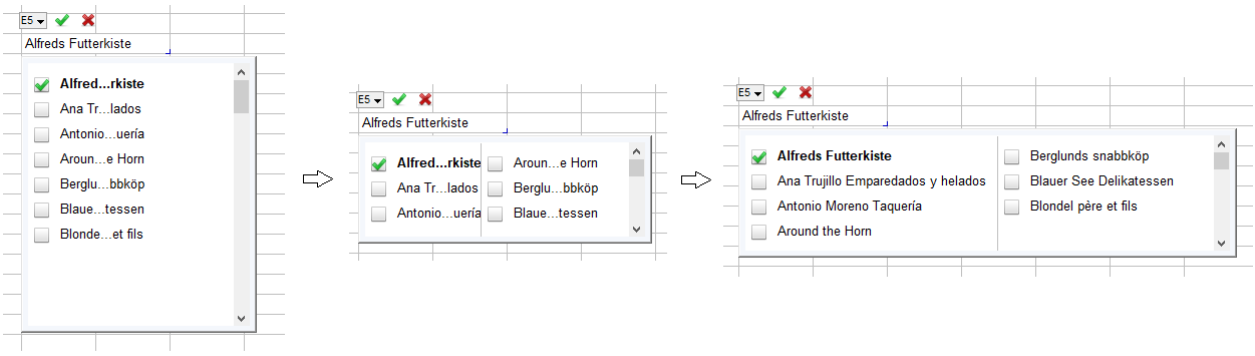
If you choose the **AutoComplete** option, the edited cell contents will be automatically completed/replaced with the full found matching text from the list after you type the initial characters.

If the **Sort items** option is not selected, the list will display all its elements in the same order as they were added/entered. Otherwise the list items will be sorted before displaying.




One list can be used by any number of cells in any worksheets in a given workbook. You can create up to 100 list in one workbook (file). To move lists between workbooks, you have to copy/paste the data from the **List Items** edit field.

Note:
If you specify the "columns" to be > 1, the splitter lines are displayed only if the list items occupy more than one column. Thus, to resize columns you might need to temporarily shrink the list window, for example:




## Using the Increment, Decrement and Operations commands

**To add/multiple/divide/subtract values in selected cells automatically**

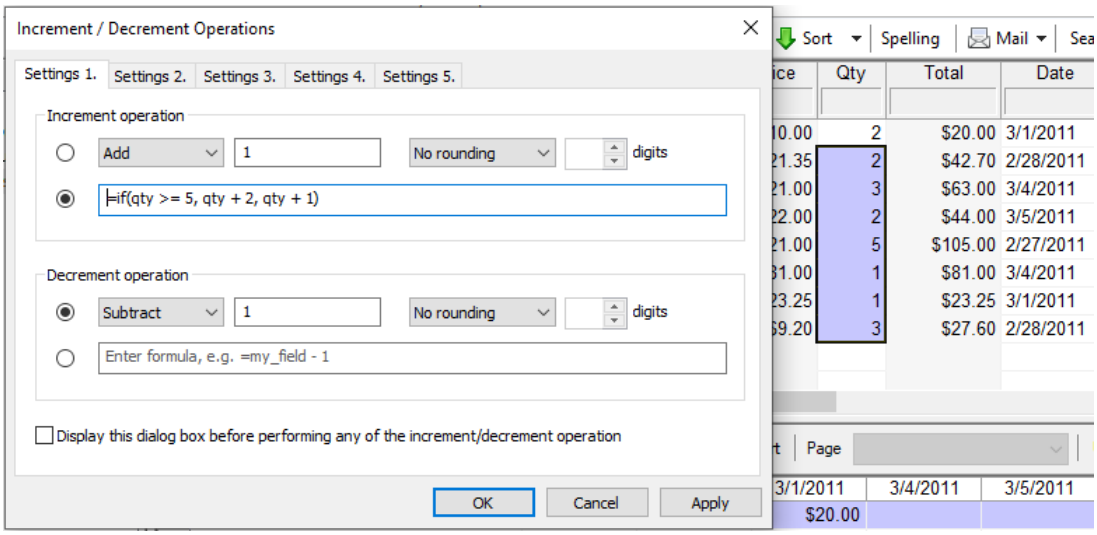
Select a cell containing a numeric value or any range of such cells use the **Edit > Increment** or **Edit > Decrement** command. By default these commands respectively add 1 and subtract 1.
The above adding and subtracting can be overwritten with the **Edit > Operations** command to perform more complex field value changes.

**To define operations executed when you use the "Edit > Increment" and "Edit > Decrement" commands**

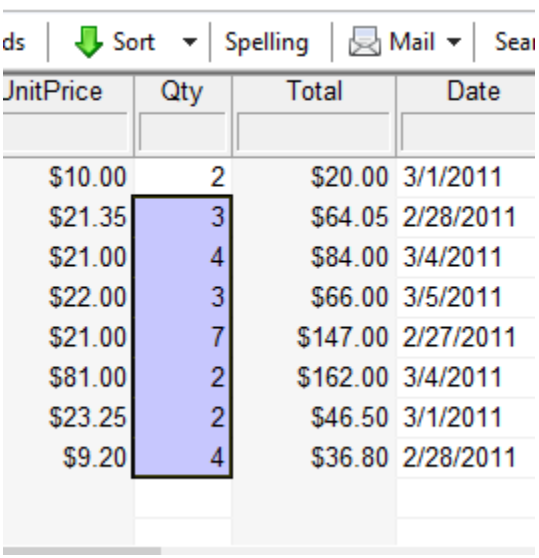
Use the **Edit > Operations** command and specify the incrementing and decrementing operations, which can be adding/subtracting/dividing/multiplying by a fixed value and at a given precision. The operation can be also defined as a formula referring to the cell(s) the command is applied to.
There are five increment/decrement pair settings and the active one is determined by the active tab in the **Operations** dialog box.

For example:

1. Define some "increment" formula-based action (and/or "decrement") on the "Settings1" tab and make it active.




2. Use the "increment' command Ctrl+Alt+"+" (and/or "decrement": Ctrl+Alt+"-"). Values in the selected fields will be modified according to the defined formula:




## Spell-checking and adding Hunspell dictionaries

### Spell-checking availability

GS-Calc 64-bit can use either the system spell checking functionality provided by Windows 8.x and later or the Hunspell spell checking engine (available for all systems).
The current selection can be changed either in the **Settings > Options > Spelling** dialog box or directly in the **Tools > Spelling** dialog box.

### Adding Hunspell dictionaries

To install a Hunspell dictionary for a given language:

1. Go to one of the web sites which distributes Hunspell dictionaries, for example:

   https://addons.mozilla.org/en-US/firefox/language-tools/

   The dictionaries are distributed as zip files typically with *.xpi or *.oxt extensions. After downloading the desirable language file, rename it to a *.zip file.

2. Each downloaded dictionary archive contains two text files with the *.aff and *.dic extensions and the name that (usually) represents the corresponding language ISO codes, e.g. en-US.aff and en-US.dic (English US).

   Copy those files to some folder and specify that folder in the GS-Calc spelling options. GS-Calc will automatically find all installed Hunspell dictionaries and list them in the **Spelling** dialog box next time you execute the **Tools > Spelling** command.
   To remove a given Hunspell dictionary, delete its *.aff and *.dic files from the above folder.

**Adding custom words to dictionaries**

Users can add their own words to existing dictionaries (or - when using the Hunspell spell checking engine - create their own full dictionaries).
The added words are saved in a text file in a folder specified in the GS-Calc spelling options. The file extension is *.txt and the file name is the language ISO code corresponding to the dictionary the file is link to. For example, if the dictionary files are

en-US.aff and en-US.dic

then the custom words will automatically (after closing the dialog box) be saved to

en-US.txt .

New words are added after clicking the **Add Word** button when performing spell checking. The *.txt files can be also edited in any text editor to add some larger number of words at once.
The text files containing added words must be saved using the UTF-8 encoding.

Other ISO codes include:

- af-ZA
- sq-AL
- ar-AE
- ar-BH
- ar-DZ
- ar-EG

- ar-IQ
- ar-JO
- ar-KW
- ar-LB
- ar-LY
- ar-MA
- ar-OM
- ar-QA
- be-BY
- bg-BG
- zh-CN
- zh-HK
- zh-MO
- zh-SG
- zh-TW
- hr-HR
- cs-CZ
- da-DK
- nl-NL
- nl-BE
- en-AU
- en-BZ
- en-CA
- en-CB
- en-IE
- en-JM
- en-NZ
- en-PH
- en-ZA
- en-US
- en-GB

- et-EE
- fi-FI
- fr-FR
- fr-BE
- fr-CA
- fr-LU
- fr-CH
- gd-IE
- de-DE
- de-AT
- de-LI
- de-LU
- de-CH
- el-GR
- he-IL
- hi-IN
- hu-HU
- is-IS
- id-ID
- it-IT
- it-CH
- ja-JP
- ko-KR
- lv-LV
- lt-LT
- mk-MK
- ms-MY
- no-NO
- pl-PL
- pt-PT
- pt-BR

- ro-RO
- ru-RU
- sr-SR
- sl-SI
- sk-SK
- es-ES
- es-AR
- es-BO
- es-CO
- es-CR
- es-DO
- es-EC
- es-GT
- es-HN
- es-MX
- es-NI
- es-PA
- es-PE
- es-PR
- es-PY
- es-UY
- es-VE
- sv-SE
- sv-FI
- th-VE
- tr-VE
- uk-UA
- uz-UZ
- vi-VN
- yi-IL
- uk-UA

## Generating passwords

The "Insert > Password" command enables you to insert / fill selected cells with unique passwords.

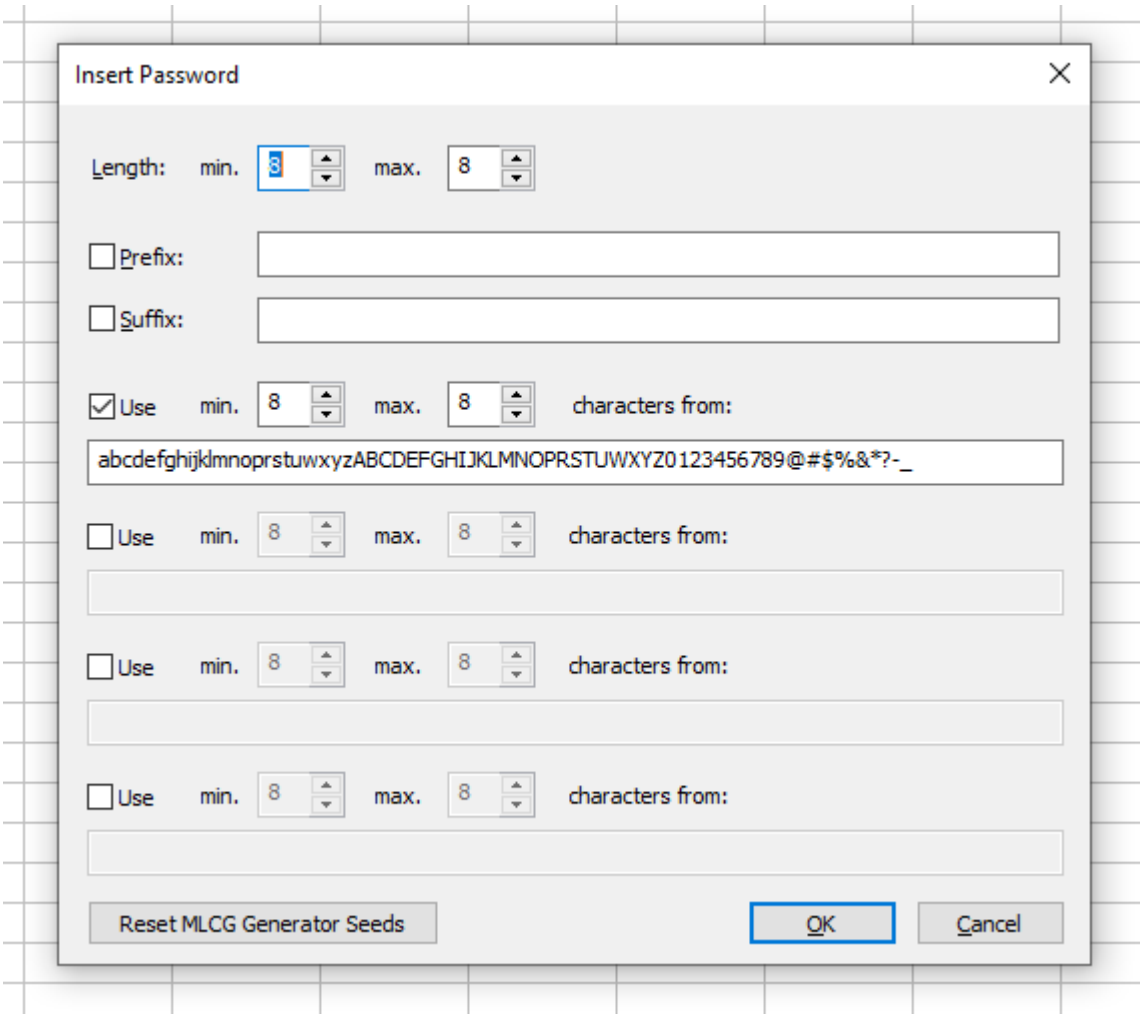
Increasing the length or the character set increases the probability that the generated passwords are unique.

For example, for the default settings, 8 character passwords generated for an entire selected 12 million row column all these passwords should be unique.
Subsequent passwords are based on the continuously generated one partial MLCG random series (~(10^18)/4 passwords).
Clicking the "Reset MLCG Generator Seeds" button starts generating another independent series.

You can use the "F3/Find > Find All > Find Duplicates" command to verify whether there are any repeated passwords.
(Please note that for the above example there might be several occurrences of passwords that differ by lower- and uppercase letters and this command will list them side by side.)
Another option is the FILTER() function with the "duplicates" filter.

Generated passwords can contain Unicode characters but non-ascii characters should be used with care as some systems (especially various recovery systems) might not support them.

Such unique text strings can be also used for testing your data models with random data.
To generate random lists of predefined strings, you can use the "Insert > Custom Series" command to load/create a list, then the "Insert > Random Data" command with that list name.




# Worksheet views

## Splitting worksheet views

Split views can display different areas/ranges of the same worksheet or different worksheets in one main worksheet window. This feature enables users to monitor/track changes in different cells and edit them without inconvenient scrolling. It can also be an equivalent to the "Freeze Rows/Columns" commands implemented in other spreadsheets.

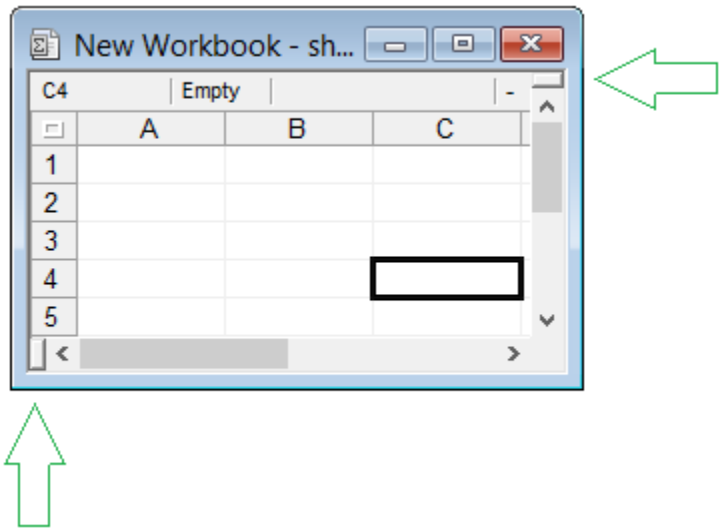
Each worksheet can contain up to 100 split views and the number of those configuration in a workbook is unlimited. The split view configuration is saved to the workbook file. Additionally, users can protect the current split view configuration with a password using the **Tools > Protect Structure** command.

The **View > Proportional Splitter** option enables you to control how the split views behave when the main frame window is resized. If this option is "on", the split views will be resized proportionally. If it's "off", the size of the split views remain the same.

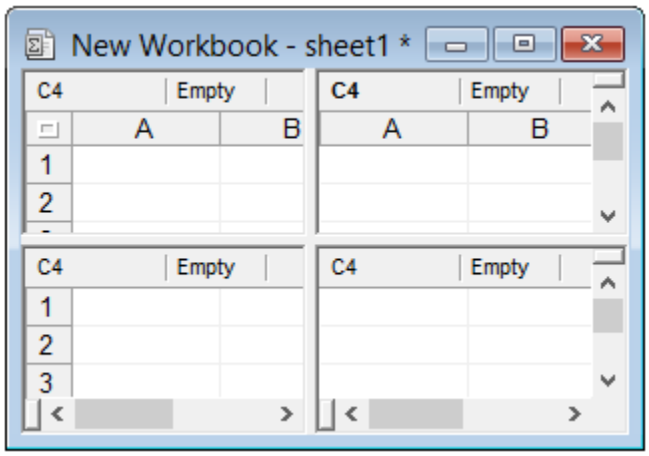
Users who need to quickly apply different splitting configurations or to "copy" a given splitting configuration to other worksheet(s) can use the **View > Pane Layout Templates** commands. These splitting configuration templates are stored as global settings.

**To split a worksheet view**

Use the **View > Split View Horizontally** and **...Split View Horizontally** commands or drag either the vertical or the horizontal splitter bar to the desirable position:

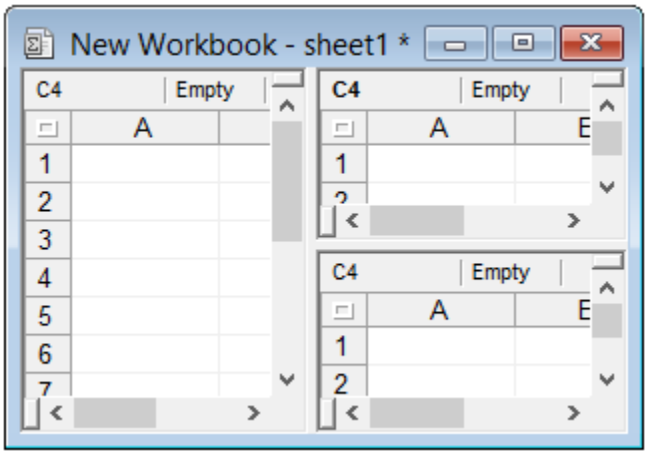



If the **<u>View > Automatic Splitting Mode</u>** is active, splitting always concerns the whole column of views (for vertical splitting) or the whole row of views (for horizontal splitting). Thus the split views always form a regular grid (e.g. 1x2, 2x1, 2x2, 3x2,...).



In this splitting mode the newly created views are synchronized by default. Scrolling columns in one view causes scrolling columns in all views within the same column of views and scrolling rows is always synchronized for views within a given row of views. Synchronization concerns both the very process of scrolling and the first visible column (for column synchronization) or the first visible row (for row synchronization).
By default, in this splitting mode the row headings are displayed only for the "left-most" views and column headings - only for the "top-most" views. To change this for a given view, click it and use the **View > Headings And Scrollbars > Column Headings** or **...Column Headings** commands.

If the **View > Manual Splitting Mode** is active, splitting always concerns only a given single view.




In this splitting mode views are not synchronized automatically. One can use the **View > Column Synchronization** and **View > Row Synchronization** commands to synchronize them in any desirable way.
Also, the left-most column (or the top row) can be different in synchronized views. For details, please see: Synchronizing views.

By default, in this splitting mode the row and column headings are displayed for all views. To change this for a given view, click it and use the **View > Headings And Scrollbars > Column Headings** or **...Column Headings** commands.

By default, in both splitting modes views display one and the same worksheet. To choose a different one for a given view, use the **View > Choose Worksheet** command.

**To remove split view(s)**

Use the **View > Split View Horizontally** and **...Split View Horizontally** commands again or drag either the vertical or the horizontal splitter bar to any of the window edges.

## Row and column synchronization options

Split views can be synchronized: scrolling columns (or rows) in one view can result in scrolling columns in all other views synchronized with it.
Synchronization commands are available via the View menu.

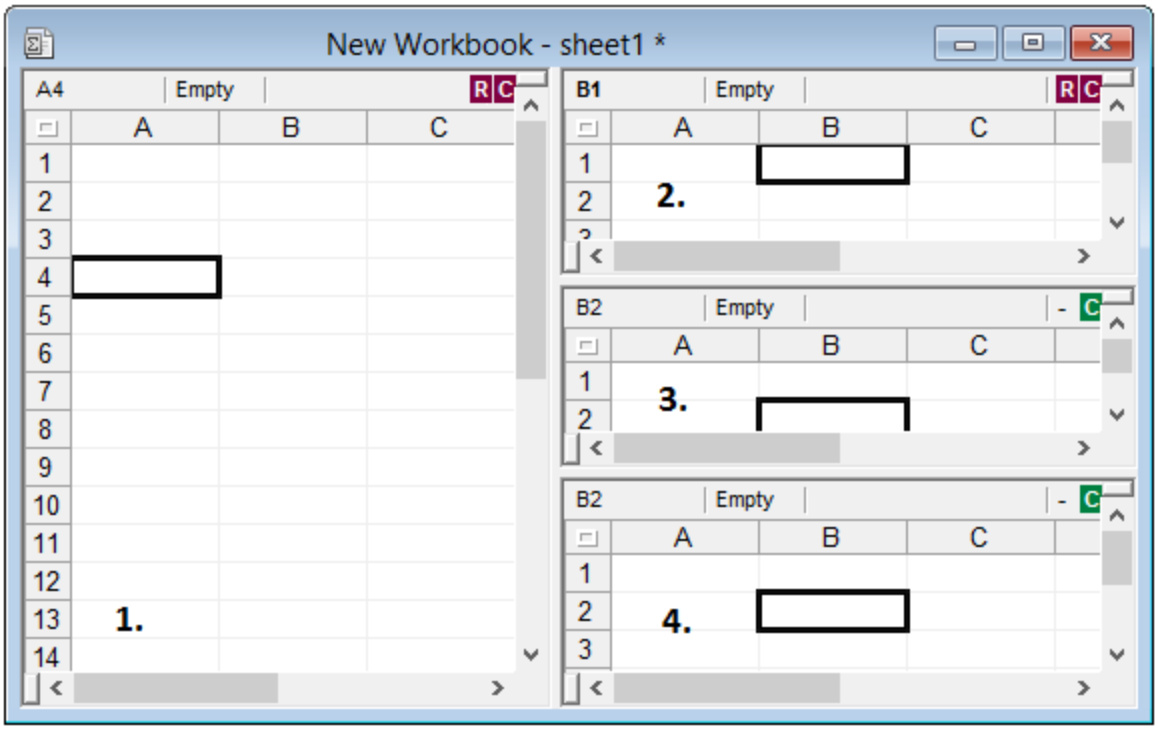
Views split in the **Automatic Splitting Mode** are all synchronized by default. The split views form a regular grid of views which are synchronized in columns and rows (of views). The default synchronization can't be turned off in this mode.

Views split in the **Manual Splitting Mode** are not synchronized initially and users have to toggle the synchronization "on" or "off" using the **Column Synchronization** and **Row Synchronization** commands.
Additionally, in this mode synchronized views don't have to use the same first column (or row), which means that the synchronization can concern any distant worksheet ranges showing both different ranges of columns and different ranges of rows.

**To toggle the view synchronization on/off in the manual splitting mode**

Click the first view to synchronize and use the **Column Synchronization** or the **Row Synchronization** command. It'll cause displaying the "Click a view to toggle the column (row) synchronization on or off" message within that view. Then click another view that should be synchronized with the current one (or to remove it from a given view synchronization chain).

If the **Synchronization Tags** option is selected, synchronized views display the "R" (rows) or "C" (columns) markers in their top-right corners. Views belonging to the same synchronization chains display their respective markers using the same colors.




In the example above the views are synchronized as follows:

- Scrolling columns is synchronized in [1.] and [2.]
- Scrolling rows is synchronized in [1.] and [2.]
- Scrolling columns is synchronized [3.] and [4.]

**To synchronize cell range selections in views**

Select the **Keep Selection Synchronized** option. Selecting cells within a given view will cause selecting the same number of columns and/or rows in views synchronized with it.

To perform a one-time synchronization of selections in synchronized views, use the **Make Same Selection** command.

**Note:** To quickly go back to the selection origin, on worksheet/table window toolbar click the address of that selection.

# Pivot Tables

## Using Pivot Tables

Pivot tables enable you to quickly and easily obtain and visualize various statistical information about data coming from flat source record tables. This includes (but is not limited to) counting, summarizing, sorting or finding partial maximum/minimum values.

In GS-Calc pivot tables are represented by **pivotData()** array formulas. Such a formula requires up to several parameters, thus it may be easier to use the **Insert > Pivot Table** composer rather then enter everything manually.

**Pivot tables usage examples:**

For a given table of customers (where that table contains among others the "Country" and "City" fields):

| | C | D | E |
|---|---|---|---|
| 2 | CompanyName | City | Country |
| 3 | Alfreds Futterkiste | Berlin | Germany |
| 4 | Ana Trujillo Emparedados y helados | México D.F. | Mexico |
| 5 | Antonio Moreno Taquería | México D.F. | Mexico |
| 6 | Around the Horn | London | UK |
| 7 | Berglunds snabbköp | Luleå | Sweden |
| 8 | Blauer See Delikatessen | Mannheim | Germany |

⋮

To find out what is the number of customers from each country:

1. Open the **Insert > Pivot Table** dialog box.

2. Select the entire customers' table range and click the **Source Range** button. Alternatively, you can enter that range manually and click the **Source Range > Reload Fields** command.

3. Choose the "Country" field as the only **Row** field.

4. Select "Count" as the default function on the functions list in the "Data fields" section.

5. Click a cell where the formula is to be inserted and click the **Insert** button.

| | Country | Grand Total |
|---|---|---|
| 1 | Argentina | 3 |
| 2 | Austria | 2 |
| 3 | Belgium | 2 |
| 4 | Brazil | 9 |
| 5 | Canada | 3 |
| 6 | Denmark | 2 |
| 7 | Finland | 2 |
| 8 | France | 11 |
| | ⋮ | |
| 22 | **Grand Total** | **91** |

To find out what is the number of customers from each country and how many of them come from a given city in that country:

1. Open the **Insert > Pivot Table** dialog box.
2. Select the entire products' table range and click the **Source Range** button. Alternatively, you can enter that range manually and click the **Source Range > Reload Fields** command.
3. Choose the "Country" field as the first **Row** field.
4. Choose the "City" field as the second **Row** field.
5. Select "Count" as the default function on the functions list in the "Data fields" section.
6. Click a cell where the formula is to be inserted and click the **Insert** button.

| | Country | City | Grand Total |
|---|---|---|---|
| 1 | Argentina | Buenos Aires | 3 |
| 2 | Argentina Total | | 3 |
| 3 | Austria | Graz | 1 |
| 4 | | Salzburg | 1 |
| 5 | Austria Total | | 2 |
| 6 | Belgium | Bruxelles | 1 |
| 7 | | Charleroi | 1 |
| 8 | Belgium Total | | 2 |
| | | ⋮ | |
| 91 | **Grand Total** | | **91** |

For a given table of products (where that table contains among others the "ProductName", (order) "Date", "Qty" and "Total" fields):

| | B | C | D | E | F | G | H |
|---|---|---|---|---|---|---|---|
| 2 | **ID** | **ProductID** | **ProductName** | **UnitPrice** | **Qty** | **Total** | **Date** |
| 3 | ALFKI | 3 | Aniseed Syrup | $10.00 | 2 | $20.00 | 3/1/2011 |
| 4 | BOLID | 5 | Chef Anton's Gumbo Mix | $21.35 | 1 | $21.35 | 2/28/2011 |
| 5 | CENTC | 11 | Queso Cabrales | $21.00 | 3 | $63.00 | 3/4/2011 |
| 6 | EASTC | 4 | Chef Anton's Cajun Seasoning | $22.00 | 2 | $44.00 | 3/5/2011 |
| 7 | BLAUS | 22 | Gustaf's Knäckebröd | $21.00 | 5 | $105.00 | 2/27/2011 |
| 8 | ERNSH | 20 | Sir Rodney's Marmalade | $81.00 | 1 | $81.00 | 3/4/2011 |

⋮

To display the products sales (sales volume only) breakdown based on days:

1. Open the **Insert > Pivot Table** dialog box.
2. Select the entire products' table range and click the **Source Range** button. Alternatively, you can enter that range manually and click the **Source Range > Reload Fields** command.
3. Choose the "ProductName" field as the only **Row** field.
4. Choose the "Date" database field as the only **Column** field.
5. Choose the "Qty" field as the only **Data** field and select the **Sum** function for it.
6. Click a cell where the formula is to be inserted and click the **Insert** button.

| | ProductName | 2011-02-27 | 2011-02-28 | 2011-03-01 | 2011-03-04 | 2011-03-05 | Grand Total |
|---|---|---|---|---|---|---|---|
| 1 | Aniseed Syrup | | | 1 | | | 1 |
| 2 | Chef Anton's Cajun Seasoning | | | | | 1 | 1 |
| 3 | Chef Anton's Gumbo Mix | | 1 | | | | 1 |
| 4 | Gustaf's Knäckebröd | 1 | | | | | 1 |
| 5 | Queso Cabrales | | | | 1 | | 1 |
| 6 | Sir Rodney's Marmalade | | | | 1 | | 1 |
| 7 | Teatime Chocolate Biscuits | | 1 | | | | 1 |
| 8 | Tofu | | | 1 | | | 1 |
| 9 | **Grand Total** | **1** | **2** | **2** | **2** | **1** | **8** |

To display the products sales (sales volume and quantity) breakdown based on days:

1. Open the **Insert > Pivot Table** dialog box.
2. Select the entire products' table range and click the **Source Range** button. Alternatively, you can enter that range manually and click the **Source Range > Reload Fields** command.
3. Choose the "ProductName" field as the only **Row** field.

4. Choose the "Date" field as the only **Column** field.
5. Choose the "Total" field as the first **Data** field and select the **Sum** function for it.
6. Choose the "Qty" field as the second **Data** field and select the **Sum** function for it.
7. Click a cell where the formula is to be inserted and click the **Insert** button.

| | ProductName | 2011-02-27 | 2011-02-28 | 2011-03-01 | 2011-03-04 | 2011-03-05 | Grand Total |
|---|---|---|---|---|---|---|---|
| 1 | Aniseed Syrup | | | $20.00 | | | $20.00 |
| 2 | | | | 2 | | | 2 |
| 3 | Chef Anton's Cajun Seasoning | | | | | $44.00 | $44.00 |
| 4 | | | | | | 2 | 2 |
| 5 | Chef Anton's Gumbo Mix | | $21.35 | | | | $21.35 |
| 6 | | | 1 | | | | 1 |
| 7 | Gustaf's Knäckebröd | $105.00 | | | | | $105.00 |
| 8 | | 5 | | | | | 5 |
| 9 | Queso Cabrales | | | | $63.00 | | $63.00 |
| 10 | | | | | 3 | | 3 |
| 11 | Sir Rodney's Marmalade | | | | $81.00 | | $81.00 |
| 12 | | | | | 1 | | 1 |
| 13 | Teatime Chocolate Biscuits | | $27.60 | | | | $27.60 |
| 14 | | | 3 | | | | 3 |
| 15 | Tofu | | | $23.25 | | | $23.25 |
| 16 | | | | 1 | | | 1 |
| 17 | **Grand Total** | **$105.00** | **$48.95** | **$43.25** | **$144.00** | **$44.00** | **$385.20** |
| 18 | | **5** | **4** | **3** | **4** | **2** | **18** |

**PivotData formula**

Pivot formulas have the following form:

**pivotData(source, rows, columns, data, functions, options [, field1, filter1, field2, filter2, ...])**

The above formula creates and returns a pivot table for the data in the 'source' range.

The 'rows', 'columns' and 'data' parameters represent arrays/ranges containing the respective pivot field indices. The indices are relative to the top-left corner of the 'source' range and the numbering starts from 1.
For example: {1, 5}, {4}, {1}
If some of the indices are incorrect, pivotData returns the #VALUE! error. In GS-Calc 9.0 the 'columns' array can contain only one element. The 'data' fields can be omitted, in which case the last specified row field and the default pivot function will be used (for the last row field).

The 'functions' argument is an array of predefined functions constants/IDs associated with the specified data fields. The possible values are:

- PIVOT::Sum
- PIVOT::SumPositive
- PIVOT::SumNegative

- PIVOT::SumSquares
- PIVOT::Count
- PIVOT::CountPositive
- PIVOT::CountNegative
- PIVOT::CountZeroes
- PIVOT::Min
- PIVOT::Max

For example: {PIVOT::Sum}, {PIVOT::Sum, PIVOT::Count}

**Note:** You must specify either the PIVOT::ColumnGrandTotals or at least one column field. Otherwise the function will return the #NUM! error code.

The 'options' parameter can be any sum of the following constants:

- PIVOT::ColumnGrandTotals - display column grand totals in the last column
- PIVOT::RowGrandTotals - include row grand totals in the last row
- PIVOT::SubTotals - include subtotals (for pivot tables with 2 or more row fields)
- PIVOT::ShowZeroes - show zeroes for empty data fields; without this option 'empty' subtotals will be displayed as the #N/A! error codes
- PIVOT::RepeatRowFields - if multiple row fields are specified, display all their values, even duplicated ones
- PIVOT::CaseSensitive - if filters are specified, use case sensitive comparison
- PIVOT::SortRowsDescending - present the output rows using the descending sort order
- PIVOT::SortColumnsDescending - present the output columns using the descending sort order
- PIVOT::NoSourceFieldNames - the source range contains no field names in the first row; use the 'Field n.' names instead
- "SEARCH::IgnorePunctuation (or 16384) - use the word sort order (ignoring certain punctuation marks) when sorting and filtering.
- "SEARCH::NeutralSortOrder (or 32768) - use neutral, language independent string comparison instead of the default language specific comparison when sorting and filtering.

- "SEARCH::Pattern (or 65536) - treat filters that don't start with (=,>,>=,<,<=) operator as simple (wildcard) patterns.

For example: {PIVOT::ColumnGrandTotals}, {PIVOT::ColumnGrandTotals + PIVOT::RowGrandTotals + PIVOT::SubTotals}

The optional [field, filter] pairs specify the 1-based field index and the condition. Only source data meeting all the specified conditions will be included in the pivot table. Conditions can have the following form:

- A text string beginning with the =,>,>=,<,<= operators.
- A number or a search pattern: a text string containing special characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

Pivot formula examples:

=pivotData(C3:H19, {1},,, {PIVOT::Sum}, PIVOT::ColumnGrandTotals)

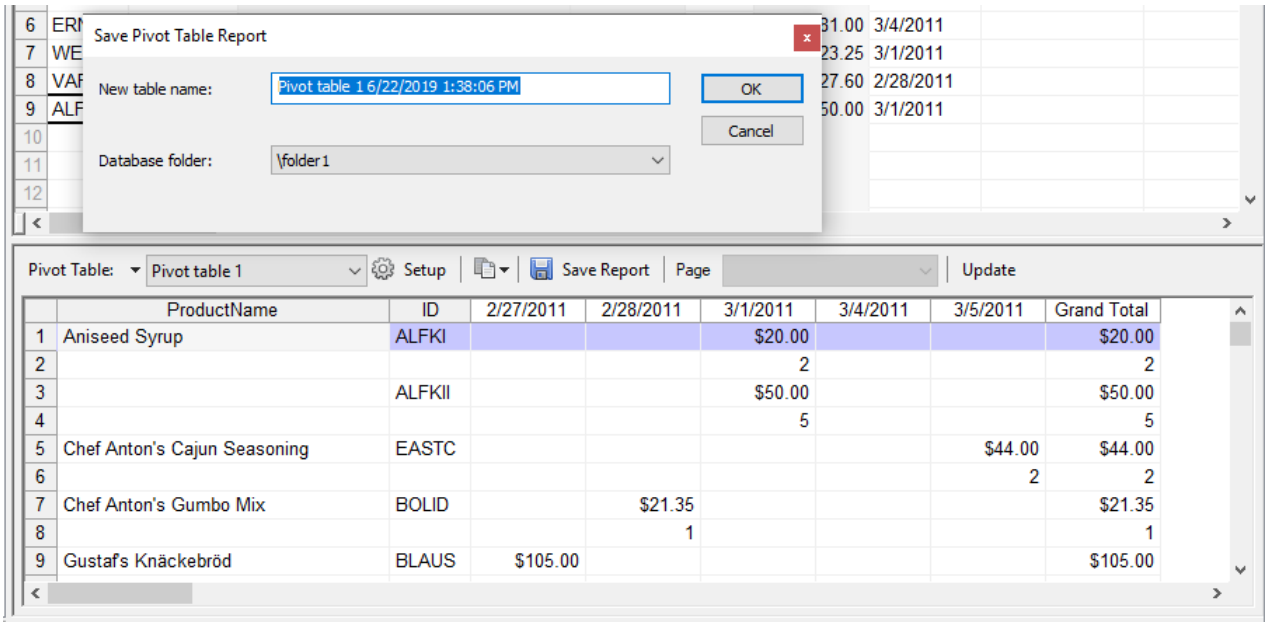
=pivotData(sheet1!C3:H19, {1}, {2}, {3, 4}, {PIVOT::Sum, PIVOT::Count}, PIVOT::RowGrandTotals + PIVOT::ColumnGrandTotals + PIVOT::SubTotals)

=pivotData(sheet1!B3:F8, {5, 1}, {3},, {PIVOT::Sum}, PIVOT::RowGrandTotals + PIVOT::ColumnGrandTotals + PIVOT::SubTotals, 2, "Jones", 4, ">2010-01-01")

## Creating Pivot Table Reports

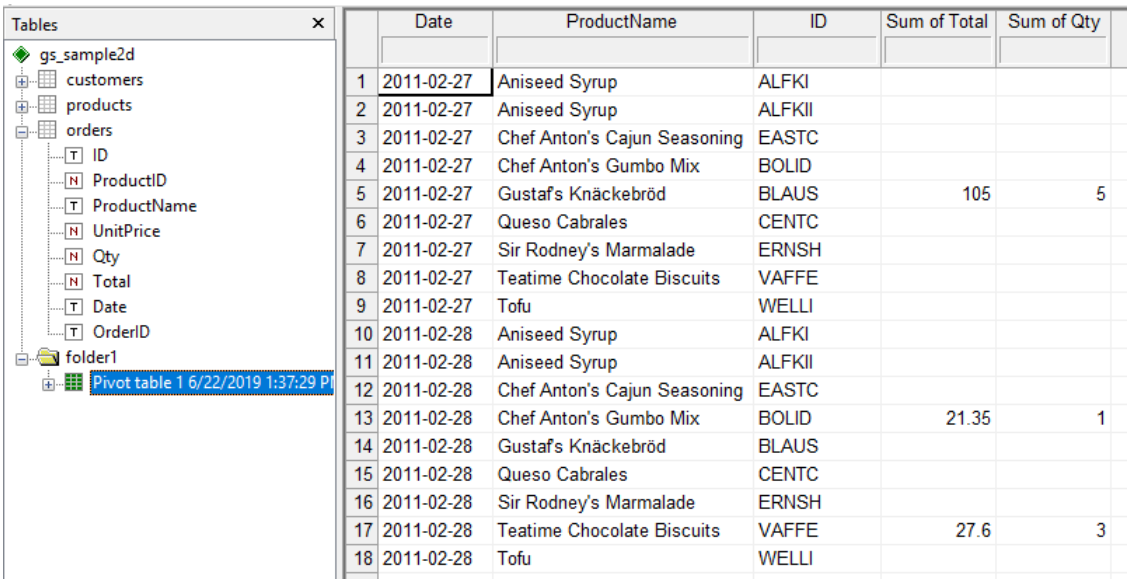
Clicking the "Save" icon in the Pivot Table View enables you to save a new time-stamp report for a given pivot table.
The procedure looks as follows:

1.



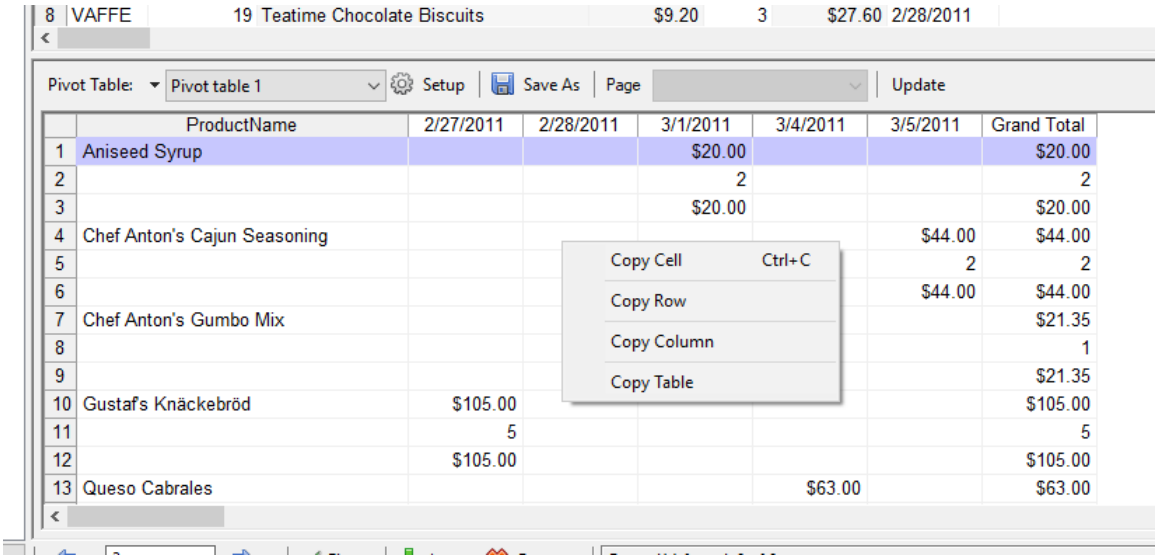

| | ProductName | ID | 2/27/2011 | 2/28/2011 | 3/1/2011 | 3/4/2011 | 3/5/2011 | Grand Total |
|---|---|---|---|---|---|---|---|---|
| 1 | Aniseed Syrup | ALFKI | | | $20.00 | | | $20.00 |
| 2 | | | | | 2 | | | 2 |
| 3 | | ALFKII | | | $50.00 | | | $50.00 |
| 4 | | | | | 5 | | | 5 |
| 5 | Chef Anton's Cajun Seasoning | EASTC | | | | | $44.00 | $44.00 |
| 6 | | | | | | | 2 | 2 |
| 7 | Chef Anton's Gumbo Mix | BOLID | | $21.35 | | | | $21.35 |
| 8 | | | | 1 | | | | 1 |
| 9 | Gustaf's Knäckebröd | BLAUS | $105.00 | | | | | $105.00 |

2.



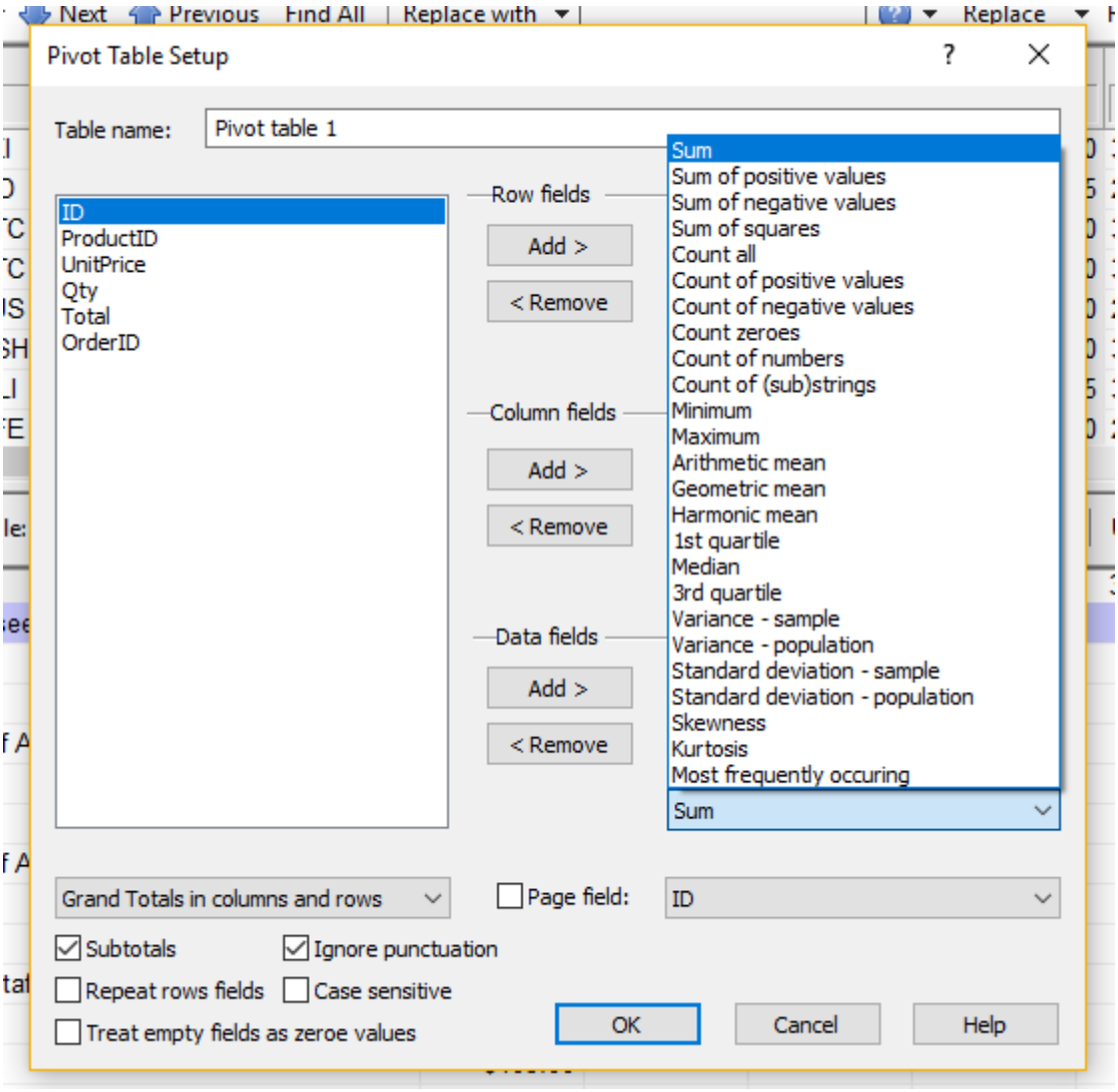

| | Date | ProductName | ID | Sum of Total | Sum of Qty |
|---|---|---|---|---|---|
| 1 | 2011-02-27 | Aniseed Syrup | ALFKI | | |
| 2 | 2011-02-27 | Aniseed Syrup | ALFKII | | |
| 3 | 2011-02-27 | Chef Anton's Cajun Seasoning | EASTC | | |
| 4 | 2011-02-27 | Chef Anton's Gumbo Mix | BOLID | | |
| 5 | 2011-02-27 | Gustaf's Knäckebröd | BLAUS | 105 | 5 |
| 6 | 2011-02-27 | Queso Cabrales | CENTC | | |
| 7 | 2011-02-27 | Sir Rodney's Marmalade | ERNSH | | |
| 8 | 2011-02-27 | Teatime Chocolate Biscuits | VAFFE | | |
| 9 | 2011-02-27 | Tofu | WELLI | | |
| 10 | 2011-02-28 | Aniseed Syrup | ALFKI | | |
| 11 | 2011-02-28 | Aniseed Syrup | ALFKII | | |
| 12 | 2011-02-28 | Chef Anton's Cajun Seasoning | EASTC | | |
| 13 | 2011-02-28 | Chef Anton's Gumbo Mix | BOLID | 21.35 | 1 |
| 14 | 2011-02-28 | Gustaf's Knäckebröd | BLAUS | | |
| 15 | 2011-02-28 | Queso Cabrales | CENTC | | |
| 16 | 2011-02-28 | Sir Rodney's Marmalade | ERNSH | | |
| 17 | 2011-02-28 | Teatime Chocolate Biscuits | VAFFE | 27.6 | 3 |
| 18 | 2011-02-28 | Tofu | WELLI | | |

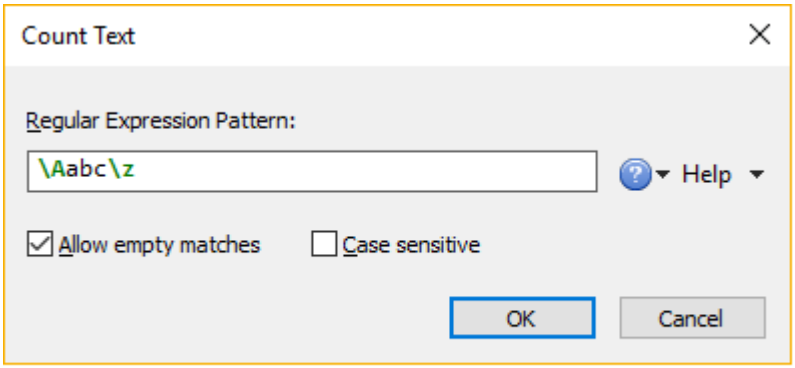
You can also simply copy the pivot table data to other programs either as image or using a few "Copy" commands from the context menu:

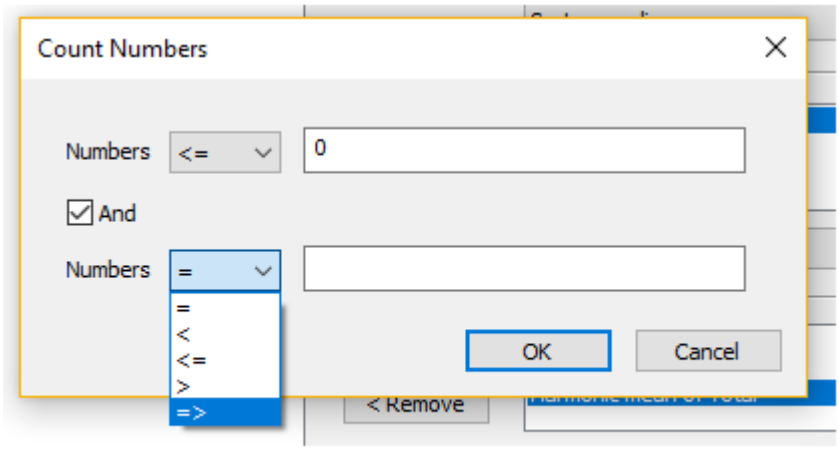



## Pivot Table Functions

GS-Calc pivot tables can use of number of "counting" functions:


Next
Previous
Find All
Replace with
Replace
Pivot Table Setup
Table name:
Pivot table 1
ID
ProductID
UnitPrice
Qty
Total
OrderID
Row fields
Add >
< Remove
Column fields
Add >
< Remove
Data fields
Add >
< Remove
Sum
Sum of positive values
Sum of negative values
Sum of squares
Count all
Count of positive values
Count of negative values
Count zeroes
Count of numbers
Count of (sub)strings
Minimum
Maximum
Arithmetic mean
Geometric mean
Harmonic mean
1st quartile
Median
3rd quartile
Variance - sample
Variance - population
Standard deviation - sample
Standard deviation - population
Skewness
Kurtosis
Most frequently occuring
Sum
Grand Totals in columns and rows
Page field:
ID
Subtotals
Ignore punctuation
Repeat rows fields
Case sensitive
Treat empty fields as zeroe values
OK
Cancel
Help




# Monte Carlo Simulations

## Using Monte Carlo Simulations

The Monte Carlo method can be used to analyze data with help of random number generators. This includes for example finding approximate maximum and minimum for a given formula/function or simulating real data like measurements or observations.

In GS-Calc the Monte Carlo simulations are executed by specifying "Input" worksheet cells or ranges which are filled with ("pseudo-") random numbers in loops and "Output" cells which contain formulas processing these numbers. Large numbers of loops enable collecting statistically meaningful number of results of those formulas and verifying how they will behave for real "Input" data.

The following menus and commands related to the MC simulations are available in GS-Calc:

- Simulations

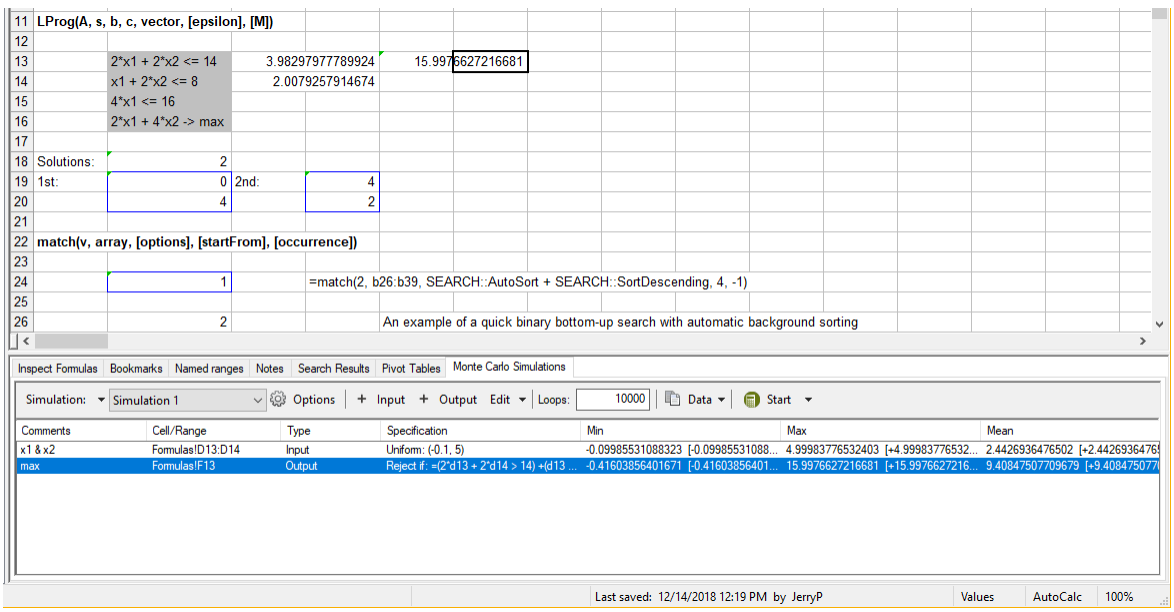
- Adding new simulations with input and output cells and optionally specifying random number generator parameters. (See: the specification of the mtxRand functions).
- Adding a copy of an existing simulation.
- Deleting the selected simulation or all simulations.

- Add Input/Output cells or ranges

  Added input worksheet cells should be empty or should contain only numbers. They are filled with random numbers in subsequent loops.
  Added output cells should contain formulas processing these numbers and returning numeric values.

  Output cells can use a "rejection" formula which determines whether the results obtained in a given loop should be retained and included in the calculated statistics.

  **Example (I):** to find an approx. solution of the linear programming problem:




2*x1 + 2*x2 <= 14
x1 + 2*x2 <= 8
4*x1 <= 16
2*x1 + 4*x2 -< max

If you specify D13 and D14 as input cells with the uniform distribution (-0.1, 5) and E13 as the output cell with the formula:

=2*d13 + 4*d14

with the following rejection criteria:

=(2*d13 + 2*d14 > 14) + (d13 + 2*d14 > 8) + (4*d13 > 16)

Then after 10000 loops (and for the default generator parameters), the found maximum of E13 will be around 15.997 for the x1=3.983 and x2=2.01 (the exact values calculated with the LProg() function are respectively 16, 4 and 2).

**Note:** The same output cells can be added many times so the above constrains doesn't have to be just one merged rejection criteria.

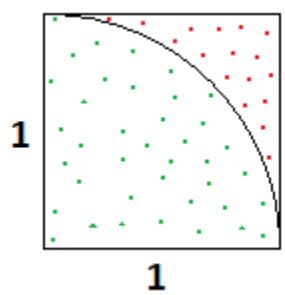
**Example (II):** to calculate the area of a circle with a radius r=1:

Specify B2 and B3 as the input cells with the uniform distribution (0, 1) and D2 as the output cell with the formula:

=sqrt(b2*b2 + b3*b3)

with the following rejection criteria:

=sqrt(b2*b2 + b3*b3) > 1




Then after 1000000 loops (and for the default generator parameters), the "counter" value for the (not rejected) output cell will be 785348. Thus the approx. area of the 1/4th of that circle is 785348/1000000 and the full area is approx. equal to 3.141392 which is the approximation of the real value of Pi*r^2 = 3.14159265358979.

- Showing min, max
  Selecting a desirable item on the "Input"/"Output" list and choosing the "Data ->Show Min/Max" commands will fill the input/output worksheet cells with the loop input/output data where this minimum or maximum was found.

- Saving generated loop data
  After performing the specified number of loops the data are ready to be saved. They are stored as a new worksheet in the specified folder. Loops are saved in rows so if the number of the selected input/output cells exceeds the maximum number of columns (4096), the saved data will be truncated.
  Input/output items specified as ranges are expanded (likewise, in rows) in that new worksheet as individual cells.

- Starting simulation, cancelling, resuming

  Executing loops can stopped and resumed at any moment.

During the procedure the list of the input/output cells is updated every 0.5s to display the current:
maximum
minimum
mean
variance
standard deviation
counter (for output cells this excludes rejected loops).

Values displayed in the [] square brackets denote the change in comparison with the previous screen update (0.5s).

Starting the simulation stops any pending workbook updates and vice-versa.

# Lookup functions

## hLookUp() function

**hLookUp(v, array, n, [type])**
**hLookUp(v, array, n, [type], [startIndex], [occurrence])**

The hLookUp() function searches the top row of 'array' for 'v' and - if found - returns a value from the same column and the n-th row of 'array'.

The 2nd Match() variant uses two additional parameters:
**startFrom** - positive numbers specify where the searching should start and negative numbers specify where it should end. For the first, top-left cell of the searched range 'startFrom'=1, for the 2nd one 'startFrom'=2 etc. For the last, bottom-right cell of the searched range 'startFrom'=-1, for the preceding cell 'startFrom'=-2 etc.
**occurrence** - specifies which occurrence of the matching/found value should be used. Positive values indicate top-down searching and counting. Negative values indicate bottom-up searching and counting. For the first match 'occurrence'=1, for the 2nd 'occurrence'=2 etc. For the last match 'occurrence'=-1, for the preceding match 'occurrence'=-2 etc. The number of occurrences is counted from the 'startFrom' index.

The 'type' argument specifies how the searching procedure should be performed. It can be either one of the three values 0, -1, 1 or a combination (sum) of various 'SEARCH::' flags:

- 0 - hLookUp searches a given range linearly for an exact match; 'v' can be a search pattern containing '?' (any single character) and '*' (any string, including an empty string); to search for '?' or '*' place a tilde (~) before them.
- 1 - if an exact match is not found, hLookUp will search for the largest value that is not greater than 'v'; no pattern matching is performed; the searched range must be sorted in the ascending order.

- -1 - if an exact match is not found, hLookUp will search for the smallest value than is not smaller than 'v'; no pattern matching is performed; the searched range must be sorted in the descending order.
- SEARCH::MatchNotGreater (or 2) - if an exact match is not found, the function will search for the largest value that is not greater than 'v'.
- SEARCH::MatchNotSmaller (or 4) - if an exact match is not found, the function will search for the smallest value than is not smaller than 'v'.
- SEARCH::SortAscending (or 8) - perform a quick binary search for a range that is sorted in the ascending order.
- SEARCH::SortDescending (or 16) - perform a quick search for a range that is sorted in the descending order.
- SEARCH::CaseSensitive (or 128) - use case sensitive string comparison.
- SEARCH::FirstMatch (or 256) - find the first match.
- SEARCH::LastMatch (or 512) - find the last match.
- SEARCH::MixedData (or 2048) - the searched range contains both text and numbers.
- SEARCH::RegEx (or 8192) - the 'v' parameter is a regular expression.
- "SEARCH::IgnorePunctuation (or 16384) - use the word sort order (ignoring certain punctuation marks).
- "SEARCH::NeutralSortOrder (or 32768) - use neutral, language independent string comparison instead of the default language specific comparison.
- "SEARCH::Pattern (or 65536) - the \"v\" parameter is a simple (wildcard) pattern; can't be used with fast binary searching.

The '1' value is an equivalent to (SEARCH::SortAscending + SEARCH::MatchNotGreater).
The '-1' value is an equivalent to (SEARCH::SortDescending + SEARCH::MatchNotSmaller).
The '0' value is an equivalent to (0).
If 'type' is omitted, it's assumed to be 1.

The SEARCH::Pattern and SEARCH::RegEx flags can't be used with SEARCH::MatchNotGreater, SEARCH::MatchNotSmaller, SEARCH::StringSort, SEARCH::CaseSensitive, SEARCH::SortAscending, SEARCH::SortDescending.

The SEARCH::MatchNotGreater and SEARCH::MatchNotSmaller flags can not be used with the SEARCH::FirstMatch and SEARCH::LastMatch flags.

If neither SEARCH::FirstMatch nor SEARCH::LastMatch is specified, the linear search returns the first match and the quick search may return any of the existing matches.

If SEARCH::SortAscending or SEARCH::SortDescending is specified, the searched range either should not contain any formulas or the formulas should not break the sort order during the recalculation. Additionally, in such a case, no circular reference will be reported for cells other than the result cell.

Typically, quick binary searches enabled by specifying the SEARCH::SortAscending or SEARCH::SortDescending flags should be significantly (tens/hundreds of times) faster then the plain linear searches.

If SEARCH::MixedData is specified, the searched range is assumed to contain both numeric and text cells. If the search value is a text string, all values from that range will be (internally) converted to text strings then compared. If it's a number, all text strings that represent numbers will be converted to numbers before the comparison takes place.

If any of the two sorting flags is used along with the SEARCH::MixedData flag, the searched range must be sorted using such a "mixed" text/numeric method.

If a given workbook is to contain an extremely large number of hLookUp() functions, for better performance, when specifying the above options one can use the resulting numeric code instead of the individual option names.

If the match is not found, it returns the #N/A! error value.

**hLookUp() examples:**

=hLookUp(2, {1, 2, 3; "a", "b", "c"}, 2, 0) returns "b"

=hLookUp(2.5, {1, 2, 3; "a", "b", "c"}, 2, -1) returns "c"

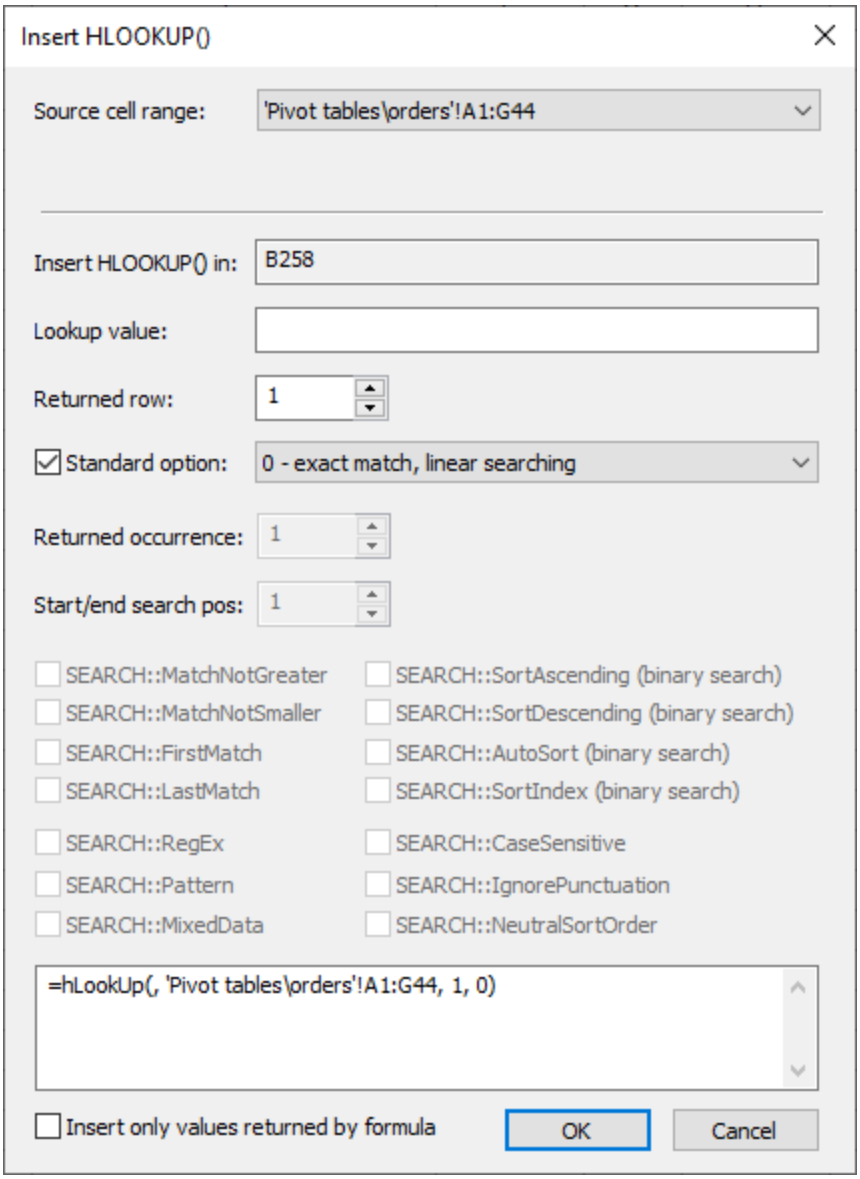
=hLookUp(2.5, {1, 2, 3; "a", "b", "c"}, 2, 1) returns "b"

=hLookUp("*bc??", {"abc", "abcde", "ac"; 1, 2, 3}, 2, 0) returns 2

=hLookUp(2.5, sheet1!b5:d10000, 2, SEARCH::SortAscending + SEARCH::MatchNotGreater)

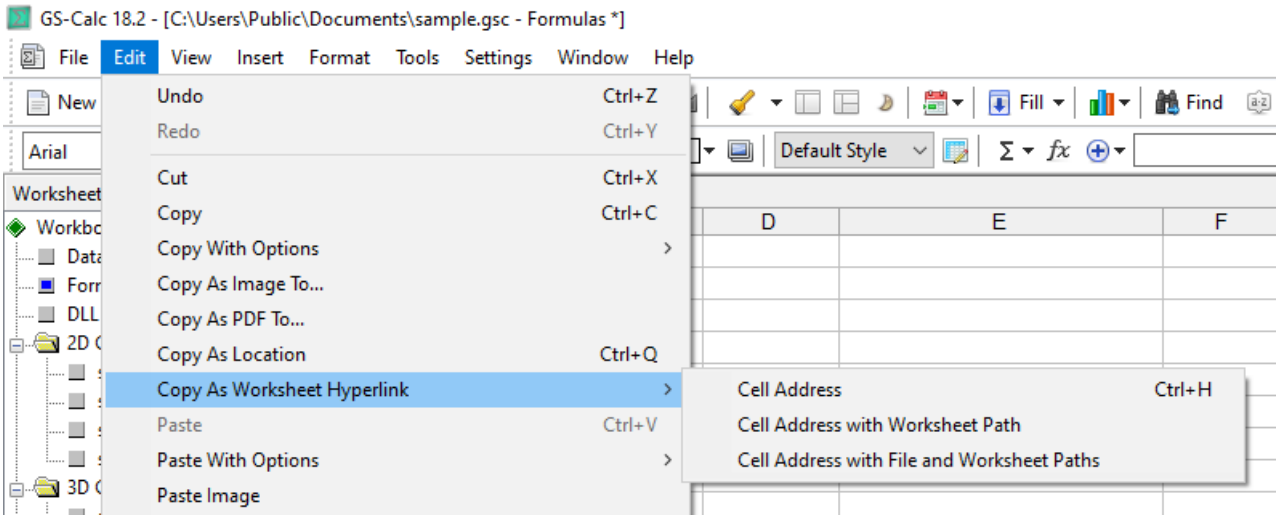
=hLookUp("bc\d", {"abc", "abcde", "abc10"; 1, 2, 3}, 2, SEARCH::RegEx,,) returns 3

You can insert the hLookUp() function with just two clicks using the **Insert > HLOOKUP()** command. The displayed "Insert HLOOKUP()" dialog box determines the optimum parameters, pre-set all the options and automatically creates the formula.

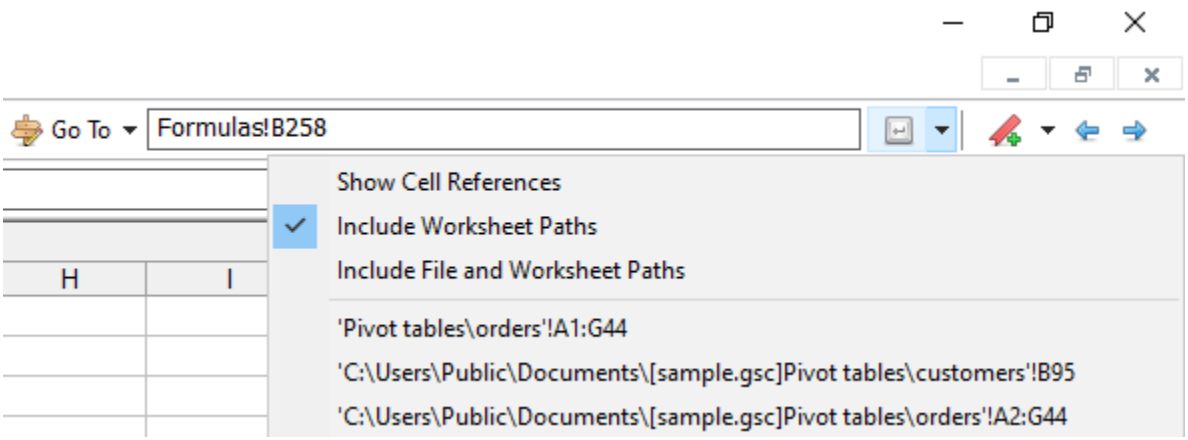



The source range 16 element list is a global list created and stored in the settings file. Subsequent source ranges are added in a circular manner by clicking one of the following:

**The "Copy as Location" menu command**




**The "Enter" toolbar button**




## vLookUp() function

**vLookUp(v, array, n, [type])**
**vLookUp(v, array, n, [type], [startIndex], [occurrence])**

The vLookUp() function searches the top row of 'array' for 'v' and - if found - returns a value from the same column and the n-th row of 'array'.

The 2nd Match() variant uses two additional parameters:
**startFrom** - positive numbers specify where the searching should start and negative numbers specify where it should end. For the first, top-left cell of the searched range 'startFrom'=1, for the 2nd one 'startFrom'=2 etc. For the last, bottom-right cell of the searched range 'startFrom'=-1, for the preceding cell 'startFrom'=-2 etc.
**occurrence** - specifies which occurrence of the matching/found value should be used. Positive values indicate top-down searching and counting. Negative values indicate bottom-up searching and counting. For the first match 'occurrence'=1, for the 2nd 'occurrence'=2

etc. For the last match 'occurrence'=-1, for the preceding match 'occurrence'=-2 etc. The number of occurrences is counted from the 'startFrom' index.

The 'type' argument specifies how the searching procedure should be performed. It can be either one of the three values 0, -1, 1 or a combination (sum) of various 'SEARCH::' flags:

- 0 - vLookUp searches a given range linearly for an exact match; 'v' can be a search pattern containing '?' (any single character) and '*' (any string, including an empty string); to search for '?' or '*' place a tilde (~) before them.
- 1 - if an exact match is not found, vLookUp will search for the largest value that is not greater than 'v'; no pattern matching is performed; the searched range must be sorted in the ascending order.
- -1 - if an exact match is not found, vLookUp will search for the smallest value than is not smaller than 'v'; no pattern matching is performed; the searched range must be sorted in the descending order.
- SEARCH::MatchNotGreater (or 2) - if an exact match is not found, the function will search for the largest value that is not greater than 'v'.
- SEARCH::MatchNotSmaller (or 4) - if an exact match is not found, the function will search for the smallest value than is not smaller than 'v'.
- SEARCH::SortAscending (or 8) - perform a quick binary search for a range that is sorted in the ascending order.
- SEARCH::SortDescending (or 16) - perform a quick binary search for a range that is sorted in the descending order.
- SEARCH::CaseSensitive (or 128) - use case sensitive string comparison.
- SEARCH::FirstMatch (or 256) - find the first match.
- SEARCH::LastMatch (or 512) - find the last match.
- SEARCH::AutoSort (or 1024) - perform background sorting automatically during the first update then use the quick binary searches. In this case the **startFrom** parameter refers to the internally sorted range.
- SEARCH::MixedData (or 2048) - the searched range contains both text and numbers.
- SEARCH::SortIndex (or 4096) - can only be used with SEARCH::AutoSort; if it's specified, Match() will return the index ralated to the internally sorted searched range, not to the actual un-sorted range in the worksheet.
- SEARCH::RegEx (or 8192) - the 'v' parameter is a regular expression.
- "SEARCH::IgnorePunctuation (or 16384) - use the word sort order (ignoring certain punctuation marks).
- "SEARCH::NeutralSortOrder (or 32768) - use neutral, language independent string comparison instead of the default language specific comparison.
- "SEARCH::Pattern (or 65536) - the \"v\" parameter is a simple (wildcard) pattern; can't be used with fast binary searching.

The '1' value is an equivalent to (SEARCH::SortAscending + SEARCH::MatchNotGreater).
The '-1' value is an equivalent to (SEARCH::SortDescending + SEARCH::MatchNotSmaller).
The '0' value is an equivalent to (0).
If 'type' is omitted, it's assumed to be 1.

The SEARCH::Pattern and SEARCH::RegEx flags can't be used with SEARCH::MatchNotGreater, SEARCH::MatchNotSmaller, SEARCH::StringSort, SEARCH::CaseSensitive, SEARCH::SortAscending, SEARCH::SortDescending.

The SEARCH::MatchNotGreater and SEARCH::MatchNotSmaller flags cannot be used with the SEARCH::FirstMatch and SEARCH::LastMatch flags.

If neither SEARCH::FirstMatch nor SEARCH::LastMatch is specified, the linear search returns the first match and the quick search may return any of the existing matches.

If SEARCH::SortAscending or SEARCH::SortDescending is specified, the searched range either must not contain any formulas or the formulas must not break the sort order during the recalculation. Additionally, in such a case, no circular reference will be reported for cells other than the result cell.

Typically, quick binary searches enabled by specifying the SEARCH::SortAscending or SEARCH::SortDescending flags should be significantly (tens/hundreds of times) faster then the plain linear searches.

If SEARCH::AutoSort is specified, GS-Calc will be creating and maintaining sort indices for the referenced searched ranges containing unsorted data. Thanks to those internally created indices it's possible to use quick binary searches for data that doesn't have to be sorted manually by the user. All sort indices are created during the first update and then they are individually updated whenever it's necessary.
The speed gain depends on how many vLookUp() formulas are there in your worksheet, how often the same ranges are re-used and how big the searched ranges are. For very large worksheets, this can be even hundreds of times (and more) faster.
The SEARCH::AutoSort flag must be used with SEARCH::SortAscending or SEARCH::SortDescending. If the searched data is already partially sorted, it's recommended that you use the sort order flag that matches that partial sorting the best.

If SEARCH::MixedData is specified, the searched range is assumed to contain both numeric and text cells. If the search value is a text string, all values from that range will be (internally) converted to text strings then compared. If it's a number, all text strings that represent numbers will be converted to numbers before the comparison takes place.
If any of the two sorting flags is used along with the SEARCH::MixedData flag, the searched range must be sorted using such a "mixed" text/numeric method. If the SEARCH::AutoSort flag is used, GS-Calc will take care about the correct internal sorting. Otherwise use the respective options in the **Tools > Sort Cell Range** dialog box.

If a given workbook is to contain an extremely large number of vLookUp() functions, for better performance, when specifying the above options one can use the resulting numeric code instead of the individual option names.

If the match is not found, it returns the #N/A! error value.

**vLookUp() examples:**

=vLookUp(2, {1, 2, 3; "a", "b", "c"}, 2, 0) returns "b"

=vLookUp(2.5, {1, 2, 3; "a", "b", "c"}, 2, -1) returns "c"

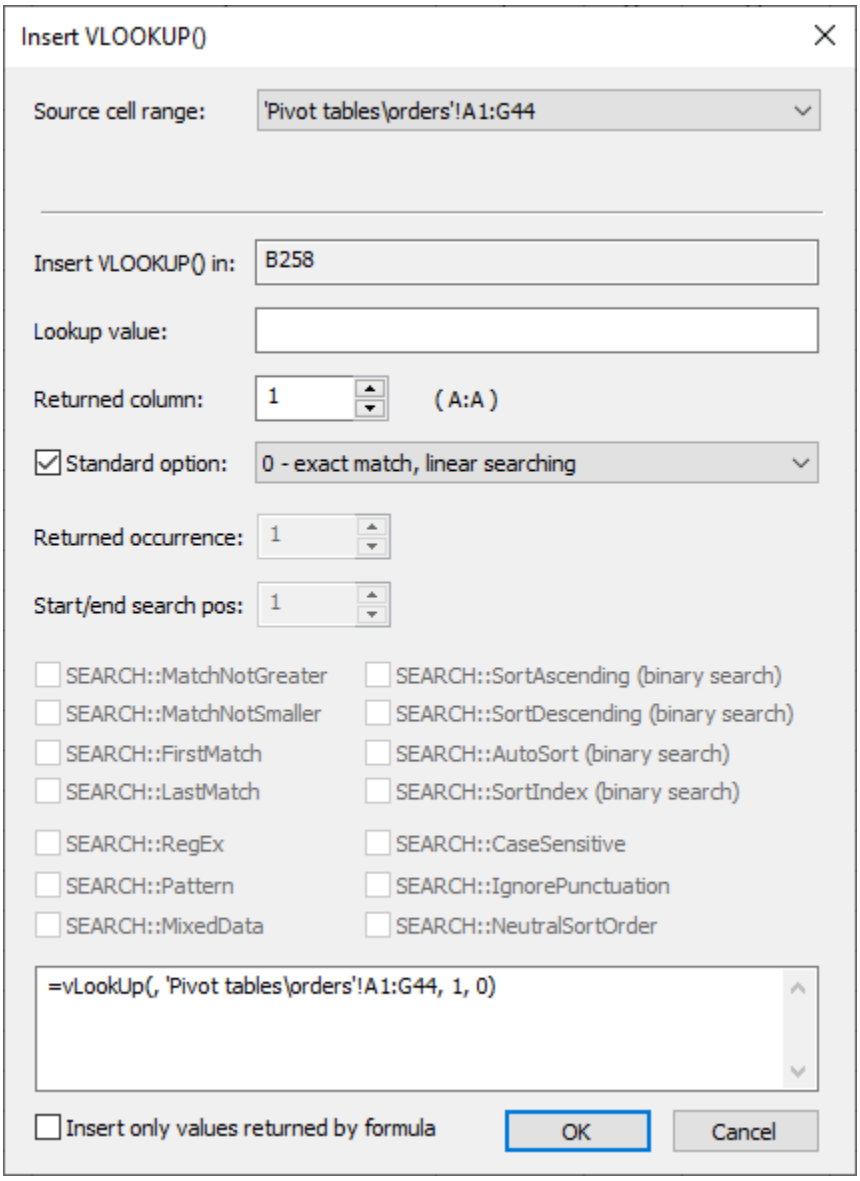
=vLookUp(2.5, {1, 2, 3; "a", "b", "c"}, 2, 1) returns "b"

=vLookUp("*bc??", {"abc", "abcde", "ac"; 1, 2, 3}, 2, 0) returns 2

=vLookUp(2.5, sheet1!b5:d10000, 2, SEARCH::SortAscending + SEARCH::MatchNotGreater)

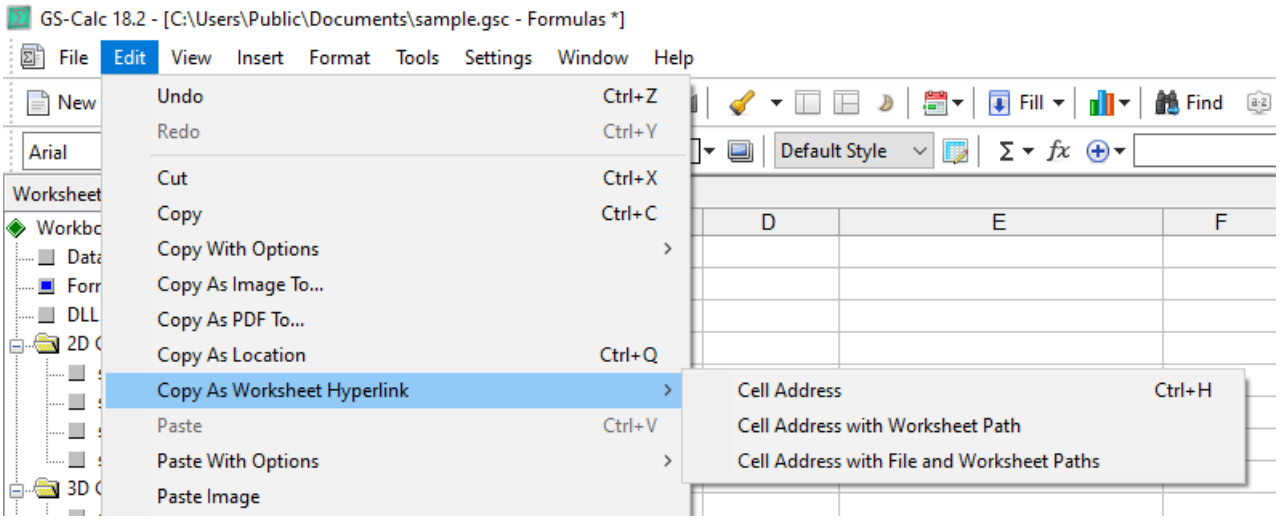
=vLookUp("bc\d", {"abc", 1; "abcde", 2; "abc10", 3}, 2, SEARCH::RegEx) returns 3

You can insert the vLookUp() function with just two clicks using the **Insert > VLOOKUP()** command. The displayed "Insert VLOOKUP()" dialog box determines the optimum parameters, pre-set all the options and automatically creates the formula.

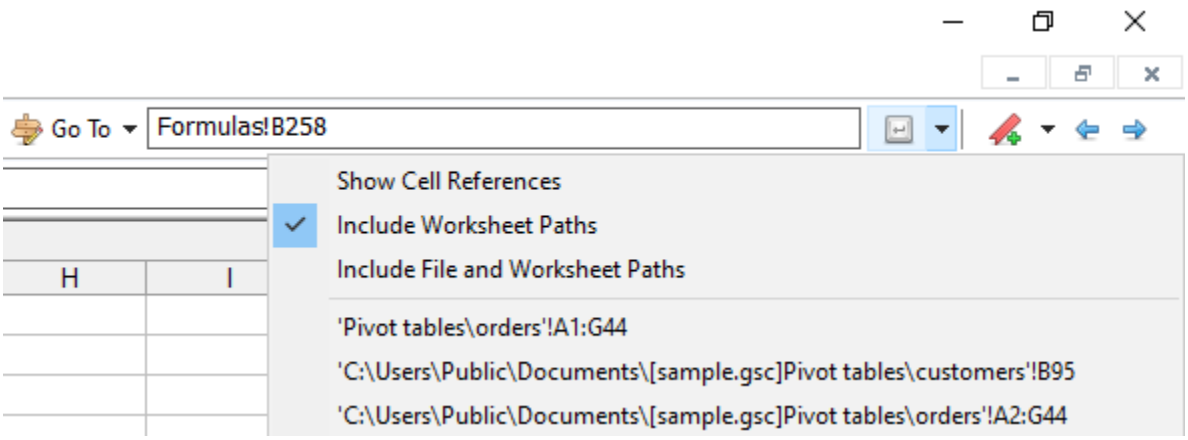


The source range 16 element list is a global list created and stored in the settings file. Subsequent source ranges are added in a circular manner by clicking one of the following:

**The "Copy as Location" menu command**




**The "Enter" toolbar button**




## Match() function

**match(v, array, [options])**
**match(v, array, [options], [startIndex], [occurrence])**

The match() function returns the relative position of a number or text 'v' in 'array'.

The 2nd Match() variant uses two additional parameters:
**startFrom** - positive numbers specify where the searching should start and negative numers specify where it should end. For the first, top-left cell of the searched range 'startFrom'=1, for the 2nd one 'startFrom'=2 etc. For the last, bottom-right cell of the

searched range 'startFrom'=-1, for the preceeding cell 'startFrom'=-2 etc.
**occurrence** - specifies which occurrence of the matching/found value should be used. Positive values indicate top-down searching and counting. Negative values indicate bottom-up searching and counting. For the first match 'occurrence'=1, for the 2nd 'occurrence'=2 etc. For the last match 'occurrence'=-1, for the preceding match 'occurrence'=-2 etc. The number of occurrences is counted from the 'startFrom' index.

The **type** argument specifies how the searching procedure should be performed. It can be either one of the three values 0, -1, 1 or a combination (sum) of various 'SEARCH::' flags:

- 0 - match() searches a given range linearly for an exact match; 'v' can be a search pattern containing '?' (any single character) and '*' (any string, including an empty string); to search for '?' or '*' place a tilde (~) before them.
- 1 - if an exact match is not found, Match() will search for the largest value that is not greater than 'v'; no pattern matching is performed; the searched range must be sorted in the ascending order.
- -1 - if an exact match is not found, Match() will search for the smallest value than is not smaller than 'v'; no pattern matching is performed; the searched range must be sorted in the descending order.
- SEARCH::MatchNotGreater (or 2) - if an exact match is not found, the function will search for the largest value that is not greater than 'v'.
- SEARCH::MatchNotSmaller (or 4) - if an exact match is not found, the function will search for the smallest value than is not smaller than 'v'.
- SEARCH::SortAscending (or 8) - perform a quick binary search for a range that is sorted in the ascending order. The searched range must have the form of a one-column or one-row vector.
- SEARCH::SortDescending (or 16) - perform a quick binary search for a range that is sorted in the descending order. The searched range must have the form of a one-column or one-row vector.
- SEARCH::CaseSensitive (or 128) - use case sensitive string comparison.
- SEARCH::FirstMatch (or 256) - find the first match.
- SEARCH::LastMatch (or 512) - find the last match.
- SEARCH::AutoSort (or 1024) - perform background sorting automatically during the first update then use the quick binary searches. In this case the **startFrom** parameter refers to the internally sorted range.
- SEARCH::MixedData (or 2048) - the searched range contains both text and numbers.
- SEARCH::SortIndex (or 4096) - can only be used with SEARCH::AutoSort; if it's specified, Match() will return the index related to the internally sorted searched range, not to the actual un-sorted range in the worksheet.
- SEARCH::RegEx (or 8192) - the 'v' parameter is a regular expression.
- "SEARCH::IgnorePunctuation (or 16384) - use the word sort order (ignoring certain punctuation marks).
- "SEARCH::NeutralSortOrder (or 32768) - use neutral, language independent string comparison instead of the default language specific comparison.
- "SEARCH::Pattern (or 65536) - the \"v\" parameter is a simple (wildcard) pattern; can't be used with fast binary searching.

The '1' value is an equivalent to (SEARCH::SortAscending + SEARCH::MatchNotGreater).
The '-1' value is an equivalent to (SEARCH::SortDescending + SEARCH::MatchNotSmaller).
The '0' value is an equivalent to (0).
If 'type' is omitted, it's assumed to be 1.

The SEARCH::Pattern and SEARCH::RegEx flags can't be used with SEARCH::MatchNotGreater, SEARCH::MatchNotSmaller, SEARCH::StringSort, SEARCH::CaseSensitive, SEARCH::SortAscending, SEARCH::SortDescending.

The SEARCH::MatchNotGreater and SEARCH::MatchNotSmaller flags can not be used with the SEARCH::FirstMatch and SEARCH::LastMatch flags.

If neither SEARCH::FirstMatch nor SEARCH::LastMatch is specified, the linear search returns the first match and the quick search may return any of the existing matches.

If SEARCH::SortAscending or SEARCH::SortDescending is specified, the searched range either must not contain any formulas or the formulas must not break the sort order during the recalculation. Additionally, in such a case, no circular reference will be reported for cells other than the result cell.

Typically, quick binary searches enabled by specifying the SEARCH::SortAscending or SEARCH::SortDescending flags should be significantly (tens/hundreds of times) faster then the plain linear searches.

If SEARCH::AutoSort is specified, GS-Calc will be creating and maintaining sort indices for the referenced searched ranges containing unsorted data. Thanks to those internally created indices it's possible to use quick binary searches for data that doesn't have to be sorted manually by the user. All sort indices are created during the first update after opening a given workbook and then they are individually updated whenever it's necessary.
The speed gain depends on how many match() formulas are there in your worksheet, how often the same ranges are re-used and how big the searched ranges are. For very large worksheets, this can be even hundreds of times (and more) faster.
The SEARCH::AutoSort flag must be used with SEARCH::SortAscending or SEARCH::SortDescending. If the searched data is already partially sorted, it's recommeneded that you use the sort order flag that matches that partial sorting the best.

If SEARCH::MixedData is specified, the searched range is assumed to contain both numeric and text cells. If the search value is a text string, all values from that range will be (internally) converted to text strings then compared. If it's a number, all text strings that represent numbers will be converted to numbers before the comparison takes place.
If any of the two sorting flags is used along with the SEARCH::MixedData flag, the searched range must be sorted using such a "mixed" text/numeric method. If the SEARCH::AutoSort flag is used, GS-Calc will take care about the correct internal sorting. Otherwise use the respective options in the **Tools > Sort Cell Range** dialog box.

If a given workbook is to contain an extremely large number of match() functions, for better performance, when specifying the above options one can use the resulting numeric code instead of the individual option names.

If the 'options' parameter is omitted, it's assumed to be 1.

If the searched value is not found, 'match' returns the #N/A! error value.

**match() examples:**

=match(2, {1, 2, 3}, 0) returns 2

=match("b", {1, 2, 3; "a", "b", "c"}, 0) returns 5

=match("*bc??", {"abc", "abcde", "ac"}, 0) returns 2

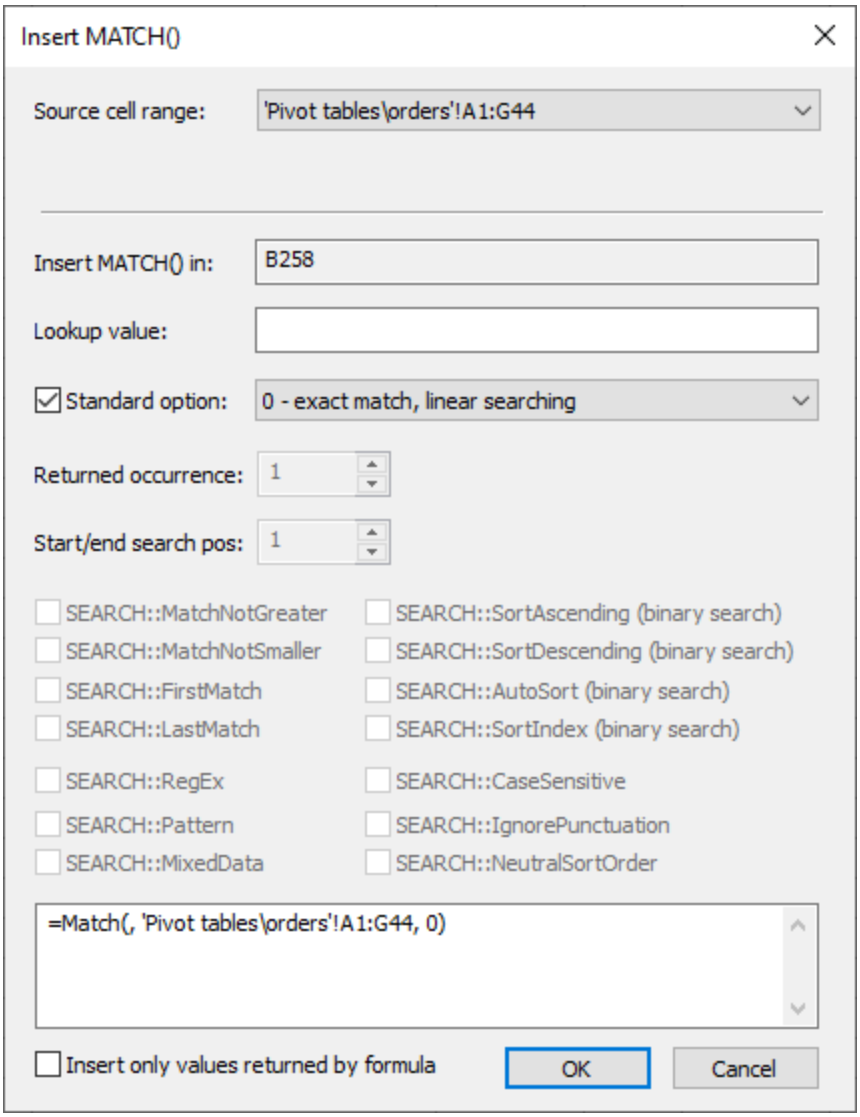
=match("*bc??", {"abc", "abcde", "ac"; 1, 2, 3}, 2, 0) returns 2

=match("b", sheet1!b5:d10000, SEARCH::SortAscending + SEARCH::MatchNotGreater)

=match(10.5, sheet1!b5:d10000, SEARCH::SortAscending + SEARCH::MatchNotGreater, 3, 2)

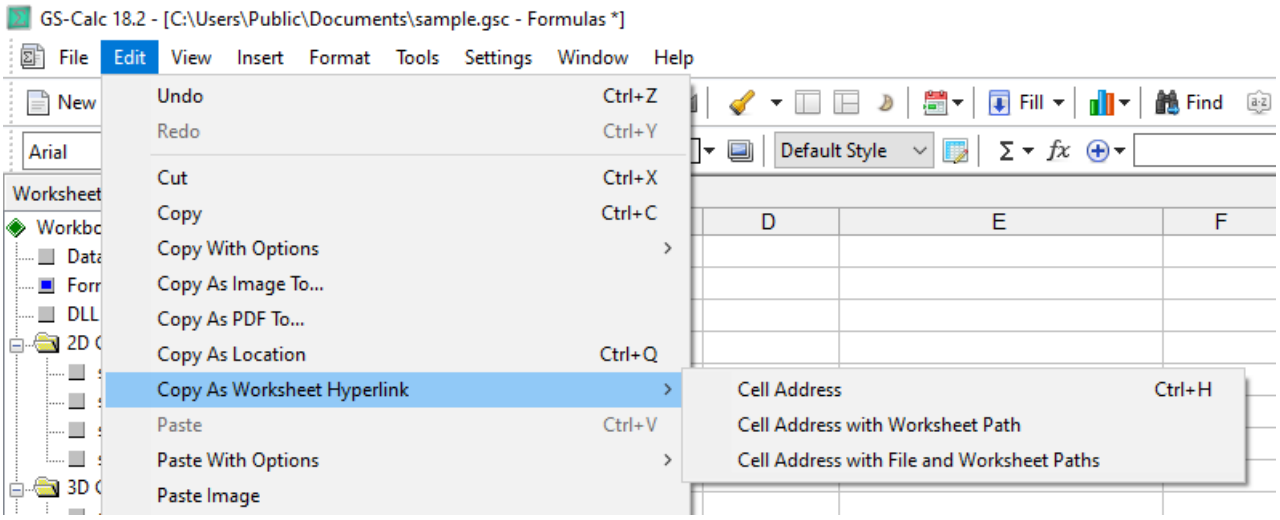
=match("bc\d", {"abc", "abcde", "abc10"}, SEARCH::RegEx) returns 3

You can insert the match() function with just two clicks using the **Insert > MATCH()** command. The displayed "Insert MATCH()" dialog box determines the optimum parameters, pre-set all the options and automatically creates the formula.

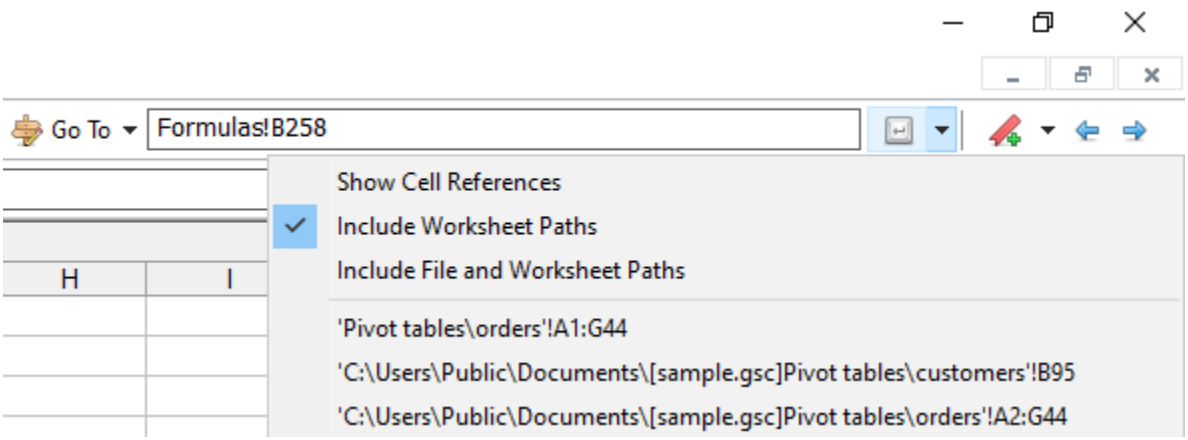



The source range 16 element list is a global list created and stored in the settings file. Subsequent source ranges are added in a circular manner by clicking one of the following:

**The "Copy as Location" menu command**




**The "Enter" toolbar button**




## Unique() function

**Unique(vector, [type])**

Searches a column (array or range) for unique values.

If 'type' is 0, the 'unique' function returns a vector of indices to the subsequent unique values found in 'vector'.

If 'type' is 1, the 'unique' function returns a vector of unique values found in 'vector'.

If 'type' is 2, the 'unique' function returns a two-column array.
The first column contains unique values, the second one: the number of occurrences of the corresponding value.

If 'type' is omitted, it's assumed to be 1.

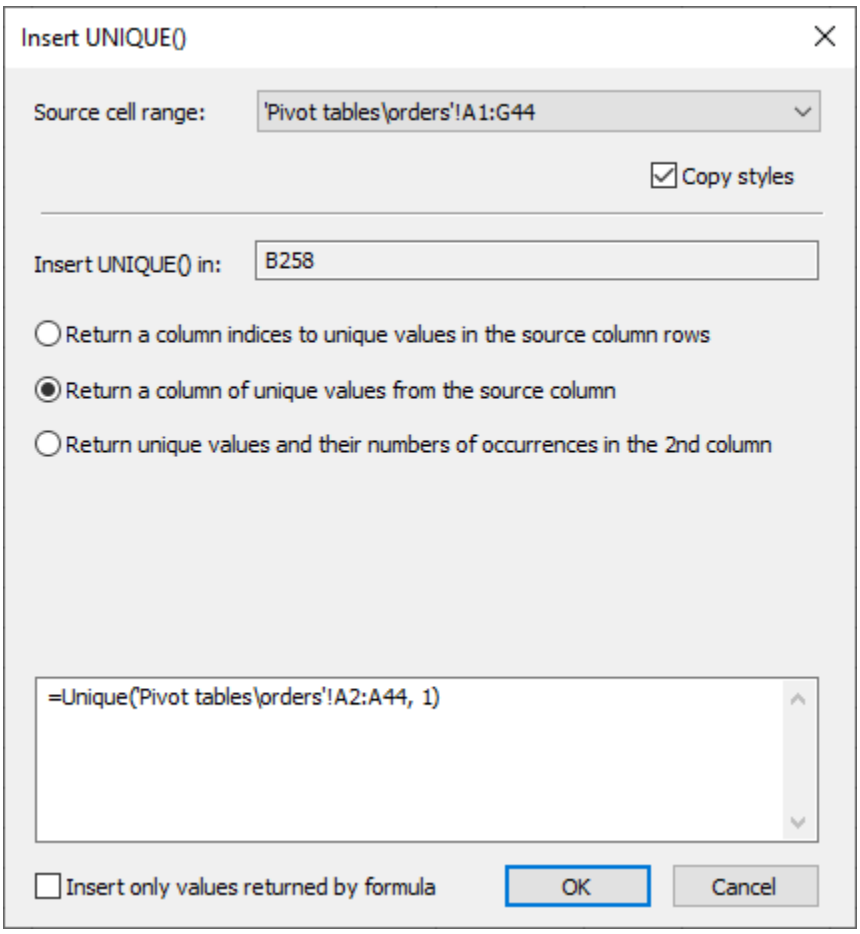
**Unique() examples:**

=unique({2; 3; 1; 2; 5; 3; 1}, 1) returns {1; 2; 3; 5}

=unique({"a"; "b"; "cd"; "a"; "cd"; "b"; "d"}, 0) returns {1; 2; 3; 5}

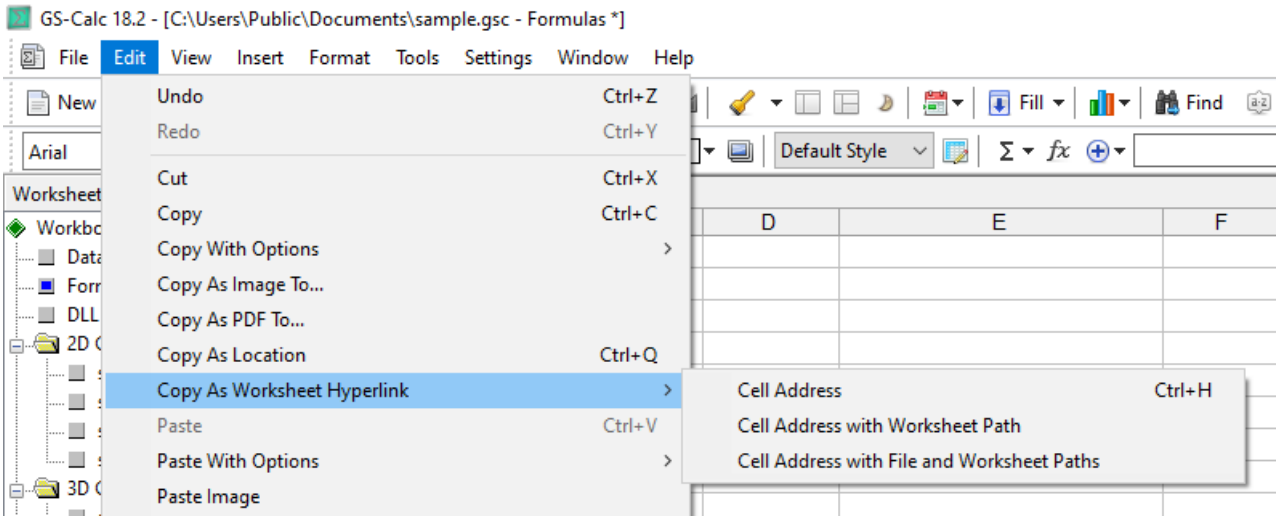
=unique({"a"; "b"; "cd"; "a"; "cd"; "b"; "d"}, 2) returns { "a", 2; "b", 2; "cd", 2; "d", 1}

You can insert the Unique() function with just two clicks using the **Insert > UNIQUE()** command. The displayed "Insert UNIQUE()" dialog box determines the optimum parameters including the option to copy styles from the selected source data range and automatically creates the formula.

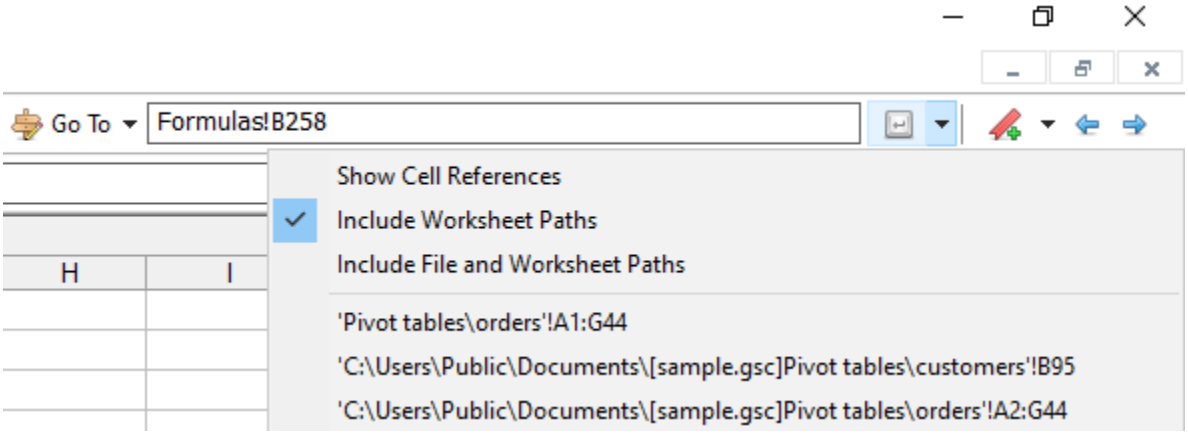


The source range 16 element list is a global list created and stored in the settings file. Subsequent source ranges are added in a circular manner by clicking one of the following:

**The "Copy as Location" menu command**




**The "Enter" toolbar button**




## Sort() function

**Sort(array, column1, order1 [, column2, order2, column3, order3])**

Sorts an array/range based on the specified (column, sorting order) pairs and returns the result as a vector of the corresponding rows indices.

The 1-based column indices specify the relative column positions within the source range.

The sorting orders be one of two values:
0 - descending order
1 - ascending order.

Use the 'index' function to obtain the final sorted array.

**Sort() examples:**

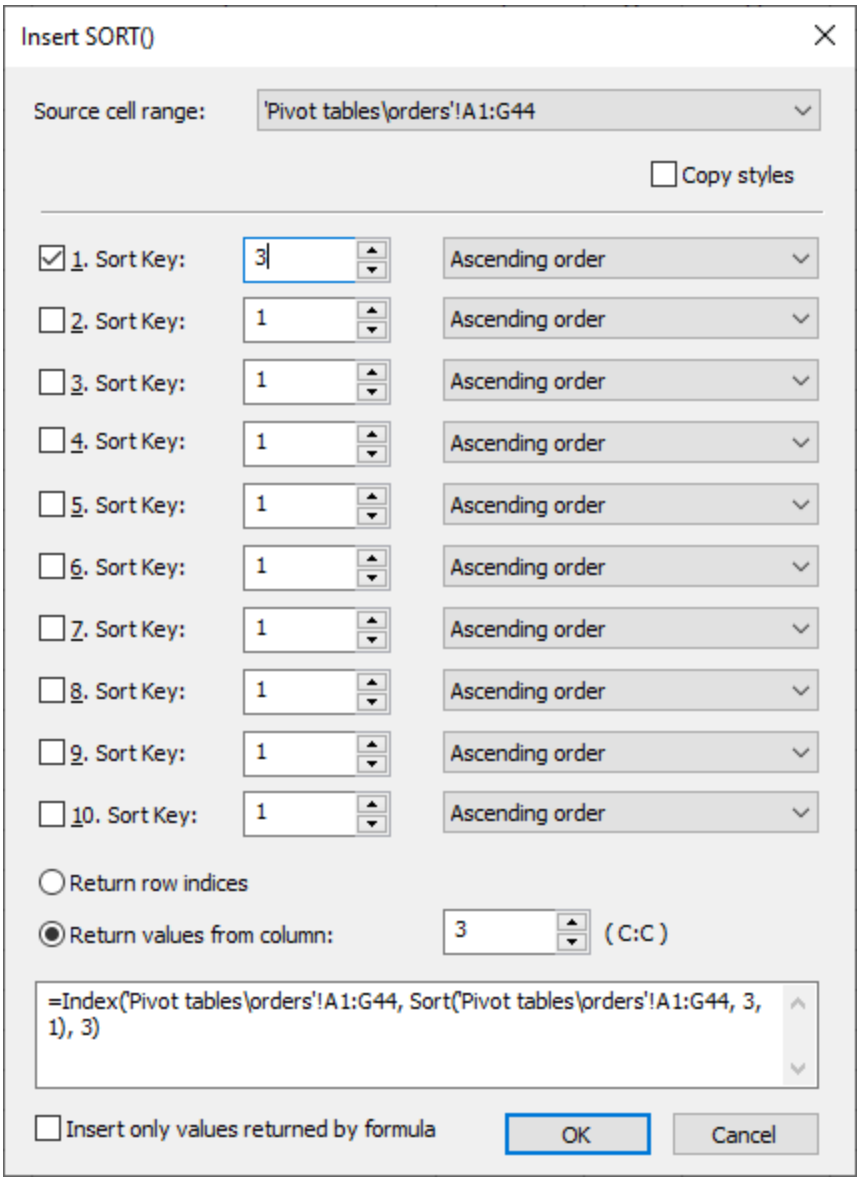
=sort({3; 1; 5; 2; 4}, 1, 1) returns {2; 4; 1; 5; 3}

=sort({3; 1; 5; 2; 4}, 1, 0) returns {3; 5; 1; 4; 2}

=sort({2, "b"; 2, "a"; 3, "d"; 3, "c"; 1, "e"}, 1, 1, 2, 1) returns { 5; 2; 1; 4; 3}

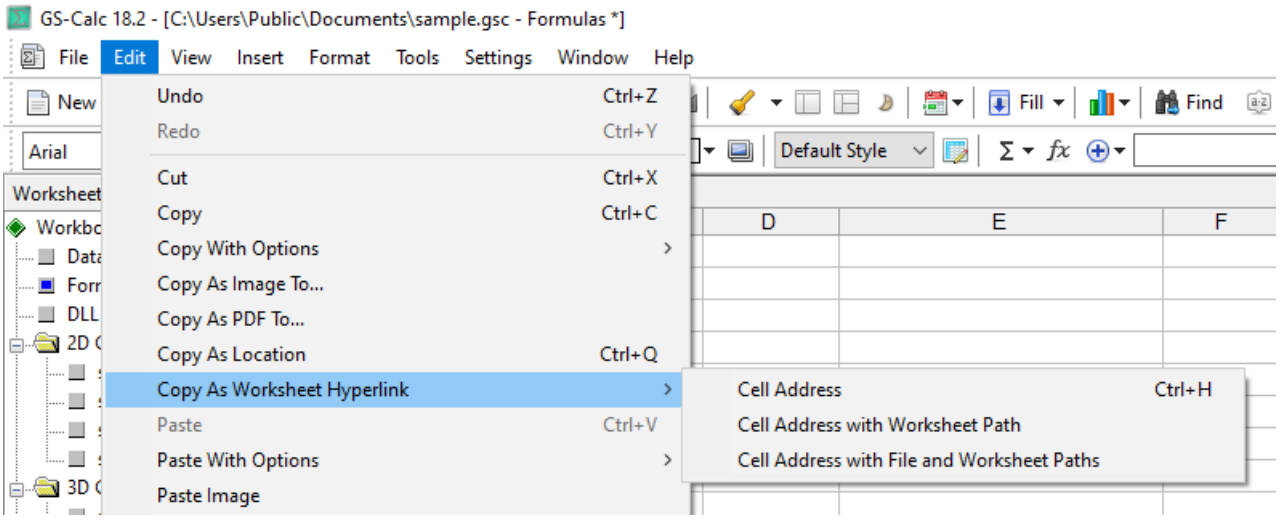
=index({2, "b"; 2, "a"; 3, "d"; 3, "c"; 1, "e"}, {5; 2; 1; 4; 3}, 1) returns {1; 2; 2; 3; 3}
where {5; 2; 1; 4; 3} is the result of the above sorting.

You can insert the Sort() function with just two clicks using the **Insert > SORT()** command. The displayed "Insert SORT()" dialog box determines the optimum parameters including the option to copy styles from the selected source data range and automatically creates the formula.

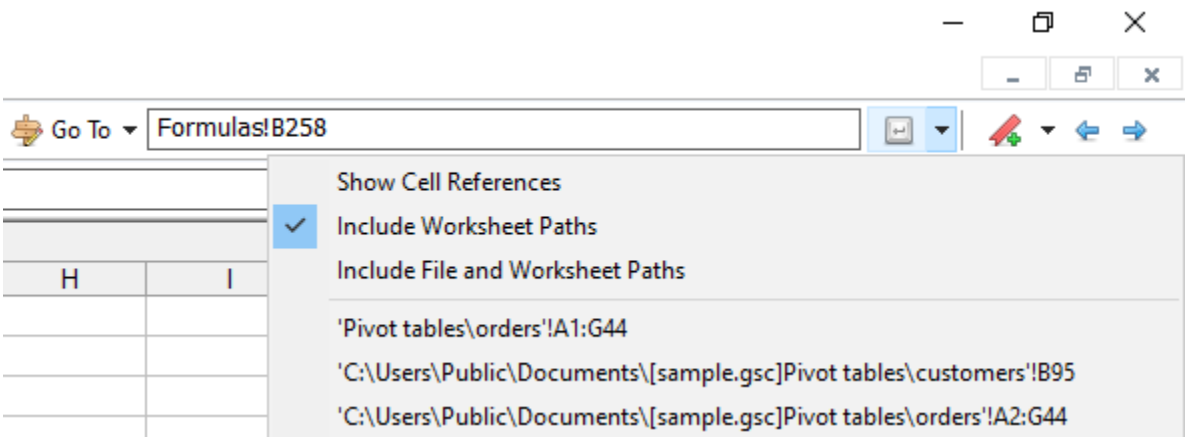



The source range 16 element list is a global list created and stored in the settings file. Subsequent source ranges are added in a circular manner by clicking one of the following:

**The "Copy as Location" menu command**




**The "Enter" toolbar button**




# Regular Expressions

## Using regular expressions in formulas

Regular expression can be used in the following formulas:

- Find() and Replace() formulas
- Lookup and matching formulas: Match(), vLookUp() and hLookUp()

To use RE, each of these functions requires you to specify the **SEARCH::RegEx** (or 8192) flag/constant as its parameter as described in the **Formula Composer** window.

In the Find() function you can also use the **SEARCH::RegExStr** flag/constant. If it's specified, instead of the index the function returns the substring that matches the pattern. For example:

=find("\w\d{2,3}", "abc4567ghr", 1, SEARCH::RegEx) returns 3

=find("\w\d{2,3}", "abc4567ghr", 1, SEARCH::RegExStr) returns "c456"

Some examples of regular expressions:

| | |
|---|---|
| abc | finds cells containing (substrings) "abc" |
| .bc | finds cells containing three character substrings consisting of any character followed by "bc" |
| \Aabc | cell contents starting with "abc" |
| abc\z | cell contents ending with "abc" |
| \Aabc.*123\z | cell contents starting with "abc" and ending with "123" with any number of other characters in between |
| abc\d\z | substrings ending with "abc" and one digit |
| ^a\d+ | cell contents containing a line that starts with "a" and at least one digit |
| ^a\d* | cell contents containing a line that starts with "a" and zero or more digits |
| a\d*$ | cell contents containing a line that ends with "a" and zero or more digits |
| [ab]+c | substrings "abc", "aabc", "abbabc" etc. and not "c" |
| [ab]*c | substrings "abc", "aabc", "abbabc" etc. and "c" |
| [^ab]+c | substrings ending with "c" and not containing "a" or "b" |
| \w\d{2,3} | substrings consisting of one letter followed by 2 or 3 digits |
| ^ab\d{2,3}c | cell contents beginning with "ab", two or three digits and "c" |
| abc\|xyz | cells containing substrings "abc" or "xyz" |

For more information, see: PCRE regular expression syntax summary

The **Replace()** function additionally accept the replacing string which - if the **SEARCH::RegEx** flag is set - can contain:

1. Absolute references (by number) to capturing sub patterns, eg. \1, \2...
   a capturing sub pattern (or a "group") is a part of the pattern enclosed in () parenthesis.
2. \l, \L, \u, \U literals to (binary) switch upper- and lower-case conversion.
3. \r, \n - 'line feed' and 'new line' literals (by default, Ctrl+Enter inserts the \r\n sequence into the text field).
4. \s - a single space character.

Example 1.

| | |
|---|---|
| **Find** pattern: | ab+ |
| Database field content: | abcdefaabb |
| **Replace** string: | x |
| Replacement result: | xcdefax |

Example 2.

| | |
|---|---|
| **Find** pattern: | (.)a+\d{1,3} |
| Database field content: | abc aa0102 |
| **Replace** string: | \1 |
| Replacement result: | abc 2 |

Example 3.

| | |
|---|---|
| **Find** pattern: | (ab) |
| Database field content: | abcdef ghijk abb123 |
| **Replace** string: | \u\1\l\1xyz\s |
| Replacement result: | ABabxyz cdef ghijk ABabxyz b123 |

Example 4.

| | |
|---|---|
| **Find** pattern: | \R |
| Database field content: | abc<br>def<br>ghi |
| **Replace** string: | \s |
| Replacement result: | abc def ghi |

## Find & Replace - full text searches and replacing

To perform full-text searching use the **Edit > Find (F3)** command and in the **Find** field of the displayed **Find & Replace** toolbar specify the data to search for. Clicking the **Options** button, you can specify how the searched data should be treated: as a regular expression or as plain text string.

Selecting a range of cells limits the searching and replacing procedure to that range. Clicking a worksheet (any cell) invalidates the active search range and searching is performed within the whole worksheet till a new selection is made.
Changing search/replace options or text strings also initiates a new search with the current (new) selection.

Searching is performed as follows:

- Text and numeric cells - searching the string representing its unformatted contents.
- Formulas - searching the string representing the formula.

There are three search modes that can be specified using the **Options** button menu:

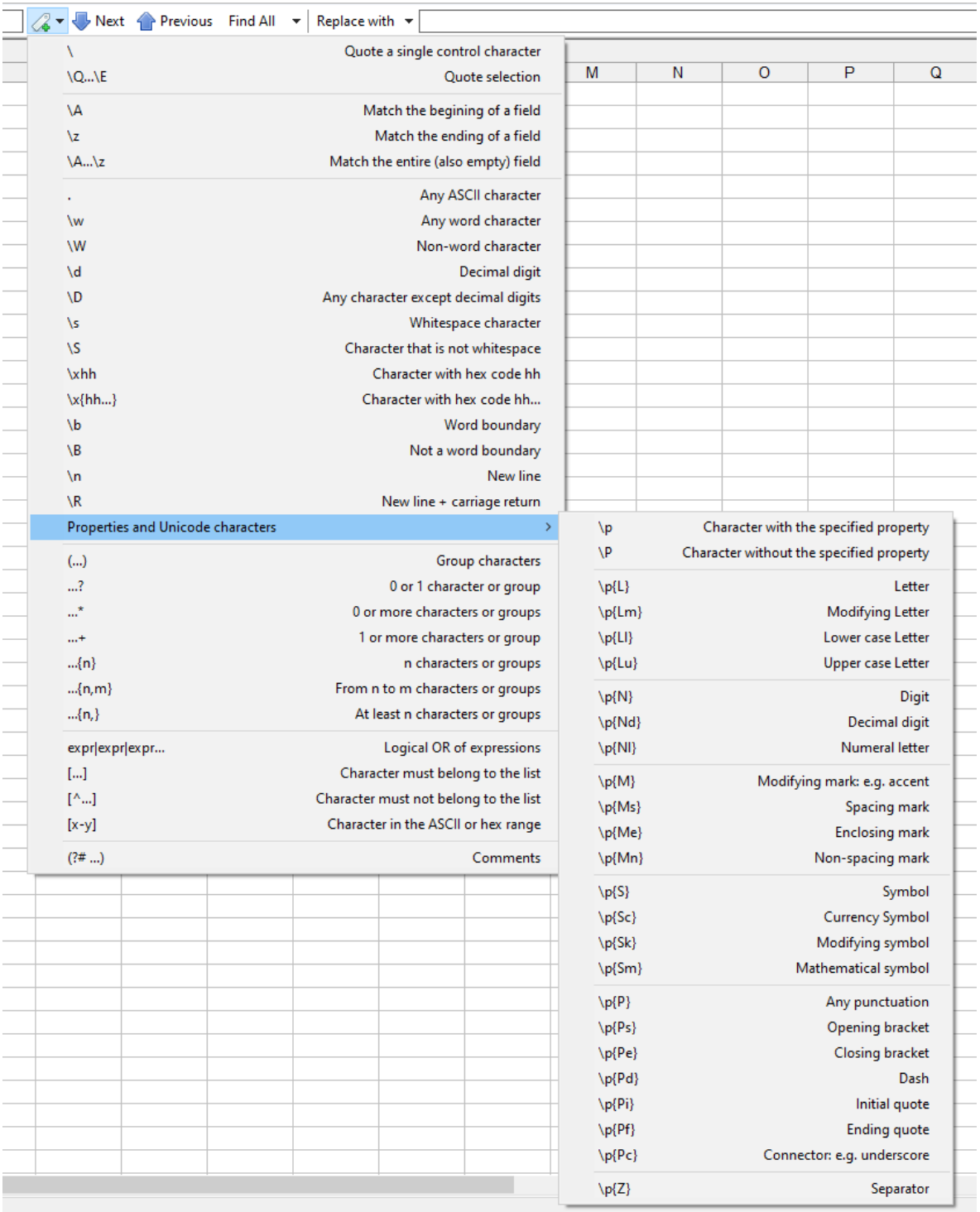
- **Regular Expressions** - the **Find** string is a data pattern based on the standard regular expression syntax. Some examples of regular expressions:

| | |
|---|---|
| abc | finds cells containing (substrings) "abc" |
| .bc | finds cells containing three character substrings consisting of any character followed by "bc" |
| \Aabc | cell contents starting with "abc" |
| abc\z | cell contents ending with "abc" |
| \Aabc.*123\z | cell contents starting with "abc" and ending with "123" with any number of other characters in between |
| abc\d\z | substrings ending with "abc" and one digit |
| ^a\d+ | cell contents containing a line that starts with "a" and at least one digit |
| ^a\d* | cell contents containing a line that starts with "a" and zero or more digits |
| a\d*$ | cell contents containing a line that ends with "a" and zero or more digits |
| [ab]+c | substrings "abc", "aabc", "abbabc" etc. and not "c" |
| [ab]*c | substrings "abc", "aabc", "abbabc" etc. and "c" |
| [^ab]+c | substrings ending with "c" and not containing "a" or "b" |
| \w\d{2,3} | substrings consisting of one letter followed by 2 or 3 digits |
| ^ab\d{2,3}c | cell contents beginning with "ab", two or three digits and "c" |
| abc\|xyz | cells containing substrings "abc" or "xyz" |
| (?=.*abc)(?=.*xyz) | matches cell contents contaning "abc" and "xyz" - logical AND |
| \A\z | matches empty cells |

- Using the **Options** button you can set two additional options: **Allow empty matches** and **Match case**.
  Empty matches occur for **Regular Expression** like a? or \A\z , which lets you search for e.g. empty cells with data blocks.
- For more information, see PCRE regular expression syntax summary
- The **Replace** string can contain:
  (1) absolute references (by number) to capturing subpatterns, e.g. \1, \2...
  a capturing subpattern (or a "group") is a part of the pattern enclosed in () parenthesis. (2) \l, \L, \u, \U literals to (binary) switch upper- and lower-case conversion
  (3) \r, \n - 'line feed' and 'new line' literals (by default, Ctrl+Enter inserts the \r\n sequence into the text field).
  (4) \s - a single space character.

- **Notes:**
  - **To find and replace Unicode/non-ascii characters, use Unicode reg. expressions,**
    **e.g. \p{L} instead of \w, \P{L} instead of \W etc.**
  - **Matching empty cells requires checking the "Options > Allow Empty Matches" option on the "Find & Replace" toolbar.**

The regular expression tags can be easily inserted via the "RegEx" menu available in GS-Base




Examples:

**Find** pattern: cat
**Replace** string: dog
Result: replaces "cat" with "dog"

**Find** pattern: \\
**Replace** string: /
Result: replaces the "\" characters with "/"

**Find** pattern: \A\z
**Replace** string: NULL
Result: fills empty fields with the "NULL" strings

**Find** pattern: \ANULL\z
**Replace** string:
Result: if a field contains only the "NULL" string, deletes its contents

**Find** pattern: \b(\w+)(?:\W+\1\b)+
**Replace** string: \1
Result: removes duplicate words in fields

**Find** pattern: (\b\w)(\w*(\W+|\z))
**Replace** string: \u\1\l\2
Result: converts first letters in words to uppercase, other to lowercase

**Find** pattern: (\b\w)(\w*(\W+|\z))
**Replace** string: \1
Result: creates abbreviations consisting of first letters of words

**Find** pattern: (\b\w(\d*))(\w*(\W+|\z))
**Replace** string: \1
Result: creates abbreviations consisting of first letters of words, leaves full numbers

**Find** pattern: \b((\d{1,3}\.){3,3})\d{1,3}\b
**Replace** string: \1*
Result: mask IP address (e.g. 11.12.13.114 to 11.12.13.*)

**Find** pattern: (\S*)(\s*)(\S*)(\s*)(\S*)
**Replace** string: \g5\g4\g3\g2\g1
Result: reverses the order of up to first 3 words in fields

**Find** pattern: \R
**Replace** string: \s
Result: replaces line-breaks with spaces

**Find** pattern: ab+
Database field content: abcdefaabb
**Replace** string: x
Replacement result: xcdefax

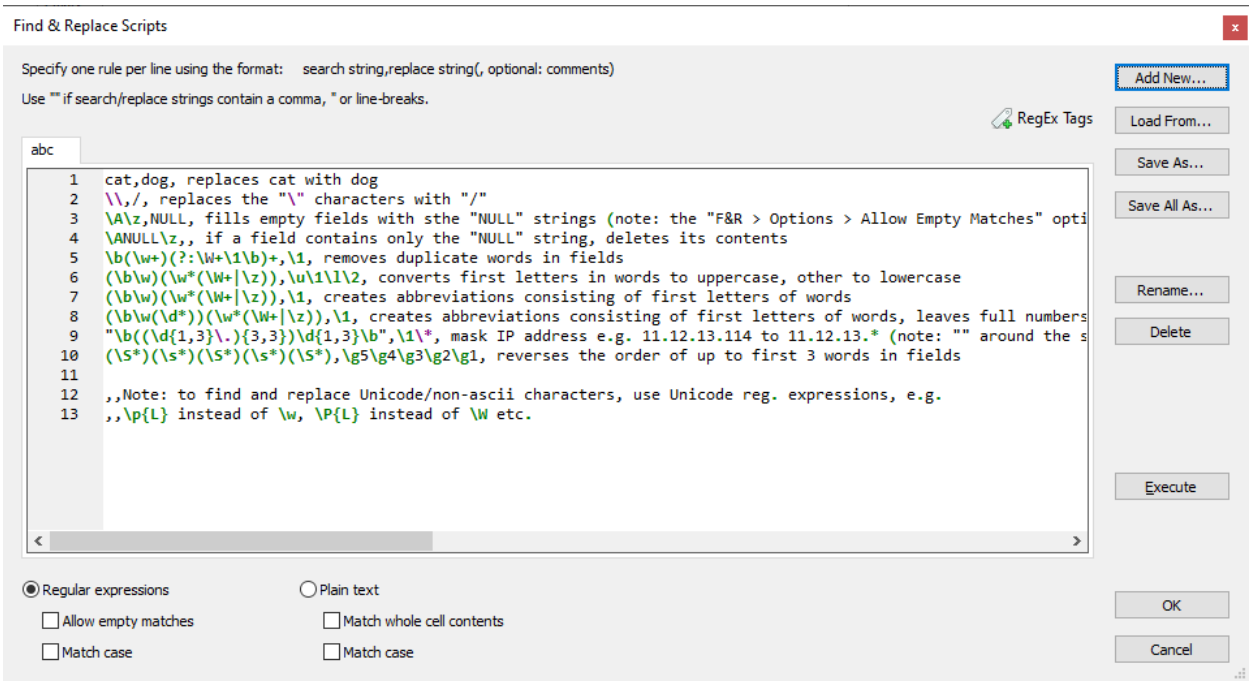
**Find** pattern: (.)a+\d{1,3}
Database field content: abc aa0102
**Replace** string: \1
Replacement result: abc 2

**Find** pattern: (ab)

| | |
|---|---|
| Database field content: | abcdef ghijk abb123 |
| **Replace** string: | \u\1\l\1xyz\s |
| Replacement result: | ABabxyz cdef ghijk ABabxyz b123 |
| **Find** pattern: | \R |
| Database field content: | abc<br>def<br>ghi |
| **Replace** string: | \s |
| Replacement result: | abc def ghi |

- **Plain Text - Partial Matching** - the **Find** string represents a plain text string and the function searches for any substring of the cell contents. The string can contain wildcard "?" (any character) and "*" (any string) characters. Any non-wildcard "?" and "*" characters must be prefixed with a tilde (~).
- **Plain Text - Full Content Matching** - the **Find** string represents a plain text string that is compared against the whole cell contents. The string can contain wildcard "?" (any character) and "*" (any string) characters. Any non-wildcard "?" and "*" characters must be prefixed with a tilde (~).
- The "Scripts" button on the "Find & Replace" toolbar can be used to performed quick mass text replacing in a given table.

**Note: the "search" and "replace" strings entered in the window below must be entered like in a csv text file, so text containing commas, line breaks or quoting symbols must be in "" and inner quoting symbols must be doubled.**



# Using the FILTER() function

## Filtering tables

The FILTER() function enables you to filter data with the number of filters up to the max. number of columns in a worksheet, sort the results using up to six keys and pre- or post-filter the data to find compound duplicates consisting of up to 100 keys/columns.

Filter types include: Regex (regular expressions), patterns (with *, ?,~), "similar to" (fuzzy searches), all of the numeric and literal =, <, >, relations, ranges, content length comparisons, empty cells, duplicates.

The function uses the following parameters:

> **=Filter**(search-range, filters, hyperlink-path, options[, empty-string])

**search-range**

The "search-range" argument represents a range (or an array) with rows to filter. It can be of any size up to the maximum number of columns in a worksheet minus one (as the 1st output column may additionally contain hyperlinks).

If there are any rows found in the search range, FILTER() returns an array with the corresponding number of columns.

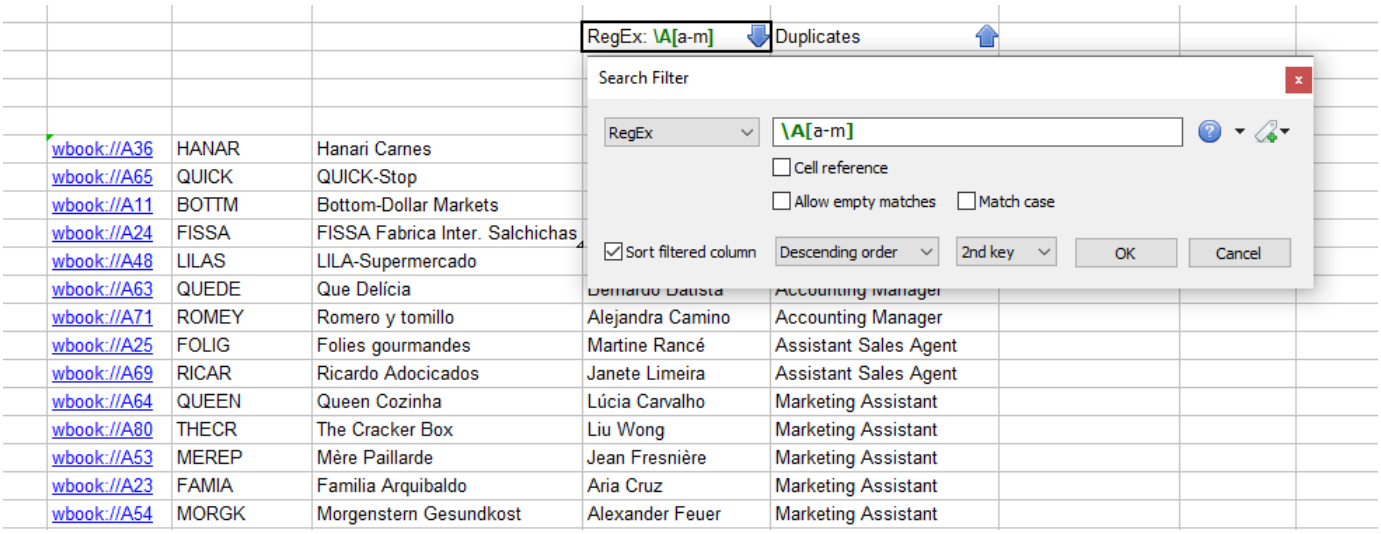
If you specify the "hyperlink" option as described below and apply the **Format > Hyperlink** style to the first column of the range when the returned array is, this first column will contain hyperlinks to the original rows in "search-range".
Thus the filtered rows/records can be easily edited. After clicking a given link (or pressing SPACE) and completing editing of the source data, you just click the "back" bookmark toolbar button (or press Alt+F11) to go back to browsing results in the same place.

**filters**

The "filters" argument represents a range or an array with filtering expressions for the subsequent columns in the **"search-range"** range. Thus the number of cells in "filters" must not exceed the number of columns in the **"search-range"** range. The n-th **cell position** in this range/array is a filter expression for the n-th column in the "search-range" data range.
If there is no filter for a given column, the corresponding filter cell can be left empty.

Entering filters in worksheet cells is easy: you can use the **Format > Search Filter** format style for the desirable cells. This will display formatted filters with e.g. syntax coloring for

Regex expressions and graphic sort order indications. When editing such cells, GS-Calc displays a cell-aligned dialog box with all the filter specification and sort options:



Example

The filter options are as follow:

- Cell reference - the filter text is cell reference (for the "between" filter type: two references)
- Allow empty matches - required by RegEx if you want to use expressions like \A\z
- Match case
- Word sort order - when sorting and searching for duplicates ignore punctuations and non-word characters
- Sorting and the sort key index - for a single column sorting it's always "1st key"; if you want to use multi-column sorting you must specify subsequent key indices ("2nd", "3rd", ..."6th") for the other sorted columns. If there are repeated indices, FILTER() returns the "#VALUE!" error code.

**Note:** The "duplicates" filters are handled slightly differently than the other filters. The "options" parameter enables you to specify whether searching for duplicates should be performed before or after all other filters are applied.

You can also specify filters directly as text cells, e.g. if you want to generate them by your formulas.

**hyperlink-path**

The "hyperlink-path" argument represents a worksheet/workbook path that will be be used along with cell references if the "include hyperlinks" options is specified (see below).

For example: sheet1, folder1\sheet1, c:\documents\[sample.gsc]sheet1

On the above example screen, the hyperlink path is left empty as both the data (A1:H100000) and the result (=FILTER(A1:H100000, ...) are in the same worksheet.

**options**

The "option" argument is a number and can be 0 or a combination (a sum from 1 to 7) of the following:

- 1 - include hyperlinks to the original search-range data rows in the first column in of the FILTER() function results
- 2 - perform searching for duplicates AFTER all other filters are applied; by default, the searches for duplicates are performed first
- 4 - as FILTER() is supposed to offer the best possible speed when handling very large tables, by default it doesn't create calculation chains with formulas placed in the "search-range" range; if e.g. some column in the "search-range" consists of formulas and you want to make sure FILTER() is executed after they're updated, add this "4" value to options. Please note that this will result in significantly slower filtering.

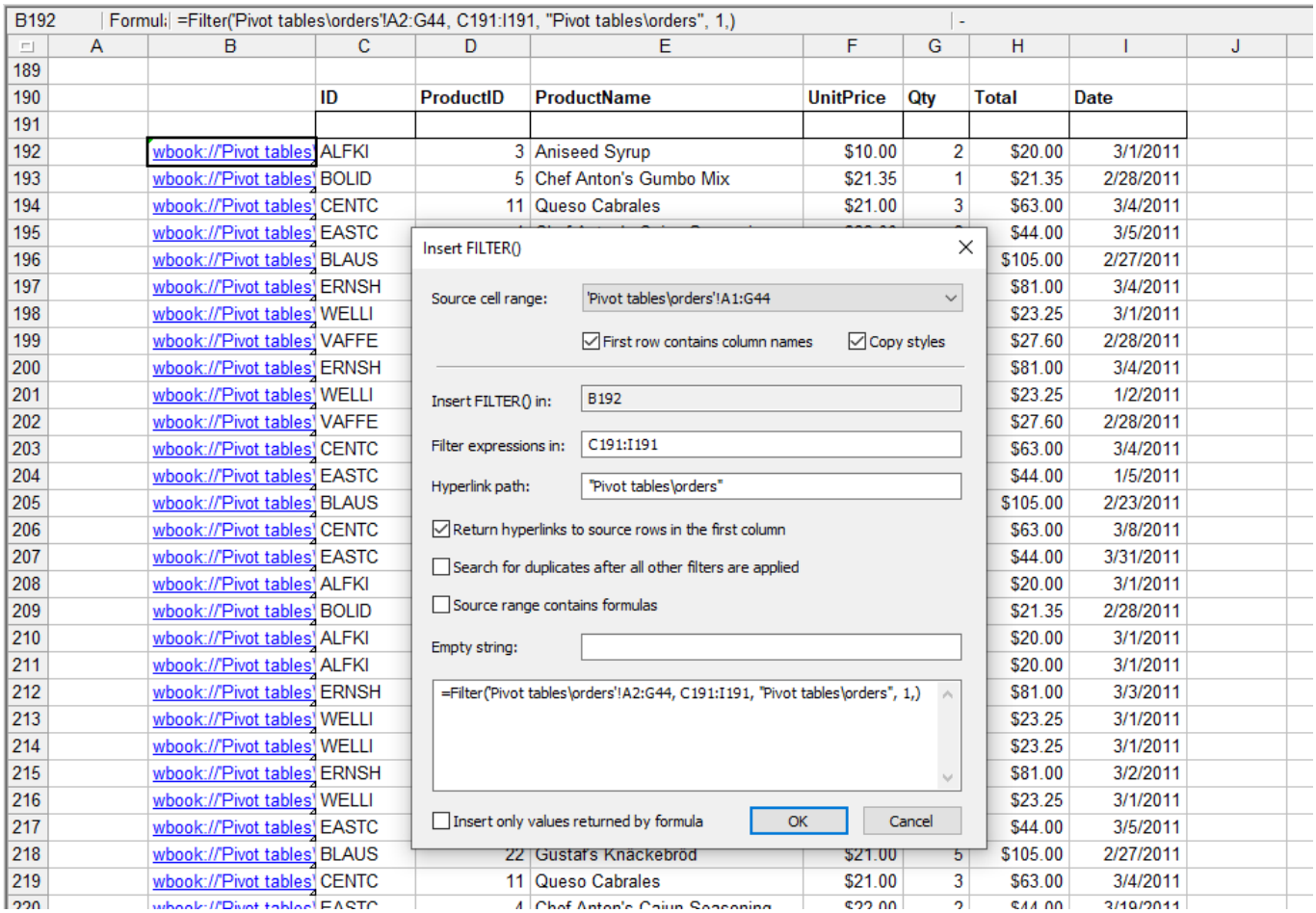
The default value is 0.

**Note:** If you're using FILTER() for very large data sets and if you expect you'll be changing this (and other) options frequently, use a cell reference for "options" in FILTER() instead of a hard-code value because re-editing the FILTER() formula cell may activate the "Undo" for most of the target range and cause unnecessary delays before actual filtering.

**empty-string**

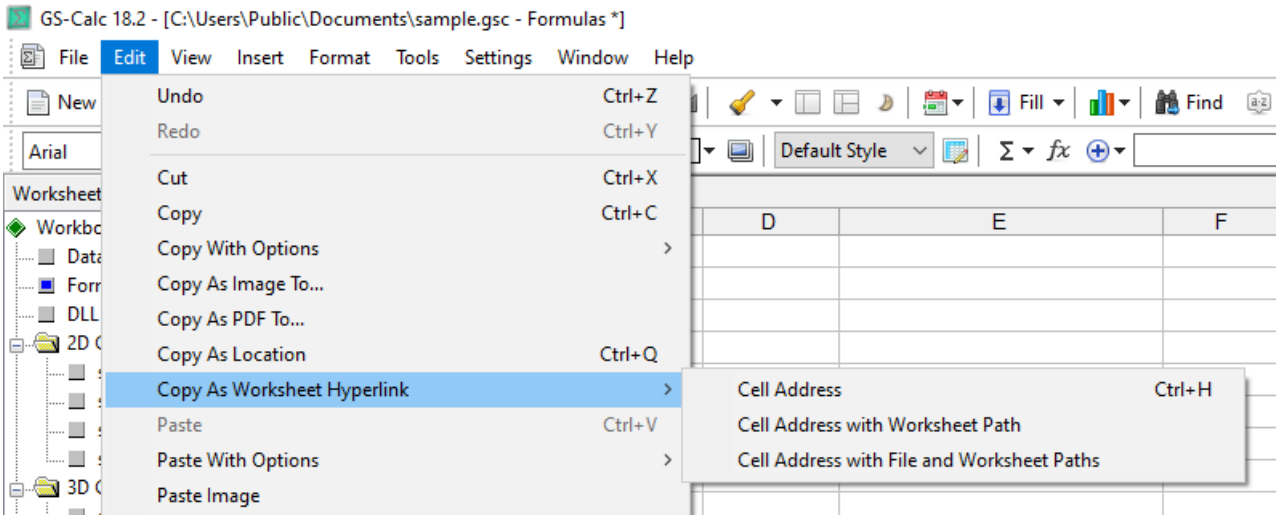
The optional "empty-string" argument represents a number or a string that the FILTER() function is to return if no matching rows are found for the specified filters. If it's omitted, FILTER() returns the "#N/A!" error code in such a case.

You can insert the Filter() function with just two clicks using the **Insert > Filter()** command. The displayed "Insert FILTER()" dialog box determines the optimum parameters including the filters range and creates the formula. On the screen below the "Insert" command was used to filter data in the "sample.gsc -> orders " worksheet with a single "OK" click. It automatically copies the source styles, adds the necessary "hyperlink" and "filter expression" formatting.

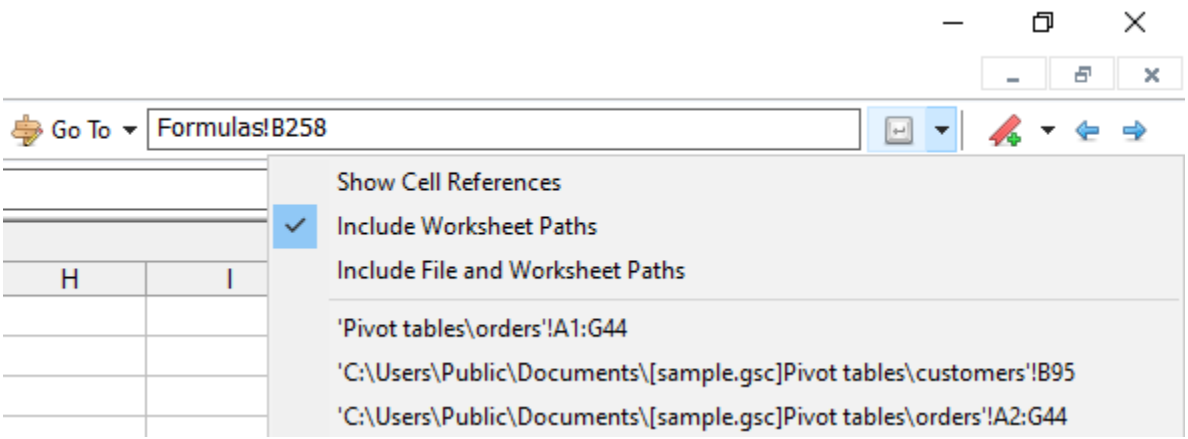



The source range 16 element list is a global list created and stored in the settings file. Subsequent source ranges are added in a circular manner by clicking one of the following:

**The "Copy as Location" menu command**




**The "Enter" toolbar button**




## Filtering tables

The easiest way to specify filters in the FILTER() function is to use the "Format > Search Expression" style for the cells containing filters.

If you want to access and construct search expressions directly e.g. by your formulas, you need to use them as plain text cells in the following form:

**filter-type[sort-key+|-][n]{:|=}[filter-text]**

where elements in [,] brackets are optional.

- **filter-type**
  - a - a regular expression

    - b - "starts with" / a pattern with *, ?, ~
    - c - equal
    - d - not equal
    - e - similar to
    - f - greater than
    - g - greater or equal
    - h - smaller than
    - i - smaller or equal
    - j - between (values/strings)
    - k - cell content equal to
    - l - cell content length greater than
    - m - cell content length smaller than
    - n - duplicated values in a column
    - o - empty cells (as opposed to "k:0" this will also include cells with spaces only)
- (optional) **sort-key** is a sort key index; for a single column sorting it's always "1"; for multi-key sorting these must be numbers from 1 to 6 where each one can occur in the "filters" array only once; the number must be followed by the "+" or "-" sort order indications (the ascending and descending sort orders);
- (optional) **n** is a digit representing (a sum of) options:
    - 1 - allow empty matches (required by RegEx if you want to use expressions like \A\z)
    - 2 - match case
    - 4 - word sort order; when sorting and searching for duplicates ignore punctuations and non-word characters

  Thus n can be any number from 1 to 7.

- The **:** or **=** character indicates the starting position of the filter text itself. If it's "=", it's assumed that this a cell reference containing that text (for the "between" filter: two references);
- (optional) **filter-text** is the actual filtering expression; it's ignored for the "n" or "o" filter types. For the other filters, specifying an empty text results in ignoring that filter.

Examples:

- a:john
- b:*smith

- e:color
- j:23 54
- j:A M
- j=$b$1 $b$2
- k:15
- n:
- o:
- a1+:john
- b1-:*smith
- e2+:color
- a5:john
- b2:*smith
- e1:color
- a1+5:john
- b1-2:*smith
- e2+1:color

Each cell in the **filters** parameter in the FILTER() function contains a single filter. That **filters** parameter can be also entered directly as an array within the FILTER() formula, like:

**=Filter(B2:H100000, B1:H1, "", 0)**

**=Filter(B1:H100000, {"a:tom","n1+:","","a:new"}, "", 3, "no records")**

**=Filter(B1:H100000, {"a:tom";"";"";"a:new"}, "sheet1", 2)**

# Inspecting cells and finding certain cell types

## Inspecting cells and finding certain cell types: Using the "Inspect Cell" pane

The **View > Inspect Formulas** pane enables you to quickly view a list of any formulas in the current workbook along their values and your comments. To add new cells to the list, click the **Add**.

Cells that are currently empty are displayed on the list in red.

# Formatting

## Using data styles and numeric formats

You can use the following standard numeric styles and their options:

- Default
  (No specific formatting. Numbers with more than 15 digits are displayed in the exponential-scientific format.)
- General
  - decimal places (0 to 14)
  - leading zeros (0 to 14)
  - thousand separators
  - how negative numbers are rendered
  - display factor (displayed numbers are divided by 1000 to that power)
- Currency
  - currency symbol position
  - currency symbol
  - how negative numbers are rendered
  - display factor (displayed numbers are divided by 1000 to that power)
- Accounting
  - currency symbol
- Date
  - date pattern
  - using fixed or system dependent year/month/day order
- Time
  - time/period pattern
- Date/Time
  - date pattern
  - time pattern
  - using fixed or system dependent year/month/day order
  - date/time order
- Percent
  - decimal places (0 to 14)
- Fraction
  - either a fixed denominator value (1 to 9,223,372,036,854,775,807) or a fixed number of denominator digits (1 to 19)
- Scientific
  - decimal places (0 to 14)
  - a fixed exponent value

Alternatively, you can use a **user-defined** numeric format. To do this, simply enter the desirable style pattern in the **Style** dialog. If you need to re-use it, you can save it as a new user style with a new name. To learn more about style patterns, please see how the standard styles are defined. For example, the accounting format is defined as:

_(\$* #,##0.00_);"($"* #,##0.00\);_(\$* \-??_);_(@_)

and the fraction format with the fixed denominator (4) can be defined as:

?\ ?/\4

User-defined formats allow you to specify font colors for numbers that meet some conditions. For example:

[green]#.#;[red]\-#.#;[blue]#.#;[gray]@

displays positive numbers in a green font, negative numbers in red, blue zeroes and gray text labels,

[red][<10]#0.00;[yellow][<=50]#0.0;[green][<400]##0;[magenta][>=400]#00

displays numbers less than 10 in a red font, numbers less or equal to 50 in a yellow font, numbers less than 400 in a green font and numbers greater or equal to 400 using a magenta font.

Color values can be expressed as RGB values (for example, [red][<10]#0.00 is the same as [#FF0000][<10]#0.00) or by one of the following color names:

| | | | |
|---|---|---|---|
| **black** | **[#000000]** | **maroon** | **[#800000]** |
| **green** | **[#008000]** | **olive** | **[#808000]** |
| **navy** | **[#000080]** | **purple** | **[#800080]** |
| **teal** | **[#008080]** | **gray** | **[#808080]** |
| **silver** | **[#C0C0C0]** | **red** | **[#FF0000]** |
| **lime** | **[#00FF00]** | **yellow;** | **[#FFFF00]** |
| **blue** | **[#0000FF]** | **fuchsia** | **[#FF00FF]** |
| **aqua** | **[#00FFFF]** | **white** | **[#FFFFFF]** |

## Using Cell Style Palettes

A cell style is a set of formatting properties (font, data style, borders, background), used to display a single cell.
Style palettes are collections of all cell styles that are currently in-use by all tables/worksheets in a given workbook (excluding charts and other inserted objects). Style palettes dynamically change as you perform various formatting actions.

GS-Calc enables you to modify all properties of existing style palette entries. In particular, you can also edit the **default** cell style, changing the appearance of all cell worksheets. For

example, if you change the background color for that default cell style, all cells that don't have any specific background set by a user will use that color to display their interiors.

GS-Calc provides various options for creating, editing and managing style palette. The related menus and commands are available either in the resizable **Format > Style Palette** window or in the **Format > Cell Format** dialog box if the **Format > Advanced Cell Format Dialog Options > Show Style Lists** option is checked.

## Using Custom Styles

A cell style is a set of formatting properties (font, data style, borders, background), used to display a single cell.
Custom styles are collections of user-defined cell styles that can be used when formatting cells in a given workbook. Custom styles are stored in workbooks but unlike palette styles, a given custom style may or may not be in use by that workbook. Custom styles don't change unless you explicitly manually edit, add or delete some of them.

GS-Calc provides various options for creating, editing and managing custom styles. The related menus and commands are available either in the resizable **Format > Custom Styles** window or in the **Format > Cell Format** dialog box if the **Format > Advanced Cell Format Dialog Options > Show Style Lists** option is checked.

## Hyperlinks

The **Format > Hyperlink** style let you treat the cell contents as a hyperlink that is activated after a single mouse click or pressing the SPACE key. To edit it, double click it or use the ENTER/F2 keys (unless you re-define the ENTER in the global or AutoScroll options).

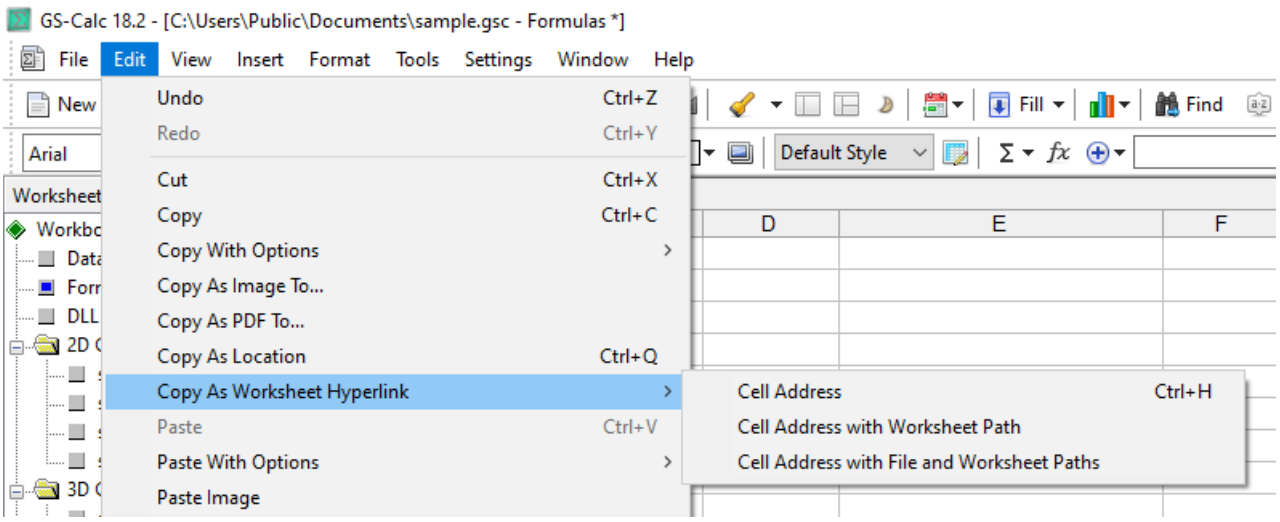
The link can be any location in the current workbook, other workbook files, www addresses, e-mail address, files on a local disc etc.
You can explicitly specify the link type/protocol adding one of the following prefixes:

- wbook:// (for example, wbook://folder2/sheet2!b100)
- http://
- https://
- file://
- mailto:
- ftp://
- gopher://
- nntp://
- news://

- wais://
- telnet://
- prospero://
- res://

**Note:** Clicking a given workbook hyperlink causes jumping to the specified location. If this is within the same workbook and you later want to go back to the original location from before that jump, click the "back" bookmark toolbar button (Alt+F11). (This jump-back option remains valid till you use any bookmarks commands for other purposes.)

The easiest way to create a hyperlink to the current cell is to use one of the **Edit > Copy as Hyperlink** commands. To paste/insert such copied and already formatted hyperlink in the same or in other workbooks you just need to use the plain **Paste** command. The Ctrl+H shortcut is automatically re-assigned to the most recent "hyperlink" command variant.



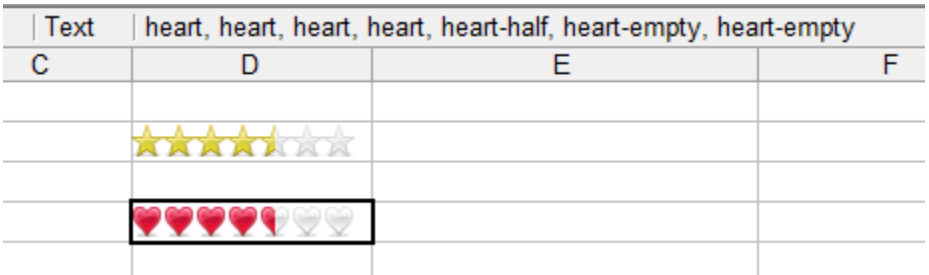

## Images in Cells

**Format > Images** style enables you to enter in a cell any sequence of image names and then display the result in one line in a cell.

The predefined image names (available in every workbook) include various buttons, flags and symbols.
For example, entering in a cell

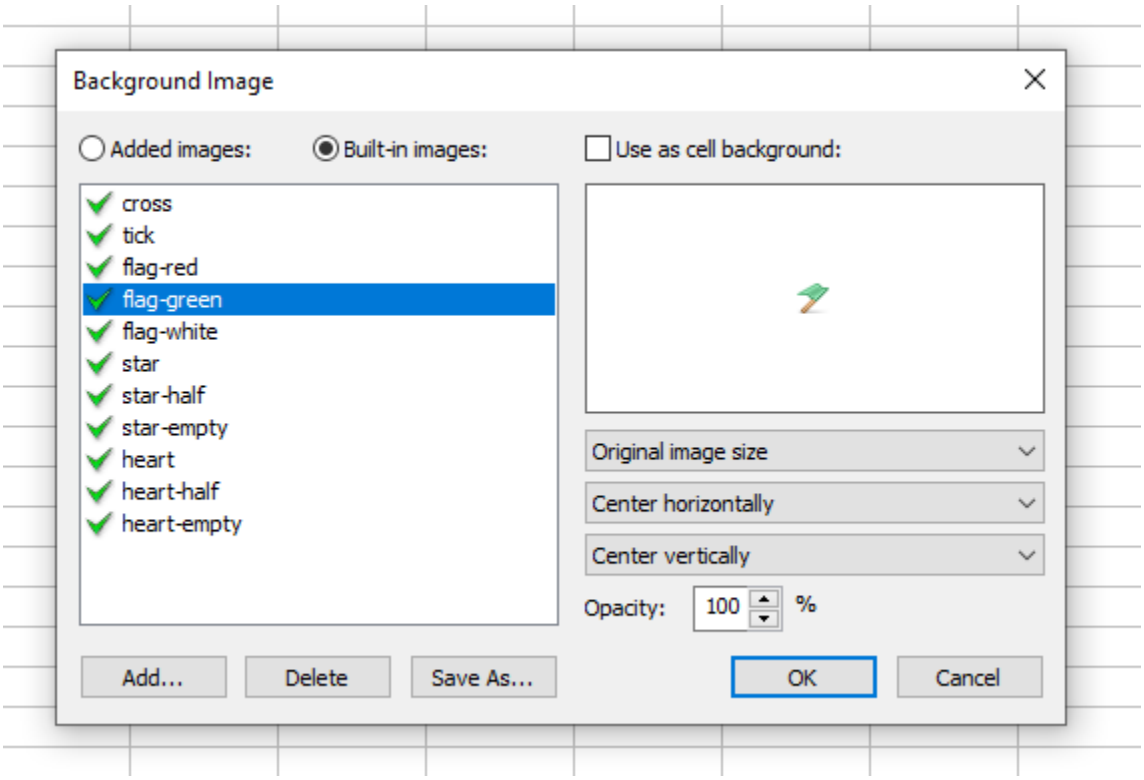
**star, star, star, start, star-half, star-empty, star-empty**

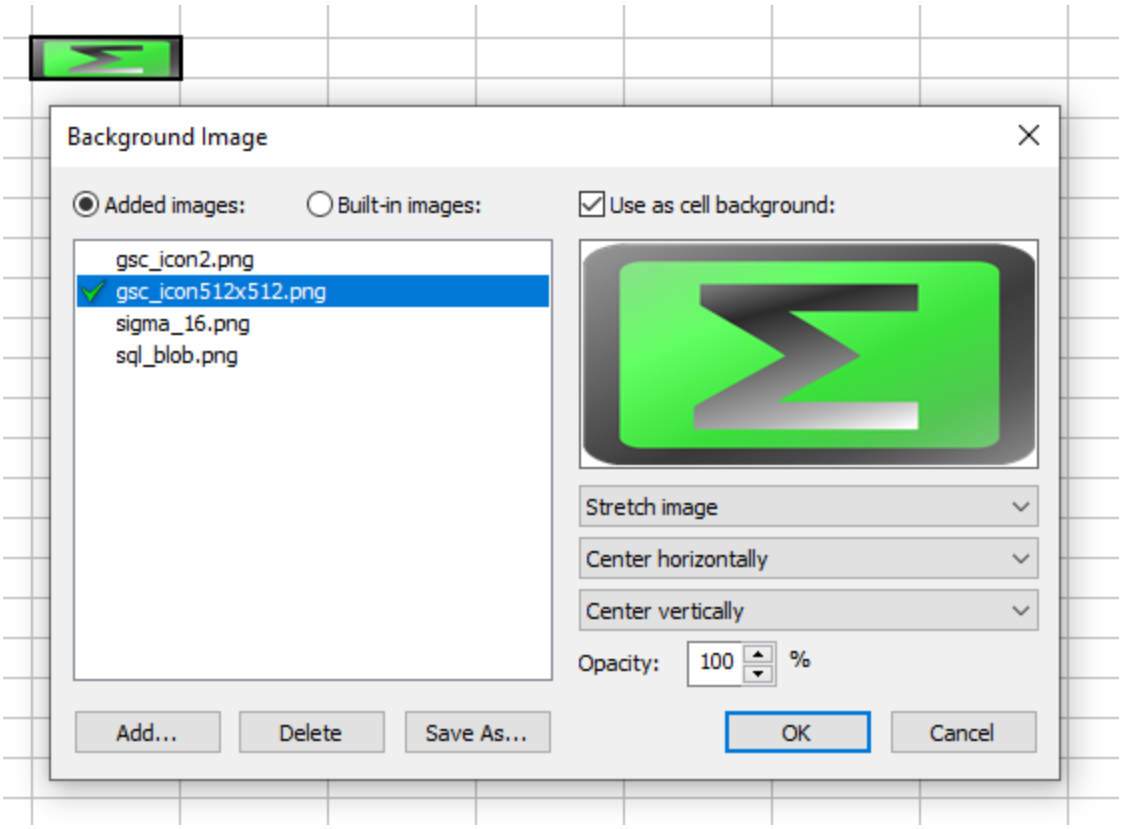
will display in a cell a 7-star-style rating with 4.5 stars:




To add and use in a given workbook your own images, use the **Images List** toolbar button or use the "background image" command in the "Cell Format" dialog box.

In the displayed "Workbook Images" dialog box, you can view and manage the added images:






# Charts and images

## Creating Charts

To specify chart data series, use one of the following two methods:

1. Select a continuous range of cells containing data series in rows or columns and use one of the **Insert Chart > Series In Rows/Columns** commands. GS-Calc will then automatically create the corresponding data series. For example:

| | | | | |
|---|---|---|---|---|
| Series 1. | 2 | 2.3 | 4.2 | 3 |
| Series 2. | -2 | 1.1 | 2.1 | 2.3 |
| Series 3. | 2 | 1 | 2 | 5.4 |

| | | | | |
|---|---|---|---|---|
| Series 1. | 2 | 2.3 | 4.2 | 3 |
| Series 2. | -2 | 1.1 | 2.1 | 2.3 |
| Series 3. | 2 | 1 | 2 | 5.4 |

| Series 1. | Series 2. | Series 3. |
|---|---|---|
| 2 | -2 | 2 |
| 2.3 | 1.1 | 1 |
| 4.2 | 2.1 | 2 |
| 3 | 2.3 | 5.4 |

| Series 1. | Series 2. | Series 3. |
|---|---|---|
| 2 | -2 | 2 |
| 2.3 | 1.1 | 1 |
| 4.2 | 2.1 | 2 |
| 3 | 2.3 | 5.4 |

2. Use the **Insert Chart** command to create an empty chart and add subsequent series manually. To add a new series, click the **New Series** button in the **Chart** dialog box.

For example, for a **2D/3D chart** and for the following data:

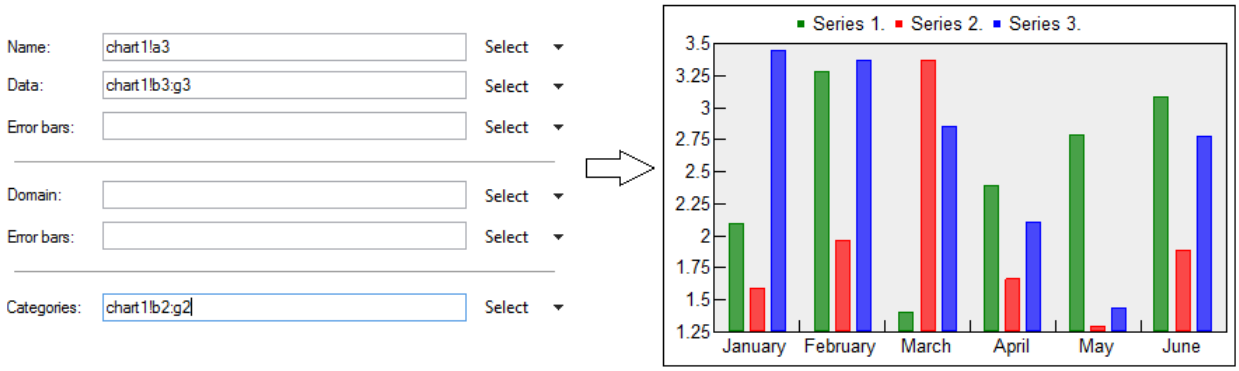
series names (a3:a5)

categories (b2:g2)

| | January | February | March | April | May | June |
|---|---|---|---|---|---|---|
| Series 1. | 2.1 | 3.3 | 1.4 | 2.4 | 2.8 | 3.1 |
| Series 2. | 1.7 | 2.2 | 4.1 | 1.8 | 1.3 | 2.1 |
| Series 3. | 3.7 | 3.6 | 2.9 | 1.9 | 1 | 2.8 |

series data in rows (b3:g5)

the chart 1. series tab in the **Chart** dialog box should look like this:



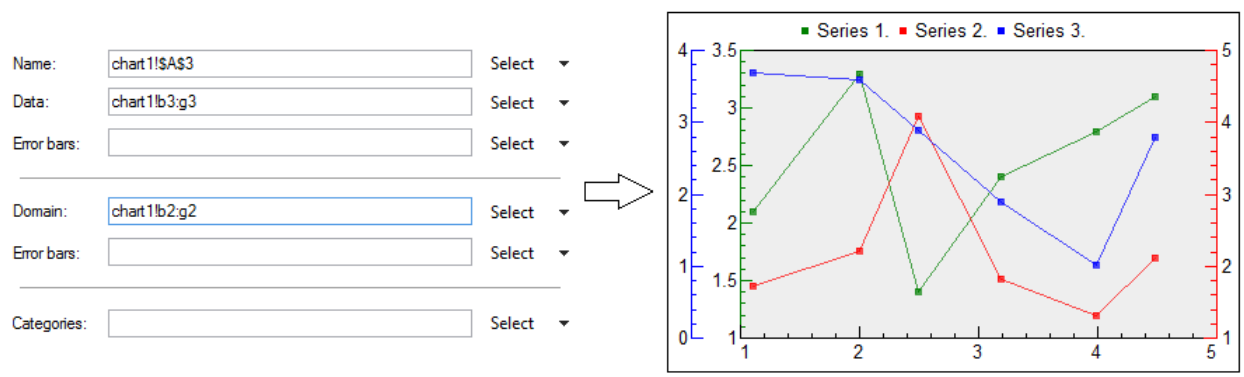

For a **scatter (XY) chart** and for the following data:

series names (a3:a5) domain (b2:g2)

| | | | | | | |
|---|---|---|---|---|---|---|
| | 1.1 | 2 | 2.5 | 3.2 | 4 | 4.5 |
| Series 1. | 2.1 | 3.3 | 1.4 | 2.4 | 2.8 | 3.1 |
| Series 2. | 1.7 | 2.2 | 4.1 | 1.8 | 1.3 | 2.1 |
| Series 3. | 3.7 | 3.6 | 2.9 | 1.9 | 1 | 2.8 |

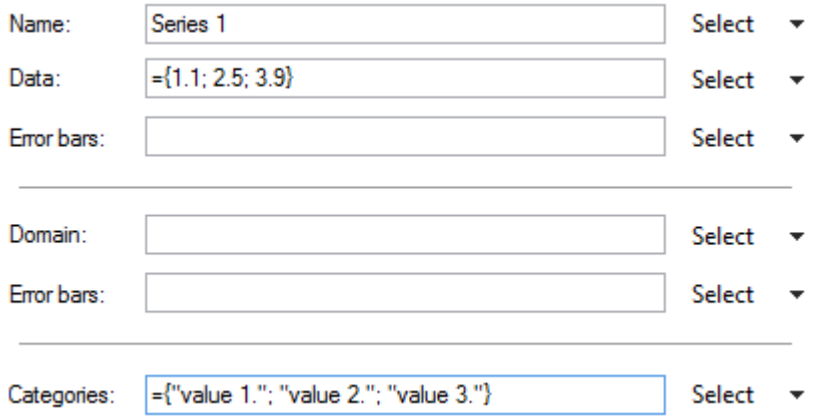
series data in rows (b3:g5)

the chart 1. series tab in the **Chart** dialog box should look like this:

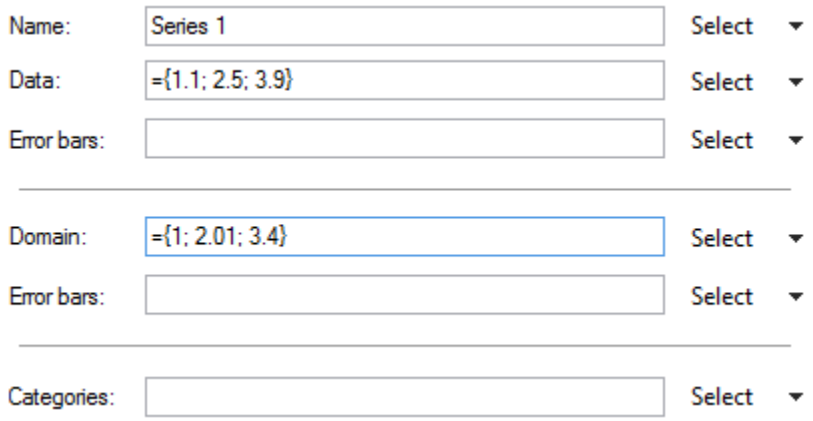



3. The chart data series can be also specified as array formulas returning vectors (one-column arrays with ; row separators) or lists of unformatted numbers separated by spaces. For example:

(1)




(2)




Data entered in the **Chart** dialog must be of the following types:

| **Chart data type** | **Must be specified as** |
|---|---|
| Series name | A single cell address or a plain text string, for example:<br>(1) Worksheet1!$E$2<br>(2) '2D Charts\sample2'!$B$24<br>(3) sales in February |
| Chart data series, domain, series and domain error bars, data categories | A single cell range, a list of unformatted numbers (for data series), a list of text strings (for categories) or an array formula, for example:<br>(1) Worksheet1!$H$1:$H$100<br>(2) 23 2.3 45 12.1 44<br>(3) ab &quotc d e" &quota1" &quotfg h" &quotin "&quotquote"""<br>(4) ={1, 2, 3.1, 4.2, 5.7}<br>(5) =row(b1:b100)*rand()<br>(6) =mtxSeries(30, 1, 1) + 4*mtxRand(30,1,1)<br>Note: When entering a list of text strings, you must use quotation marks for |

| | |
|---|---|
| | those text strings that contain spaces or quotation marks. Additionally, such inner quotation marks must be doubled. |
| Chart tiles, axis titles | A plain text string or a single cell address, for example:<br>(1) sample chart title<br>(2) Worksheet1!E2 |

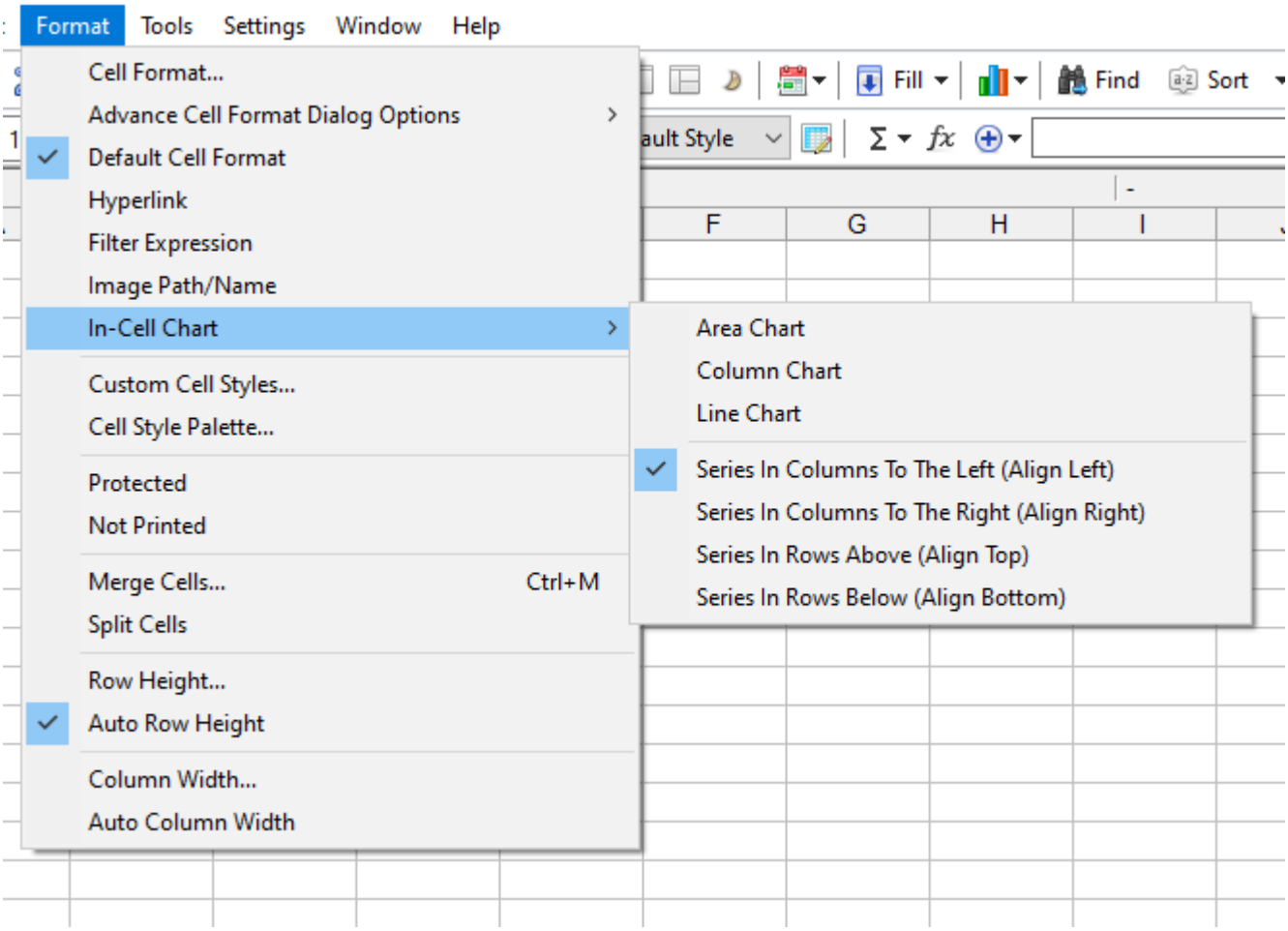
Notes:

- All changes specified in the **Chart** dialog box are applied only after you click the "Apply" button. The exceptions are: (3d) rotation and lamp parameters - they are applied instantly automatically.
- The **Chart** dialog box always requires cell references in the A1 notation.
- **Shared axes:** By default each data series has its own visible "y" axis and "x"/category axes and the automatically calculated axis ranges can be different for different series. To ensure different series are displayed using the same axis limits, do one of the following:
    1. Specify the same fixed axis limits for each series (and leave only one of them visible).
    2. Specify a shared axis for given series. This is done by choosing/entering the name/id of another axis of another series on the **Chart > Axis** tab. If a given series is to use the axis of another series, in its **Chart > Axis > Shared axis id** field choose the name/id of that axis. Shared axes are displayed only once for each group of series sharing them.
- Specify only the minimum (or only the maximum) axis limit with other parameters calculated automatically may result in displaying the minimum (or the maximum) limit that will be actually smaller (or greater for maximum limits) by one "interval" value. To make sure the limits are exactly as entered, you need to specify all the following three axis parameters: the minimum limit, the maximum limit and the interval value.
- Newly created charts show only one set of horizontal gridlines for the first series and axes for the first two series. It's up to you to add other gridlines and to make other axes visible and/or change their style and class.
- If you leave the **Categories** (for 2D/3D charts) or **Domain** (for XY-scatter charts) fields blank, GS-Calc will use a series of integers starting from 1.

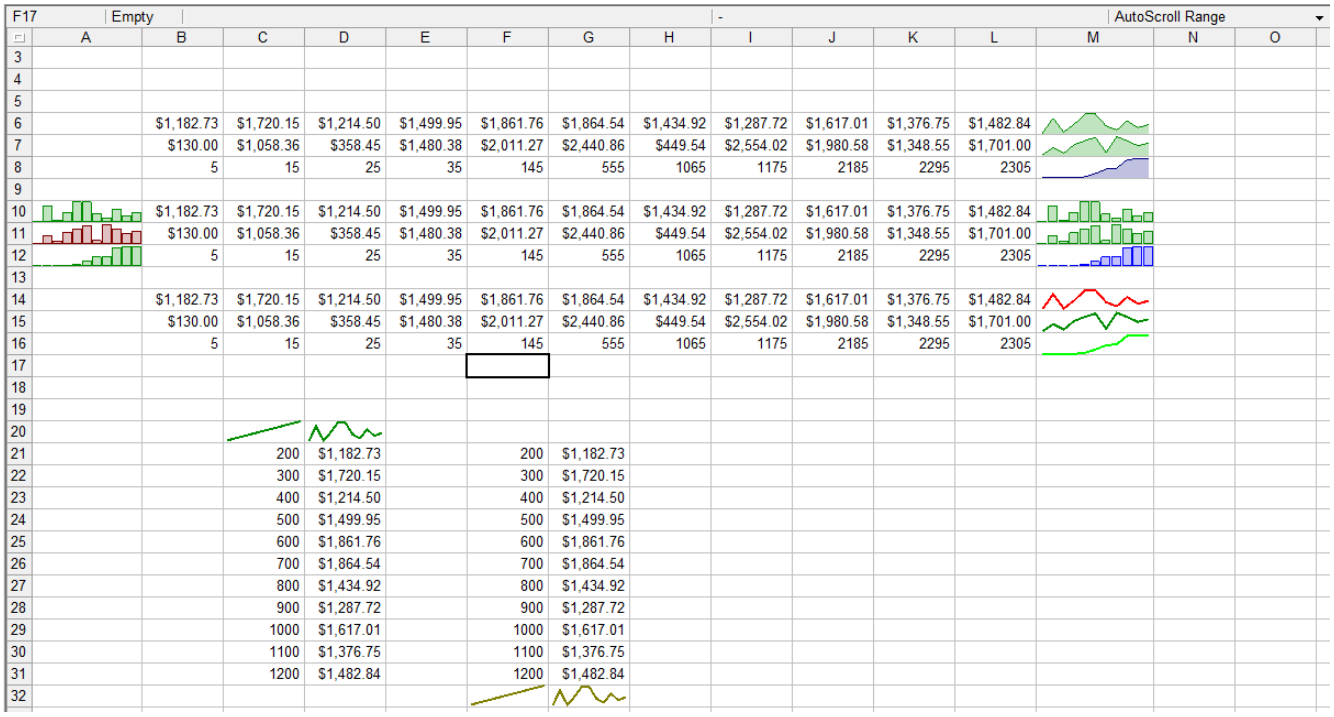
## Charts in Cells

The **Format > In-Cell Chart** command enables you to create any number of cell mini-charts displayed with a single click.
All parameters and display options are set up automatically. By default, you apply this in-cell format to a cell(s) at the end of a data series placed in a row. However, the chart(s) can be also displayed at the beginning of a series in a row or above or below a series in a column - a plain cell text alignment determines this:

The **Format > In-Cell Chart menu:**




**Sample In-Cell charts:**




# Password protection

## Protecting and encrypting workbooks

To encrypt and password-protect a given file, use the **File > Encrypt** command, specify your password consisting of at least 6 characters and choose the encryption algorithm.

If a workbook is encrypted and password-protected, the "shield" toolbar button becomes green.

Note: GS-Calc uses standard ODF (OpenDocument) encryption procedure and encrypt all information (tables, styles, charts, images, settings etc.) that a workbook can contains. However, (file)names of the inserted images are left unencrypted in the *.ods file.

## Protecting workbook structure

To disable/enable adding, deleting or moving individual worksheets, use the **Tools > Protect Structure** command. The password must consist of at least 6 characters.

Note: Unless your workbook is encrypted at the same time, a person having full access to your file can use another software capable of reading/writing *.ods files to change these settings.

### Protecting and encrypting workbooks

To disable/enable cell protection settings for a given worksheet, use the **Tools > Cell Protection** command. Your password must consist of at least 6 characters.

Once the cell protection is active, you can use the **Format > Protected** command to turn on/off the individual cell protection. Protected cells cannot be deleted.
Each time you (un-)protect some cell(s), you'll be asked to supply a password defined as explained above.

Note: Unless your workbook is encrypted at the same time, a person having full access to your file can use another software capable of reading/writing *.ods files to change these settings.

## Opening and saving files

### Content and statistics

To view the file statistics and/or to change some of the options concerning the saved content for the current workbook, use the **File > Content & Statistics** command.

**Save values of formulas**

Specifies whether the current values of formulas should be saved. Turning this option off may result in significantly smaller files. If this option is turned off, you need to perform an initial update/recalculation after opening a given file to re-create formula values.
This option is available only for the GS-Calc *.gsc file format. Default: On

**Save values of formulas**

If the **Save values of formulas** check box is on, this option can be used to explicitly specify cells or ranges containing formulas that are to be saved. Cells and ranges should be entered as a comma separated list, for example:
  a10, d:d, sheet1!f20, 5:5, dk1:dk10, "first sheet"!ab100, "second sheet"!ab100:ab200
If a given cell/range reference doesn't contain the worksheet path, it's assumed that it refers to the currently active worksheet.
This option is available only for the GS-Calc *.gsc file format. Default: Off

**Save thumbnail**

Specifies whether the thumbnail image should be added to the saved workbook. Thumbnail images can be saved in the "*.gsc" and *.ods files. They represent the portion of the worksheet visible when that worksheet was saved.
For encrypted files the thumbnail is replaced by a 128x128 pixels static image informing users that the content is encrypted.
Both GS-Calc *.gsc and ODF *.ods files are zip files and saved thumbnail images are available in those zips as the following zip stream: **Thumbnails/thumbnail.png**.
**Note:** The ODF specification requires thumbnails in the *.ods files to have a fixed size of 128x128 pixels.
Default: On

**Cell range**

If the **Save thumbnail** check box is on, this option specifies the cell range to be used for the thumbnail image. It can contain a simple cell range, a cell range along with the full worksheet reference or only the worksheet reference/path without any specific cell range. If not specified, the thumbnail will represent the portion of the worksheet visible when that worksheet was saved.
Default: Off

## GS-Calc and ODF files

GS-Calc can optionally use both its private binary file format and the ODF format as its native format.

**GS-Calc (*.gsc)**

Typically, GS-Calc *.gsc files should be several times smaller than the corresponding ODF files. Thanks to various special compression techniques for certain files that difference can even reach several hundred times or more. Similarly loading/saving times are much faster when this file format is used.
GS-Calc files are plain zip files that contains several zip streams including the workbook data, used images, the list of files (along with optional encryption information), the current file version number, the minimal GS-Calc version required to open it and the thumbnail image. The thumbnail path is: **Thumbnail/thumbnails.png**.
If the file is encrypted, both the workbook data and all used images are encrypted.
The *.gsc format is a private binary format and it's not supported by other applications.

To associate the *.gsc file type with GS-Calc in Windows, you must register that file type in the Windows registry database. To do this, run GS-Calc as an administrator (e.g. right-click GS-Calc shortcut/icon and on the context menu choose "Run as administrator") and use the **Settings > Register GS-Calc *.gsc File Type** command.
Similarly, if you no longer need that registration, you can use the **Settings > Unregister GS-Calc *.gsc File Type** command to remove all registry entries added earlier.

**Open Document Format (*.ods)**

Saved files conform to the ODF 1.2 specification. This is a standard file format used by several notable applications. There are several GS-Calc-specific features that depends on

preserving some element naming (which otherwise are not required to be preserved when a given file is re-edited by other applications). Those features include:

- The accounting data style.
- Fixed exponent values in the scientific data styles.
- The user defined data style specified via the format code.
- Using array formulas in the Chart dialog to specify directly chart data series, categories, domain and errors.
- Secondary chart axes (e.g. axes placed at the top or the right side of the chart wall).

GS-Calc enables you to specifies which formula notation should be used when saving newly created *.ods files.
You can choose one of the following formula "namespaces":
Excel 2010/2013 ("msxol:"),
OpenFormula ("of:"),
Google Docs ("oooc:").
When editing existing *.ods files GS-Calc will preserve their current formula notation. To change it you can use the **Save As** command.
Choosing the right formula namespace might be important as, for example, opening an "OpenFormula" file in Excel might result in removing the formulas and replacing them with their current values.
This options can be specified either during the setup or later via the **Setting > Options** dialog.

## Opening and saving GS-Base databases (saved as *.xls workbooks)

**To save a workbook as a GS-Base database in the *.xlsx/*.xls format**

Use the **File > Save As** command and choose the **GS-Base (*.xlsx, *.xls)** format from the **File of type** list.

GS-Calc uses the *.xlsx/*.xls format instead of the native GS-Base *.gsb/*.zip format because the *.xlsx/*.xls files are the simplest among the standard formats that can still transfer the complete common data between the two programs.

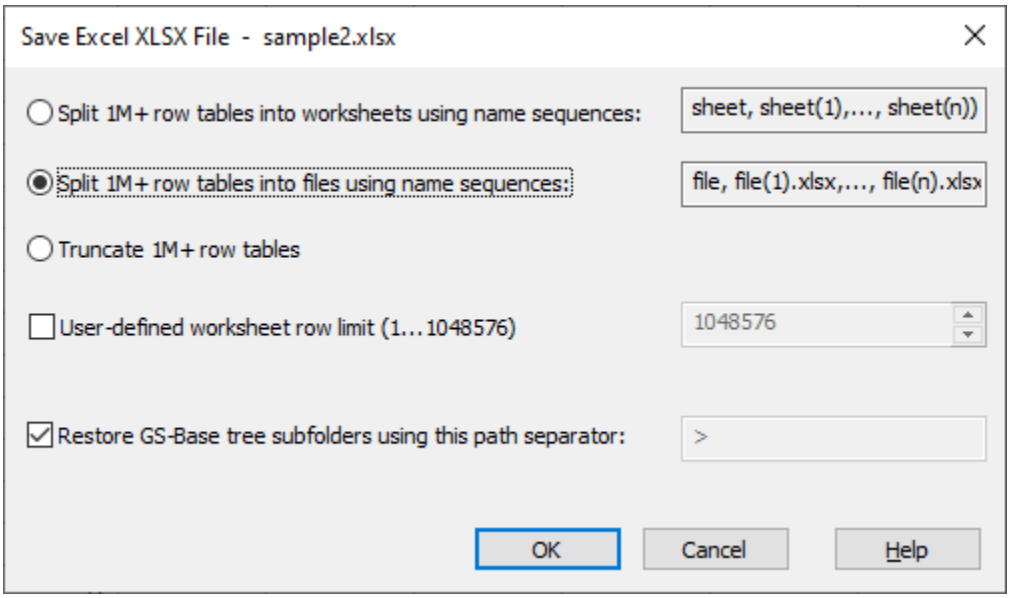
In the **Save GS-Base File** dialog box specify the following options:

- **Save field names in the first row**
  If this option is selected, the first row of each saved sheet will contain the corresponding record field names that will be reused if you open the workbook back in GS-Base.
- **Split tables exceeding the 1M/65K Excel *.xlsx/*.xls limit in files [...]**
  If this option is selected and if the number of records in one or more of the saved tables exceeds the 1M/65K Excel *.xlsx/*.xls limit, GS-Calc will split such tables and save multiple *.xls workbooks:
  file_name.xlsx
  file_name(1).xlsx
  file_name(2).xlsx

...
file_name(n).xlsx
If there are any previously saved files with larger indices (e.g. file_name(n+1).xls and on), they will be deleted.
If you open the file_name.xls file again in GS-Calc or GS-Base, the remaining partial workbooks will be automatically (internally) loaded to form the original GS-Calc/GS-Base tables.

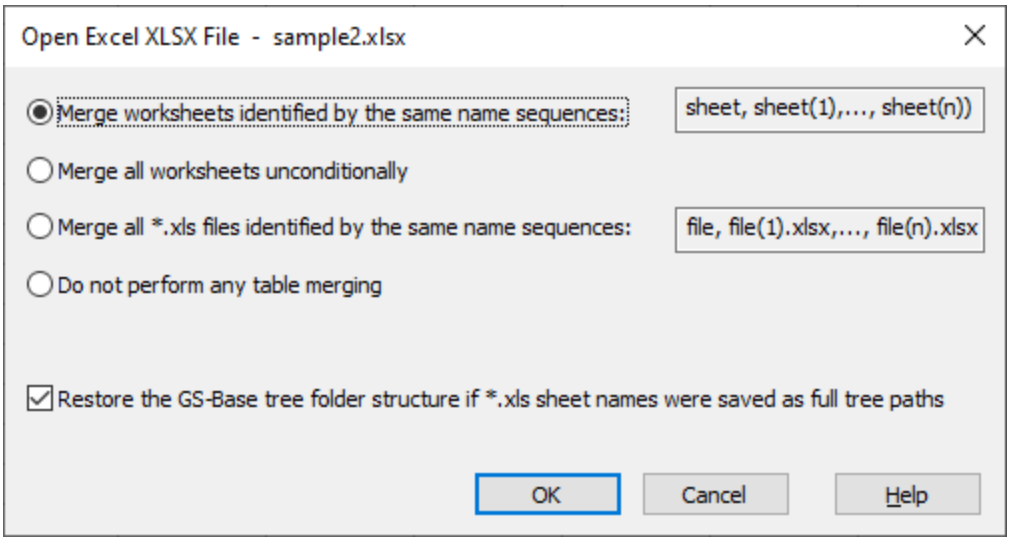
- **Enable restoring the GS-Calc folder structure [...]**

  If this option is selected and if you created folders in the GS-Calc worksheet tree pane for some tables, the Excel sheet names will be formed as whole such table tree paths which will enable GS-Calc to re-created the original tree folder structure when you open the saved Excel *.xlsx/*.xls workbook back in GS-Calc (or GS-Base).
  As the "\" path separator can not be used in Excel names, it's replaced with ">".




**To open a GS-Base file saved in the *.xlsx/*.xls format**

Use the **File > Open** command and choose the **GS-Base (*.xlsx, *.xls)** format from the **File of type** list.




## Opening and saving Excel 2003 XML files

**To open an Excel XML file**

Use the **File > Open** command and choose the Excel XML format from the **File of type** list.

**To save a file in one of the supported formats**

Use the **File > Open** command and choose the Excel XML format from the **File of type** list.

**Note:** After saving a document in a different format you may need to use the **File > Reload** command to refresh the view.

## Saving files: PDF files

To save data in the PDF format, do one of the following:

- Use the **File > Save Worksheet As PDF** and **File > Save All As PDF** commands. The former saves the current worksheet and the later - the entire workbook. You can change the page layout and other content options using the **File > Page Settings** command.

- Use the **Edit > Copy As PDF To...** command to save currently selected range, chart or image.

- Use the **File > Preview** command.
  In the preview window, click either the **Save Page** button to save the currently

displayed single page or the **Save All** button to save all generated pages. **Note:** Pages generated in the preview window depend also on the current **File > Print > General** settings. For example, if you select the
**File > Print > General > Printed Data > Selection**
option, the preview page(s) will contain only the selected cells or the selected object (a chart or an image occupying the whole page).

**Note:** When saving a PDF file GS-Calc maps all fonts used in a workbook to a standard set of 14 PostScript Type 1 fonts which are required to be present in every PDF viewing application and in every operating system.

To specify the advanced PDF save options, use the **File > Advanced PDF Save Options** command. GS-Calc enables you to choose the language/code page that should be used when saving text data to a PDF file and to specify the custom character encoding list to correctly display all glyphs for that language.

For example, the full **iso-8859-1** list is:

33 /exclam /quotedbl /numbersign /dollar /percent /ampersand /quotesingle /parenleft /parenright /asterisk /plus /comma /hyphen /period /slash /zero /one /two /three /four /five /six /seven /eight /nine /colon /semicolon /less /equal /greater /question /at /A /B /C /D /E /F /G /H /I /J /K /L /M /N /O /P /Q /R /S /T /U /V /W /X /Y /Z /bracketleft /backslash /bracketright /asciicircum /underscore /grave /a /b /c /d /e /f /g /h /i /j /k /l /m /n /o /p /q /r /s /t /u /v /w /x /y /z /braceleft /bar /braceright /asciitilde /bullet /Euro /bullet /quotesinglbase /florin /quotedblbase /ellipsis /dagger /daggerdbl /circumflex /perthousand /Scaron /guilsinglleft /OE /bullet /Zcaron /bullet /bullet /quoteleft /quoteright /quotedblleft /quotedblright /bullet /endash /emdash /tilde /trademark /scaron /guilsinglright /oe /bullet /zcaron /Ydieresis /space /exclamdown /cent /sterling /currency /yen /brokenbar /section /dieresis /copyright /ordfeminine /guillemotleft /logicalnot /hyphen /registered /uni203E /degree /plusminus /twosuperior /threesuperior /acute /mu /paragraph /periodcentered /cedilla /onesuperior /ordmasculine /guillemotright /onequarter /onehalf /threequarters /questiondown /Agrave /Aacute /Acircumflex /Atilde /Adieresis /Aring /AE /Ccedilla /Egrave /Eacute /Ecircumflex /Edieresis /Igrave /Iacute /Icircumflex /Idieresis /Eth /Ntilde /Ograve /Oacute /Ocircumflex /Otilde /Odieresis /multiply /Oslash /Ugrave /Uacute /Ucircumflex /Udieresis /Yacute /Thorn /germandbls /agrave /aacute /acircumflex /atilde /adieresis /aring /ae /ccedilla /egrave /eacute /ecircumflex /edieresis /igrave /iacute /icircumflex /idieresis /eth /ntilde /ograve /oacute /ocircumflex /otilde /odieresis /divide /oslash /ugrave /uacute /ucircumflex /udieresis /yacute /thorn /ydieresis

The encoding list the **Windows-1250** code page could be:

32/space 40/parenleft/parenright 44/comma/hyphen/period/slash/zero/one/two/three/four /five/six/seven/eight/nine/colon 65/A/B/C/D/E/F/G/H/I/J/K/L/M/N/O/P 82/R/S/T/U 87/W/X/Y/Z 97/a 99/c/d/e/f/g/h/i/j/k/l/m/n/o/p 114/r/s/t/u 119/w 121/y/z 140/Sacute 143 /Zacute 156/sacute 159/zacute 163/Lslash 165/Aogonek 175/Zdotaccent 179/lslash 185/aogonek 191/zdotaccent 202/Eogonek 198/Cacute 209/Nacute 211/Oacute 230/cacute 234/eogonek 241/nacute 243/oacute

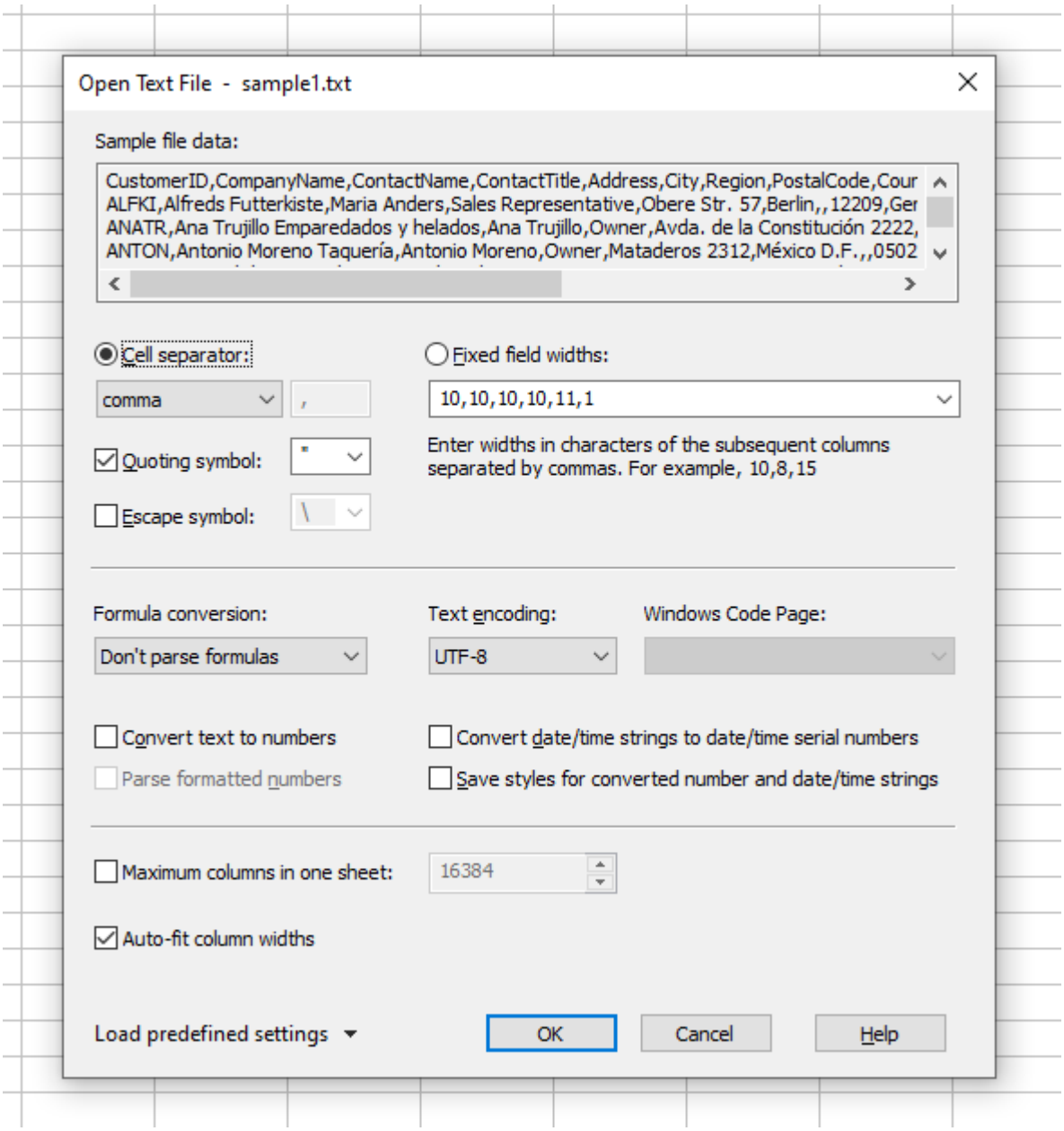
## Text files

### To open a text file

Use the **File > Open** command and choose **Text** from the **File of type** list. Next, specify additional text file options:

| | |
|---|---|
| **Cell separator** | A single character used to separate cells in one line. |
| **Fixed cell widths** | A list of fixed cell width values. For example:<br>10,30,15,40<br>If not all widths are specified, the remaining cells are assumed to contain 1024 characters. |
| **Quoting symbol** | A symbol used to delimit fields containing separator characters. Cells containing quoting symbols, separators or new line characters must be delimited by quoting symbols. The inner quoting symbols must be doubled.<br><br>**If you need to open a text file that doesn't conform to this standard, consider either de-selecting this option or specifying some unique quoting symbol that doesn't occur within that file.** |
| **Escaping symbol** | A symbol used to mark the occurrence of a single byte value specified as a two-digit hexadecimal number with the preceding **x** letter. For example:<br>\x66<br>\xA0\x45<br>Escaping symbols not representing the above sequences must be doubled.<br><br>**Escaping can be selected when opening a text file. When saving GS-Calc always uses the more common quoting method along with utf8 encoding.**<br>**By default escaping is turned off - choose it only if you're certain that a given file uses this form of encoding and uses the specified escaping symbol.** |
| **Formula conversion** | Specifies whether loaded text strings should be evaluated and converted to formulas. Leaving the default **No Conversion** option results in treating all cells as text labels. |
| **Convert text to numbers** | Specifies whether loaded text strings should be evaluated and converted to numbers.<br>Only unformatted numbers can be converted. The decimal point is determined by the **Settings > Locales** menu options and by the current system regional settings. |
| **Parse formatted numbers** | Specifies whether text strings representing not only plain numbers but also formatted numbers should be converted to numbers.<br>The decimal point is determined by the **Settings > Locales** menu options and by the current system regional settings. |

| | |
|---|---|
| **Convert date/time strings to date serial numbers** | Specifies whether text strings representing dates and/or times should be converted to date serial numbers. |
| **Convert date/time strings to date serial numbers** | Specifies whether cells containing numbers (obtained after the text) should retain the corresponding data styles. |
| **Encoding** | Specifies the text encoding for a given text file: UTF-8, ANSI 8-bit, ISO/OEM. |
| **Maximum columns in sheet** | By default, if a given text file has more than 16K columns, multiple sheets are created to contain and show all the data. You can specify any other splitting limit from 1 to 16384 columns. The opened text file can have up to 1 million columns that will be split in 16384-column sheets.<br>If multiple worksheets are created by GS-Base, trying to save back such a set of split sheets as a text file will result in prompting for a new ZIP (zip64) file archive file name instead of overwriting the original source text file. To load such ZIP text archive later for editing, simply choose the "Text \| Text Archive (*.txt, *.csv, *tsv, *.zip)" filter in the "Open File" dialog box. |




**Note:**
If a given text file is to be opened entirely as text (with no numbers and formulas parsing), all worksheet columns created from that file are automatically formatted using the "Text" style. To delete the "Text" file from a given column, click its header and either press Del then "Formatting" or choose the "Default style" on the formatting toolbar.

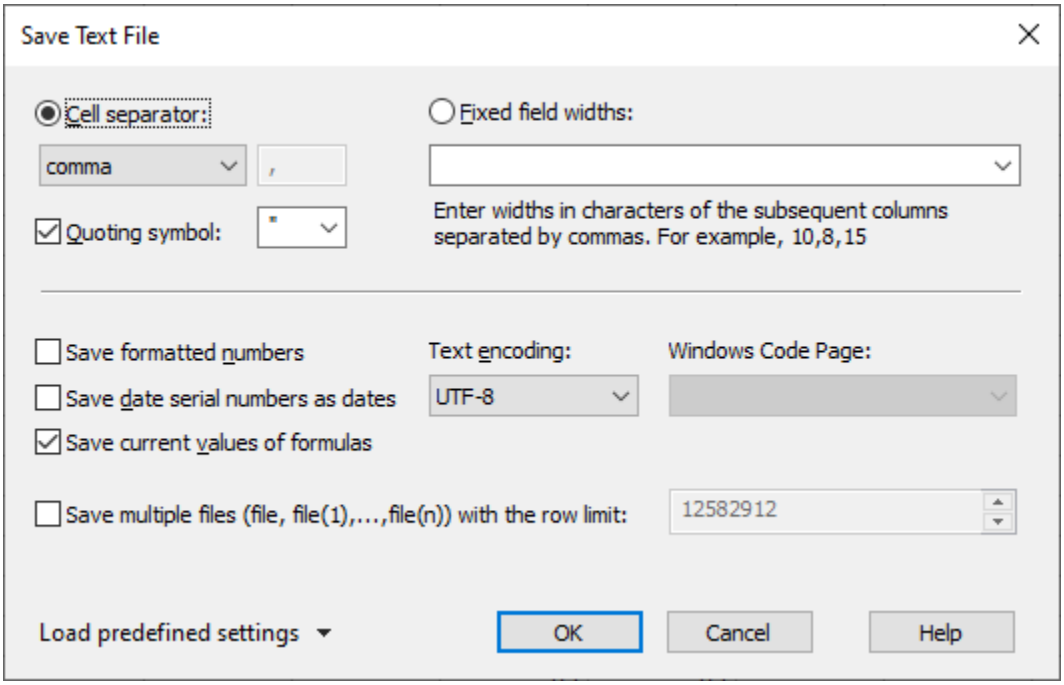
**To resize columns in the opened text file automatically**

Select the **Settings > Options > General > Auto-fit column widths in imported files** option.
(See Settings)

**To save a text file**

Use the **File > Save** command and choose **Text** from the **File of type** list. Next, specify the same additional text file options as above.

By default, if you choose the "Text | Text Archive (*.txt, *.csv, *tsv, *.zip)" filter in the "Save File" dialog box to save a given *.gsc workbook as text, only the current sheet is saved as a standalone text file.
If you want to save all sheets at once, you need to specify a file name with the *.zip extension. This will tell GS-Base to save all created text files to one ZIP64 text archive (along with a short XML file that specifies used text parameters so you won't have to re-enter them when opening that ZIP64 file in the future).
When saving ZIP text archives the options to split sheets based on the number of rows is inactive.




## Text *.zip archives

GS-Calc can now both load and save multiple text files in one zip file. The corresponding "Files of Type" filter name in the "(File) Open" dialog box is:

Text | Text Archive (*.txt;*.csv;*.tsv;*.tab;*.zip)

To save such a zip for the first time using "Save As", you just need to select this filter and enter both the file name and the ".zip" extension.

The advantages of this functionality include:

- All worksheets can be saved at once as text files in one zip; there is no need for all the merging and copying.
- You don't have to repeatedly enter or switch between various text file options (separators, encoding, parsing data etc.); when opening such a zipped set of text files, the "Text File Options" dialog box is displayed only once for all files (although you can choose to display it for any of the files individually).
- Once the zip file is saved, with one or any number of text files, GS-Calc saves in it a short *.xml file with parameters so you can keep on editing/processing/saving/loading such text files without the "Text File Options" dialog boxes and without bothering with re-selecting proper text options.
- All nested subfolders of the worksheet tree are retained and recreated when you save a *.gsc file to such a zip archive.
- You can make using such archives even more automatic generating such zip with multiple *.xml options (for some or all text files).

**To open a *.zip file containing a number of text files**

Use the **File > Open** command and choose **Text | Text Archive** from the **File of type** list.

Text *.zip archives must contain text files with the *.txt, *.csv, *.tsv, *.tab extensions (optionally in zip's subfolders which will be retained in the imported workbook)

The text file options (separators, encoding, parsing data, formatting etc.) are specified as follows:

- by one [zip-archive-name].xml file in this zip archive with text file options for all included text files
- by one or more [included-text-file-name].xml files in this archive with text file for one or more included text files
- in the **Open Text File** dialog boxes displayed for one or more included text file which text options doesn't result from any of the two previous methods; checking the "apply to all (...)" checkbox in the dialog box causes applying it to the rest of the included text files.

The *.xml files must use the following construction:

```
<?xml version="1.0" encoding="UTF-8"?>
<text-file-options
    use-separator="true"

    separator="tab"            // comma | tab | point | semicolon | space |
any char.
    use-quoting="true"
    quoting-symbol="&quot;"    // any char.

    use-escaping="true"
    escape-symbol="&quot;"     // any char.

    use-fixed-widths="false"
    fixed-width=""             // 10 | 20,40,30
```

```
    formula-conversion="none"  // none | 3.x-6.x | 7.x-16.x

    text-to-numbers="true"            // e.g. "123.45" -> 123.45

    text-to-formatted-numbers="true"    // e.g. "($123.45)" -> 123.45

    text-to-dates="false"             // "6/1/2017" -> a serial date number

    keep-styles="false"               // after parsing a formatted
date/number retain its style

    encoding="utf-8"          // utf-8 | utf-16 | ansi

    iso-code-page="Default Ansi code page"/>
                              // a code page name as in the "Open Text File"
dialog box
                              // "Default ANSI code page" ... "ISO 8859-15
Latin 9"
```

**Note:**
If a given text file is to be opened entirely as text (with no numbers and formulas parsing), all worksheet columns created from that file are automatically formatted using the "Text" style. To delete the "Text" file from a given column, click its header and either press Del then "Formatting" or choose the "Default style" on the formatting toolbar.

**To save a text *.zip archive file containing all worksheets:**

Use the **File > Save As** command, choose **Text | Text Archive** from the **File of type** list and after specifying the file name, add the ".zip" extension to it.

The **File > Save As** command causes displaying the **Save Text File** dialog box to let you specify the text file options. Subsequent **File > Save** commands reuses these settings, skipping this dialog box.

Text *.zip archives are saved with one [zip-archive-name].xml file in this zip archive with text file options for all included text files.

**To resize columns in the opened text file automatically**

Select either the **Auto-fit column widths** checkbox in the **Open Text File** dialog boc on in the **Settings > Options > General** dialog box.
(See Settings)

**To resize columns in the opened text file manually**

- For all columns at once: click a row heading to select a row and use the **Format > Auto-fit column widths** command
- For one or more column: select some cells from the desirable columns and use the **Format > Auto-fit column widths** command

## xBase files

**To open an xBase file**

Use the **File > Open** command and choose **dBaseIII, dBaseIV, FoxPro 2.x or Clipper** from the **File of type** list. Next, specify additional xBase file options:

**Encoding** Specifies the text encoding for a given xBase file: ANSI 8-bit or ISO/OEM.

xBase records marked as 'deleted' are displayed in red. To clear the 'deleted' flag in the saved file, remove/clear the font color from corresponding worksheet row.

**To resize columns in the opened xBase file automatically**

Select the **Settings > Options > General > Auto-fit column widths in imported files** option.
(See Settings)

**To save an xBase file**

Use the **File > Save** command and choose **dBaseIII, dBaseIV, FoxPro 2.x or Clipper** from the **File of type** list. Next, specify additional xBase file options:

**Encoding** Specifies the text encoding for a given xBase file: ANSI 8-bit or ISO/OEM.

To mark some xBase records as 'deleted' in the saved file, format the corresponding worksheet rows using a red font.

**Note:** After saving a document in a different format you may need to use the **File > Reload** command to refresh the view.

# Printing workbooks

## Printing and print previewing

By default, after choosing the **Print** command, GS-Calc prints the entire current worksheet using content options/settings specified in the **Print Worksheet > Content** dialog.

**To print a range of cells**

Select the desirable range and choose the **Selection** option in the **Print Worksheet > General** dialog.

**To print selected pages**

specify the list of pages in in the **Print Worksheet > General** dialog.
The list can contain single pages or page ranges separated by commas. For example: 1,2,4-8,10

**To print an inserted chart or image**

Select (click) the desirable object before choosing the **Print** command. When printing a chart, it'll be stretched/shrunk to fit the printed page.

**To include a worksheet name, page number, date etc. in the printed pages headers and footers**

In the **Print > Layout** dialog box, enter the header/footer text inserting the following special codes:

| | |
|---|---|
| **&p** | Prints the current page number |
| **&p+number** | Prints the current page number plus 'number' |
| **&p-number** | Prints the current page number minus 'number' |
| **&f** | Prints the full file path |
| **&a** | Prints the name of the worksheet |
| **&d** | Prints the current date |
| **&t** | Prints the current time |
| **&&** | Prints a single ampersand |

**To change the view scale in the print-preview window**

Click the **Zoom In** button and pressing the right mouse button select a rectangle that should be zoom in.

# Settings

## Settings

To modify GS-Calc settings, use the **Settings > Options** dialog box.

Note that GS-Calc doesn't store any information in the Windows registry except adding COM/shell entries required by other applications. These entries are added (optionally) during the installation and you can remove/add them at any time using the **Settings > Register-Unregister** commands. All program settings are saved to the 'settings.xml' file.

When starting, GS-Calc looks for this file in the following places:

1. The file system folder for local application data (e.g. C:\Documents and Settings*username*\Local Settings\Application Data\GS-Calc).
2. The installation folder of GS-Calc.

Before closing, the settings are saved as follows:

1. The initial/original file, if it exists.
2. The file system folder for local application data, if GS-Calc is installed in the default system "Programs Folder" folder.
3. The installation folder of GS-Calc, if GS-Calc is not installed in the default system "Programs Folder" folder.

You can use the **Settings > Save - Load Profile** commands to save or load different profile files.

**Undo/Redo level**

The number of actions that can be undone via the **Edit/Undo-Redo** commands.
Valid range:
Default: 20

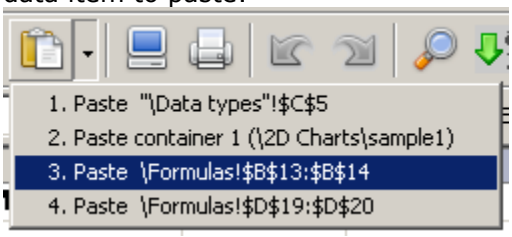
**Notes:**

- If you set a very high undo level and copy/paste very large amounts of data, the system memory usage may jump significantly as well.
- The drag-drop copying is treated as a two-step action so the minimum undo level is 2.

**Copy/Paste level**

The number of data 'slots' that can be filled cyclically via subsequent copy operations.
Valid range:
Default: 1

By default, the **Edit/Paste** command inserts the last copied data. To paste any data available in the Copy/Paste buffer, click the arrow button associated with the **Paste** button and choose the data item to paste:




**When saving new ODF (*.ods) files, use formula notation compatible with**

Specifies which formula notation should be used when saving newly created *.ods files.
You can choose one of the following formula "namespaces":
Excel 2010/2013,
OpenFormula,
Google Docs.
When editing existing *.ods files GS-Calc preserves their current formula notation. To change it you can use the **Save As** command.
Saving a file with the right formula namespace might be important because opening an ODF file in other applications that don't support it might result in removing the formulas and replacing them with their current values.
Default: Excel 2010/2013

**Auto-format cells**

If this option is **on**, GS-Calc tries to identify the style of each newly entered data and changes the previous cell format if it's different.
Default: On

**Use date picker when editing dates**

If this option is **on** the system "Date Picker" control is displayed when you edit cell containing dates.
For more information on using the date picker, see: editing and filling cells.
Default: On

**Parse dates in formatted cells only**

If this option is **on** text strings representing dates will be converted to date serial numbers only in cells having the **Date**, **Time** or **Date/Time** format set.
Default: Off

**Enable syntax coloring**

If this option is **on** formulas edited in cells will be displayed using syntax coloring. Cell references, strings and errors will be displayed in various colors and/or using various font attributes. This functionality is provided thanks to the © **Scintilla** project.
Default: Off

**Update copied absolute cell references**

Specifies whether absolute cell references like $A$1 in formulas should be modified when copying cells and inserting/deleting columns/rows.
Default: Off

**Update copied relative cell references**

Specifies whether relative cell references like A1 in formulas should be modified when copying cells and inserting/deleting columns/rows.
Default: On

**Hide cell markers**

Hides cell markers that are normally displayed in cells which contain formulas, drop-down lists, comments, circular references (>1) and truncated text.
Default: On

**Show progress when saving/loading files**

If this option is **on**, GS-Calc displays a progress bar when saving and opening files.
Default: On

**Display formulas instead of their values**

Displays formulas instead of their values.
Default: Off

**Display references using R1C1 notation**

Changes the current cell address notation. See: Cell references.
You must use the current notation when entering formulas. Note that this doesn't affect existing/saved workbooks - internally, always the A1 notation is used.
Default: Off (Use A1)

**Extended toolbar button styles**

Adds the 3rd "indeterminate" button state for toolbar formatting buttons.
By default, if a range is selected the button states reflect the binary format/state of the top-left cell of the selection, e.g. "Bold" or "Not Bold". With the above option selected, the entire selected range is checked and if that range contains e.g. both "Bold" or "Not Bold", the button is set to the "indeterminate" state.
**Note:** Choosing this option may significantly slow down selecting very large ranges. Default: Off

**Auto-fit column widths in imported files**

If this option is turned on, after opening a text or xBase file all not empty columns will be automatically resized to fit their widths to their contents.
The resizing is performed within the limits defined by the <AutoFitMin> and <AutoFitMax> elements in the GS-Base **settings.xml** configuration file. By default, these values are equal respectively to 20 pixels and 640 pixels and they can be modified within the <2; 2000> range.
**Note:** Choosing this option may slow down opening very large text and xBase files.
Default: Off

**Automatically close Chart dialog box**

If this option is turned on, the **Chart** dialog box is closed when you switch to another worksheet. Leaving that dialog box opened is helpful if the chart resides in one worksheet and its data is selected in another.
Default: Off

**Cell text overflow**

Specifies whether and how truncate the text which is longer that the cell width.
Default: Allow if adjoining cells are empty.

**After selecting cells**

Specifies whether GS-Base should scroll back to the cell where selecting cells started. If this option is turned off, to scroll back to the selection origin, click the address displayed on worksheet window toolbar.
Default: No action

**Mouse wheel**

Specifies how GS-Base should react to a single mouse wheel move. Available options are:

- scrolling lines,
- scrolling pages.

Default: Scroll lines

**After pressing Enter**

Specifies whether (and how) the current position should be scrolled after you finish editing a given cell and press **Enter**. Default: No action

**Start folder**

Specifies the folder displayed in the **Open File** dialog box when you open it for the first time (after launching GS-Calc).

**AutoSave after**

Specifies the period of user inactivity after which GS-Calc will automatically save all open workbooks.
Default: Never

**AutoClose after**

Specifies the period of user inactivity after which GS-Calc will automatically close all open workbooks. If any of the workbooks was modified, it'll be saved before closing. Default: Never

**Windows opacity**

Specifies the opacity of dialog boxes. Decreasing this value means making the windows more transparent.
Valid range:
Default: 100% (no transparency)

**Default font**

Specifies the default font name.
Default: Arial

**Recursion level**

If there are circular formula references in a workbook, this parameter specifies how many such circular iterations should be performed when the workbook is updated. The default value of 1 means that after detecting such a cycle GS-Calc stops and exits such a calculation loop after detecting it. Circular cell markers are displayed if level >= 2.
To list all formulas with circular references, use the "Inspect Cells" pane and click the "Add" button.
Valid range:
Default: 1

**Perform background calculations**

If this option is **on**, GS-Calc updates formulas using processes running in the "background", that is, not blocking any other GS-Calc's functions. Updated results of formulas become available only after the procedure finishes. It's not possible to end up with or use partially updated data. Actions resulting in modifying cell data causes restarting of the active update cycle. Other actions, including formatting, don't have any impact on background updating.
The status bar shows the progress bar for the current re-calculation process and the total number of active processes.
Background updating requires more memory than the non-background mode. Default: On

**Keep workbooks referenced in calculations open**

If this option is **on** and cells contain references to external workbooks, GS-Calc will open and retain such workbooks in memory until the referencing workbook is closed. If this option is **off** the referenced workbooks are open and close each time a referencing cell is updated during the calculation process/loop.
Default: On

**Number of CPUs**

Specifies the number of CPUs/threads to be used when updating formulas, auto-fitting column widths and performing some editing actions. If two or more processor cores are available, allowing GS-Calc to use them will result in faster recalculation.
Choosing a value that is greater than the actual number of cores of your processor doesn't improve the performance.
The value specified here is used by workbooks that have their individual **Number of CPUs** option set to **Default**. You can overwrite this value for each workbook.
Default: 2

**Threads priority**

Specifies the priority of the update threads. The available priority classes correspond to the standard Windows settings for threads.
Note that choosing the highest priorities can significantly hamper all other processes in your system.
The value specified here is used by workbooks that have their individual **Threads priority** option set to **Default**. You can overwrite this value for each workbook.
Default: 2

**Automatic update mode**

Forces re-calculation of the entire workbook after each cell editing action.
The value specified here is used by workbooks that have their individual **Update Mode**

option set to **Default**. You can overwrite this value for each workbook.
Default: on

**Favor compound formulas**

Optimizes the calculations assuming that most of the formulas use both compound built-in functions and references.
The value specified here is used by workbooks that have their individual **Update Optimization Options** option set to **Default**. You can overwrite this value for each workbook.
Default: on

**Assume short calculation chains**

Optimizes the calculations assuming that worksheets contain short calculation chains (groups of linked cells).
The value specified here is used by workbooks that have their individual **Update Optimization Options** option set to **Default**. You can overwrite this value for each workbook.
Default: on

**Bottom-up calculation**

Recalculates cells/formulas moving from the right to the left and towards the top.
(The opposite order the default.)
The value specified here is used by workbooks that have their individual **Update Optimization Options** option set to **Default**. You can overwrite this value for each workbook.
Default: off

## Legacy compatibility

GS-Calc 11 introduced the following changes concerning the behavior of a few formulas to ensure better ODF files compatibility. If you're opening a file created by the version 10.x and older, you may need to check the **Settings > Legacy Compatibility** option.

- SUM()
  - **GS-Calc 10.x and older:**
    Text representations of numbers in references and arrays are inlcuded.
  - **GS-Calc 11 and newer:**
    Text representations of numbers in references and arrays are not inlcuded. You can also use the SUMA() function to include text representations of numbers.
- AVERAGE(), MIN(), MAX(), COUNT()
  - **GS-Calc 10.x and older:**
    Text representations of numbers in references and arrays are inlcuded.
  - **GS-Calc 11 and newer:**
    Text representations of numbers in references and arrays are not inlcuded. You can also use respectively the AVERAGEA(), MINA(), MAXA(), COUNTA() function to include text representations of numbers.
- INDEX(range, row, column)

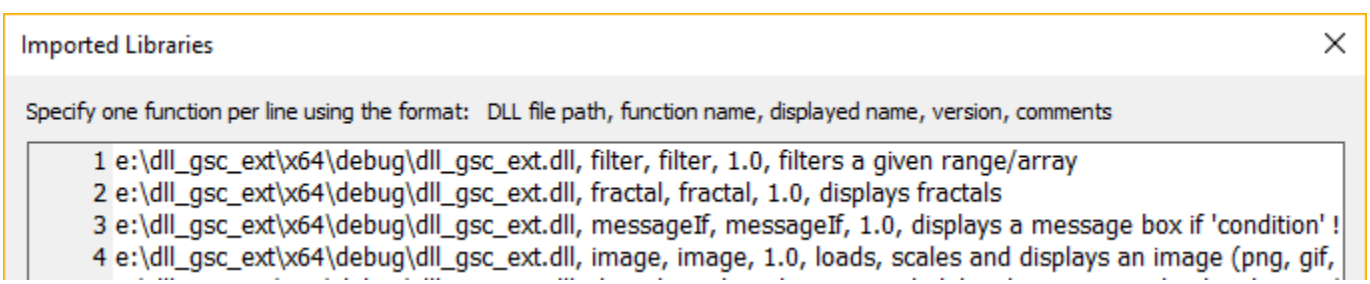
    - **GS-Calc 10.x and older:**
      Specifying 0 either as the row or as the column argument causes returning the #NUM! error.
    - **GS-Calc 11 and newer:**
      Specifying 0 either as the row or as the column argument have the same meaning as specifying blank parameters: it causes returning respectively either the full column or the full row of the range.
- MATCH(), VLOOKUP(), HLOOKUP()
    - **GS-Calc 10.x and older:**
      Specifying 1 and -1 as the search mode argument requires the searched range to be in the ascending order.
    - **GS-Calc 11 and newer:**
      Specifying -1 as the search mode argument requires the searched range to be in the descending order.
- NoPattern and StringSort flags in MATCH(), VLOOKUP(), HLOOKUP(), PIVOTDATA()
    - **GS-Calc 11.x:**
      Those flags are no longer required and are no longer documented.
    - **GS-Calc 14 and newer:**
      By default no pattern matching is used and string sorting is used (that is, punctation marks are not ignored).

# Custom DLL libraries

## Adding functions in DLL libraries

GS-Calc (ver. 17 and later) can be configured to use functions written by users in C/C++ as plain DLL libraries. Such functions can be even used as a replacement for the default built-in functions. They can accept and return all the argument types and additionally they can also "return" images displayed directly in worksheets or messages displayed after each update. What's important, using these functions GS-Calc retains the complete multicore support, the same speed and memory requirements (e.g. no extra arrays are allocated when passing ranges as arguments). You can add virtually any number of such functions.

Libraries are added by the Settings > Imported Function Libraries command as shown on the screenshot below. The internal function names MUST match the original C/C++ names. The displayed names are what is entered in worksheet cells and the only requirement is that they must be unique.



The added functions must be declared as

```
typedef double(__cdecl *DLL_EXT_FUNCTION)(GSCalcArg *arg);
```

where GSCalcArg is a structure enabling you to pass up to 15 arguments: numbers, strings and ranges/arrays.

```
struct ArrayItem
{
	uint8_t type;		// DLL_ARGT_EMPTY | DLL_ARGT_DOUBLE |
DLL_ARGT_TEXT
	uint8_t err_code;	// ERROR1_DIV_BY_ZERO...ERROR1_SYNTAX_ERROR
	uint16_t col;		// [0, 4095]
	uint32_t row;		// [0, 12582911]
	double num;		// numeric cell
	char text[MAXINPUT + 1];// text cell; max. 1024+NULL char. utf-8
string
};

typedef int(__cdecl *READ_ARRAY)(void *env, int array_index, ArrayItem *val);
typedef int(__cdecl *WRITE_ARRAY)(void *env, int array_index, ArrayItem
*val);

typedef void* (__cdecl *MALLOC)(size_t s);
typedef void(__cdecl *MFREE)(void *_Memory);

struct GSCalcArg
{
	uint8_t types[16];		// input: DLL_ARGT_NUMBER |
DLL_ARGT_TEXT | DLL_ARGT_ARRAY | DLL_ARGT_EMPTY
					// output: as above + DLL_ARGT_IMAGE |
DLL_ARGT_MESSAGE
	uint8_t errors[16];		//
ERROR1_DIV_BY_ZERO...ERROR1_SYNTAX_ERROR
	double numbers[16];
	char *strings[16];		// input (UTF-8) 0...14 strings are
preallocated by GS-Calc and must not be overwritten;
					// strings[15] is used for output
strings and already points to a MAXINPUT+1 char. temp. buffer
	struct Dims
	{
		uint32_t cx;
		uint32_t cy;
	} array_dims[16];		// dimensions of the subsequent and
grouped together range/array arguments passed from GS-Calc
	struct
	{
		BYTE *data;		// DIB data if you return types[15] =
DLL_ARGT_IMAGE; must be (de-)allocated with memory_alloc/memory_free
		size_t size;		// DIB data size
		Dims dims;		// optional resizing when displaying
the image (returned either as a DIB or a file path)
		MALLOC memory_alloc;
		MFREE memory_free;
```

```
        } image;
        READ_ARRAY read_array;         // returns ArrayItem.type
        WRITE_ARRAY write_array;       // returns ArrayItem.type or -1 if the
out-of-memory condition occurs
        void *env;                     // internal data that must be passed
back in read_array/write_array calls
};
```

Indices 0...14 in types[] and errors[] in GSCalcArg represent all subsequent argument types and errors from left to right passed from GS-Calc.

Indices 0...14 in numbers[], strings[] and array_dims[] point to argument values. The original parameters are already divided into these 3 groups (numbers, strings and ranges/arrays) when your functions is called. For example, if some worksheet cell execute your function

```
my_function(1.0, 2, "abc", a5:b15,)
```

the parameters passed to the DLL in the GSCalcArg structure will be as follows:

```
types[0] = DLL_ARGT_DOUBLE, types[1] = DLL_ARGT_DOUBLE, types[2] =
DLL_ARGT_TEXT, types[3] = DLL_ARGT_ARRAY, types[4] = DLL_ARGT_EMPTY
numbers[0] = 1, numbers[1] = 2, strings[0] = "abc", array_dims[0] = {2, 10}
```

You can access numbers and strings directly. Array elements can be read (and write on output) using the read_array() (and write_array()) functions as shown below in the **filter** function example. The first argument is the "env" internal pointer from the GSCalcArg structure, the 2nd is the array index (0-14) as passed in array_dims[] and the last is a pointer to the ArrayItem structure with the "col" and "row" of the requested element.

**Note: for very large array, for best possible performance while retrieving data from data from arrays or writing data to arrays, subsequent read_array() (and write_array()) calls MUST access the array elements in the left-to-right and top-to-bottom order.**

On output, the index 15 (DLL_ARGC_RET) is used to specify the returned type (in types[15]), possible error code (in errors[15]) and value (in numbers[15], strings[15] or array_dims[15]). The values types[15] and errors[15] MUST always be set correctly on return.
Note: the strings[15] already contain a valid buffer (up to 1024 characters + NULL) allocated by GS-Base and if a given function returns a text value, it must be copied to that buffer as a NULL terminated string.

**Examples:**

```
// messageIf(condition, text)
//
// arguments:
//                     condition, message text
```

```
// returns:
//                      if condition != 0, 'messageIf' displays 'text' as a
message box after each update/recalculation
//                      otherwise the message text is displayed in a cell as
normal text

__declspec(dllexport) double __cdecl messageIf(GSCalcArg *arg)
{
        uint8_t errorCode = 0;

        if (arg->types[0] != DLL_ARGT_DOUBLE && arg->types[0] !=
DLL_ARGT_EMPTY || arg->types[1] != DLL_ARGT_TEXT)
                errorCode = ERROR1_INVALID_VALUE;
        if (arg->types[2] != DLL_ARGT_NONE)   // more than 2 arguments
detected
                errorCode = ERROR1_SYNTAX_ERROR;

        if (!errorCode && arg->errors[0])
                errorCode = arg->errors[0];
        if (!errorCode && arg->errors[1])
                errorCode = arg->errors[1];

        if (!errorCode)
        {
                ::strncpy(arg->strings[DLL_ARGC_RET], arg->strings[0], 1024);
                arg->strings[DLL_ARGC_RET][1024] = 0;
                return arg->types[DLL_ARGC_RET] = (arg->numbers[0] ?
DLL_ARGT_MESSAGE : DLL_ARGT_TEXT);
        }
        else
        {
                arg->errors[DLL_ARGC_RET] = errorCode;
                return arg->types[DLL_ARGC_RET] = DLL_ARGT_EMPTY;
        }

        return 0;
}


// filter(range, column, number)
//
// arguments:
//                      range - a range/array
//                      column - column index (from 0 to the number of columns
in 'range' - 1)
//                      number - numeric value to perform equality check
// returns:
//                      an array consisting of rows of 'range' containing
'number' in the specified column

__declspec(dllexport) double __cdecl filter(GSCalcArg *arg)
{
        uint8_t errorCode = 0;

        if (arg->types[0] != DLL_ARGT_ARRAY || arg->types[1] !=
DLL_ARGT_DOUBLE || arg->types[2] != DLL_ARGT_DOUBLE)
                errorCode = ERROR1_INVALID_VALUE;
```

```
	if (arg->types[3] != DLL_ARGT_NONE)
		errorCode = ERROR1_SYNTAX_ERROR;

	if (!errorCode && arg->numbers[0] >= arg->array_dims[0].cx)
		errorCode = ERROR1_INVALID_VALUE;
	if (!errorCode && arg->errors[0])
		errorCode = arg->errors[0];
	if (!errorCode && arg->errors[1])
		errorCode = arg->errors[1];
	if (!errorCode && arg->errors[2])
		errorCode = arg->errors[2];

	ArrayItem x = { 0 };
	int outRow = 0;

	for (uint32_t inRow = 0; inRow < arg->array_dims[0].cy && !errorCode;
++inRow)
	{
		x.row = inRow;
		x.col = static_cast<uint16_t>(arg->numbers[0]);
		if (arg->read_array(arg->env, 0, &x) == -1)
		{
			errorCode = x.err_code;
			break;
		}

		bool match = false;
		if (x.type == DLL_ARGT_DOUBLE)
			match = (x.num == arg->numbers[1] || x.err_code &&
x.err_code == arg->errors[2]);
		else if (x.type == DLL_ARGT_EMPTY)
			match = (arg->numbers[1] == 0);
		else
			errorCode = ERROR1_INVALID_VALUE;

		if (match)
		{
			for (x.col = 0; x.col < arg->array_dims[0].cx &&
!errorCode; ++x.col)
			{
				x.row = inRow;
				arg->read_array(arg->env, 0, &x);
				x.row = outRow;

				// for best performance when writing to very
large arrays, try to write in the left-to-right and top-to-bottom order

				if (arg->write_array(arg->env, DLL_ARGC_RET,
&x) == -1) // -1 means the out-of-memory condition or an invalid row/column
					errorCode = x.err_code;
			}
			++outRow;
		}
	}

	arg->errors[DLL_ARGC_RET] = (!outRow ? ERROR1_NULL_VALUE : 0);
	return arg->types[DLL_ARGC_RET] = DLL_ARGT_ARRAY;
```

```
        return 0;
}



// image(path, cx, cy)
//
// arguments:
//                     path - a bmp, jpg, png, gif file path
//                     cx/cy - display image size (optional)
// returns:
//                     loads an image (bmp, jpg, png, gif) from a disk and
displays it, optionally resizing it
//                     to the cx/cy dimensions (in screen pixels)

__declspec(dllexport) double __cdecl image(GSCalcArg *arg)
{
        uint8_t errorCode = 0;

        if (arg->types[0] != DLL_ARGT_TEXT ||
                arg->types[1] != DLL_ARGT_DOUBLE && arg->types[1] !=
DLL_ARGT_EMPTY ||
                arg->types[2] != DLL_ARGT_DOUBLE && arg->types[2] !=
DLL_ARGT_EMPTY)
                errorCode = ERROR1_INVALID_VALUE;
        if (arg->types[3] != DLL_ARGT_NONE)
                errorCode = ERROR1_SYNTAX_ERROR;

        if (!errorCode && arg->errors[0])
                errorCode = arg->errors[0];

        ::strcpy_s(arg->strings[DLL_ARGC_RET], MAXINPUT, arg->strings[0]);
        arg->image.dims.cx = static_cast<uint32_t>(arg->numbers[0]);
        arg->image.dims.cy = static_cast<uint32_t>(arg->numbers[1]);

        // arg->image.data must remain NULL

        arg->errors[DLL_ARGC_RET] = errorCode;
        return arg->types[DLL_ARGC_RET] = DLL_ARGT_IMAGE;
}
```

Complete gsclib.h header file that must be included in each DLL project

A sample complete project with the **messageIf**, **filter**, **image** and **fractal** functions:

## gsclib.h

```
#pragma once

#define DLL_ARGC_RET           15      // index of the return value for
GSCalcArg.types[]...array_dims[]
```

```
#define DLL_ARGT_EMPTY        0
#define DLL_ARGT_DOUBLE               1
#define DLL_ARGT_TEXT         2
#define DLL_ARGT_ARRAY        4
#define DLL_ARGT_IMAGE        8
#define DLL_ARGT_MESSAGE      16
#define DLL_ARGT_NONE         128

#define MAXINPUT       1024

#define ERROR1_DIV_BY_ZERO            1       // #DIV/0!
#define ERROR1_INVALID_NAME           2       // #NAME?, invalid names in
formulas
#define ERROR1_NULL_VALUE             3       // #NULL!, empty intersection,
empty array result
#define ERROR1_NO_DATA                4       // #N/A!, no lookup results / no
data in a cell
#define ERROR1_INVALID_NUMBER  5      // #NUM!, wrong number values,
overflow, underflow
#define ERROR1_INVALID_REF            6       // #REF!, wrong row/col numbers,
array result exceeding the sheet
#define ERROR1_INVALID_VALUE   7      // #VALUE!, wrong argument types, too
long text strings, str_stack overflow
#define ERROR1_OUT_OF_MEMORY   8      // #MEMORY!, out of memory in an array
formula
#define ERROR1_ARRAY_NO_FILL   9      // #FILL!, array result area already
occupied by some 'foreign' cells
#define ERROR1_ARRAY_CIRC             10      // #CIRC!, circular reference
#define ERROR1_NOT_IMPLEMENTED 11     // #IMPL! function not yet implemented
#define ERROR1_SYNTAX_ERROR           12      // #SYNTAX! wrong number of
parameters, operators or an invalid name

struct ArrayItem
{
        uint8_t type;          // DLL_ARGT_EMPTY | DLL_ARGT_DOUBLE |
DLL_ARGT_TEXT
        uint8_t err_code;      // ERROR1_DIV_BY_ZERO...ERROR1_SYNTAX_ERROR
        uint16_t col;          // [0, 4095]
        uint32_t row;          // [0, 12582911]
        double num;                   // numeric cell
        char text[MAXINPUT + 1];// text cell; max. 1024+NULL char. utf-8
string
};

typedef void* (__cdecl *MALLOC)(size_t s);
typedef void(__cdecl *MFREE)(void *_Memory);

typedef int(__cdecl *READ_ARRAY)(void *env, int array_index, ArrayItem *val);
typedef int(__cdecl *WRITE_ARRAY)(void *env, int array_index, ArrayItem
*val);

struct GSCalcArg
{
        uint8_t types[16];            // input: as in ArrayItem::type |
DLL_ARGT_ARRAY
```

```
                                        // output: as above | DLL_ARGT_IMAGE |
DLL_ARGT_MESSAGE
        uint8_t errors[16];             //
ERROR1_DIV_BY_ZERO...ERROR1_SYNTAX_ERROR
        double numbers[16];
        char *strings[16];              // input (UTF-8) strings are pre-
allocated by GS-Calc and must not be overwritten
                                                // strings[15] used for output
strings already points to a MAXINPUT+1 char. temp. buffer
        struct Dims
        {
                uint32_t cx;
                uint32_t cy;
        } array_dims[16];               // dimensions of the subsequent group
together range/array arguments passed from GS-Calc
        struct
        {
                BYTE *data;                     // DIB data if you return
types[15] = DLL_ARGT_IMAGE; must be (de-)allocated with
memory_alloc/memory_free
                size_t size;            // DIB data size
                Dims dims;                      // optional resizing when
displaying the image (returned either as a DIB or a file path)
                MALLOC memory_alloc;
                MFREE memory_free;
        } image;
        READ_ARRAY read_array;          // returns ArrayItem.type
        WRITE_ARRAY write_array;        // returns ArrayItem.type or -1 if the
out-of-memory condition occurs
        void *env;                                      // internal data that
must be passed back in read_array/write_array calls
};


//typedef double(__cdecl *DLL_EXT_FUNCTION)(GSCalcArg *arg);
```

## dllmain.cpp

```
// dllmain.cpp : Defines the entry point for the DLL application.
#include "stdafx.h"

BOOL APIENTRY DllMain( HMODULE hModule,
                       DWORD  ul_reason_for_call,
                       LPVOID lpReserved
                     )
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
                break;
    case DLL_THREAD_ATTACH:
```

```
                break;
        case DLL_THREAD_DETACH:
                break;
        case DLL_PROCESS_DETACH:
                break;
    }

    return TRUE;
}
```

## gsclib.cpp

```
// gsclib.cpp : Defines the exported functions for the DLL application.
//

#include "stdafx.h"
#include "stdlib.h"
#include "stdint.h"
#include "stdio.h"
#include <assert.h>

#include "gsclib.h"

struct FractalParams
{
        COLORREF colors[16];
        int expand;                             // palette expansion factor: <1,
64>; 1 - use basic 16 colors only, 64 - use 16 x 64 shades
        int step;                               // RGB step for subsequent
expaned color
        double zr, zi;
        double xmin, xmax;
        double ymin, ymax;
};

COLORREF defaultColors[] =
{
        RGB(0, 255, 0),         // Green
        RGB(255, 255, 0),       // Yellow
        RGB(255, 0, 0),         // Red
        RGB(0, 0, 255),         // Blue
        RGB(0, 255, 255),       // Cyan
        RGB(128, 128, 128),     // Grey
        RGB(0, 128, 128),       // Aqua
        RGB(255, 0, 255),       // Magenta
        RGB(128, 128, 0),       // Brown
        RGB(128, 0, 128),       // Purple
        RGB(255, 255, 255),
```

```
        RGB(128, 0, 0),
        RGB(0, 128, 0),
        RGB(0, 0, 128),
        RGB(192, 192, 192),
        RGB(0, 0, 0)            // Black
};


void DrawFractal(HDC hdc, const RECT& rect, FractalParams *p)
{
        //double p = /*-1.25*/0.32, q = /*0.1*/0.043;
        //double p = 0.1, q = 0.43;
        //double p = p->zr;
        //double q = p->zi;

        double m = 100;
        double zoom_a = (p->xmax - p->xmin) / (rect.right - rect.left + 1);
        double zoom_b = (p->ymax - p->ymin) / (rect.bottom - rect.top + 1);

        int palette = p->expand * 16;

        COLORREF colorsExt[1024];

        for (int c = 0; c < 16; ++c)
        {
                BYTE r = GetRValue(p->colors[c]);
                BYTE g = GetGValue(p->colors[c]);
                BYTE b = GetBValue(p->colors[c]);

                colorsExt[p->expand*c + p->expand / 2] = p->colors[c];

                for (int i = 1; i < p->expand / 2 + p->expand % 2; ++i)
                        colorsExt[p->expand*c + p->expand / 2 + i] = RGB(min(r
+ i * p->step, 255), min(g + i * p->step, 255), min(b + i * p->step, 255));
                for (int i = 1; i < p->expand / 2 + 1; ++i)
                        colorsExt[p->expand*c + p->expand / 2 - i] =
RGB(max((int)r - i * p->step, 0), max((int)g - i * p->step, 0), max((int)b -
i * p->step, 0));
        }

        for (int j = rect.top; j <= rect.bottom; ++j)
                for (int i = rect.left; i <= rect.right; ++i)
                {
                        double a = p->xmin + (i - rect.left)*zoom_a;
                        double b = p->ymin + (j - rect.top)*zoom_b;
                        double aN = 0, bN = 0;

                        USHORT n = 0;

                        while (true)
                        {
                                aN = a*a - b*b + p->zr/*p*/;
                                bN = 2 * a*b + p->zi/*q*/;

                                ++n;

                                a = aN;
```

```
                                b = bN;

                                double s = a*a + b*b;

                                if (m < s)
                                {
                                        ::SetPixelV(hdc, i, j, colorsExt[n]);
                                        break;
                                }
                                if (m == s)
                                {
                                        ::SetPixelV(hdc, i, j, colorsExt[0]);
                                        break;
                                }

                                if (n >= palette - 1)
                                {
                                        ::SetPixelV(hdc, i, j, colorsExt[n]);
                                        break;
                                }
                        }
                }
}

bool SaveFractalAsDIB(GSCalcArg *arg, BYTE*& dib, size_t& dibLen, const RECT&
rect, FractalParams *p, uint8_t& errorCode)
{
        int width = rect.right - rect.left + 1;
        int height = rect.bottom - rect.top + 1;
        int rowBytes = 4 * width;
        dibLen = sizeof(BITMAPINFOHEADER) + height*(rowBytes + (rowBytes % 4 ?
4 - rowBytes % 4 : 0)) * sizeof(BYTE);

        if (!(dib = static_cast<BYTE*>(arg->image.memory_alloc(dibLen))))
        {
                errorCode = ERROR1_OUT_OF_MEMORY;
                return false;
        }

        BYTE *bits = dib + sizeof(BITMAPINFOHEADER);
        BITMAPINFOHEADER *header = reinterpret_cast<BITMAPINFOHEADER*>(dib);
        BITMAPINFO *bmpInfo = reinterpret_cast<BITMAPINFO*>(dib);

        ::memset(header, 0, sizeof(BITMAPINFOHEADER));

        header->biSize = sizeof(BITMAPINFOHEADER);
        header->biWidth = width;
        header->biHeight = -height;
        header->biCompression = BI_RGB;
        header->biPlanes = 1;
        header->biBitCount = 32;
        header->biSizeImage = static_cast<DWORD>(dibLen) -
sizeof(BITMAPINFOHEADER);

        HDC screenDC = ::GetDC(NULL);

        HDC memoryDC = ::CreateCompatibleDC(screenDC);
```

```
        HBITMAP bmpNew = ::CreateCompatibleBitmap(screenDC, width, height);

        ::ReleaseDC(NULL, screenDC);

        if (!memoryDC || !bmpNew)
        {
                arg->image.memory_free(dib);
                dib = NULL;
                dibLen = 0;
                errorCode = ERROR1_OUT_OF_MEMORY;
                if (bmpNew)
                        ::DeleteObject(bmpNew);
                if (memoryDC)
                        ::DeleteDC(memoryDC);
                return false;
        }
        else
        {
                HBITMAP bmpOld = (HBITMAP)::SelectObject(memoryDC, bmpNew);

                DrawFractal(memoryDC, rect, p);

                ::SelectObject(memoryDC, bmpOld);
                ::GetDIBits(memoryDC, bmpNew, 0, height, bits, bmpInfo,
DIB_RGB_COLORS);
                ::DeleteObject(bmpNew);
                ::DeleteDC(memoryDC);
                return true;
        }
}


extern "C" {

// fractal(zr, zi, xmin, xmax, ymin, ymax, width, height, [colors], [expand],
[step])
//
// arguments:
//                      zr, zi - transformation
//                      xmin, xmax, ymin, ymax - zoom/offset
//                      width, height - physical bitmap size
//                      [colors] - 16-item RGB array/range with basic colors
(optional)
//                      [expand] - expands the number of colors 2-64 times
(optional)
//                      [step] - RGB step to obtain subsequent expanded colors
out of the basic colors (optional)
// returns:
//                      a pointer to the allocated memory containing the
fractal DIB bitmap;
//                      to obtain a higher resulution image when printing use
arg->image.dims.cx/.cy to specify the display rectangle
//                      and width/height to specify the actual DIB bitmap
dimensions

__declspec(dllexport) double __cdecl fractal(GSCalcArg *arg)
{
```

```
	uint8_t errorCode = 0;

	for (int i = 0; i < 8; ++i)
		if (arg->types[i] != DLL_ARGT_DOUBLE)
		{
			errorCode = ERROR1_INVALID_VALUE;
			break;
		}
		else if ((errorCode = arg->errors[i]) != 0)
			break;

	if (arg->types[11]/*...*/)
		errorCode = ERROR1_SYNTAX_ERROR;

	if (errorCode)
	{
		arg->errors[DLL_ARGC_RET] = errorCode;
		return arg->types[DLL_ARGC_RET] = DLL_ARGT_EMPTY;
	}

	struct FractalParams p = { 0 };
	RECT r = { 0, 0, 0, 0 };

	p.zr = arg->numbers[0];		// 1st DLL_ARGT_DOUBLE argument
	p.zi = arg->numbers[1];
	p.xmin = arg->numbers[2];
	p.xmax = arg->numbers[3];
	p.ymin = arg->numbers[4];
	p.ymax = arg->numbers[5];
	r.right = static_cast<int>(arg->numbers[6]);
	r.bottom = static_cast<int>(arg->numbers[7]);

	if (!errorCode && !(errorCode = arg->errors[8]))
	{
		if (arg->types[8] == DLL_ARGT_ARRAY)
		{
			if (arg->array_dims[0].cy*arg->array_dims[0].cx != 16)
// 0 - 1st array/range of the 11 arguments
				errorCode = ERROR1_INVALID_VALUE;
			ArrayItem x = { 0 };
			int index = 0;
			for (x.row = 0; x.row < arg->array_dims[0].cy && index
< 16 && !errorCode; ++x.row)
			{
				for (x.col = 0; x.col < arg->array_dims[0].cx
&& index < 16 && !errorCode; ++x.col)
				{
					arg->read_array(arg->env, 0, &x);
					if (x.type == DLL_ARGT_DOUBLE)
						p.colors[index++] =
static_cast<long>(x.num);
					else if (x.type == DLL_ARGT_EMPTY)
						p.colors[index++] =
::defaultColors[index];
					else
						errorCode =
ERROR1_INVALID_VALUE;
```

```
                                }
                        }
                }
                else if (arg->types[8] == DLL_ARGT_EMPTY)
                        ::memcpy(p.colors, ::defaultColors, 16 *
sizeof(COLORREF));
                else
                        errorCode = ERROR1_INVALID_VALUE;
        }

        if (!errorCode && !(errorCode = arg->errors[9]))
        {
                if (arg->types[9] == DLL_ARGT_DOUBLE)
                {
                        p.expand = static_cast<int>(arg->numbers[8]); // 8 -
9th number of the 11 arguments
                        errorCode = arg->errors[8];
                }
                else if (arg->types[9] == DLL_ARGT_EMPTY)
                        p.expand = 64;
                else
                        errorCode = ERROR1_INVALID_VALUE;
        }

        if (!errorCode && !(errorCode = arg->errors[10]))
        {
                if (arg->types[10] == DLL_ARGT_DOUBLE)
                {
                        p.step = static_cast<int>(arg->numbers[9]); // 9 - 10th
number of the 11 arguments
                        errorCode = arg->errors[9];
                }
                else if (arg->types[10] == DLL_ARGT_EMPTY)
                        p.step = 4;
                else
                        errorCode = ERROR1_INVALID_VALUE;
        }

        assert(p.expand > 0 && p.expand <= 64 && p.xmax > p.xmin && p.ymax >
p.ymin && p.step >= 1 && p.step <= 127);
        if (p.expand < 1 || p.expand > 64 || p.xmax <= p.xmin || p.ymax <=
p.ymin || p.step < 1 || p.step > 127 || r.right <= 0 || r.bottom <= 0)
                errorCode = ERROR1_INVALID_NUMBER;

        if (!errorCode)
                SaveFractalAsDIB(arg, arg->image.data, arg->image.size, r, &p,
errorCode);

        arg->errors[DLL_ARGC_RET] = errorCode;
        return arg->types[DLL_ARGC_RET] = DLL_ARGT_PNG;
}


// image(path, cx, cy)
//
// arguments:
//                      path - a bmp, jpg, png, gif file path
```

```
//                      cx/cy - display image size (optional)
// returns:
//                      loads an image (bmp, jpg, png, gif) from a disk and
displays it, optionally resing it
//                      to the cx/cy dimensions (in screen pixels)

__declspec(dllexport) double __cdecl image(GSCalcArg *arg)
{
        uint8_t errorCode = 0;

        if (arg->types[0] != DLL_ARGT_TEXT ||
                arg->types[1] != DLL_ARGT_DOUBLE && arg->types[1] !=
DLL_ARGT_EMPTY ||
                arg->types[2] != DLL_ARGT_DOUBLE && arg->types[2] !=
DLL_ARGT_EMPTY)
                errorCode = ERROR1_INVALID_VALUE;
        if (arg->types[3]/*...*/)
                errorCode = ERROR1_SYNTAX_ERROR;

        if (!errorCode && arg->errors[0])
                errorCode = arg->errors[0];

        ::strcpy_s(arg->strings[DLL_ARGC_RET], MAXINPUT, arg->strings[0]);
        arg->image.dims.cx = static_cast<uint32_t>(arg->numbers[0]);
        arg->image.dims.cy = static_cast<uint32_t>(arg->numbers[1]);

        // arg->image.data must remain NULL

        arg->errors[DLL_ARGC_RET] = errorCode;
        return arg->types[DLL_ARGC_RET] = DLL_ARGT_PNG;
}


// filter(range, column, number)
//
// arguments:
//                      range - a range/array
//                      column - column index (from 0 to the number of columns
in 'range' - 1)
//                      number - numeric value to perform equality check
// returns:
//                      rows of 'range' containing 'value' in the specified
column

__declspec(dllexport) double __cdecl filter(GSCalcArg *arg)
{
        uint8_t errorCode = 0;

        if (arg->types[0] != DLL_ARGT_ARRAY || arg->types[1] !=
DLL_ARGT_DOUBLE || arg->types[2] != DLL_ARGT_DOUBLE)
                errorCode = ERROR1_INVALID_VALUE;
        if (arg->types[3]/*...*/)
                errorCode = ERROR1_SYNTAX_ERROR;

        if (!errorCode && arg->errors[0])
                errorCode = arg->errors[0];
        if (!errorCode && arg->errors[1])
```

```
		errorCode = arg->errors[1];
	if (!errorCode && arg->errors[2])
		errorCode = arg->errors[2];

	ArrayItem x = { 0 };
	int outRow = 0;

	for (uint32_t inRow = 0; inRow < arg->array_dims[0].cy && !errorCode;
++inRow)
	{
		x.row = inRow;
		x.col = static_cast<int>(arg->numbers[0]);
		if (arg->read_array(arg->env, 0, &x) == -1)
		{
			errorCode = static_cast<BYTE>(x.err_code);
			break;
		}

		bool match = false;
		if (x.type == DLL_ARGT_DOUBLE)
			match = (x.num == arg->numbers[1] || x.err_code &&
x.err_code == arg->errors[2]);
		else if (x.type == DLL_ARGT_EMPTY)
			match = (arg->numbers[1] == 0);
		else
			errorCode = ERROR1_INVALID_VALUE;

		if (match)
		{
			for (x.col = 0; x.col < arg->array_dims[0].cx &&
!errorCode; ++x.col)
			{
				x.row = inRow;
				arg->read_array(arg->env, 0, &x);
				x.row = outRow;
				if (arg->write_array(arg->env, DLL_ARGC_RET,
&x) == -1)
					errorCode =
static_cast<BYTE>(x.err_code);
			}
			++outRow;
		}
	}

	arg->errors[DLL_ARGC_RET] = (!outRow ? ERROR1_NULL_VALUE : 0);
	return arg->types[DLL_ARGC_RET] = DLL_ARGT_ARRAY;

	return 0;
}


// messageIf(condition, text)
//
// arguments:
//		condition, message text
// returns:
```

```
//                      if condition != 0, 'messageIf' displays 'text' as a
message box after each update/recalculation

__declspec(dllexport) double __cdecl messageIf(GSCalcArg *arg)
{
	uint8_t errorCode = 0;

	if (arg->types[0] != DLL_ARGT_DOUBLE && arg->types[0] !=
DLL_ARGT_EMPTY || arg->types[1] != DLL_ARGT_TEXT)
		errorCode = ERROR1_INVALID_VALUE;
	if (arg->types[2]/*...*/)
		errorCode = ERROR1_SYNTAX_ERROR;

	if (!errorCode && arg->errors[0])
		errorCode = arg->errors[0];
	if (!errorCode && arg->errors[1])
		errorCode = arg->errors[1];

	if (!errorCode)
	{
		::strncpy(arg->strings[DLL_ARGC_RET], arg->strings[0], 1024);
		arg->strings[DLL_ARGC_RET][1024] = 0; // actually it's
guaranteed on the entry
		return arg->types[DLL_ARGC_RET] = (arg->numbers[0] ?
DLL_ARGT_MESSAGE : DLL_ARGT_TEXT);
	}
	else
	{
		arg->errors[DLL_ARGC_RET] = errorCode;
		return arg->types[DLL_ARGC_RET] = DLL_ARGT_EMPTY;
	}

	return 0;
}
```

# Scripting

## Interfaces and methods

### Registering scripting interfaces

To be able to use scripting, you must register the scripting interfaces in the Windows registry database. To do this, run GS-Calc as an administrator (e.g. right-click GS-Calc shortcut/icon and on the context menu choose "Run as administrator") and use the **Settings > Register Scripting Interfaces** command.
Similarly, if you no longer need scripting, you can use the **Settings > Unregister**

**Scripting Interfaces** command to remove all registry entries added earlier. Uninstalling GS-Calc will unregister those interfaces as well.

**Creating scripts**

You can create your scripts either as global scripts saved in the program settings and available to all databases or you can create scripts stored in a given GS-Calc workbook and available only after you open that workbook.
To create these scripts use the "File > Application Scripts" and "File > Database Scripts" commands.

The "(...) Scripts" dialog box enables you to organize your scripts in subfolders, test them to locate errors, copy/import/export them etc.

Sample "Scripts" screen.

Notes:

- If a global script is executed when there is no open workbook, it should start with one of the application Open(...)/New(..) functions
- If a workbook is already loaded, an executed script, either global or local, automatically refers to that workbook and using the Open(...) functions with the same workbook path has no effect.
- The backslash occurring in the script text mostly in file path parameters must always be doubled for example:
  f:\\my_folder\\file01.gsc

**Interfaces provided by GS-Calc**

- XBaseParams
- TextParams
- MergeParams
- FormatParams
- PrintSettings
- Worksheet
- Workbook
- Application

---

## XBaseParams

**format**

A string specifying the xBase file format:

- dbaseIII
- dbaseIV
- clipper
- foxpro

Example:

```
var xBaseParams = GSCalc.CreateXBaseParams();
xBaseParams.format = "dbaseIV";
```

**encoding**

A string specifying the character encoding in the xBase file:

- windows
- dos

Example:

```
var xBaseParams = GSCalc.CreateXBaseParams();
xBaseParams.encoding = "dos";
```

**GetFieldCount()**

Returns the number of defined database fields.

Example:

```
var xBaseParams = GSCalc.CreateXBaseParams();
xBaseParams.AddField("name", "C", 40, 0);
var counter = GetFieldCount();
```

**SetField(index, name, type, length, decimals)**

**AddField(name, type, length, decimals)**

**InsertField(index, name, type, length, decimals)**

Update the name and type of an existing database fields.

**index**

One-based field index.

**name**

A field name consisting of up to 10 characters.

**type**

A string specifying any of the xBase field types:

- C - character field
- N - numeric field
- F - numeric field
- L - logical field

- D - date field
- M - memo field

**length**

The field length. The specified value is used only the "C" field (up to 65534 for FoxPro and up to 254 otherwise) and the "N"/"F" fields.
The length of other fields are fixed and can't be changed.

**decimals**

The number of decimal places in the "N" fields.

Example:

```
var xBaseParams = GSCalc.CreateXBaseParams();
xBaseParams.AddField("item", "C", 40, 0);
xBaseParams.AddField("price", "N", 5, 2);
xBaseParams.SetField(1, "item", "C", 50, 0);
```

**GetFieldName(index)**

Returns the name of a given database field.

**index**

A one-based index of the field.

Example:

```
var xBaseParams = GSCalc.CreateXBaseParams();
xBaseParams.AddField("name", "C", 40, 0);
var name = GetFieldName(1); // returns "name"
```

**GetFieldType(index)**

Returns the type of a given database field.

**index**

A one-based index of the field.

Example:

```
var xBaseParams = GSCalc.CreateXBaseParams();
xBaseParams.AddField("name", "C", 40, 0);
var name = GetFieldType(1); // returns "C"
```

**GetFieldLength(index)**

Returns the length of a given database field.

**index**

A one-based index of the field.

Example:

```
var xBaseParams = GSCalc.CreateXBaseParams();
xBaseParams.AddField("name", "C", 40, 0);
var name = GetFieldLength(1); // returns 40
```

**GetFieldDecimals(index)**

Returns the length of a given numeric ("N") database field.

**index**

A one-based index of the field.

Example:

```
var xBaseParams = GSCalc.CreateXBaseParams();
xBaseParams.AddField("name", "C", 40, 0);
var name = GetFieldLength(1); // returns 40
```

**DeleteField(index)**

Deletes a given database field.

**index**

A one-based index of the field.

Example:

```
var xBaseParams = GSCalc.CreateXBaseParams();
xBaseParams.AddField("first_name", "C", 40, 0);
xBaseParams.AddField("last_name", "C", 40, 0);
xBaseParams.AddField("street", "C", 40, 0);
xBaseParams.AddField("city", "C", 40, 0);
var name = DeleteField(3);
```

---

## TextParams

**separator**

Specifies a character separating column in a text file.

Example:

```
var textParams1 = GSCalc.CreateTextParams();
textParams1.separator = "\t"; // tab-separeted values
var textParams2 = GSCalc.CreateTextParams();
textParams2.separator = ","; // command-separated values
```

**encoding**

A string specifying the character encoding in the text file:

- utf8
- utf16
- windows
- dos

Example:

```
var textParams = GSCalc.CreateTextParams();
textParams.encoding = "utf8";
```

**quotingSymbol**

Specifies a character used to quote values containing column/value separators. The inner quoting symbols are doubled.

Example:

```
var textParams = GSCalc.CreateTextParams();
textParams.quotingSymbol = "\"";
```

**loadNumbers**

A logical value specifying whether strings representing unformatted numbers should be converted to numbers. If the value is set to false, a text file will be open/saved faster and all worksheet cells will be text cells.

Example:

```
var textParams = GSCalc.CreateTextParams();
textParams.loadNumbers = true;
```

**loadFmtNumbers**

A logical value specifying whether strings representing formatted numbers should be converted to numbers. If the value is set to false, a text file will be open/saved faster and formatted numbers will become text cells.

Example:

```
var textParams = GSCalc.CreateTextParams();
textParams.loadFmtNumbers = true;
```

**loadDates**

A logical value specifying whether strings representing date/time values should be converted to date serial numbers. If the value is set to false, a text file will be open/saved faster and date strings will become text cells.
If the date data is to be sorted correctly, it must be converted to date serial numbers.

Example:

```
var textParams = GSCalc.CreateTextParams();
textParams.loadDates = false;
```

**loadDateStyles**

A logical value specifying whether styles of the date/time values converted to date serial numbers should be preserved in the opened worksheet. If the value is set to false, cells containing date serial numbers will be displaying them as numbers.

Example:

```
var textParams = GSCalc.CreateTextParams();
textParams.loadDateStyless = false;
```

**parsingMode**

A value specifying if and how possible formulas should be parsed when opening a text file:

- 1 - no parsing; possible formula strings will become text cells
- 2 - parsing GS-Calc 3.x-6.x formulas
- 3 - parsing GS-Calc 7.x-x.x formulas

Example:

```
var textParams = GSCalc.CreateTextParams();
textParams.parsingMode = 3;
```

**columnWidths**

A string specifying fixed column/value widths in the text file. Setting this value overwrites previously defined column/value separator. If a text file line contains more characters than the specified widths, the remaining ones will be forming subsequent max-1024-character fields.

Example:

```
var textParams = GSCalc.CreateTextParams();
textParams.columnWidths = "10, 30, 15, 20";
```

**saveFmtNumbers**

A logical value specifying whether numbers from formatted cells should be saved as formatted numbers or as generic non-formatted numbers (e.g. 1,123.10 vs 1123.1).

Example:

```
var textParams = GSCalc.CreateTextParams();
textParams.saveFmtNumbers = false;
```

**saveFmtDates**

A logical value specifying whether formatted dates (serial numbers and generic date/time strings) should be saved as the resulting formatted strings.

Example:

```
var textParams = GSCalc.CreateTextParams();
textParams.saveFmtDates = true;
```

**saveFormulaValues**

A logical value specifying whether formulas values should be saved in a text file. If it's set to "false", the very formulas will be saved instead.

Example:

```
var textParams = GSCalc.CreateTextParams();
textParams.saveFormulaValues = false;
```

**autoFitCols**

A logical value specifying whether after opening a given file columns should be automatically resized to fit their contents. Note: specifying 1/true will result in additional calculations after opening that file which may the entire opening process slightly longer.

Example:

```
var textParams = GSCalc.CreateTextParams();
textParams.autoFitCols = false;
```

---

## MergeParams

A set of parameters used by rows-merging functions.

- **path**

  Specifies a file(s) with records to merge. The path can contain a file name with wildcard (*, ?) characters, enabling you to merge certain files from a given folder or all files from that folder.

- **table**

  A string specifying the table with records to merge (for file formats capable of storing multiple tables).
  If it's null/empty string, the default/active table is used.

- **fieldNames**

  Specifies whether merged tables contain field/column names in the 1st row (that should be excluded from merging). 1/0, true/false
  Default: 0

- **matchFieldNames**

  A logical value specifying whether cells/fields from the merged table should be added to these columns where names in the first row in both tables match.
  Default: 0

- **enableUndo**

  A logical value specifying whether the Undo action should be possible.
  For mass rows merging it pays off to turn this option off to save memory.
  Default: 0

  Example:

```
var mergeParams = GSCalc.CreateMergeParams();
mergeParams.path = "e:\\data_files\\*.gsc";
mergeParams.table = "customers";
```

```
mergeParams.table = "pivot table reports\\data";
mergeParams.fieldNames = 1;
mergeParams.matchFieldNames = 0;
mergeParams.enableUndo = 0;
```

---

## FormatParams

**dataStyleName**

A string specifying the predefined data style name:

- Default
- General
- Currency
- Accounting
- Date
- Time
- Date/Time
- Fraction
- Percentage
- Scientific
- Custom

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.dataStyleName = "Currency";
```

**SetGeneralNumberFormat(decimals, zeroes, brackets, inRed, separators, scaling)**

In addition to setting the value of the **dataStyleName** property, this function enables you to modify the default options for the selected data style. **decimals**
The number of decimal places. This can be "auto" or any number from 0 to 14.
Default value: "auto"
**zeroes**
The number of leading zeroes. This can be any number from 0 to 14.
Default value: 1
**brackets**
A logical value specifying whether negative numbers should be enclosed in brackets.
Default value: false
**inRed**
A logical value specifying whether negative numbers should be displayed in red.
Default value: false
**separators**
A logical value specifying whether thousand separators should be used.
Default value: false
**scaling**

The display factor as a power of 1000. This can be any number from 0 to 5.
Default value: 0

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.SetGeneralNumberFormat(4, 1, false, false, false, 0);
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**SetCurrencyFormat(decimals, position, symbol, brackets, inRed, scaling)**

In addition to setting the value of the **dataStyleName** property, this function enables you to modify the default options for the selected data style. **decimals**
The number of decimal places. This can be "auto" or any number from 0 to 14.
Default value: "auto" (=2)
**position**
A string specifying the position of the currency symbol:

- "$1.1"
- "$ 1.1"
- "1.1$"
- "1.1 $"

Default value: "$1.1"
**symbol**
A string specifying the currency symbol.
Default value: "$"
**brackets**
A logical value specifying whether negative numbers should be enclosed in brackets.
Default value: false
**inRed**
A logical value specifying whether negative numbers should be displayed in red.
Default value: false
**scaling**
The display factor as a power of 1000. This can be any number from 0 to 5.
Default value: 0

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.SetCurrencyFormat(0, "$1.1", "$", false, false, 0);
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**SetAccountingFormat(decimals, symbol, scaling)**

In addition to setting the value of the **dataStyleName** property, this function enables you to modify the default options for the selected data style. **decimals**
The number of decimal places. This can be "auto" or any number from 0 to 14.
Default value: "auto" (=2)
**symbol**
A string specifying the currency symbol.
Default value: "$"
**scaling**
The display factor as a power of 1000. This can be any number from 0 to 5.
Default value: 0

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.SetAccountingFormat(0, "$", 0);
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**SetDateFormat(pattern, systemOrder)**

In addition to setting the value of the **dataStyleName** property, this function enables you to modify the default options for the selected data style. **pattern**
A string specifying the date format:
"m/d/yyyy", "mm/dd/yyyy", "m/d/yy", "mm/dd/yy", "m/d", "mm/yy", "mmm-d", "mmm-d-yyyy", "mmm-d-yy" "mmmm d, yyyy", "dddd, mmmm dd, yyyy".
Default value: "m/d/yyyy"
**systemOrder**
A logical value specifying whether day-month display order should be determined by the current system settings.
Default value: true

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.SetDateFormat("m/d", true);
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**SetTimeFormat(pattern)**

In addition to setting the value of the **dataStyleName** property, this function enables you to modify the default options for the selected data style. **pattern**
A string specifying the date format:

"h:mm", "h:mm AM/PM", "hh:mm", "hh:mm AM/PM", "h:mm:ss", "h:mm:ss AM/PM", "hh:mm:ss", "hh:mm:ss AM/PM", "[m]:ss.00", "[h]:mm:ss", "[d] hh:mm"
Default value: "h:mm"

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.SetTimeFormat("hh:mm AM/PM");
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**SetDateTimeFormat(datePattern, timePattern, systemOrder, timeFirst)**

In addition to setting the value of the **dataStyleName** property, this function enables you to modify the default options for the selected data style. The meaning of first three arguments is the same as in **SetDateFormat** and **SetTimeFormat** methods listed above.
**timeFirst**
A logical value specifying whether the time string should precede the date string.
Default value: false

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.SetDateTimeFormat("m/d","hh:mm AM/PM", true, false);
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**SetPercentFormat(decimals, scaling)**

In addition to setting the value of the **dataStyleName** property, this function enables you to modify the default options for the selected data style. **decimals**
The number of decimal places. This can be "auto" or any number from 0 to 14.
Default value: "auto"
**scaling**
The display factor as a power of 1000. This can be any number from 0 to 5.
Default value: 0

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.SetPercentFormat(0, 0);
wsheet.selectedRange = "d4";
```

```
wsheet.SetCellFormat(formatParams);
```

**SetFractionFormat(denominator, digits)**

In addition to setting the value of the **dataStyleName** property, this function enables you to modify the default options for the selected data style. **denominator**
If **digits** is true, this argument specifies the fixed number of denominator digits.
If **digits** is false, this argument specifies the fixed denominator value.
Default value: digits: false, denominator: 1

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.SetFractionFormat(2, false);
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**SetScientificFormat(decimals, exponent)**

In addition to setting the value of the **dataStyleName** property, this function enables you to modify the default options for the selected data style. **decimals**
The number of decimal places. This can be "auto" or any number between 0 and 14.
Default value: "auto"
**exponent**
The value of the exponent. This can be "auto" or any number between -99 and 99.
Default value: "auto"

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.SetScientificFormat("auto", 12);
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**fontName**

A string specifying the font name. To specify the default font name, use an empty string.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
```

```
formatParams.fontName = "Tahoma";
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**fontSize**

The size of the font in points. To specify the default size, use zero.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.fontSize = 8;
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**language**

A string specifying the language related to the font. This is a standard language code typically consisting of the country-language pair, eg: en-US, en-GB, en-AU, de-DE, de-AU, es-ES, pl-PL, ru-RU etc.
If the submitted string is invalid, it'll default to en-US.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.language = "en-GB";
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**boldFont**

Toggles the "bold" font attribute on/off.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.boldFont = true;
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**italicFont**

Toggles the "italic" font attribute on/off.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.italicFont = true;
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**underlineFont**

Toggles the "underline" font attribute on/off.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.underlineFont = true;
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**strikeoutFont**

Toggles the "strikeout" font attribute on/off.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.strikeoutFont = true;
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**fontColor**

Specifies the font color. Accepted values are: the default system "auto" color, predefined color names and strings representing 3-byte RGB color values using the hex notation.
The predefined values include: "black", "maroon", "green", "olive", "navy", "purple", "teal", "gray", "silver", "red", "lime", "yellow", "blue", "fuchsia", "aqua", "white".

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.fontColor = "#FF0000";
formatParams.fontColor = "green";
formatParams.fontColor = "auto";
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**horzAlignment**

Specifies the horizontal text alignment in cells. Accepted values are the following strings:

- left
- center
- right
- default

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.horzAlignment = "center";
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**vertAlignment**

Specifies the vertical text alignment in cells. Accepted values are the following strings:

- top
- center
- bottom
- base-line
- default

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.vertAlignment = "center";
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**horzIndent**

Specifies the horizontal text indent in cells. Accepted values are between 0 and 255. The default value is 4.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.horzIndent = 4;
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**vertIndent**

Specifies the vertical text indent in cells. Accepted values are between 0 and 255. The default value is 2.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.vertIndent = 1;
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**wrapText**

A logical value that toggles wrapping text in cells on/off. Setting this property to true automatically sets the shrinkText property to false.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.wrapText = true;
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**shrinkText**

A logical value that toggles shrinking text that overflows the width of the cells on/off. Setting this property to true automatically sets the wrapText property to false.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.shrinkText = true;
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**textRotation**

Specifies the rotation of the text in cells. Acceptable values are from -90 to 90 degrees.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.textRotation = 25;
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**hideFormula**

A logical value that toggles hiding formulas on/off.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.hideFormula = true;
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**protectedCell**

A logical value that toggles cell protection on/off.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.protectedCell = true;
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**printedCell**

A logical value that toggles printing cell contents on/off.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.printedCell = true;
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**borderStyle**

Specifies the cell border style. The following style names are accepted:

- none
- solid
- dot
- dash
- dash-dot
- dash-dot-dot

Specifying "none" remove all borders around the cell.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.borderColor = "green";
formatParams.borderPosition = 15;
formatParams.borderStyle = "solid";
formatParams.borderWidth = 1;
formatParams.borderDoubleLine = true;
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**borderWidth**

Specifies the cell border width. Accepted values are from 1 to 16 logical pixels.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.borderColor = "green";
formatParams.borderPosition = 15;
formatParams.borderStyle = "solid";
```

```
formatParams.borderWidth = 1;
formatParams.borderDoubleLine = true;
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**borderDoubleLine**

A logical value that toggles displaying double borders.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.borderColor = "green";
formatParams.borderPosition = 15;
formatParams.borderStyle = "solid";
formatParams.borderWidth = 1;
formatParams.borderDoubleLine = true;
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**borderColor**

Specifies the cell border color. Accepted values are: the default system "auto" color, predefined color names and strings representing 3-byte RGB color values using the hex notation.
The predefined values include: "black", "maroon", "green", "olive", "navy", "purple", "teal", "gray", "silver", "red", "lime", "yellow", "blue", "fuchsia", "aqua", "white".

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.borderColor = "green";
formatParams.borderPosition = 15;
formatParams.borderStyle = "solid";
formatParams.borderWidth = 1;
formatParams.borderDoubleLine = true;
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**borderPosition**

Specifies the cell border color. The following numeric values and their combinations are accepted:

- 0 - removes all borders

- 1 - top
- 2 - bottom
- 4 - left
- 8 - right
- 16 - diagonal left
- 32 - diagonal right
- 64 - borders around a range of cells

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.borderColor = "green";
formatParams.borderPosition = 15;
formatParams.borderStyle = "solid";
formatParams.borderWidth = 1;
formatParams.borderDoubleLine = true;
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**bkgColor**

Specifies the cell background color. Accepted values are: the default system "auto" color, predefined color names and strings representing 3-byte RGB color values using the hex notation.
The predefined values include: "black", "maroon", "green", "olive", "navy", "purple", "teal", "gray", "silver", "red", "lime", "yellow", "blue", "fuchsia", "aqua", "white".

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.bkgColor = "#FF0000";
formatParams.bkgColor = "green";
formatParams.bkgColor = "auto";
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**bkgImageName**

Specifies the cell background image. The name must represent an image available via the workbook list of images (see: "Cell Format > Background > Images").

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
```

```
var formatParams = wsheet.CreateFormatParams();
formatParams.bkgImageName = "blue-sphere.png";
formatParams.bkgImageOpacity = 100;
formatParams.bkgImageHorzPos = "left";
formatParams.bkgImageVertPos = "center";
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**bkgImageRepeatt**

Specifies how the cell background image should be displayed within the cell. The following text strings values are accepted:

- default - use original image size
- repeat - use original image size repeating the image within the whole cell
- stretch - stretch the image to fit it to the cell size

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.bkgImageName = "blue-sphere.png";
formatParams.bkgImageRepeat = "repeat";
formatParams.bkgImageOpacity = 100;
formatParams.bkgImageHorzPos = "left";
formatParams.bkgImageVertPos = "center";
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**bkgImageHorzPos**

Specifies the horizontal position of the cell background image. The following text string values are accepted:

- left
- center
- right

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.bkgImageName = "blue-sphere.png";
formatParams.bkgImageHorzPos = "right";
formatParams.bkgImageOpacity = 100;
formatParams.bkgImageHorzPos = "left";
formatParams.bkgImageVertPos = "center";
wsheet.selectedRange = "d4";
```

```
wsheet.SetCellFormat(formatParams);
```

**bkgImageVertPos**

Specifies the vertical position of the cell background image. The following text string values are accepted:

- top
- center
- bottom

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.bkgImageName = "blue-sphere.png";
formatParams.bkgImageVertPos = "top";
formatParams.bkgImageOpacity = 100;
formatParams.bkgImageHorzPos = "left";
formatParams.bkgImageVertPos = "center";
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**bkgImageOpacity**

Specifies the cell background image opacity. This is numeric value between 0 (a fully transparent image) to 100 (a fully opaque image).

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.bkgImageName = "blue-sphere.png";
formatParams.bkgImageVertPos = "top";
formatParams.bkgImageOpacity = 70;
formatParams.bkgImageHorzPos = "right";
formatParams.bkgImageVertPos = "center";
wsheet.selectedRange = "d4";
wsheet.SetCellFormat(formatParams);
```

**Reset()**

Resets the FormatParams object and initiates it with default or empty attributes. When using such an object with the formatting method of the Worksheet object, only submitting explicitly a given FormatParams property triggers a corresponding formatting action. For example, the following code:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.boldFont = true;
formatParams.italicFont = true;
wsheet.selectedRange = "b10:b20";
wsheet.SetCellFormat(formatParams);
```

is an equivalent to clicking the "Bold" toolbar button, then the "Italic" format button. If the following code re-use the same FormatParams object without resetting:

```
formatParams.dataStyleName = "Currency";
wsheet.selectedRange = "b10:b20";
wsheet.SetCellFormat(formatParams);
```

it will be an equivalent to clicking the "Bold" toolbar button, then the "Italic" format button, then choosing the "Currency" style.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.Reset();
```

---

## PrintSettings

**printedData**

Specifies which data are to be printed. The following text string values are accepted:

- all - printing all pages and all data
- default - same as "all"
- pages - printing a page range specified by the "printedPages" property
- selection - printing a currently selected range of cells

"All" is the default value.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.printedData = "all";
wsheet.SetPrintSettings(printSettings);
```

```
wsheet.Print(false);
```

**printedPages**

Specifies the range of printed pages as a comma-separated list of pages and page ranges.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.printedData = "pages";
printSettings.printedPages = "1,3,5-8";
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**orientation**

Specifies the printed pages orientation. The following text string values are accepted:

- landscape
- portrait

"Portrait" is the default value.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.orientation = "landscape";
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**horzPageAlign**

Specifies the horizontal alignment of the printed pages. The following text string values are accepted:

- left
- center
- right

The default value is "left".

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.horzPageAlign = "center";
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**vertPageAlign**

Specifies the vertical alignment of the printed pages. The following text string values are accepted:

- top
- center
- bottom

The default value is "top".

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.vertPageAlign = "center";
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**marginLeft**

Specifies the left printed page margin in mm. The default value is an equivalent to 0.5".

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.marginLeft = 6;
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**marginTop**

Specifies the top printed page margin in mm. The default value is an equivalent to 0.8".

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.marginTop = 6;
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**marginRight**

Specifies the right printed page margin in mm. The default value is an equivalent to 0.5".

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.marginRight = 6;
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**marginBottom**

Specifies the bottom printed page margin in mm. The default value is an equivalent to 1.1".

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.marginBottom = 6;
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**headerMargin**

Specifies in mm the margin between the header and the top edge of the printed page. The default value is an equivalent to 0.1".

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.headerMargin = 6;
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**footerMargin**

Specifies in mm the margin between the footer and the bottom edge of the printed page. The default value is an equivalent to 0.1".

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.footerMargin = 6;
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**header**

Specifies the header text. The header can contain special content fields and alignment controlling codes. For more information, please see the documentation/help for the "Print" dialog box.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.header = "&c Document: &f";
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**footer**

Specifies the footer text. The footer can contain special content fields and alignment controlling codes. For more information, please see the documentation/help for the "Print" dialog box.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.header = "&c &p";
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**scale**

Specifies the printing scale. The valid values are from 10% to 999%.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.scale = 120;
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**autoFit**

A logical value specifying whether the printed worksheet data should be shrunk to fit the page.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.autoFit = true;
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**printHeadings**

A logical value specifying whether the (table) row and column headings should be printed.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
```

```
printSettings.printHeadings = true;
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**printObjects**

A logical value specifying whether objects (charts and images) should be printed.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.printObjects = true;
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**printGrid**

A logical value specifying whether gridlines should be printed.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.printGrid = true;
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**printFormulas**

A logical value specifying whether cells containing formulas should be printed (either as their values or the very formulas).

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.printFormulas = true;
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**printZeroes**

A logical value specifying whether zero cell values should be printed.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.printZeroes = true;
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**bkgColor**

Specifies the page background color. Accepted values are: the default system "auto" color, predefined color names and strings representing 3-byte RGB color values using the hex notation.
The predefined values include: "black", "maroon", "green", "olive", "navy", "purple", "teal", "gray", "silver", "red", "lime", "yellow", "blue", "fuchsia", "aqua", "white".

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.bkgColor = "#FDFDFD";
//printSettings.bkgColor = "gray";
//printSettings.bkgColor = "auto";
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**bkgImageName**

Specifies the page background image. The name must represent an image available via the workbook list of images (see: "Cell Format > Background > Images").

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.bkgImageName = "blue-sphere.png";
printSettings.bkgImageOpacity = 100;
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**bkgImageRepeat**

Specifies how the page background image should be displayed within the page. The following text strings values are accepted:

- default - use original image size
- repeat - use original image size repeating the image within the whole page
- stretch - stretch the image to fit it to the page size

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.bkgImageName = "blue-sphere.png";
printSettings.bkgImageRepeat = "repeat";
printSettings.bkgImageOpacity = 100;
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**bkgImageHorzPos**

Specifies the horizontal position of the page background image. The following text string values are accepted:

- left
- center
- right

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.bkgImageName = "blue-sphere.png";
printSettings.bkgImageOpacity = 100;
printSettings.bkgImageHorzPos = "right";
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**bkgImageVertPos**

Specifies the vertical position of the page background image. The following text string values are accepted:

- top
- center
- bottom

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.bkgImageName = "blue-sphere.png";
printSettings.bkgImageOpacity = 100;
printSettings.bkgImageVertPos = "center";
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**bkgImageOpacity**

Specifies the page background image opacity. This is numeric value between 0 (a fully transparent image) to 100 (a fully opaque image).

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
wsheet.GetPrintSettings(printSettings);
printSettings.bkgImageName = "blue-sphere.png";
printSettings.bkgImageOpacity = 50;
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

---

## Worksheet

**selectedRange**

Specifies the current cell range selection.

After selecting a new range, the corresponding active worksheet will be scrolled to ensure that range (or at least its top-left cell) is visible.
Setting a selection automatically de-select any currently selected objects.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
wsheet.selectedRange = "c20";
wsheet.UpdateWindows();
GSCalc.Sleep(2000);
wsheet.selectedRange = "c20:d22";
wsheet.UpdateWindows();
GSCalc.Sleep(2000);
wsheet.selectedRange = "2:2";
```

```
wsheet.UpdateWindows();
GSCalc.Sleep(2000);
wsheet.selectedRange = "H:I";
```

**topLeftCell**

Specifies the top-left cell of the worksheet window.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
wsheet.topLeftCell = "c20";
```

**Scroll(type, scrollSelection)**

Scrolls the worksheet view. If **scrollSelection** is **false**, calling this method corresponds to dragging the scroll boxes of the view scrollbars (and to pressing scrolling keys with the "Scroll Lock" key on.
If **scrollSelection** is **false** (and the "Scroll Lock" key off), calling this method corresponds to pressing the cursor keys.
Unlike other methods, **Scroll** enforces screen updates during the excecution of the script so **UpdateWindow()** doesn't have to be used.

The **type** argument can be of the following text strings:

- line-down
- line-up
- line-right
- line-left
- page-down
- page-up
- page-right
- page-left
- top
- bottom
- home
- end

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var i;
for ( i = 0; i < 50; ++i )
{
        wsheet.Scroll("line-down", true);
        GSCalc.Sleep(200);
}
```

**Address(column, row)**

Creates and returns and a string representing the specified relative cell address. To obtain only the column or row name/number, specify 0 as the 2nd argument.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
ver cell = wsheet.Address(3, 20); // returns "C20"
ver cell2 = wsheet.Address(3, 0); // returns "C"
```

**AddressEx(column, relativeColumn, row, relativeRow)**

Creates and returns and a string representing the specified relative or absolute cell address, depending on the logical **relativeColumn** and **relativeRow** arguments. To obtain only the column or row name/number, specify 0 as the 2nd argument.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
ver cell = wsheet.AddressEx(3, true, 20, false); // returns "C$20"
ver cell = wsheet.AddressEx(3, false, 0, false); // returns "$C"
```

**Column(cell)**

Returns the column number of the specified cell address.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
ver column = wsheet.Column("C20"); // returns 3
```

**Row(cell)**

Returns the column number of the specified cell address.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
ver column = wsheet.Column("C20"); // returns 20
```

**GetFirstColumn()**

Returns the number of the first non-empty column.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
ver column = wsheet.GetFirstColumn();
```

**GetFirstRow()**

Returns the number of the first non-empty row.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
ver row = wsheet.GetFirstRow();
```

**GetLastColumn()**

Returns the number of the last non empty column.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
ver column = wsheet.GetLastColumn();
```

**GetLastRow()**

Returns the number of the last non empty row.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
ver row = wsheet.GetLastRow();
```

**GetMaxColumn()**

Returns the maximum column number.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
ver column = wsheet.GetMaxColumn();
```

**GetMaxRow()**

Returns the maximum row number.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
ver row = wsheet.GetMaxRow();
```

**CreateFormatParams()**

Creates and returns an object containing cell formatting parameters. The **FormatParams** object is then used to retrieve existing cell formats and to set new ones.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.boldFont = true;
formatParams.italicFont = true;
wsheet.selectedRange = "b1:b20";
wsheet.SetCellFormat(formatParams);
```

**GetCellFormat(formatParams)**

Retrieves a cell format for the selected cell or range. The corresponding column or row format is ignored and must be obtained using the GetColumnFormat() and GetRowFormat() methods.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
wsheet.selectedRange = "b10:b20";
wsheet.GetCellFormat(formatParams);
```

**GetColumnFormat(formatParams)**

Retrieves a column format relative to the seleted cell or range.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
wsheet.selectedRange = "b10:b20";
wsheet.GetColumnFormat("b10:b20", formatParams);
```

**GetRowFormat(formatParams)**

Retrieves a row format relative to the selected cell or range.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
wsheet.selectedRange = "b10:b20";
wsheet.GetRowFormat(formatParams);
```

**SetCellFormat(formatParams)**

Formats the selected cell or range of cells using formatting attributes set in the FormatParams object. If the range doesn't include any complete columns or rows, the formatting will be applied to subsequent individual cells.
If the selected range specifies a column selection (e.g. c1:c12582912, h1:12582912, c:c, c:h), the formatting will be applied to whole columns.
If the selected range specifies a row selection (e.g. a10:fan10, 1:1, 4:6), the formatting will be applied to whole rows.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var formatParams = wsheet.CreateFormatParams();
formatParams.boldFont = true;
formatParams.italicFont = true;
wsheet.selectedRange = "b10:b20";
wsheet.SetCellFormat(formatParams);
```

**SetCellCustomStyle(name)**

Formats the selected cell or a range of cells using a user-defined custom cell style/format. Cell styles can be added, edited and deleted via the "Format > Custom Cell Styles" dialog box or via the "Format > Cell Format" dialog box.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
wsheet.selectedRange = "b10:b20";
wsheet.SetCellCustomStyle("cell style 1.");
```

**DeleteCells(showDialogBox, contentType)**

Deletes selected cell(s). The **showDialogBox** argument is a logical value that specifies whether the **Delete** dialog box should be displayed.
The **contentType** argument is a text string specifying the type of the deleted data. It can be any combination of the

- numbers
- labels
- formulas
- data
- formatting
- lists
- comments

The **data** option is an equivalent to "number, labels, formulas".

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
wsheet.selectedRange = "c10:c20";
wsheet.DeleteCells(false, "data");
wsheet.selectedRange = "d10";
wsheet.DeleteCells(false, "formatting");
wsheet.selectedRange = "b4";
wsheet.DeleteCells(true, "formatting, lists, comments");
```

**CopyCells(copyType)**

Copies selected cell(s). The **contentType** argument is a numeric 0-7 value specifying the type of the copy action:

- 1 - simple copy action
- 2 - copy and transpose
- 3 - copy and reverse rows
- 4 - copy and reverse columns
- 5 - simple copy with formulas replaced by their values
- 6 - copy values and transpose
- 7 - copy values and reverse rows
- 8 - copy values and reverse columns

Example:

```
  var wbook = GSCalc.ThisWorkbook();
  var wsheet = wbook.GetActiveWorksheet();
  wsheet.selectedRange = "c10:c20";
  wsheet.CopyCells(0);
  wsheet.selectedRange = "d10";
  wsheet.PasteCells(true);
  wsheet.selectedRange = "b4";
  wsheet.CopyCells(0);
  wsheet.selectedRange = "b5";
  wsheet.PasteCells(true);
  wbook.WaitForUpdate(); // to allow the background updating to complete
before further actions
  // ...
```

**PasteCells(textAndFormatting)**

Pastes previously copied cell(s) or the textual data from the Clipboard. The **textAndFormatting** argument is a logical value specifying whether the copied cells should be pasted along with their formatting.
If the current selection is a single cell, the copied and pasted data block will be the same. If the current selection spans multiple rows and/or columns, the pasted data will be duplicated, filling the selected range if it's bigger than the source range.

Executing a large number of the **PasteCells** actions will be faster if the **updateMode** property is set to **manual** and the application/workbook window is minimized.

Example:

```
  var wbook = GSCalc.ThisWorkbook();
  var wsheet = wbook.GetActiveWorksheet();
  wsheet.selectedRange = "c10:c20";
  wsheet.CopyCells(0);
  wsheet.selectedRange = "d10";
  wsheet.PasteCells(true);
  wsheet.selectedRange = "b4";
  wsheet.CopyCells(0);
  wsheet.selectedRange = "b5:b10";
  wsheet.PasteCells(true);
```

**GetColumnWidth()**

Returns width of the current column (a column related to the top-left cell of the selection) in pixels. The width is calculated for the view scale = 100%. If the column width is automatic, the "auto" string is returned.

Example:

```
  var wbook = GSCalc.ThisWorkbook();
```

```
var wsheet = wbook.GetActiveWorksheet();
var width = wsheet.GetColumnWidth();
```

**SetColumnWidth(width)**

Sets the width for the columns included in the currently selected range. The **width** argument is a text string that can represent either a width in pixels related to the view scale = 100% or the "auto" text string. The latter causes adjusting the widths to the contained data.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
wsheet.selectedRange = "b5:e5";
wsheet.SetColumnWidth("auto");
wsheet.selectedRange = "f5";
wsheet.SetColumnWidth("30");
```

**GetRowHeight()**

Returns the height of the current row (a row related to the top-left cell of the selection) in pixels. The height is calculated for the view scale = 100%. If the row height is automatic, the "auto" string is returned.

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var height = wsheet.GetRowHeigh();
```

**SetRowHeight(height)**

Sets the height for the rows included in the currently selected range. The **height** argument is a text string that can represent either a width in pixels related to the view scale = 100% or the "auto" text string. The latter turns on automatic heights (that is, heights adjusted to the contained data).

Example:

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
wsheet.selectedRange = "b5:e5";
wsheet.SetRowHeight("auto");
wsheet.selectedRange = "f5";
wsheet.SetRowHeight("30");
```

**InsertColumns(type)**

Inserts new columns based on the currently selected range. The **argument** is a numeric value that specifies the following:

- 1 - inserts a blank column(s) before the selection
- 2 - inserts a column(s) before the selection with the same formatting
- 3 - inserts a column(s) before the selection with the same formatting and size
- 4 - inserts a column(s) before the selection with the same formatting, size and unmodified formulas
- 5 - inserts a blank column(s) after the selection
- 6 - inserts a column(s) after the selection with the same formatting
- 7 - inserts a column(s) after the selection with the same formatting and size
- 8 - inserts a column(s) after the selection with the same formatting, size and unmodified formulas

Example:

```
  var wbook = GSCalc.ThisWorkbook();
  var wsheet = wbook.GetActiveWorksheet();
  wsheet.selectedRange = "b5:e5";
  wsheet.InsertColumns(1);
  wsheet.selectedRange = "f5";
  wsheet.InsertColumns(3);
  wbook.WaitForUpdate(); // to allow the background updating to complete
before further actions
  // ...
```

**DeleteColumns()**

Deletes columns based on the currently selected range.

Example:

```
  var wbook = GSCalc.ThisWorkbook();
  var wsheet = wbook.GetActiveWorksheet();
  wsheet.selectedRange = "b5:e5";
  wsheet.DeleteColumns();
  wsheet.selectedRange = "f5";
  wsheet.DeleteColumns();
```

**InsertRows(type)**

Inserts new rows based on the currently selected range. The **argument** is a numeric value that specifies the following:

- 1 - inserts a blank row(s) before the selection
- 2 - inserts a row(s) before the selection with the same formatting
- 3 - inserts a row(s) before the selection with the same formatting and size

- 4 - inserts a row(s) before the selection with the same formatting, size and unmodified formulas
- 5 - inserts a blank row(s) after the selection
- 6 - inserts a row(s) after the selection with the same formatting
- 7 - inserts a row(s) after the selection with the same formatting and size
- 8 - inserts a row(s) after the selection with the same formatting, size and unmodified formulas

Example:

```
  var wbook = GSCalc.ThisWorkbook();
  var wsheet = wbook.GetActiveWorksheet();
  wsheet.selectedRange = "b5:b10";
  wsheet.InsertRows(1);
  wsheet.selectedRange = "b11";
  wsheet.InsertRows(3);
  wbook.WaitForUpdate(); // to allow the background updating to complete
before further actions
  // ...
```

**DeleteRows()**

Deletes rows based on the currently selected range.

Example:

```
  var wbook = GSCalc.ThisWorkbook();
  var wsheet = wbook.GetActiveWorksheet();
  wsheet.selectedRange = "b5:b10";
  wsheet.DeleteRows(1);
  wsheet.selectedRange = "b11";
  wsheet.DeleteRows(3);
```

**InsertSeries()**

Insert a new data series based on the currently selected range. For the detailed description of this function, please see the "Entering data > Inserting series" help topic.

Example:

```
  var wbook = GSCalc.ThisWorkbook();
  var wsheet = wbook.GetActiveWorksheet();
  wsheet.selectedRange = "b5:b100";
  wsheet.InsertSeries();
  wbook.WaitForUpdate(); // to allow the background updating to complete
before further actions
  // ...
```

**IsText(cell)**

**Returns** true is the specified cell contains a text/label.
If cell is a cell range, the method returns true if that range contains at least one text string and zero or more empty cells.
Otherwise the false value is returned.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var check1 = wsheet.IsText("b5:b100");
var check2 = wsheet.IsText("b101");
```

**IsNumber(cell)**

Returns **true** is the specified cell contains a number.
If cell is a cell range, the method returns true if that range contains at least one number and zero or more empty cells.
Otherwise the **false** value is returned.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var check1 = wsheet.IsNumber("b5:b100");
var check2 = wsheet.IsNumber("b101");
```

**IsFormula(cell)**

Returns **true** is the specified cell contains a formula. If cell is a cell range, the method returns true if that range contains at least one formula and zero or more empty cells.
Otherwise the **false** value is returned.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var check1 = wsheet.IsFormula("b5:b100");
var check2 = wsheet.IsFormula("b101");
```

**IsError(cell)**

Returns the first found error code for the specified cell or range or 0 if the cell or range doesn't contain any cells/formulas returning errors. The possible error codes are as follows:

- 1 - #DIV/0!
- 2 - #NAME?
- 3 - #NULL!
- 4 - #N/A!
- 5 - #NUM!
- 6 - #REF!
- 7 - #VALUE!
- 8 - #MEMORY!
- 9 - #FILL!
- 12 - #SYNTAX!

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var check1 = wsheet.IsError("b5:b100");
var check2 = wsheet.IsError("b101");
```

**InsertData(cell, data, parse)**

Inserts data into the specified cell or range. The data argument can be any text string containing up to 1024 characters. The parse argument is a logical value that specifies whether the submitted data should be parsed automatically and inserted as a number, date, a text string or a formula. If it's set to false, the entered data will be always treated as text - same as pressing Enter vs Enter+Shift when entering cell contents.
This function fully simulates manual data entering.

When entering large data sets, it's typically much more faster to use the Copy/Paste and InsertSeries methods and their variants.
Using InsertData() method will be also much faster if the **updateMode** property is set to **manual** and the application/workbook window is minimized as show in the 2nd example below.

**Example 1.:**

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
wsheet.InsertData("b5", "11.5", true);
wsheet.InsertData("b6", "1.7", true);
wsheet.InsertData("b7:b10", "-0.1", true);
wsheet.InsertData("b11", "=sum(b5:b10)", true);
wbook.WaitForUpdate(); // to allow the background updating to complete
before further actions
// ...
```

**Example 2.:**

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
GSCalc.MinimizeAppWindow();
var updateMode = wbook.updateMode;
wbook.updateMode = "manual";
var i;
for ( i = 1; i < 200000; ++i )
{
        var ref = "b" + i;
        wsheet.InsertData(ref, "some text data", false);
}
wbook.updateMode = updateMode;
GSCalc.RestoreAppWindow();
```

**InsertComments(cell, comments)**

Adds comments to specified cell. The comments argument can be any text string containing up to 1024 characters.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
wsheet.InsertComments("b5", "comments in b5");
wsheet.InsertComments("b6", "comments in b6");
```

**GetData(cell)**

Returns data from the specified cell. The returned data can represent:

- a numeric or textual cell value
- a numeric or textual result of a formula
- a text string representing an error code
- an undefined value for an empty cell

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
wsheet.InsertData("b5", "11.5", true);
wsheet.InsertData("b6", "1.7", true);
wsheet.InsertData("b7:b10", "-0.1", true);
wsheet.InsertData("b11", "=sum(b5:b10)", true);
var sum = wsheet.GetData("b5:b10"); // sum = 12.8
```

**GetFormula(cell)**

Returns formula from the specified cell that contains some formula.
If the specify cell doesn't contain a formula, an exception is thrown.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
wsheet.InsertData("b5", "11.5", true);
wsheet.InsertData("b6", "1.7", true);
wsheet.InsertData("b7:b10", "-0.1", true);
wsheet.InsertData("b11", "=sum(b5:b10)", true);
var formula = wsheet.GetFormula("b11"); // formula = "sum(b5:b10)"
```

**GetComments(cell)**

Returns comments for the specified cell. If the cell doesn't have comments, an empty string is returned.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
wsheet.InsertComments("b5", "comments in b5");
var comments = wsheet.GetComments("b5"); // comments = "comments in
b5"
```

**UpdateWindow()**

UpdateWindow enforces GS-Calc to perform any pending screen updates of the workbook window before the script terminates and passes the control back to it.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
wsheet.topLeftCell = "cv100";
wsheet.UpdateWindow();
GSCalc.Sleep(3000);
wsheet.selectedRange = "ab10";
wsheet.UpdateWindow();
```

**SaveSelectionAsImage(file)**

Saves the currently selected chart or image (if an object is selected) or the current cell range (if no object is selected) as an image to the specified file. The image format is determined by the file extension: *.png, *.jpg, *.gif, *.bmp, *.tiff.
If the **file** argument is an empty string, the standard **Save File** dialog box is displayed.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
wsheet.selectedRange = "b5:b10";
wsheet.SaveSelectionAsImage("d:\\image1.png");
```

**CreatePrintSettings()**

Creates and returns an object containing page/print settings. The **PrintSettings** object is then used to retrieve existing worksheets print settings and to set new ones.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
printSettings.printHeadings = false;
wsheet.Print(false);
```

**GetPrintSettings(printSettings)**

Returns an object containing page/print settings. The **PrintSettings** object is then used to retrieve existing worksheet print settings and to set new ones.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
printSettings.printHeadings = false;
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**SetPrintSettings(printSettings)**

Sets the print settings for the active worksheet.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
printSettings.printHeadings = false;
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

**Print(showPrintDialog)**

Prints the active worksheet.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var wsheet = wbook.GetActiveWorksheet();
var printSettings = wsheet.CreatePrintSettings();
printSettings.printHeadings = false;
printSettings.printedData = "range";
printSettings.printedPages = "1,3,5-8";
wsheet.SetPrintSettings(printSettings);
wsheet.Print(false);
```

---

## Workbook

### GetActiveWorksheet()

Returns the active worksheet object. All methods called on this object always refers to the currently selected worksheet.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.SelectTreeItem("sheet1");
var wsheet = wbook.GetActiveWorksheet();
wsheet.InsertData("b5", "data entered in sheet1!b5", true);
wbook.SelectTreeItem("sheet2");
wsheet.InsertData("b5", "data entered in sheet2!b5", true);
```

### GetWorksheetCount()

Returns the number of worksheets in the workbook.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var counter = wbook.GetWorksheetCount();
```

### GetFolderCount()

Returns the number of folders in the workbook.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var counter = wbook.GetFolderCount();
```

**IsFolderEmpty(folder)**

Returns **true** if the specified folder doesn't contain any tree items (worksheets or folders) and **false** otherwise.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var empty = wbook.IsFolderEmpty("folder1");
```

**GetFirstTreeItem(folder)**

Returns the first tree item (worksheets or folders) of the specified folder. If the **folder** argument is an empty string, the first item of the entire worksheet tree is returned. The returned value has a form of the full tree tree path.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var item1 = wbook.GetFirstTreeItem("");
if ( wbook.IsFolder(item1) )
{
        var item2 = wbook.GetFirstTreeItem(item1);
        // ...
}
```

**GetPrevTreeItem(treeItem)**

Returns the previous sibling tree item (a worksheet or a folder) for the specified worksheet tree item.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var item1 = wbook.GetFirstTreeItem("");
var item2 = wbook.GetNextTreeItem(item1);
var item3 = wbook.GetPrevTreeItem(item2);
// item1 == item3 if there are at least two sibling items in the main
folder of the worksheet tree.
var item1 = wbook.GetFirstTreeItem("folder1\\subfolder2");
var item2 = wbook.GetNextTreeItem(item1);
var item3 = wbook.GetPrevTreeItem(item2);
// item1 == item3 if there are at least two sibling items in the
"subfolder2" folder.
```

**GetNextTreeItem(treeItem)**

Returns the next sibling tree item (a worksheet or a folder) for the specified worksheet tree item.

**Example:**

```
  var wbook = GSCalc.ThisWorkbook();
  var item1 = wbook.GetFirstTreeItem("");
  var item2 = wbook.GetNextTreeItem(item1);
  var item3 = wbook.GetPrevTreeItem(item2);
  // item1 == item3 if there are at least two sibling items in the main
folder of the worksheet tree.
  var item1 = wbook.GetFirstTreeItem("folder1\\subfolder2");
  var item2 = wbook.GetNextTreeItem(item1);
  var item3 = wbook.GetPrevTreeItem(item2);
  // item1 == item3 if there are at least two sibling items in the
"subfolder2" folder.
```

**GetParentFolder(treeItem)**

Returns the full path of the folder that contains the specified worksheet tree item (either a folder or a worksheet). If **treeItem** is top/root element, an empty string is returned.

**Example:**

```
  var wbook = GSCalc.ThisWorkbook();
  var item = wbook.GetFirstTreeItem("folder1\\subfolder2");
  var parent = wbook.GetParentFolder(item);
  // parent == "folder1\subfolder2"
```

**IsWorksheet(treeItem)**

Returns the logical **true** value id the specified worksheet tree item is a worksheet. Otherwise **false** is returned.

**Example:**

```
  var wbook = GSCalc.ThisWorkbook();
  var check1 = wbook.IsWorksheet("folder1\\sheet10");
  var check2 = wbook.IsWorksheet("abc");
```

**IsFolder(treeItem)**

Returns the logical **true** value id the specified worksheet tree item is a folder. Otherwise **false** is returned.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var check1 = wbook.IsFolder("folder1\\sheet10");
var check2 = wbook.IsFolder("abc");
```

**ExpandFolder(folder)**

Expand the specified folder in the worksheet tree window.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.ExpandFolder("folder1");
```

**CollapseFolder(folder)**

Collapse the specified folder in the worksheet tree window.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.ColapseFolder("folder1");
```

**SelectTreeItem(path)**

Changes selection in the worksheet tree window. The new selected tree item is specified by the full **path** argument. If it points to a worksheet, the current active worksheet is changed automatically as well.
If **path** is an empty string, the top/root folder is selected.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.SelectTreeItem("sheet1");
var wsheet = wbook.GetActiveWorksheet();
wsheet.InsertData("b5", "data entered in sheet1!b5", true);
wbook.SelectTreeItem("sheet2");
wsheet.InsertData("b5", "data entered in sheet2!b5", true);
```

**InsertFolder(name)**

Inserts a new folder into the current worksheet tree folder. The **name** argument must not include tree path elements.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.SelectTreeItem("folder2");
wbook.InsertFolder("nested-folder3");
```

### InsertWorksheet(name)

Inserts a new worksheet into the current worksheet tree folder. The **name** argument must not include tree path elements.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.SelectTreeItem("");
wbook.InsertFolder("sheet1");
```

### DeleteTreeItem()

Deletes the currently selected worksheet tree item (which can be a single worksheet or a folder containing other folders and worksheets).

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.SelectTreeItem("folder4");
wbook.DeleteTreeItem();
wbook.SelectTreeItem("folder5\\sheet1");
wbook.DeleteTreeItem();
```

### MoveTreeItem(sourcePath, targetPath)

Moves a worksheet tree item performing the equivalent of the drag-and-drop operation. If the **targetPath** specifies a worksheet, the moved item(s) are insert before it in the same folder.
If the **targetPath** specifies a folder, the moved item(s) are inserted as the last item(s) in that folder. An empty **targetPath** specifies the top/root folder.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.MoveTreeItem("folder1\\sheet11", "folder2");
wbook.MoveTreeItem("folder1\\sheet10", "");
```

### RenameTreeItem(name)

Changes the name of the currently selected worksheet tree item (except the top/root item).

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.SelectTreeItem("folder4");
wbook.RenameTreeItem("folder_4");
wbook.SelectTreeItem("pivot tables\\customers")
wbook.RenameTreeItem("new customers");
```

### GetNamedRangeCount()

Returns the number of defined named ranges.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.GetNamedRangeCount();
```

### GetNamedRange(index, propertyName)

Returns the name and cell range of the named expression specified by **index**. The type of the returned string is determined by the **propertyName** argumement and can be "name" or "range".

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var name = wbook.GetNamedRange(2, "name");
var cell = wbook.GetNamedRange(2, "range");
```

### SetNamedRange(index, rangeName, range)

Set the existing named range specified by **index**.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
if ( wbook.GetNamedRangeCount() >= 3 )
{
        wbook.SetNamedRange(1, "start", "a10");
        wbook.SetNamedRange(2, "name", sheet1!c4);
        wbook.SetNamedRange(3, "sales",
"\"d:\\[sample.gsc]Folder1\"!$A$1:$E$5");
}
```

### AddNamedRange(rangeName, range)

Adds a new named range specified by **index.**

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.AddNamedRange("start", "a10");
wbook.AddNamedRange("name", sheet1!c4);
wbook.AddNamedRange("sales", "\"d:\\[sample.gsc]Folder1\"!$A$1:$E$5");
```

**RemoveNamedRange(index)**

Removes an existing named range specified by **index**.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
while ( wbook.GetNamedRangeCount() >= 1 )
{
        wbook.RemoveNamedRange(1);
}
```

**RemoveAllNamedRanges()**

Removes all defined named ranges.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.RemoveAllNamedRanges();
```

**Save()**

Saves the workbook using the current file format and file path. If this is a new workbook, the **Save As** dialog box is displayed.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.Save();
```

SaveAs(path)

Saves the current workbook to a new file. If path is an empty string, Save As dialog box is displayed.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.SaveAs("d:\\wbook1.gsc");
```

**SaveAsPDFFile(path, saveAllWorksheets)**

Saves the current workbook to the specified PDF file. If the logical **saveAllWorksheets** value is true, all worksheets are saved and the tree structure is preserved in that PDF file.
If **saveAllWorksheets** is false, only the current/active worksheet is saved.
When saving to PDFs the current worksheets print/page settings determine the layout of PDF pages.
The method doesn't change the file path information or the modification state of the original workbook.
If **path** is an empty string, the **Save As** dialog box is displayed.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.SaveAsPDFFile("d:\\copy_of_wbook1.pdf");
```

**SaveAsExcelFile(path)**

**Saves the current workbook as a new Excel XML file.**
**If path is an empty string, the Save As dialog box is displayed.**

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.SaveAsExcelFile("d:\\wbook1.xml");
```

**SaveAsTextFile(path, showDialogBox, textParams)**

Saves the current workbook as a new text file using the provided TextParams object.
If **showDialogBox** is **true**, the **Save Text File** dialog box is displayed.
If **path** is an empty string, the **Save As** dialog box is displayed.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var textParams = GSCalc.CreateTextParams();
textParams.separator = "\t"; // tab-separeted values
textParams.quotingSymbol = "\"";
textParams.encoding = "utf8";
wbook.SaveAsTextFile("d:\\wbook1.txt", false, textParams);
wbook.SaveAsTextFile("d:\\wbook2.txt", true, textParams);
```

**SaveAsXBaseFile(path, showDialogBox, xBaseParams)**

Saves the current workbook as a new xBase file using the provided XBaseParams object. If **showDialogBox** is **true**, the **Save xBase File** dialog box is displayed and it contains a suggested set of fields based on the data the worksheet contains.
If **path** is an empty string, the **Save As** dialog box is displayed.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var xBaseParams = GSCalc.CreateXBaseParams();
xBaseParams.format = "dbaseIV";
xBaseParams.AddField("item", "C", 40, 0);
xBaseParams.AddField("price", "N", 5, 2);
xBaseParams.encoding = "windows";
wbook.SaveAsXBaseFile("d:\\wbook1.dbf", false, xBaseParams);
wbook.SaveAsXBaseFile("d:\\wbook2.dbf", true, xBaseParams);
```

**Reload()**

**Reloads the workbook. All changes are discarded.**

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.Reload();
```

**Close()**

Closes the workbook. If the **modified** property is **true**, users are prompted to save it. Trying use a workbook object after it's closed results in throwing the E_POINTER exception.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.Close();
try
{
        var counter = wbook.GetWorksheetCount();
}
catch(e)
{
        GSCalc.MessageBox(e, "ok", 1, "error");
}
```

**GetFileInfo(path, type, value)**

**// 1 - size, 2 - mod. dateReleaseFile**

**modified.**

**Example:**

```
  // file size in bytes
  var size = GSCalc.GetFileInfo("c:\\file01.gsc", 1);
  // last modification date in the format YYYY-MM-DD (YYYY-MM-
DDTHH:MM:SS)
  var modificationData = GSCalc.GetFileInfo("c:\\file01.gsc", 2);
```

**ReleaseFile()**

Detaches and closes the currently open workbook file. The workbook and its window remain open and you keep on editing it allowing other programs to access that file at the same time.
If you try to save it without calling the AttachFile() function first, the Save As dialog box will be displayed.

**Example:**

```
  var wbook = GSCalc.ThisWorkbook();
  var path = wbook.ReleaseFile();
```

**AttachFile(path)**

Attaches previously detached file and open it. Returns 1 if a given file was modified after ReleaseFile(), 0 if not and -1 if the file can't be opened and attached

**Example:**

```
  var wbook = GSCalc.ThisWorkbook();
  wbook.AttachFile("c:\\text_file01.csv");
```

**MergeRows(mergeParams)**

Merges/adds rows from other GS-Calc *.gsc files and tables to the current worksheet. The mergeParams parameters are created using the CreateMergeParams() method.
For all parameters see the MergeParams description.

**Example:**

```
  var wbook = GSCalc.ThisWorkbook();
  var merge = GSCalc.CreateMergeParams();
```

```
  // merge records from all *.gsc files with names starting with
"gsc_file"
  merge.path = "c:\\gsc_file*.gsc";
  merge.table = "";
  wbook.MergeRows(mergeParams)
```

**MergeRowsFromODSFile(mergeParams)**

Merges/adds rows from other *.ods files and tables to the current worksheet. The mergeParams parameters are created using the CreateMergeParams() method.
For all parameters see the MergeParams description.

**Example:**

```
  var wbook = GSCalc.ThisWorkbook();
  var merge = GSCalc.CreateMergeParams();
  // merge records from all *.ods files with names starting with
"ods_file"
  merge.path = "c:\\ods_file*.ods";
  merge.table = "";
  wbook.MergeRowsFromODSFile(mergeParams)
```

**MergeRowsFromTextFile(mergeParams, textParams)**

Merges/adds rows from other *.txt files to the current worksheet. The mergeParams parameters are created using the CreateMergeParams() method and textParams - with the CreateTextParams() method.
For all parameters see the MergeParams and TextParams descriptions.

**Example:**

```
  var wbook = GSCalc.ThisWorkbook();
  var textParams = GSCalc.CreateTextParams();
  textParams.separator = ",";
  var mergeParams = GSCalc.CreateMergeParams();
  // merge records from all *.csv files with names starting with
"csv_file"
  merge.path = "c:\\csv_file*.csv";
  merge.table = "";
  wbook.MergeRowsFromTextFile(mergeParams, textParams)
```

**MergeRowsFromExcelFile(mergeParams)**

Merges/adds rows from other *.xlsx files and tables to the current worksheet. The mergeParams parameters are created using the CreateMergeParams() method.
For all parameters see the MergeParams description.

**Example:**

```
  var wbook = GSCalc.ThisWorkbook();
  var merge = GSCalc.CreateMergeParams();
  // merge records from all *.xlsx files with names starting with
"xlsx_file"
  merge.path = "c:\\xlsx_file*.xlsx";
  merge.table = "";
  wbook.MergeRowsFromExcelFile(mergeParams)
```

**MergeRowsFromXBaseFile(mergeParams, xBaseParams)**

Merges/adds rows from other dBaseIV *.dbf files and tables to the current worksheet. The mergeParams parameters are created using the CreateMergeParams() method and xBaseParams - CreateXBaseParams.
For all parameters see the MergeParams description.

**Example:**

```
  var wbook = GSCalc.ThisWorkbook();
  var merge = GSCalc.CreateMergeParams();
  // merge records from all *.dbf files with names starting with
"dbf_file"
  merge.path = "c:\\dbf_file*.dbf";
  merge.table = "";
  var xbaseParams = GSCalc.CreateXBaseParams();
  xbaseParams.encoding = "windows";
  wbook.MergeRowsFromXBaseFile(mergeParams)
```

**MergeTable(path, table)**

Merges/adds tables from other GS-Calc *.gsc files to the current workbook and currently selected folder.
"Path" specifies a file(s) with tables to merge. The path can contain a file name with wildcard (*, ?) characters, enabling you to merge tables from multiple files from a given folder or all files from that folder.
If no table name is specified (table="" / NULL), then the default/current table from a given file is added.

**Example:**

```
  var wbook = GSCalc.ThisWorkbook();
  // merge tables from all *.gsc files with names starting with
"gsc_file"
  wbook.MergeTable("c:\\gsc_file*.gsc", "customers");
```

**MergeTableFromODSFile(path, table)**

Merges/adds tables from other *.ods files to the current workbook and currently selected folder.

"Path" specifies a file(s) with tables to merge. The path can contain a file name with wildcard (*, ?) characters, enabling you to merge tables from multiple files from a given folder or all files from that folder.
If no table name is specified (table="" / NULL), then the default/current table from a given file is added.

**Example:**

```
  var wbook = GSCalc.ThisWorkbook();
  // merge tables from all *.ods files with names starting with
"ods_file"
  wbook.MergeTable("c:\\ods_file*.ods", "customers");
```

**MergeTableFromTextFile(path, table, textParams)**

Merges/adds tables from other text files to the current workbook and currently selected folder.
"Path" specifies a file(s) with tables to merge. The path can contain a file name with wildcard (*, ?) characters, enabling you to merge tables from multiple files from a given folder or all files from that folder.
If no table name is specified (table="" / NULL), then the default/current table from a given file is added.

**Example:**

```
  var textParams = GSCalc.CreateTextParams();
  textParams.separator = ",";
  var wbook = GSCalc.ThisWorkbook();
  // merge tables from all *.csv files with names starting with
"csv_file"
  wbook.MergeTable("c:\\csv_file*.csv", "", textParams);
```

**MergeTableFromExcelFile(path, table)**

Merges/adds tables from other *.xlsx files to the current workbook and currently selected folder.
"Path" specifies a file(s) with tables to merge. The path can contain a file name with wildcard (*, ?) characters, enabling you to merge tables from multiple files from a given folder or all files from that folder.
If no table name is specified (table="" / NULL), then the default/current table from a given file is added.

**Example:**

```
  var wbook = GSCalc.ThisWorkbook();
  // merge default tables from all *.xlsx files with names starting with
"xlsx_file"
  wbook.MergeTable("c:\\xlsx_file*.xlsx", "");
```

**MergeTableFromXBaseFile(path, table, xBaseParams)**

Merges/adds tables from other dBaseIV *.dbf files to the current workbook and currently selected folder.
"Path" specifies a file(s) with tables to merge. The path can contain a file name with wildcard (*, ?) characters, enabling you to merge tables from multiple files from a given folder or all files from that folder.
If no table name is specified (table="" / NULL), then the default/current table from a given file is added

**Example:**

```
  var xBaseParams = GSCalc.CreateXBaseParams();
  xBaseParams.encoding = "windows";
  var wbook = GSCalc.ThisWorkbook();
  // merge tables from all *.dbf files with names starting with
"dbf_file"
  wbook.MergeTable("c:\\dbf_file*.dbf", "");
```

**SetFilePassword(enable, method, oldPassword, newPassword)**

Sets, modifies or removes password protection for the workbook.
If the logical **enable** argument is **true**, the password protection is added, if it's **false**, the protection is removed.
The **method** argument can currently be only the **"blowfish"** encryption method name.
If the workbook is already password protected, the **oldPassword** argument must contain the existing password.
If **enable** is **true**, the **newPassword** argument argument must contain a new password consisting of at least 6 characters.

**Example:**

```
  var wbook = GSCalc.ThisWorkbook();
  wbook.SetFilePassword(true, "blowfish", "Df8ER1", "97uYwp");
  var wbook = GSCalc.NewWorkbook();
  wbook.SetPassword(true, "blowfish", "", "97uYwp");
```

**SetCellPassword(enable, oldPassword, newPassword)**

Sets, modifies or removes password protection for individual cells.
Once this password is set, the **Format > Protect** command can be used to set the cell protection.

If the logical **enable** argument is **true**, the password protection is added, if it's **false**, the protection is removed.
If **enable** is **true**, the **newPassword** argument must contain a new password consisting of at least 6 characters.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.SetCellPassword(true, "Df8ER1", "97uYwp");
```

**SetTreePassword(enable, oldPassword, newPassword)**

Sets, modifies or removes password protection for the worksheet tree structure and view splitters.
If the logical **enable** argument is **true**, the password protection is added, if it's **false**, the protection is removed.
If **enable** is **true**, the **newPassword** argument must contain a new password consisting of at least 6 characters.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.SetTreePassword(true, "Df8ER1", "97uYwp");
```

**UpdateAllWorksheets()**

Updates all formulas in all worksheets in this workbook. (Same as the **Update All - F9** command.)

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.UpdateAllWorksheets();
wbook.WaitForUpdate(); // to allow the background updating to complete
before further actions
```

**UpdateActiveWorksheet()**

Updates all formulas in the active worksheet. (Same as the **Update Worksheet Shift+F9** command.)

GS-Calc enables you to load/save quickly extremely large files and use multicore calculations for any formulas using dynamic references because it doesn't pre-build calculation trees. However, the disadvantage of this is that if you use the **UpdateAllWorksheets()** method, GS-Calc will be unconditionally updating all formulas in all worksheets, no matter whether the current data change requires it. Setting the **updateMode** to **manual** and using sequences of the **UpdateActiveWorksheet()** and/or **UpdateAllWorksheetRegion()** methods relevant to your worksheets design, you can greatly increase the update speed.

**Example:**

```
  var wbook = GSCalc.ThisWorkbook();
  wbook.UpdateActiveWorksheet();
  wbook.WaitForUpdate(); // to allow the background updating to complete
before further actions
```

**UpdateActiveWorksheetRegion(range)**

Updates all formulas in the specified range in the active worksheet.

GS-Calc enables you to load/save quickly extremely large files and use multicore calculations for any formulas using dynamic references because it doesn't pre-build calculation trees. However, the disadvantage of this is that if you use the **UpdateAllWorksheets()** method, GS-Calc will be unconditionally updating all formulas in all worksheets, no matter whether the current data change requires it. Setting the **updateMode** to **manual** and using sequences of the **UpdateActiveWorksheet()** and/or **UpdateAllWorksheetRegion()** methods relevant to your worksheets design, you can greatly increase the update speed.

**Example:**

```
  var wbook = GSCalc.ThisWorkbook();
  wbook.UpdateActiveWorksheetRegion("a5:b1000000");
  wbook.WaitForUpdate();  // to allow the background updating to
complete before further actions
  wbook.UpdateActiveWorksheetRegion("g5:m1000000");
  wbook.WaitForUpdate();  // to allow the background updating to
complete before further actions
```

**WaitForUpdate()**

If the background updating mode in turned on and there is an updating process running, the method waits till that process completes.
If the background updating mode is turned off, the method returns immediately.
Performing any editing action before the update completes restarts the updating process.
Some actions (like saving) can't be carried out before the pending update completes.

For efficiency reasons it's recommended that you set the **updateMode** property to "manual" before performing a massive data editing/entering action.

**Example:**

```
  var wbook = GSCalc.ThisWorkbook();
  wbook.UpdateAllWorksheets();
  wbook.WaitForUpdate();
```

**MaximizeWindow()**

Maximizes the workbook window.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.MaximizeWindow();
```

### RestoreWindow()

Restores the previous size of the workbook window after it's maximized.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.RestoreWindow();
```

### modified

A logical value specifying whether the contents of the workbook was modified. It's managed automatically by the program. Typically, modifying it explicitly might be necessary only to avoid notifications about closing an unsaved workbook.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.modified = false;
wbook.Close()
```

### updateMode

Specifies whether formulas are updated automatically or manually. The method accepts the following strings as its argument:

- default - application defaults
- manual - manual updating (to update, use GUI commands or "update" script methods)
- automatic - automatic updating (formulas are updated whenever any workbook data changes)

For more details, please see the **Settings** help topic.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.updateMode = "manual";
```

**updateThreads**

Specifies the number of threads/CPUs used when updating workbook formulas:

- default - application defaults
- any number from 1 to 16

For more details, please see the **Settings** help topic.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.updateThreads = 4;
// ...
wbook.updateThreads = "default";
```

**updatePriority**

Specifies the update threads priority:

- default - application defaults
- realTime
- high
- aboveNormal
- normal
- belowNormal
- idle

For more details, please see the **Settings** help topic.

Example:

```
var wbook = GSCalc.ThisWorkbook();
wbook.updatePriority = "high";
// ...
wbook.updatePriority = "default";
```

**updateOptimization**

Specifies the update optimization. This can be either the **default** string or any combination of numeric values listed below.

- default - application defaults
- 1 - optimize for long formulas
- 2 - optimize for short formula chains
- 4 - perform bottom-up updating

For more details, please see the **Settings** help topic.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
wbook.updateOptimization = 3;
// ...
wbook.updateOptimization = "default";
```

---

## Application

### CreateXBaseParams()

Creates and returns an object containing xBase (dBase, Clipper, FoxPro) file format details. The **XBaseParams** object is then used to retrieve header/fields information of an existing opened database file or to create header/fields structure for a new database file.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var xBaseParams = GSCalc.CreateXBaseParams();
xBaseParams.format = "dbaseIV";
xBaseParams.AddField("item", "C", 40, 0);
xBaseParams.AddField("price", "N", 5, 2);
xBaseParams.encoding = "windows";
wbook.SaveAsXBaseFile("d:\\wbook1.dbf", false, xBaseParams);
wbook.SaveAsXBaseFile("d:\\wbook2.dbf", true, xBaseParams);
```

### CreateTextParams()

Creates and returns an object containing text file format details. The **TextParams** object is then used to open existing text files and to create new **ones.**

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
var textParams = GSCalc.CreateTextParams();
textParams.separator = "\t"; // tab-separeted values
textParams.quotingSymbol = "\"";
textParams.encoding = "utf8";
wbook.SaveAsTextFile("d:\\wbook1.txt", false, textParams);
wbook.SaveAsTextFile("d:\\wbook2.txt", true, textParams);
```

### CreateMergeParams()

A set of parameters used by rows-merging functions.

- path

Specifies a file(s) with records to merge. The path can contain a file name with wildcard (*, ?) characters, enabling you to merge certain files from a given folder or all files from that folder.

- table

  A string specifying the table with records to merge (for file formats capable of storing multiple tables).
  If it's null/empty string, the default/active table is used.

- fieldNames

  Specifies whether merged tables contain field/column names in the 1st row (that should be excluded from merging). 1/0, true/false
  Default: 0

- **matchFieldNames**

  A logical value specifying whether cells/fields from the merged table should be added to these columns where names in the first row in both tables match.
  Default: 0

- **enableUndo**

  A logical value specifying whether the Undo action should be possible.
  For mass rows merging it pays off to turn this option off to save memory.
  Default: 0

**Example:**

```
var mergeParams = GSCalc.CreateMergeParams();
mergeParams.path = "e:\\data_files\\*.gsc";
mergeParams.table = "customers";
mergeParams.table = "pivot table reports\\data";
mergeParams.fieldNames = 1;
mergeParams.matchFieldNames = 0;
mergeParams.enableUndo = 0;
```

**ThisWorkbook()**

Returns the **Workbook** object corresponding to the workbook that contains the executed script or (if the script is not a part of the workbook contents) to the current workbook.

**Example:**

```
var wbook1 = GSCalc.ThisWorkbook();
var wbook2 = GSCalc.NewWorkbook();
```

```
   var wbook3 = GSCalc.OpenWorkbook("d:\\sales2013\\june.gsc",
"");
```

**NewWorkbook()**

Creates a new workbook and returns the corresponding **Workbook** object.

**Example:**

```
   var wbook1 = GSCalc.ThisWorkbook();
   var wbook2 = GSCalc.NewWorkbook();
   var wbook3 = GSCalc.OpenWorkbook("d:\\sales2013\\june.gsc",
"");
```

**OpenWorkbook(path, password)**

Opens an existing workbook in the *.gsc or *.odf file format and returns the corresponding **Workbook** object.
If **path** is an empty string, the **Open File** dialog box is displayed.

**Example:**

```
   var wbook1 = GSCalc.ThisWorkbook();
   var wbook2 = GSCalc.NewWorkbook();
   var wbook3 = GSCalc.OpenWorkbook("d:\\sales2013\\june.gsc",
"");
   var wbook4 = GSCalc.OpenWorkbook("d:\\sales2013\\june.ods",
"FE2OkP");
```

**OpenTextFile(path, showDialogBox, textParams)**

Opens an existing text file and returns the corresponding **Workbook** object.
If **showDialogBox** is **true**, the **Open Text File** dialog box is displayed.
If **path** is an empty string, the **Open File** dialog box is displayed.

**Example:**

```
   var textParams = GSCalc.CreateTextParams();
   textParams.separator = "\t"; // tab-separeted values
   textParams.quotingSymbol = "\"";
   textParams.encoding = "utf8";
   var wbook1 = GSCalc.OpenTextFile("d:\\wbook1.txt", false,
textParams);
   var wbook2 = GSCalc.OpenTextFile("d:\\wbook2.txt", true,
textParams);
```

**OpenExcelFile(path)**

Opens an existing workbook in the Excel *.xlsx file format and returns the corresponding **Workbook** object.
If **path** is an empty string, the **Open File** dialog box is displayed.

**Example:**

```
var wbook = GSCalc.OpenExcelFile("d:\\sales2020\\june.xlsx");
```

**OpenXBaseFile(path, showDialogBox, xBaseParams)**

Opens an existing xBase file and returns the corresponding **Workbook** object. If **showDialogBox** is **true**, the **Open xBase File** dialog box is displayed.
If **path** is an empty string, the **Open File** dialog box is displayed.

**Example:**

```
var xBaseParams = GSCalc.CreateXBaseParams();
xBaseParams.format = "dbaseIV";
var wbook1 = GSCalc.OpenXBaseFile("d:\\wbook1.dbf", false,
xBaseParams);
xBaseParams.format = "clipper";
var wbook2 = GSCalc.OpenXBaseFile("d:\\wbook2.dbf", true,
xBaseParams);
```

**GetBookmarkCount()**

Returns the number of bookmarks defined for the current application profile.

**Example:**

```
var counter = GSCalc.GetBookmarkCount();
```

**GetBookmark(index)**

Returns the string representing the bookmark defined at the **index** position in the current application profile.

**Example:**

```
var counter = GSCalc.GetBookmarkCount();
var i;
for ( i = 1; i <= counter; ++i )
{
        var bookmark = GSCalc.GetBookmark(i);
        GSCalc.MessageBox(bookmark, "ok", 1, "information");
}
```

### SetBookmark(index, bookMark)

Modifies an existing bookmark defined at the **index** position in the current application profile.

**Example:**

```
if ( GSCalc.GetBookmarkCount() > 0 )
{
        GSCalc.SetBookmark(1, "AH1");
}
```

### AddBookmark(bookMark)

Adds a new bookmark for the current application profile.

**Example:**

```
GSCalc.RemoveAllBookmarks();
GSCalc.AddBookmark("AH1");
```

### RemoveBookmark(index)

Adds a new bookmark for the current application profile.

**Example:**

```
while ( GSCalc.GetBookmarkCount() > 0 )
{
        GSCalc.RemoveBookmark(1);
}
```

### RemoveAllBookmarks()

Adds a new bookmark for the current application profile.

**Example:**

```
GSCalc.RemoveAllBookmarks();
```

### LoadProfile(path)

Loads profile information from the specified XML file. If **path** is an empty string, the **Open File** dialog box is displayed.

**Example:**

```
GSCalc.LoadProfile("d:\\profiles\\prof1.xml");
```

**SaveProfile(path)**

Save profile information to the specified XML file. If **path** is an empty string, the **Save** dialog box is displayed.

**Example:**

```
GSCalc.SaveProfile("d:\\profiles\\prof1.xml");
```

**GetWorkbookCount()**

Returns the number of open workbooks.

**Example:**

```
var counter = GSCalc.GetWorkbookCount();
```

**CloseAllWorkbooks()**

Closes all open workbooks. Trying use a workbook object after it's closed results in throwing the E_POINTER exception.

**Example:**

```
var wbook = GSCalc.ThisWorkbook();
GSCalc.CloseAllWorkbooks();
try
{
        var counter = wbook.GetWorksheetCount();
}
catch(e)
{
        GSCalc.MessageBox(e, "ok", 1, "error");
}
```

**TileWorkbooks(tileVertically)**

Tiles open workbook windows vertically or horizontally.

**Example:**

```
GSCalc.TileWorkbooks(true);
GSCalc.TileWorkbooks(false);
```

### MaximizeAppWindow()

Maximizes the application window.

**Example:**

```
GSCalc.MaximizeAppWindow();
```

### MinimizeAppWindow()

Minimizes the application window.

**Example:**

```
GSCalc.MinimizeAppWindow();
```

### RestoreAppWindow()

Restores the previous size and position of the application window after it's minimized or maximized.

**Example:**

```
GSCalc.RestoreAppWindow();
```

### { "GetFolder(title, folder)", 1 },

### GetFilePath(saveAs, title, folder, name, ext)

Displays the "File" dialog box and returns selected file path.

- saveAs
  1/0, true/false - if it's 1, the "Save As" type dialog box will be displayed. Otherwise, the "Open" file dialog box, which requires a given file to exist.
- title
  The dialog box title such as "Save As" or "Open File".
- folder
  The initial folder that the dialog box shows. You can also use a full file path for this. For example, "d:\\docs\\files", "e:\\reports\\doc_file\\feb\\file01.xlsx"
- name
  The initial file name displayed in the "File name" field.

- ext
  The initial file extension filter that determines which the initial "Files of type" combo box selection.

  You can simply put here the full file path or just the extension:
  ".gsc",
  ".ods",
  ".txt", ".csv", ".tsv",".tab", ".zip",
  ".xlsx",
  ".xml",
  ".dbf",
  "*.*",

The dialog box returns the full file path. If the file selection is cancelled, an empty string is returned.

**Example:**

```
var openFilePath = GSCalc.GetFilePath(false, "Open XLSX File",
"d:\\", "d:\\newfile.xlsx", ".xlsx");
```

**GetFolder(title, folder)**

Displays the "Folder" dialog box and returns selected folder path.

- title
  The dialog box title such as "Select Folder".
- folder
  The initial folder that the dialog box shows. You can also use a full file path for this. For example, "d:\\docs\\files", "e:\\reports\\doc_file\\feb\\file01.xlsx"

The dialog box returns the full folder path. If the folder selection is cancelled, an empty string is returned.

**Example:**

```
var folderPath = GSCalc.GetFolder("Select Folder", "d:\\");
```

**MessageBox(message, buttons, button, icon)**

*Copyrights* **Example:**

```
var wbook;
if ( GSCalc.MessageBox("Open this file: calc1.gsc", "yes-no",
1, "question") == "yes" )
{
        wbook = GSCalc.OpenWorkbook("d:\\sales2013\\calc1.gsc",
"");
```

```
}
```

**InputBox(title, password, initValue)**

Displays a simple, one-line input box using the following arguments:

- title
  The input field title.
- password
  A logical value specifying whether the edit control should use the password style, masking the entered characters.
- initValue
  The initial string displayed in the edit control.

The method returns the entered text. If a use click the Cancel button, the E_FAIL exception is thrown.

**Example:**

```
var name = GSCalc.InputBox("Enter your name", false, "");
```

**Sleep(time)**

Suspend the execution of the script for the specified **time** in milliseconds.

**Example:**

```
GSCalc.Sleep(5000);
```

**statusBar**

Specifies the text displayed in the 1st pane of status bar.

**Example:**

```
GSCalc.statusBar = "Executing script...";
```

**startFolder**

Specifies the default workbook folder.

**Example:**

```
GSCalc.startFolder = "d:\\users\\kt";
```

## Samples

### Registering scripting interfaces

To be able to use scripting, you must register the scripting interfaces in the Windows registry database. To do this, run GS-Calc as an administrator (e.g. right-click GS-Calc shortcut/icon and on the context menu choose "Run as administrator") and use the **Settings > Register Scripting Interfaces** command.
Similarly, if you no longer need scripting, you can use the **Settings > Unregister Scripting Interfaces** command to remove all registry entries added earlier. Uninstalling GS-Calc will unregister those interfaces as well.

### Creating scripts

You can create your scripts either as global scripts saved in the program settings and available to all databases or you can create scripts stored in a given GS-Calc workbook and available only after you open that workbook.
To create these scripts use the "File > Application Scripts" and "File > Database Scripts" commands.

The "(...) Scripts" dialog box enables you to organize your scripts in subfolders, test them to locate errors, copy/import/export them etc.

Sample "Scripts" screen.

Notes:

- If a global script is executed when there is no open workbook, it should start with one of the application Open(...)/New(..) functions
- If a workbook is already loaded, an executed script, either global or local, automatically refers to that workbook and the active worksheet and using the Open(...) functions with the same workbook path has no effect.
- The backslash occurring in the script text mostly in file path parameters must always be doubled for example:
  f:\\my_folder\\file01.gsc

To add the following sample scripts to your GS-Calc, click **File > Application Scripts**, then in the displayed dialog box click **New > JScript** and copy/paste the script text - it'll be saved automatically in your global program settings.
To execute the added scripts, open that dialog box and click **Run** or assign some Ctrl+Shift+... shortcut to execute it using the keyboard.

- Merging rows from multiple text files from a given folder to the current worksheet
- Merging rows from multiple Excel XLSX files from a given folder to the current worksheet
- Importing tables from multiple text files from a given folder to the current workbook

- Importing tables from multiple Excel XLSX files from a given folder to the current workbook
- Releasing open workbook (gsc/xlsx/txt/csv...) files to allow other programs to update them at the same time
- Attaching a released workbook file (gsc/xlsx/txt/csv...) to check for changes and enable saving that file
- Activating the n-th worksheet in in the current workbook (ignoring subfolders)
- Iterating through the entire tree of subfolders of a given the current workbook
- Editing worksheets
- Adding worksheets and folders
- Opening a text file and saving it to new *.gsc file

---

**Merging rows from multiple text files from a given folder to the current worksheet**

```
var wbook = GSCalc.ThisWorkbook();

var merge = GSCalc.CreateMergeParams();
merge.path = "e:\\my_folder\\some_text_file??.txt";
// or e.g. merge.path = "e:\\my_folder\\some_text_file*.txt";
// or e.g. merge.path = "e:\\my_folder\\some_text_file_01.txt";
// or e.g. merge.path = "e:\\my_folder\\*.csv";
// or e.g. merge.path = "e:\\my_folder\\";
// or e.g. merge.path = GSCalc.GetFolder("Find folder", "e:\\");
// or e.g. merge.path = GSCalc.GetFilePath(false, "Merge from text file",
"e:\\", "d:\\my_folder", ".csv")
merge.table = "";
merge.fieldNames = false;

var text = GSCalc.CreateTextParams();
text.separator = ",";
text.encoding = "utf8";
// or text.encoding = "windows";
text.autoFitColumns = false;

wbook.MergeRowsFromTextFile(merge, text);
```

---

**Merging rows from multiple Excel XLSX files from a given folder to the current worksheet**

```
var wbook = GSCalc.ThisWorkbook();

var merge = GSCalc.CreateMergeParams();
merge.path = "e:\\my_folder\\some_text_file??.xlsx";
// or e.g. merge.path = "e:\\my_folder\\some_text_file*.xlsx";
// or e.g. merge.path = "e:\\my_folder\\some_text_file_01.xlsx";
```

```
// or e.g. merge.path = "e:\\my_folder\\*.xlsx";
// or e.g. merge.path = "e:\\my_folder\\";
// or e.g. merge.path = GSCalc.GetFolder("Find folder", "e:\\");
// or e.g. merge.path = GSCalc.GetFilePath(false, "Merge from text file",
"e:\\", "d:\\my_folder", ".xlsx")

// no table name - merge rows from the default worksheet in each XLSX file
merge.table = "";
merge.fieldNames = false;

wbook.MergeRowsFromExcelFile(merge);
// or e.g. wbook.MergeRowsFromMySQLFile(merge);
// or e.g. wbook.MergeRowsFromFile(merge);
```

---

### Importing tables from multiple text files from a given folder to the current workbook

```
var wbook = GSCalc.ThisWorkbook();

var text = GSCalc.CreateTextParams();
text.separator = ",";
text.encoding = "utf8";
// or text.encoding = "windows";
text.autoFitColumns = false;

wbook.MergeTableFromTextFile("e:\\my_folder\\some_text_file??.txt", "",
text);
// or e.g. wbook.MergeTableFromTextFile("e:\\my_folder\\some_text_file*.txt",
"", text);
// or e.g.
wbook.MergeTableFromTextFile("e:\\my_folder\\some_text_file_01.txt", "",
text);
// or e.g. wbook.MergeTableFromTextFile("e:\\my_folder\\*.csv", "", text);
// or e.g. wbook.MergeTableFromTextFile("e:\\my_folder\\", "", text);
// or e.g.
wbook.MergeTableFromTextFile("e:\\my_folder\\some_text_file??.txt", "",
text);
// or e.g. wbook.MergeTableFromTextFile(GSCalc.GetFolder("Find folder",
"e:\\"), "", text);
// or e.g. wbook.MergeTableFromTextFile(GSCalc.GetFilePath(false, "Merge from
text file", "e:\\", "d:\\my_folder", ".csv"), "", text);
```

---

### Importing tables from multiple Excel XLSX files from a given folder to the current workbook

```
var wbook = GSCalc.ThisWorkbook();

wbook.MergeTableFromExcelFile("e:\\my_folder\\some_text_file??.xlsx", "");
```

```
// or e.g.
wbook.MergeTableFromExcelFile("e:\\my_folder\\some_text_file*.xlsx", "");
// or e.g.
wbook.MergeTableFromExcelFile("e:\\my_folder\\some_text_file_01.xlsx", "");
// or e.g. wbook.MergeTableFromExcelFile("e:\\my_folder\\*.csv", "");
// or e.g. wbook.MergeTableFromExcelFile("e:\\my_folder\\", "");
// or e.g.
wbook.MergeTableFromExcelFile("e:\\my_folder\\some_text_file??.xlsx", "");
// or e.g. wbook.MergeTableFromExcelFile(GSCalc.GetFolder("Find folder",
"e:\\"), "");
// or e.g. wbook.MergeTableFromExcelFile(GSCalc.GetFilePath(false, "Merge
from Excel XLSX file", "e:\\", "d:\\my_folder", ".xlsx"), "");
```

---

### Releasing open workbook (gsc/xlsx/txt/csv...) files to allow other programs to update them at the same time

```
var wbook = GSCalc.ThisWorkbook();

var filePath = wbook.Release();
GSCalc.MessageBox("Closed file: " + filePath, "ok", 1, "information");

// You can keep on editing the workbook as it's already loaded. You have to
use the Attach() method to open/connect it again
// if you want to save it or otherwise the "Save File As" message box will be
displayed.
```

---

### Attaching a released workbook file (gsc/xlsx/txt/csv...) to check for changes and enable saving that file

```
var wbook = GSCalc.ThisWorkbook();

if (wbook.AttachFile("") == 1)
{
   // An empty parameter - file path indicates that we're attaching a file
from the recent ReleaseFile().
   // AttachFile() returns 1 if the file was modified after you used
ReleaseFile().
   // In that case you might want to reload it to see the changes
   wbook.Reload();
}

// After checking and reloading you can release it again
wbook.Release();
```

---

### Activating the n-th worksheetin in the current workbook (ignoring subfolders)

```
var wbook = GSCalc.ThisWorkbook();
```

```
// get the first "child" element in the specified folder - in this case: the
main folder
// if path.length = 0 then there is no more worksheets or subfolders
var path = wbook.GetFirstTreeItem("\\");

// iterate through direct "child" elements of the specified folder (ignoring
nested subfolders)
// to find the 10th worksheet
var counter = 0;

while (path.length)
{
        if (path.charAt(path.length - 1) == '\\')
        {
                // the terminating '\\' means it's a folder
                GSCalc.MessageBox("folder - " + path, "ok", 1, "information");
        }
        else
        {
                // worksheet
                if (++counter == 10)
                {
                        break;
                }
        }
        path = GSBase.GetNextDatabaseItem(path);
}

if (path.length)
{
        // activate the 10th worksheet
        wbook.SetActiveWorksheet(path);
}
```

---

**Iterating through the entire tree of subfolders of a given the current workbook. (max. 10 subfolder nesting levels)**

```
var wbook = GSCalc.ThisWorkbook();

var folders = new Array(10);
var i = 0;

var treeItem = wbook.GetFirstTreeItem("");

do
{
        if ( wbook.IsFolder(treeItem) )
        {
                GSCalc.MessageBox("Folder: " + treeItem, "ok", 1,
"information");
                if ( !wbook.IsFolderEmpty(treeItem) )
                {
                        folders[i++] = treeItem;
```

```
                        treeItem = wbook.GetFirstTreeItem(treeItem);
                        continue;
                }
        }
        else
        if ( wbook.IsWorksheet(treeItem) )
        {
                GSCalc.MessageBox("Worksheet: " + treeItem, "ok", 1,
"information");
        }

        treeItem = wbook.GetNextTreeItem(treeItem);
        if ( treeItem == "" && i > 0 )
        {
                treeItem = wbook.GetNextTreeItem(folders[--i]);
        }
} while ( treeItem );
```

---

### Editing worksheets

```
try
{
        var wbook = GSCalc.OpenWorkbook("d:\\sample.gsc", "");
        var wsheet = wbook.GetActiveWorksheet();

        wbook.SelectTreeItem("formulas");

        // display amount in "b9" as thousands of dollars

        var formatParams = wsheet.CreateFormatParams();
        formatParams.SetCurrencyFormat(0, "$ 1.1", "$", false, false, 1);

        wsheet.selectedRange = "b9";
        wsheet.SetCellFormat(formatParams);

        // if the window is to be updated before the script terminates,
UpdateWindow() must be called

        wsheet.UpdateWindow();

        // insert a new formula in b10

        wsheet.InsertData("b10", "=fv(0.65%, 36, -500, -5500, 0)", true);
        wbook.WaitForUpdate();

        // repeat the formatting

        wsheet.selectedRange = "b10";
        wsheet.SetCellFormat(formatParams);

        // save all as a new encrypted file, specifying no path to display the
"Save" dialog box

        wbook.SetFilePassword(true, "blowfish", "", "dk3zPi");
        wbook.SaveAs("");
```

```
}
catch(e)
{
	GSCalc.MessageBox(e, "ok", 1, "error");
}
```

---

### Adding worksheets and folders

```
try
{
	// delete the "sample4" worksheet from the "2d charts" folder

	var wbook = GSCalc.OpenWorkbook("d:\\sample.gsc", "");
	wbook.SelectTreeItem("2d charts\\sample4");
	wbook.DeleteTreeItem();

	// insert a new "general" folder as the last folder of the tree

	wbook.SelectTreeItem("");
	wbook.InsertFolder("general");

	// move the folder to the beginning of the tree

	wbook.MoveTreeItem("general", wbook.GetFirstTreeItem(""));

	// move the "formulas" and "data types" worksheets to "general"

	wbook.MoveTreeItem("data types", "general");
	wbook.MoveTreeItem("formulas", "general");

	// insert a new worksheet as the last one in "general"

	wbook.SelectTreeItem("general");
	wbook.InsertWorksheet("more formulas");

	wbook.Save();
	wbook.Close();
}
catch(e)
{
	GSCalc.MessageBox(e, "ok", 1, "error");
}
```

---

### Opening a text file and saving it to new *.gsc file

```
var textParams = GSCalc.CreateTextParams();
textParams.separator = ",";
textParams.quotingSymbol = "\"";
textParams.encoding = "utf8";
var wbook = GSCalc.OpenTextFile("d:\\test456.txt", false, textParams);
wbook.SaveAs("d:\\test456.gsc");
```

# COM Programming

## Interfaces and methods

```
import "oaidl.idl";
import "ocidl.idl";
	[
		helpstring("IXBaseParams Interface"),
		pointer_default(unique)
	]
	interface IXBaseParams : IDispatch
	{
		[propget, id(1), helpstring("property format")] HRESULT
format([out, retval] BSTR *pFormat);
		[propput, id(1), helpstring("property format")] HRESULT
format([in] BSTR format);

		[propget, id(2), helpstring("property encoding")] HRESULT
encoding([out, retval] BSTR *pEncoding);
		[propput, id(2), helpstring("property encoding")] HRESULT
encoding([in] BSTR encoding);

		[id(3), helpstring("method GetFieldCount")] HRESULT
GetFieldCount([out, retval] int *pCounter);

		[id(4), helpstring("method SetField")] HRESULT SetField([in]
int index, [in] BSTR name, [in] BSTR type, [in] int length, [in] int
decimals);
		[id(5), helpstring("method AddField")] HRESULT AddField([in]
BSTR name, [in] BSTR type, [in] int length, [in] int decimals);
		[id(6), helpstring("method InsertField")] HRESULT
InsertField([in] int index, [in] BSTR name, [in] BSTR type, [in] int length,
[in] int decimals);

		[id(7), helpstring("method GetFieldName")] HRESULT
GetFieldName([in] int index, [out, retval] BSTR *pName);
		[id(8), helpstring("method GetFieldType")] HRESULT
GetFieldType([in] int index, [out, retval] BSTR *pType);
		[id(9), helpstring("method GetFieldLength")] HRESULT
GetFieldLength([in] int index, [out, retval] int *pLength);
		[id(10), helpstring("method GetFieldDecimals")] HRESULT
GetFieldDecimals([in] int index, [out, retval] int *pDecimals);

		[id(11), helpstring("method DeleteField")] HRESULT
DeleteField([in] int index);
	};

	[
		dual,
```

```
                helpstring("ITextParams Interface"),
                pointer_default(unique)
        ]
        interface ITextParams : IDispatch
        {
                [propget, id(1), helpstring("property separator")] HRESULT
separator([out, retval] BSTR *pSeparator);
                [propput, id(1), helpstring("property separator")] HRESULT
separator([in] BSTR separator);

                [propget, id(2), helpstring("property encoding")] HRESULT
encoding([out, retval] BSTR *pEncoding);
                [propput, id(2), helpstring("property encoding")] HRESULT
encoding([in] BSTR encoding);

                [propget, id(3), helpstring("property quotingSymbol")] HRESULT
quotingSymbol([out, retval] BSTR *pSymbol);
                [propput, id(3), helpstring("property quotingSymbol")] HRESULT
quotingSymbol([in] BSTR symbol);

                [propget, id(4), helpstring("property loadNumbers")] HRESULT
loadNumbers([out, retval] BOOL *pValue);
                [propput, id(4), helpstring("property loadNumbers")] HRESULT
loadNumbers([in] BOOL value);

                [propget, id(5), helpstring("property loadFmtNumbers")]
HRESULT loadFmtNumbers([out, retval] BOOL *pValue);
                [propput, id(5), helpstring("property loadFmtNumbers")]
HRESULT loadFmtNumbers([in] BOOL value);

                [propget, id(6), helpstring("property loadDates")] HRESULT
loadDates([out, retval] BOOL *pValue);
                [propput, id(6), helpstring("property loadDates")] HRESULT
loadDates([in] BOOL value);

                [propget, id(7), helpstring("property loadDateStyles")]
HRESULT loadDateStyles([out, retval] BOOL *pValue);
                [propput, id(7), helpstring("property loadDateStyles")]
HRESULT loadDateStyles([in] BOOL value);

                [propget, id(8), helpstring("property parsingMode")] HRESULT
parsingMode([out, retval] int *pParsing);
                [propput, id(8), helpstring("property parsingMode")] HRESULT
parsingMode([in] int parsing);

                [propget, id(9), helpstring("property columnWidths")] HRESULT
columnWidths([out, retval] BSTR *pColumnWidths);
                [propput, id(9), helpstring("property columnWidths")] HRESULT
columnWidths([in] BSTR columnWidths);

                [propget, id(10), helpstring("property saveFmtNumbers")]
HRESULT saveFmtNumbers([out, retval] BOOL *pValue);
                [propput, id(10), helpstring("property saveFmtNumbers")]
HRESULT saveFmtNumbers([in] BOOL value);

                [propget, id(11), helpstring("property saveFmtDates")] HRESULT
saveFmtDates([out, retval] BOOL *pValue);
```

```
                [propput, id(11), helpstring("property saveFmtDates")] HRESULT
saveFmtDates([in] BOOL value);

                [propget, id(12), helpstring("property saveFormulaValues")]
HRESULT saveFormulaValues([out, retval] BOOL *pValue);
                [propput, id(12), helpstring("property saveFormulaValues")]
HRESULT saveFormulaValues([in] BOOL value);

                [propget, id(15), helpstring("property autoFitColumns")]
HRESULT autoFitColumns([out, retval] BOOL* pValue);
                [propput, id(15), helpstring("property autoFitColumns")]
HRESULT autoFitColumns([in] BOOL value);
        };

        [
                dual,
                helpstring("IMergeParams Interface"),
                pointer_default(unique)
        ]
        interface IMergeParams : IDispatch
        {
                [propput, id(1), helpstring("property path pattern")] HRESULT
path([in] BSTR val);
                [propget, id(1), helpstring("property path pattern")] HRESULT
path([out, retval] BSTR* val);

                [propput, id(2), helpstring("property table name")] HRESULT
table([in] BSTR val);
                [propget, id(2), helpstring("property table name")] HRESULT
table([out, retval] BSTR* val);

                [propput, id(3), helpstring("property master field index")]
HRESULT masterField([in] int field);
                [propget, id(3), helpstring("property master field index")]
HRESULT masterField([out, retval] int* field);

                [propput, id(4), helpstring("property slave field index")]
HRESULT slaveField([in] int field);
                [propget, id(4), helpstring("property slave field index")]
HRESULT slaveField([out, retval] int* field);

                [propput, id(5), helpstring("property merge type")] HRESULT
mergeType([in] int val);
                [propget, id(5), helpstring("property merge type")] HRESULT
mergeType([out, retval] int* val);

                [propput, id(11), helpstring("property namesInTopRow")]
HRESULT fieldNames([in] BOOL val);
                [propget, id(11), helpstring("property namesInTopRow")]
HRESULT fieldNames([out, retval] BOOL* val);

                [propput, id(6), helpstring("property matchFieldNames")]
HRESULT matchFieldNames([in] int val);
                [propget, id(6), helpstring("property matchFieldNames")]
HRESULT matchFieldNames([out, retval] int* val);
```

```
                [propput, id(7), helpstring("property ignore empty fields")]
HRESULT ignoreEmpty([in] BOOL val);
                [propget, id(7), helpstring("property ignore empty fields")]
HRESULT ignoreEmpty([out, retval] BOOL* val);

                [propput, id(8), helpstring("property allow undo")] HRESULT
enableUndo([in] BOOL val);
                [propget, id(8), helpstring("property allow undo")] HRESULT
enableUndo([out, retval] BOOL* val);
        };

        [
                dual,
                helpstring("IFormatParams Interface"),
                pointer_default(unique)
        ]
        interface IFormatParams : IDispatch
        {
                [propget, id(1), helpstring("property dataStyleName")] HRESULT
dataStyleName([out, retval] BSTR *pName);
                [propput, id(1), helpstring("property dataStyleName")] HRESULT
dataStyleName([in] BSTR name);

                [id(2), helpstring("method SetGeneralNumberFormat")] HRESULT
SetGeneralNumberFormat([in] BSTR decimals, [in] BYTE zeroes, [in] BOOL
brackets, [in] BOOL inRed, [in] BOOL separators, [in] int scaling);
                [id(3), helpstring("method SetCurrencyFormat")] HRESULT
SetCurrencyFormat([in] BSTR decimals, [in] BSTR position, [in] BSTR symbol,
[in] BOOL brackets, [in] BOOL inRed, [in] int scaling);
                [id(4), helpstring("method SetAccountingFormat")] HRESULT
SetAccountingFormat([in] BSTR decimals, [in] BSTR symbol, [in] int scaling);
                [id(5), helpstring("method SetDateFormat")] HRESULT
SetDateFormat([in] BSTR pattern, [in] BOOL systemOrder);
                [id(6), helpstring("method SetTimeFormat")] HRESULT
SetTimeFormat([in] BSTR pattern);
                [id(7), helpstring("method SetDateTimeFormat")] HRESULT
SetDateTimeFormat([in] BSTR datePattern, [in] BSTR timePattern, [in] BOOL
systemOrder, [in] BOOL timeFirst);
                [id(8), helpstring("method SetPercentFormat")] HRESULT
SetPercentFormat([in] BSTR decimals, [in] int scaling);
                [id(9), helpstring("method SetFractionFormat")] HRESULT
SetFractionFormat([in] long denominator, [in] BOOL digits);
                [id(10), helpstring("method SetScientificFormat")] HRESULT
SetScientificFormat([in] BSTR decimals, [in] BSTR exponent);

                [propget, id(11), helpstring("property fontName")] HRESULT
fontName([out, retval] BSTR *pVal);
                [propput, id(11), helpstring("property fontName")] HRESULT
fontName([in] BSTR newVal);
                [propget, id(12), helpstring("property fontSize")] HRESULT
fontSize([out, retval] int *pVal);
                [propput, id(12), helpstring("property fontSize")] HRESULT
fontSize([in] int newVal);

                [propget, id(13), helpstring("property language")] HRESULT
language([out, retval] BSTR *pLanguage);
```

```
                [propput, id(13), helpstring("property language")] HRESULT
language([in] BSTR language);

                [propget, id(14), helpstring("property boldFont")] HRESULT
boldFont([out, retval] BOOL *pVal);
                [propput, id(14), helpstring("property boldFont")] HRESULT
boldFont([in] BOOL newVal);
                [propget, id(15), helpstring("property italicFont")] HRESULT
italicFont([out, retval] BOOL *pVal);
                [propput, id(15), helpstring("property italicFont")] HRESULT
italicFont([in] BOOL newVal);
                [propget, id(16), helpstring("property underlineFont")]
HRESULT underlineFont([out, retval] BOOL *pVal);
                [propput, id(16), helpstring("property underlineFont")]
HRESULT underlineFont([in] BOOL newVal);
                [propget, id(17), helpstring("property strikeoutFont")]
HRESULT strikeoutFont([out, retval] BOOL *pVal);
                [propput, id(17), helpstring("property strikeoutFont")]
HRESULT strikeoutFont([in] BOOL newVal);
                [propget, id(18), helpstring("property fontColor")] HRESULT
fontColor([out, retval] BSTR *pColor);
                [propput, id(18), helpstring("property fontColor")] HRESULT
fontColor([in] BSTR color);

                [propget, id(19), helpstring("property horzAlignment")]
HRESULT horzAlignment([out, retval] BSTR *pHorzAlign);
                [propput, id(19), helpstring("property horzAlignment")]
HRESULT horzAlignment([in] BSTR horzAlign);
                [propget, id(20), helpstring("property vertAlignment")]
HRESULT vertAlignment([out, retval] BSTR *pVertAlign);
                [propput, id(20), helpstring("property vertAlignment")]
HRESULT vertAlignment([in] BSTR vertAlign);

                [propget, id(21), helpstring("property horzIndent")] HRESULT
horzIndent([out, retval] int *pIndent);
                [propput, id(21), helpstring("property horzIndent")] HRESULT
horzIndent([in] int indent);
                [propget, id(22), helpstring("property vertIndent")] HRESULT
vertIndent([out, retval] int *pIndent);
                [propput, id(22), helpstring("property vertIndent")] HRESULT
vertIndent([in] int indent);

                [propget, id(23), helpstring("property wrapText")] HRESULT
wrapText([out, retval] BOOL *pValue);
                [propput, id(23), helpstring("property wrapText")] HRESULT
wrapText([in] BOOL value);

                [propget, id(24), helpstring("property shrinkText")] HRESULT
shrinkText([out, retval] BOOL *pValue);
                [propput, id(24), helpstring("property shrinkText")] HRESULT
shrinkText([in] BOOL value);

                [propget, id(25), helpstring("property textRotation")] HRESULT
textRotation([out, retval] int *pRotation);
                [propput, id(25), helpstring("property textRotation")] HRESULT
textRotation([in] int rotation);
```

```
                [propget, id(26), helpstring("property hideFormula")] HRESULT
hideFormula([out, retval] BOOL *pValue);
                [propput, id(26), helpstring("property hideFormula")] HRESULT
hideFormula([in] BOOL value);

                [propget, id(27), helpstring("property protectedCell")]
HRESULT protectedCell([out, retval] BOOL *pValue);
                [propput, id(27), helpstring("property protectedCell")]
HRESULT protectedCell([in] BOOL value);

                [propget, id(28), helpstring("property printedCell")] HRESULT
printedCell([out, retval] BOOL *pValue);
                [propput, id(28), helpstring("property printedCell")] HRESULT
printedCell([in] BOOL value);

                [propget, id(29), helpstring("property borderStyle")] HRESULT
borderStyle([out, retval] BSTR *pStyle);
                [propput, id(29), helpstring("property borderStyle")] HRESULT
borderStyle([in] BSTR style);
                [propget, id(30), helpstring("property borderWidth")] HRESULT
borderWidth([out, retval] int *width);
                [propput, id(30), helpstring("property borderWidth")] HRESULT
borderWidth([in] int width);
                [propget, id(31), helpstring("property borderDoubleLine")]
HRESULT borderDoubleLine([out, retval] BOOL *doubleLine);
                [propput, id(31), helpstring("property borderDoubleLine")]
HRESULT borderDoubleLine([in] int doubleLine);
                [propget, id(32), helpstring("property borderColor")] HRESULT
borderColor([out, retval] BSTR *pColor);
                [propput, id(32), helpstring("property borderColor")] HRESULT
borderColor([in] BSTR color);

                [propget, id(33), helpstring("property borderPosition")]
HRESULT borderPosition([out, retval] int *pPos);
                [propput, id(33), helpstring("property borderPosition")]
HRESULT borderPosition([in] int pos);

                [propget, id(34), helpstring("property bkgColor")] HRESULT
bkgColor([out, retval] BSTR *pColor);
                [propput, id(34), helpstring("property bkgColor")] HRESULT
bkgColor([in] BSTR color);

                [propget, id(35), helpstring("property bkgImageName")] HRESULT
bkgImageName([out, retval] BSTR *pName);
                [propput, id(35), helpstring("property bkgImageName")] HRESULT
bkgImageName([in] BSTR name);

                [propget, id(36), helpstring("property bkgImageRepeat")]
HRESULT bkgImageRepeat([out, retval] BSTR *pRepeat);
                [propput, id(36), helpstring("property bkgImageRepeat")]
HRESULT bkgImageRepeat([in] BSTR repeat);

                [propget, id(37), helpstring("property bkgImageHorzPos")]
HRESULT bkgImageHorzPos([out, retval] BSTR *pHorzPos);
                [propput, id(37), helpstring("property bkgImageHorzPos")]
HRESULT bkgImageHorzPos([in] BSTR horzPos);
```

```
                [propget, id(38), helpstring("property bkgImageVertPos")]
HRESULT bkgImageVertPos([out, retval] BSTR *pVertPos);
                [propput, id(38), helpstring("property bkgImageVertPos")]
HRESULT bkgImageVertPos([in] BSTR vertPos);

                [propget, id(39), helpstring("property bkgImageOpacity")]
HRESULT bkgImageOpacity([out, retval] int *pValue);
                [propput, id(39), helpstring("property bkgImageOpacity")]
HRESULT bkgImageOpacity([in] int value);

                //[id(31), helpstring("method GetState")] HRESULT
GetState([in] BSTR propName, [out, retval] BYTE *pState);
                [id(40), helpstring("method Reset")] HRESULT Reset();
        };

        [
                dual,
                helpstring("IPrintSettings Interface"),
                pointer_default(unique)
        ]
        interface IPrintSettings : IDispatch
        {
                [propget, id(1), helpstring("property printedData")] HRESULT
printedData([out, retval] BSTR *data);
                [propput, id(1), helpstring("property printedData")] HRESULT
printedData([in] BSTR data);

                [propget, id(2), helpstring("property printedPages")] HRESULT
printedPages([out, retval] BSTR *pPages);
                [propput, id(2), helpstring("property printedPages")] HRESULT
printedPages([in] BSTR pages);

                [propget, id(3), helpstring("property orientation")] HRESULT
orientation([out, retval] BSTR *pOrientation);
                [propput, id(3), helpstring("property orientation")] HRESULT
orientation([in] BSTR orientation);

                [propget, id(4), helpstring("property horzPageAlign")] HRESULT
horzPageAlign([out, retval] BSTR *pAlignment);
                [propput, id(4), helpstring("property horzPageAlign")] HRESULT
horzPageAlign([in] BSTR pAlignment);
                [propget, id(5), helpstring("property vertPageAlign")] HRESULT
vertPageAlign([out, retval] BSTR *pAlignment);
                [propput, id(5), helpstring("property vertPageAlign")] HRESULT
vertPageAlign([in] BSTR pAlignment);

                [propget, id(6), helpstring("property marginLeft")] HRESULT
marginLeft([out, retval] double *pMargin);
                [propput, id(6), helpstring("property marginLeft")] HRESULT
marginLeft([in] double margin);
                [propget, id(7), helpstring("property marginTop")] HRESULT
marginTop([out, retval] double *pMargin);
                [propput, id(7), helpstring("property marginTop")] HRESULT
marginTop([in] double margin);
                [propget, id(8), helpstring("property marginRight")] HRESULT
marginRight([out, retval] double *pMargin);
```

```
                [propput, id(8), helpstring("property marginRight")] HRESULT
marginRight([in] double margin);
                [propget, id(9), helpstring("property marginBottom")] HRESULT
marginBottom([out, retval] double *pMargin);
                [propput, id(9), helpstring("property marginBottom")] HRESULT
marginBottom([in] double margin);

                [propget, id(10), helpstring("property headerMargin")] HRESULT
headerMargin([out, retval] double *pMargin);
                [propput, id(10), helpstring("property headerMargin")] HRESULT
headerMargin([in] double margin);
                [propget, id(11), helpstring("property footerMargin")] HRESULT
footerMargin([out, retval] double *pMargin);
                [propput, id(11), helpstring("property footerMargin")] HRESULT
footerMargin([in] double margin);
                [propget, id(12), helpstring("property header")] HRESULT
header([out, retval] BSTR *pText);
                [propput, id(12), helpstring("property header")] HRESULT
header([in] BSTR text);
                [propget, id(13), helpstring("property footer")] HRESULT
footer([out, retval] BSTR *pText);
                [propput, id(13), helpstring("property footer")] HRESULT
footer([in] BSTR text);

                [propget, id(14), helpstring("property scale")] HRESULT
scale([out, retval] int *pScale);
                [propput, id(14), helpstring("property scale")] HRESULT
scale([in] int scale);

                [propget, id(15), helpstring("property autoFit")] HRESULT
autoFit([out, retval] BOOL *pValue);
                [propput, id(15), helpstring("property autoFit")] HRESULT
autoFit([in] BOOL value);

                [propget, id(16), helpstring("property printHeadings")]
HRESULT printHeadings([out, retval] BOOL *pValue);
                [propput, id(16), helpstring("property printHeadings")]
HRESULT printHeadings([in] BOOL value);

                [propget, id(17), helpstring("property printObjects")] HRESULT
printObjects([out, retval] BOOL *pValue);
                [propput, id(17), helpstring("property printObjects")] HRESULT
printObjects([in] BOOL value);

                [propget, id(18), helpstring("property printGrid")] HRESULT
printGrid([out, retval] BOOL *pValue);
                [propput, id(18), helpstring("property printGrid")] HRESULT
printGrid([in] BOOL value);

                [propget, id(19), helpstring("property printFormulas")]
HRESULT printFormulas([out, retval] BOOL *pValue);
                [propput, id(19), helpstring("property printFormulas")]
HRESULT printFormulas([in] BOOL value);

                [propget, id(20), helpstring("property printZeroes")] HRESULT
printZeroes([out, retval] BOOL *pValue);
```

```
                [propput, id(20), helpstring("property printZeroes")] HRESULT
printZeroes([in] BOOL value);

                [propget, id(21), helpstring("property bkgColor")] HRESULT
bkgColor([out, retval] BSTR *pValue);
                [propput, id(21), helpstring("property bkgColor")] HRESULT
bkgColor([in] BSTR value);

                [propget, id(22), helpstring("property bkgImageName")] HRESULT
bkgImageName([out, retval] BSTR *pName);
                [propput, id(22), helpstring("property bkgImageName")] HRESULT
bkgImageName([in] BSTR name);

                [propget, id(23), helpstring("property bkgImageRept")] HRESULT
bkgImageRept([out, retval] BSTR *pRepeat);
                [propput, id(23), helpstring("property bkgImageRept")] HRESULT
bkgImageRept([in] BSTR repeat);

                [propget, id(24), helpstring("property bkgImageHorzPos")]
HRESULT bkgImageHorzPos([out, retval] BSTR *pHorzPos);
                [propput, id(24), helpstring("property bkgImageHorzPos")]
HRESULT bkgImageHorzPos([in] BSTR horzPos);

                [propget, id(25), helpstring("property bkgImageVertPos")]
HRESULT bkgImageVertPos([out, retval] BSTR *pVertPos);
                [propput, id(25), helpstring("property bkgImageVertPos")]
HRESULT bkgImageVertPos([in] BSTR vertPos);

                [propget, id(26), helpstring("property bkgImageOpacity")]
HRESULT bkgImageOpacity([out, retval] int *pValue);
                [propput, id(26), helpstring("property bkgImageOpacity")]
HRESULT bkgImageOpacity([in] int value);
        };

        [
                dual,
                helpstring("IWorksheet Interface"),
                pointer_default(unique)
        ]
        interface IWorksheet : IDispatch
        {
                [propget, id(1), helpstring("property selectedRange")] HRESULT
selectedRange([out, retval] BSTR *pRange);
                [propput, id(1), helpstring("property selectedRange")] HRESULT
selectedRange([in] BSTR range);

                [propget, id(2), helpstring("property topLeftCell")] HRESULT
topLeftCell([out, retval] BSTR *pRange);
                [propput, id(2), helpstring("property topLeftCell")] HRESULT
topLeftCell([in] BSTR range);

                [id(3), helpstring("method Scroll")] HRESULT Scroll([in] BSTR
type, [in] BOOL scrollSelection);

                [id(4), helpstring("method Address")] HRESULT Address([in] int
column, [in] int row, [out, retval] BSTR *pCell);
```

```
                [id(60), helpstring("method AddressRC")] HRESULT
AddressRC([in] __int64 row, [in] unsigned short int column, [out, retval]
BSTR* pCell);
                [id(61), helpstring("method AddressExRC")] HRESULT
AddressExRC([in] __int64 row, [in] BOOL relativeRow, [in] unsigned short int
column, [in] BOOL relativeColumn, [out, retval] BSTR* pCell);

                [id(5), helpstring("method AddressEx")] HRESULT AddressEx([in]
int column, [in] BOOL relativeColumn, [in] int row, [in] BOOL relativeRow,
[out, retval] BSTR *pCell);
                [id(6), helpstring("method Column")] HRESULT Column([in] BSTR
cell, [out, retval] int *pColumn);
                [id(7), helpstring("method Row")] HRESULT Row([in] BSTR cell,
[out, retval] int *pRow);

                [id(8), helpstring("method GetFirstColumn")] HRESULT
GetFirstColumn([out, retval] unsigned short *pColumn);
                [id(9), helpstring("method GetFirstRow")] HRESULT
GetFirstRow([out, retval] unsigned long *pRow);
                [id(10), helpstring("method GetLastColumn")] HRESULT
GetLastColumn([out, retval] unsigned short *pColumn);
                [id(11), helpstring("method GetLastRow")] HRESULT
GetLastRow([out, retval] unsigned long *pRow);
                [id(12), helpstring("method GetMaxColumn")] HRESULT
GetMaxColumn([out, retval] unsigned short *pColumn);
                [id(13), helpstring("method GetMaxRow")] HRESULT
GetMaxRow([out, retval] unsigned long *pRow);

                [id(14), helpstring("method CreateFormatParams")] HRESULT
CreateFormatParams([out, retval] IFormatParams** ppFormatParams);
                [id(15), helpstring("method GetCellFormat")] HRESULT
GetCellFormat([in] IFormatParams* pFormatParams);
                [id(16), helpstring("method GetColumnFormat")] HRESULT
GetColumnFormat([in] IFormatParams* pFormatParams);
                [id(17), helpstring("method GetRowFormat")] HRESULT
GetRowFormat([in] IFormatParams* pFormatParams);

                [id(18), helpstring("method SetCellFormat")] HRESULT
SetCellFormat([in] IFormatParams* pFormatParams);
                [id(19), helpstring("method SetCellCustomStyle")] HRESULT
SetCellCustomStyle([in] BSTR name);

                [id(20), helpstring("method DeleteCells")] HRESULT
DeleteCells([in] BOOL showDialogBox, [in] BSTR contentType);
                [id(21), helpstring("method CopyCells")] HRESULT
CopyCells([in] int copyType);
                [id(22), helpstring("method PasteCells")] HRESULT
PasteCells([in] BOOL textAndFormatting);

                [id(23), helpstring("method GetColumnWidth")] HRESULT
GetColumnWidth([out, retval] BSTR *pWidth);
                [id(24), helpstring("method SetColumnWidth")] HRESULT
SetColumnWidth([in] BSTR width);
                [id(25), helpstring("method GetRowHeight")] HRESULT
GetRowHeight([out, retval] BSTR *pHeight);
                [id(26), helpstring("method GetRowHeight")] HRESULT
SetRowHeight([in] BSTR height);
```

```
                [id(27), helpstring("method InsertColumns")] HRESULT
InsertColumns([in] int type);
                [id(28), helpstring("method DeleteColumns")] HRESULT
DeleteColumns();
                [id(29), helpstring("method InsertRows")] HRESULT
InsertRows([in] int type);
                [id(30), helpstring("method DeleteRows")] HRESULT
DeleteRows();

                [id(31), helpstring("method InsertSeries")] HRESULT
InsertSeries();

                [id(32), helpstring("method IsText")] HRESULT IsText([in] BSTR
cell, [out, retval] BOOL *pValue);
                [id(33), helpstring("method IsNumber")] HRESULT IsNumber([in]
BSTR cell, [out, retval] BOOL *pValue);
                [id(34), helpstring("method IsFormula")] HRESULT
IsFormula([in] BSTR cell, [out, retval] BOOL *pValue);
                [id(35), helpstring("method IsError")] HRESULT IsError([in]
BSTR cell, [out, retval] BYTE *pCode);
                [id(36), helpstring("method InsertData")] HRESULT
InsertData([in] BSTR cell, [in] BSTR data, [in] BOOL parse);
                [id(37), helpstring("method InsertComments")] HRESULT
InsertComments([in] BSTR cell, [in] BSTR comments);
                [id(38), helpstring("method GetData")] HRESULT GetData([in]
BSTR cell, [out, retval] VARIANT *pValue);
                [id(39), helpstring("method GetFormula")] HRESULT
GetFormula([in] BSTR cell, [out, retval] BSTR *pFormula);
                [id(40), helpstring("method GetComments")] HRESULT
GetComments([in] BSTR cell, [out, retval] BSTR *pComments);

                [id(48), helpstring("method IsTextRC")] HRESULT IsTextRC([in]
__int64 row, [in] unsigned short int column, [out, retval] BOOL* pValue);
                [id(49), helpstring("method IsNumberRC")] HRESULT
IsNumberRC([in] __int64 row, [in] unsigned short int column, [out, retval]
BOOL* pValue);
                [id(50), helpstring("method IsFormulaRC")] HRESULT
IsFormulaRC([in] __int64 row, [in] unsigned short int column, [out, retval]
BOOL* pValue);

                [id(62), helpstring("method IsFormulaNumericRC")] HRESULT
IsFormulaNumericRC([in] __int64 row, [in] unsigned short int column, [out,
retval] BOOL* pValue);
                [id(63), helpstring("method IsFormulaTextualRC")] HRESULT
IsFormulaTextualRC([in] __int64 row, [in] unsigned short int column, [out,
retval] BOOL* pValue);

                [id(51), helpstring("method IsErrorRC")] HRESULT
IsErrorRC([in] __int64 row, [in] unsigned short int column, [out, retval]
BYTE* pCode);

                [id(52), helpstring("method InsertNumberRC")] HRESULT
InsertNumberRC([in] __int64 row, [in] unsigned short int column, [in] double
value);
```

```
                [id(53), helpstring("method GetNumberRC")] HRESULT
GetNumberRC([in] __int64 row, [in] unsigned short int column, [out, retval]
double* pValue);

                [id(54), helpstring("method InsertTextRC")] HRESULT
InsertTextRC([in] __int64 row, [in] unsigned short int column, [in] BSTR
text);
                [id(55), helpstring("method GetTextRC")] HRESULT
GetTextRC([in] __int64 row, [in] unsigned short int column, [out, retval]
BSTR* pText);

                [id(56), helpstring("method InsertFormulaRC")] HRESULT
InsertFormulaRC([in] __int64 row, [in] unsigned short int column, [in] BSTR
text);
                [id(57), helpstring("method GetFormulaRC")] HRESULT
GetFormulaRC([in] __int64 row, [in] unsigned short int column, [out, retval]
BSTR* pText);

                [id(58), helpstring("method InsertCommentsRC")] HRESULT
InsertCommentsRC([in] __int64 row, [in] unsigned short int column, [in] BSTR
comments);
                [id(59), helpstring("method GetCommentsRC")] HRESULT
GetCommentsRC([in] __int64 row, [in] unsigned short int column, [out, retval]
BSTR* pComments);
                [id(41), helpstring("method UpdateWindow")] HRESULT
UpdateWindow();

                [id(42), helpstring("method SaveSelectionAsImage")] HRESULT
SaveSelectionAsImage([in] BSTR file);

                [id(43), helpstring("method CreatePrintSettings")] HRESULT
CreatePrintSettings([out, retval] IPrintSettings **ppPrintSettings);
                [id(44), helpstring("method GetPrintSettings")] HRESULT
GetPrintSettings([in] IPrintSettings *pPrintSettings);
                [id(45), helpstring("method SetPrintSettings")] HRESULT
SetPrintSettings([in] IPrintSettings *pPrintSettings);
                [id(46), helpstring("method Print")] HRESULT Print([in] BOOL
showPrintDialog);
        };

        [
                dual,
                helpstring("IWorkbook Interface"),
                pointer_default(unique)
        ]
        interface IWorkbook : IDispatch
        {
                [id(1), helpstring("method GetActiveWorksheet")] HRESULT
GetActiveWorksheet([out, retval] IWorksheet **ppWorksheet);

                [id(49), helpstring("method SetActiveWorksheet")] HRESULT
SetActiveWorksheet([in] BSTR path);
                [id(50), helpstring("method SetActiveFolder")] HRESULT
SetActiveFolder([in] BSTR path);

                [id(51), helpstring("method GetActiveFolderPath")] HRESULT
GetActiveFolderPath([out, retval] BSTR* pPath);
```

```
                [id(52), helpstring("method GetActiveWorksheetPath")] HRESULT
GetActiveWorksheetPath([out, retval] BSTR* pPath);

                [id(2), helpstring("method GetWorksheetCount")] HRESULT
GetWorksheetCount([out, retval] unsigned long* pCounter);
                [id(3), helpstring("method GetFolderCount")] HRESULT
GetFolderCount([out, retval] unsigned long* pCounter);

                [id(4), helpstring("method IsFolderEmpty")] HRESULT
IsFolderEmpty([in] BSTR folder, [out, retval] BOOL *pHasChildItems);
                [id(5), helpstring("method GetFirstTreeItem")] HRESULT
GetFirstTreeItem([in] BSTR folder, [out, retval] BSTR *pChild);
                [id(6), helpstring("method GetNextTreeItem")] HRESULT
GetNextTreeItem([in] BSTR treeItem, [out, retval] BSTR *pSibling);
                [id(7), helpstring("method GetPrevTreeItem")] HRESULT
GetPrevTreeItem([in] BSTR treeItem, [out, retval] BSTR *pSibling);
                [id(8), helpstring("method GetParentFolder")] HRESULT
GetParentFolder([in] BSTR treeItem, [out, retval] BSTR *pParent);
                [id(9), helpstring("method IsWorksheet")] HRESULT
IsWorksheet([in] BSTR treeItem, [out, retval] BOOL *pBool);
                [id(10), helpstring("method IsFolder")] HRESULT IsFolder([in]
BSTR treeItem, [out, retval] BOOL *pBool);

                [id(11), helpstring("method ExpandFolder")] HRESULT
ExpandFolder([in] BSTR folder);
                [id(12), helpstring("method CollapseFolder")] HRESULT
CollapseFolder([in] BSTR folder);

                [id(13), helpstring("method SelectTreeItem")] HRESULT
SelectTreeItem([in] BSTR path);

                [id(14), helpstring("method InsertFolder")] HRESULT
InsertFolder([in] BSTR name);
                [id(15), helpstring("method InsertWorksheet")] HRESULT
InsertWorksheet([in] BSTR name);

                [id(16), helpstring("method DeleteTreeItem")] HRESULT
DeleteTreeItem();
                [id(17), helpstring("method MoveTreeItem")] HRESULT
MoveTreeItem([in] BSTR sourcePath, [in] BSTR targetPath);
                [id(53), helpstring("method CopyTreeItem")] HRESULT
CopyTreeItem([in] BSTR sourcePath, [in] BSTR targetPath);
                [id(18), helpstring("method RenameTreeItem")] HRESULT
RenameTreeItem([in] BSTR name);

                [id(19), helpstring("method GetNamedRangeCount")] HRESULT
GetNamedRangeCount([out, retval] unsigned long *pCounter);
                [id(20), helpstring("method GetNamedRange")] HRESULT
GetNamedRange([in] unsigned long index, [in] BSTR propertyName, [out, retval]
BSTR *pProperty);
                [id(21), helpstring("method SetNamedRange")] HRESULT
SetNamedRange([in] unsigned long index, [in] BSTR rangeName, [in] BSTR
range);
                [id(22), helpstring("method AddNamedRange")] HRESULT
AddNamedRange([in] BSTR rangeName, [in] BSTR range);
                [id(23), helpstring("method RemoveNamedRange")] HRESULT
RemoveNamedRange([in] unsigned long index);
```

```
                [id(24), helpstring("method RemoveAllNamedRanges")] HRESULT
RemoveAllNamedRanges();

                [id(25), helpstring("method Save")] HRESULT Save();
                [id(26), helpstring("method SaveAs")] HRESULT SaveAs([in] BSTR
path);
                [id(27), helpstring("method SaveAsPDFFile")] HRESULT
SaveAsPDFFile([in] BSTR path, [in] BOOL saveAllWorksheets);
                [id(28), helpstring("method SaveAsExcelFile")] HRESULT
SaveAsExcelFile([in] BSTR path);
                [id(29), helpstring("method SaveAsTextFile")] HRESULT
SaveAsTextFile([in] BSTR path, [in] BOOL showDialogBox, [in] ITextParams
*pTextParams);
                [id(30), helpstring("method SaveAsXBaseFile")] HRESULT
SaveAsXBaseFile([in] BSTR path, [in] BOOL showDialogBox, [in] IXBaseParams
*pXBaseParams);
                [id(31), helpstring("method Reload")] HRESULT Reload();
                [id(32), helpstring("method Close")] HRESULT Close();

                [id(33), helpstring("method SetFilePassword")] HRESULT
SetFilePassword([in] BOOL enable, [in] BSTR cryptMethod, [in] BSTR
oldPassword, [in] BSTR newPassword);
                [id(34), helpstring("method SetCellPassword")] HRESULT
SetCellPassword([in] BOOL enable, [in] BSTR oldPassword, [in] BSTR
newPassword);
                [id(35), helpstring("method SetTreePassword")] HRESULT
SetTreePassword([in] BOOL enable, [in] BSTR oldPassword, [in] BSTR
newPassword);

                [id(36), helpstring("method UpdateAllWorksheets")] HRESULT
UpdateAllWorksheets();
                [id(37), helpstring("method UpdateActiveWorksheet")] HRESULT
UpdateActiveWorksheet();
                [id(38), helpstring("method UpdateActiveWorksheetRegion")]
HRESULT UpdateActiveWorksheetRegion([in] BSTR range);
                [id(39), helpstring("method WaitForUpdate")] HRESULT
WaitForUpdate();

                [id(40), helpstring("method MaximizeWindow")] HRESULT
MaximizeWindow();
                [id(41), helpstring("method RestoreWindow")] HRESULT
RestoreWindow();

                [propget, id(42), helpstring("property modified")] HRESULT
modified([out, retval] BOOL *pValue);
                [propput, id(42), helpstring("property modified")] HRESULT
modified([in] BOOL value);

                [propget, id(43), helpstring("property updateMode")] HRESULT
updateMode([out, retval] BSTR *pMode);
                [propput, id(43), helpstring("property updateMode")] HRESULT
updateMode([in] BSTR mode);

                [propget, id(44), helpstring("property updateThreads")]
HRESULT updateThreads([out, retval] BSTR *pThreads);
                [propput, id(44), helpstring("property updateThreads")]
HRESULT updateThreads([in] BSTR threads);
```

```
                [propget, id(45), helpstring("property updatePriority")]
HRESULT updatePriority([out, retval] BSTR *pPriority);
                [propput, id(45), helpstring("property updatePriority")]
HRESULT updatePriority([in] BSTR priority);

                [propget, id(46), helpstring("property updateOptimization")]
HRESULT updateOptimization([out, retval] BSTR *pOptimization);
                [propput, id(46), helpstring("property updateOptimization")]
HRESULT updateOptimization([in] BSTR optimization);

                [propget, id(47), helpstring("property lastError")] HRESULT
lastError([out, retval] BYTE *pCode);
                [propput, id(47), helpstring("property lastError")] HRESULT
lastError([in] BYTE code);

                [id(54), helpstring("method GetFileInfo")] HRESULT
GetFileInfo([in] BSTR path, [in] BYTE type, [out, retval] BSTR* value); // 1
- size, 2 - mod. date
                [id(55), helpstring("method ReleaseFile")] HRESULT
ReleaseFile([out, retval] BSTR* value);
                [id(56), helpstring("method AttachFile")] HRESULT
AttachFile([in] BSTR path, [out, retval] int* modified);

                [id(57), helpstring("method MergeRows")] HRESULT
MergeRows([in] IMergeParams* mergeParams);
                [id(58), helpstring("method MergeRowsFromODSFile")] HRESULT
MergeRowsFromODSFile([in] IMergeParams* mergeParams);
                [id(59), helpstring("method MergeRowsFromTextFile")] HRESULT
MergeRowsFromTextFile([in] IMergeParams* mergeParams, [in] ITextParams*
params);
                [id(60), helpstring("method MergeRowsFromExcelFile")] HRESULT
MergeRowsFromExcelFile([in] IMergeParams* mergeParams);
                [id(61), helpstring("method MergeRowsFromXBaseFile")] HRESULT
MergeRowsFromXBaseFile([in] IMergeParams* mergeParams, [in] IXBaseParams*
params);

                [id(62), helpstring("method MergeTable")] HRESULT
MergeTable([in] BSTR path, [in] BSTR table);
                [id(63), helpstring("method MergeTableFromODSFile")] HRESULT
MergeTableFromODSFile([in] BSTR path, [in] BSTR table);
                [id(64), helpstring("method MergeTableFromTextFile")] HRESULT
MergeTableFromTextFile([in] BSTR path, [in] BSTR table, [in] ITextParams*
params);
                [id(65), helpstring("method MergeTableFromExcelFile")] HRESULT
MergeTableFromExcelFile([in] BSTR path, [in] BSTR table);
                [id(66), helpstring("method MergeTableFromXBaseFile")] HRESULT
MergeTableFromXBaseFile([in] BSTR path, [in] BSTR table, [in] IXBaseParams*
params);
        };


        [
                dual,
                helpstring("IDocuments Interface"),
                pointer_default(unique)
        ]
```

```
        interface IApplication : IDispatch
        {
                [id(1), helpstring("method CreateXBaseParams")] HRESULT
CreateXBaseParams([out, retval] IXBaseParams **ppXBaseParams);
                [id(2), helpstring("method CreateTextParams")] HRESULT
CreateTextParams([out, retval] ITextParams **ppTextParams);
                [id(30), helpstring("method CreateMergeParams")] HRESULT
CreateMergeParams([out, retval] IMergeParams** ppMergeParams);

                [id(3), helpstring("method ThisWorkbook")] HRESULT
ThisWorkbook([out, retval] IWorkbook** ppWorkbook);
                [id(4), helpstring("method NewWorkbook")] HRESULT
NewWorkbook([out, retval] IWorkbook** ppWorkbook);
                [id(5), helpstring("method OpenWorkbook")] HRESULT
OpenWorkbook([in] BSTR path, [in] BSTR password, [out, retval] IWorkbook**
ppWorkbook);
                [id(6), helpstring("method OpenTextFile")] HRESULT
OpenTextFile([in] BSTR path, [in] BOOL showDialogBox, [in] ITextParams
*pTextParams, [out, retval] IWorkbook **ppWorkbook);
                [id(7), helpstring("method OpenExcelFile")] HRESULT
OpenExcelFile([in] BSTR path, [out, retval] IWorkbook** ppWorkbook);
                [id(8), helpstring("method OpenXBaseFile")] HRESULT
OpenXBaseFile([in] BSTR path, [in] BOOL showDialogBox, [in] IXBaseParams*
pXBaseParams, [out, retval] IWorkbook **ppWorkbook);
                [id(9), helpstring("method GetBookmarkCount")] HRESULT
GetBookmarkCount([out, retval] int *pCounter);
                [id(10), helpstring("method GetBookmark")] HRESULT
GetBookmark([in] int index, [out, retval] BSTR *bookMark);
                [id(11), helpstring("method SetBookmark")] HRESULT
SetBookmark([in] int index, [in] BSTR bookMark);
                [id(12), helpstring("method AddBookmark")] HRESULT
AddBookmark([in] BSTR bookMark);
                [id(13), helpstring("method RemoveBookmark")] HRESULT
RemoveBookmark([in] int index);
                [id(14), helpstring("method RemoveAllBookmarks")] HRESULT
RemoveAllBookmarks();

                [id(15), helpstring("method LoadProfile")] HRESULT
LoadProfile([in] BSTR path);
                [id(16), helpstring("method SaveProfile")] HRESULT
SaveProfile([in] BSTR path);

                [id(17), helpstring("method GetWorkbookCount")] HRESULT
GetWorkbookCount([out, retval] int *pCounter);
                [id(18), helpstring("method CloseAllWorkbooks")] HRESULT
CloseAllWorkbooks();
                [id(19), helpstring("method TileWorkbooks")] HRESULT
TileWorkbooks([in] BOOL tileVertically);
                [id(20), helpstring("method MaximizeAppWindow")] HRESULT
MaximizeAppWindow();
                [id(21), helpstring("method MinimizeAppWindow")] HRESULT
MinimizeAppWindow();
                [id(22), helpstring("method RestoreAppWindow")] HRESULT
RestoreAppWindow();
```

```
                [id(23), helpstring("method MessageBox")] HRESULT
MessageBox([in] BSTR message, [in] BSTR buttons, [in] int button, [in] BSTR
icon, [out, retval] BSTR *retValue);
                [id(24), helpstring("method InputBox")] HRESULT InputBox([in]
BSTR title, [in] BOOL password, [in] BSTR initValue, [out, retval] BSTR
*retValue);
                [id(25), helpstring("method Sleep")] HRESULT Sleep([in]
unsigned long time);

                [id(28), helpstring("method GetFilePath")] HRESULT
GetFilePath([in] BOOL saveAs, [in] BSTR title, [in] BSTR folder, [in] BSTR
name, [in] BSTR ext, [out, retval] BSTR* filePath);
                [id(29), helpstring("method GetFolder")] HRESULT
GetFolder([in] BSTR title, [in] BSTR folder, [out, retval] BSTR* filePath);

                [propget, id(26), helpstring("property statusBar")] HRESULT
statusBar([out, retval] BSTR *pText);
                [propput, id(26), helpstring("property statusBar")] HRESULT
statusBar([in] BSTR text);

                [propget, id(27), helpstring("property startFolder")] HRESULT
startFolder([out, retval] BSTR *pFolder);
                [propput, id(27), helpstring("property startFolder")] HRESULT
startFolder([in] BSTR folder);
        };
```

## Samples

Creating and saving a new workbook
Editing an existing workbook
Copying and pasting cell ranges
Formatting
Browsing the worksheet tree
Inserting, copying and removing worksheets and folders
Password protection and encryption
Saving and editing text files
Printing
Saving ranges as *.png images

The following examples were created in MS Visual Studio C++. (The "community" version can be downloaded from the MS website at no cost.) The examples use "smart pointers" and function wrappers with declarations and definitions automatically generated as header and source files by MS VS C++ when you use the #import directive. The "main()" definitions are skipped. These function wrappers use exceptions as a method of handling errors. If you don't want to use exceptions, please see those headers files for how to use the generic COM system function calls instead. For the complete list of interface definitions, please see **Interfaces, methods and properties**.

These interfaces are also used in GS-Calc to support scripting. In scripting languages like JScript and VBScript the rules for creating and using the "GS-Calc.Application" object and the other available objects and their interfaces remain in general the same except that you

can access object "properties" directly, without the "get_..." and "put_..." functions. In both cases the methods/functions are the same and they use the same parameters described in see the **Methods, properties and sample usage** scripting help topic.

Before the interfaces will be accessible to other programs in the Windows system, they must be registered by GS-Calc with the "Register GS-Calc COM Interfaces" command (on the GS-Calc "Settings" menu).

---

**Creating and saving a new workbook**

```
#include <stdio.h>
#include <stdio.h>
#include <comdef.h>

#import "E:\gscalc\gscalc.exe"

using namespace GSCALCLib;

struct StartOle
{
        StartOle() { ::CoInitialize(NULL); }
        ~StartOle() { ::CoUninitialize(); }
} _startOle;

try
{
        IApplicationPtr app;

        app.CreateInstance(L"GSCalc.Application");

        //create a new workbook with the default "sheet1" worksheet
        IWorkbookPtr wbook = app->NewWorkbook();

        //alternatively, add another worksheet
        //wbook->InsertWorksheet(_bstr_t("sheet2"));

        IWorksheetPtr wsheet = wbook->GetActiveWorksheet();

        //turn off automatic updating of formulas for best performance
        //
        //"default" - as defined in the global application settings
        //"automatic" - updating after each cell data change
        //"manual" - updating only with the "Update All/Worksheet" commands
        wbook->put_updateMode(_bstr_t("manual"));

        //insert a series of numbers 1 to 10000 in c1:c10000
        for (int i = 1; i <= 10000; ++i)
        {
                wchar_t buffer[20] = { 0 };
                wsheet->InsertNumberRC(i, 3, i);
        }
```

```
        //alternatively, the InsertData() function can be used, however
        //it performs additional checkings, conversions, autoformatting and
activating "undo"
        //which results in a much slower (up to several hundreds of times)
execution (see the corresponding help topic for details)
        /*
        for (int i = 1; i <= 10000; ++i)
        {
                char buffer[20] = { 0 };
                ::_itoa(i, buffer, 10);
                wsheet->InsertData(wsheet->Address(3, i), _bstr_t(buffer),
true);
        }
        */

        //insert a label and a formula to sum the above numbers
        wsheet->InsertTextRC(10001, 1, _bstr_t("sample sum"));
        wsheet->InsertFormulaRC(10001, 3, _bstr_t("=sum(c1:c1000)"));

        //add a comment
        wsheet->InsertCommentsRC(10001, 3, _bstr_t("sum of numbers from c1 to
c10000"));

        //if it's necessary recalculate all worksheets and update the views;
        wbook->UpdateAllWorksheets();

        //if updating is performed in the background, wait till it's complete
        wbook->WaitForUpdate();

        //or simply update the screen (as the "Insert[...]RC" functions don't
refresh the screen)
        //wsheet->UpdateWindow();

        //save a workbook using the native *.gsc format;
        //specifying the *.ods extension causes saving it in the ODF format
        wbook->SaveAs(_bstr_t("e:\\sample111.gsc"));
        wbook->Close();
}
catch(_com_error error)
{
        // ...
}
```

---

**Editing an existing workbook**

```
#include <stdio.h>
#include <stdio.h>
#include <comdef.h>

#import "E:\gscalc\gscalc.exe"

using namespace GSCALCLib;
```

```
struct StartOle
{
	StartOle() { ::CoInitialize(NULL); }
	~StartOle() { ::CoUninitialize(); }
} _startOle;

try
{
	IApplicationPtr app;

	app.CreateInstance(L"GSCalc.Application");

	//open the sample.gsc workbook
	IWorkbookPtr wbook = app->OpenWorkbook(_bstr_t(L"e:\\sample.gsc"),
(BSTR)NULL);

	//set the default active worksheet;
	//once it's set and the file is saved it'll remain the active one
after re-opening;

	if (::_wcsicmp(wbook->GetActiveWorksheetPath(), _bstr_t(L"data
types")))
		wbook->SetActiveWorksheet(_bstr_t(L"data types"));

	IWorksheetPtr wsheet = wbook->GetActiveWorksheet();

	//add 1.0 to the cell C1
	double v1 = wsheet->GetNumberRC(1, 3);
	v1 += 1.0;
	wsheet->InsertNumberRC(1, 3, v1);

	//not using the WaitForUpdate() function will cause terminating the
current updating process
	//and restarting it after another cell change below
	//wbook->WaitForUpdate();

	//insert formula in C4
	wsheet->InsertFormulaRC(5, 3, _bstr_t("=sum(c1:c4)"));

	//if the update mode it set to "manual", update formulas excplicitely;
	//wbook->UpdateActiveWorksheet();

	//if updating is performed in the background, wait till it's complete
	wbook->WaitForUpdate();

	//from the B6 cell get the formula, its value and optionally an error
code;
	//note: if a formula returns a text string, the IsFormulaTextualRC()
functions returns TRUE
	//and the GetTextRC() function must be used to obtain its current
value
	_bstr_t formula = wsheet->GetFormulaRC(5, 3);
	double v2 = wsheet->GetNumberRC(5, 3);
	BYTE errorCode = wsheet->IsErrorRC(5, 3);

	//assuming it's a console app, print the data
	if (!errorCode)
```

```
                ::wprintf(L"%s  %s -> %g", static_cast<wchar_t*>(wsheet-
>AddressRC(5, 3)), static_cast<wchar_t*>(formula), v2);
        else
                ::wprintf(L"%s  %s -> error (%u)",
static_cast<wchar_t*>(wsheet->AddressRC(5, 3)),
static_cast<wchar_t*>(formula), errorCode);

        //clear the B1 cell;
        //deleting and formatting cells/ranges requires selecting the
respective range;
        //the 2nd parameter in DeleteCells can be a string with the following
substrings:
        //data numbers labels formulas formatting lists comments;
        //data = numbers labels formulas
        wsheet->put_selectedRange(wsheet->AddressRC(1, 3));
        wsheet->DeleteCells(false, _bstr_t("data"));

        //if the update mode it set to "manual", update formulas excplicitely;
        //wbook->UpdateActiveWorksheet();

        //if updating is performed in the background, wait till it's complete
        wbook->WaitForUpdate();

        errorCode = wsheet->IsErrorRC(5, 3);

        //print the data again
        if (!errorCode)
                ::wprintf(L"\n%s  %s -> %g", static_cast<wchar_t*>(wsheet-
>AddressRC(5, 3)), static_cast<wchar_t*>(formula), wsheet->GetNumberRC(5,
3));
        else
                ::wprintf(L"\n%s  %s -> error (%u)",
static_cast<wchar_t*>(wsheet->AddressRC(5, 3)),
static_cast<wchar_t*>(formula), errorCode);

        //insert the n/a! error code in B1
        wsheet->InsertData(wsheet->Address(3, 1), _bstr_t("#N/A!"), true);

        //if the update mode it set to "manual", update formulas excplicitely;
        //wbook->UpdateActiveWorksheet();

        //if updating is performed in the background, wait till it's complete
        wbook->WaitForUpdate();

        errorCode = wsheet->IsErrorRC(5, 3);

        //print the data again
        if (!errorCode)
                ::wprintf(L"\n%s  %s -> %g", static_cast<wchar_t*>(wsheet-
>AddressRC(5, 3)), static_cast<wchar_t*>(formula), wsheet->GetNumberRC(5,
3));
        else
                ::wprintf(L"\n%s  %s -> error (%u)",
static_cast<wchar_t*>(wsheet->AddressRC(5, 3)),
static_cast<wchar_t*>(formula), errorCode);

        //save changes
```

```
        wbook->Save();
        wbook->Close();
}
catch(_com_error error)
{
        // ...
}
```

---

**Copying and pasting cell ranges**

```
#include <stdio.h>
#include <stdio.h>
#include <comdef.h>

#import "E:\gscalc\gscalc.exe"

using namespace GSCALCLib;

struct StartOle
{
        StartOle() { ::CoInitialize(NULL); }
        ~StartOle() { ::CoUninitialize(); }
} _startOle;


try
{
        IApplicationPtr app;

        app.CreateInstance(L"GSCalc.Application");

        //open the sample.gsc workbook
        IWorkbookPtr wbook = app->OpenWorkbook(_bstr_t(L"e:\\sample.gsc"),
(BSTR)NULL);

        //set the default active worksheet;
        //once it's set and the file is saved it'll remain the active one
after re-opening;

        if (::_wcsicmp(wbook->GetActiveWorksheetPath(), _bstr_t(L"data
types")))
                wbook->SetActiveWorksheet(_bstr_t(L"data types"));

        IWorksheetPtr wsheet = wbook->GetActiveWorksheet();

        //select the range to copy
        wsheet->put_selectedRange(wsheet->AddressRC(1, 3) + _bstr_t(L":") +
wsheet->AddressRC(5, 3));

        //copy
        //1 - copy all data including formulas
        //5 - copy all data with formula values instead of formulas
        wsheet->CopyCells(1);
```

```
        //select the top-left cell for the inserted data
        wsheet->put_selectedRange(wsheet->AddressRC(1, 4));

        //paste the copied cells
        //true - paste data and formatting
        //false - paste only data, preserving the existing formatting
        wsheet->PasteCells(true);

        //select the range to paste
        wsheet->put_selectedRange(wsheet->AddressRC(1, 5) + _bstr_t(L":") +
wsheet->AddressRC(15, 6));

        //paste the copied cells;
        //the data will be duplicated within that range
        wsheet->PasteCells(true);

        //if updating is performed in the background, wait till it's complete
        wbook->WaitForUpdate();

        //save changes
        wbook->Save();
        wbook->Close();
}
catch(_com_error error)
{
        // ...
}
```

---

**Formatting**

```
#include <stdio.h>
#include <stdio.h>
#include <comdef.h>

#import "E:\gscalc\gscalc.exe"

using namespace GSCALCLib;

struct StartOle
{
        StartOle() { ::CoInitialize(NULL); }
        ~StartOle() { ::CoUninitialize(); }
} _startOle;


try
{
        IApplicationPtr app;

        app.CreateInstance(L"GSCalc.Application");

        //open the sample.gsc workbook
```

```
        IWorkbookPtr wbook = app->OpenWorkbook(_bstr_t(L"e:\\sample.gsc"),
(BSTR)NULL);

        //set the default active worksheet;
        //once it's set and the file is saved it'll remain the active one
after re-opening;
        if (::_wcsicmp(wbook->GetActiveWorksheetPath(), _bstr_t(L"data
types")))
                wbook->SetActiveWorksheet(_bstr_t(L"data types"));

        IWorksheetPtr wsheet = wbook->GetActiveWorksheet();

        //select the range C1:C5
        wsheet->put_selectedRange(wsheet->AddressRC(1, 3) + _bstr_t(L":") +
wsheet->AddressRC(5, 3));

        IFormatParamsPtr format = wsheet->CreateFormatParams();

        //set the scientific format leaving the default options
        //parameters:
        //1. a number of decimals: auto|0...14
        //2. an exponent: auto|-99...99
        format->SetScientificFormat(_bstr_t("auto"), _bstr_t("auto"));

        //format C1:C5
        wsheet->SetCellFormat(format);

        //clear previously used formatting attributes
        format->Reset();

        // set the italic attribute
        format->put_italicFont(true);

        wsheet->SetCellFormat(format);

        format->Reset();

        // set a new font and all its attributes
        format->put_italicFont(TRUE);
        format->put_underlineFont(TRUE);
        format->put_strikeoutFont(TRUE);
        format->put_boldFont(TRUE);
        format->put_fontColor(_bstr_t("white"));//or format-
>put_fontColor(_bstr_t("#FFFFFF"));
        format->put_fontSize(15);
        format->put_fontName(_bstr_t("Courier"));

        wsheet->SetCellFormat(format);

        format->Reset();

        //set new cell filling
        format->put_bkgColor(_bstr_t("green")); //or format-
>put_fontColor(_bstr_t("#00FF00"));

        //set new cell border attributes;
        //boder type parameter:
```

```
        //0 - remove borders
        //1 - top
        //2 - bottom
        //4 - left
        //8 - right
        //16 - diagonal left
        //32 - diagonal right
        //64 - auto
        format->put_borderPosition(1 | 2 | 4 | 8);
        format->put_borderColor(_bstr_t("blue"));
        //boder style parameter:
        //none | solid | dot | dash | dash-dot | dash-dot-dot
        format->put_borderStyle(_bstr_t("solid"));
        format->put_borderWidth(2);

        wsheet->SetCellFormat(format);

        //save changes
        wbook->Save();
        wbook->Close();
}
catch(_com_error error)
{
        // ...
}
```

---

**Browsing the worksheet tree**

```
#include <stdio.h>
#include <stdio.h>
#include <comdef.h>

#import "E:\gscalc\gscalc.exe"

using namespace GSCALCLib;

struct StartOle
{
        StartOle() { ::CoInitialize(NULL); }
        ~StartOle() { ::CoUninitialize(); }
} _startOle;


try
{
        IApplicationPtr app;

        app.CreateInstance(L"GSCalc.Application");

        //open the sample.gsc workbook
        IWorkbookPtr wbook = app->OpenWorkbook(_bstr_t(L"e:\\sample.gsc"),
(BSTR)NULL);
```

```
        // 1. browse the folder containing the active worksheet; don't list
nested folders

        _bstr_t folder = wbook->GetParentFolder(wbook-
>GetActiveWorksheetPath());

        _bstr_t itemPath = wbook->GetFirstTreeItem(folder);

        size_t counter = 0;

        while (itemPath.length())
        {
                if (wbook->IsFolder(itemPath))
                {
                        ::wprintf(L"\n%zu.  Folder: %s", ++counter,
static_cast<wchar_t*>(itemPath));
                }
                else
                {
                        ::wprintf(L"\n%zu. Worksheet: %s", ++counter,
static_cast<wchar_t*>(itemPath));
                        //
                        //... e.g. wbook->SetActiveWorksheet(itemPath)...
                        //
                }
                itemPath = wbook->(itemPath);
        }

        ::wprintf(L"\n\n");

        // 2. browse the entire worksheet tree; list nested folders

        //an arbitrary chosen max. nesting level for the sample.gsc folders
        _bstr_t stack[5];
        size_t level = 0;

        itemPath = wbook->GetFirstTreeItem((BSTR)NULL);

        counter = 0;

        while (itemPath.length())
        {
                if (wbook->IsFolder(itemPath))
                {
                        ::wprintf(L"\n%zu. Folder: %s", ++counter,
static_cast<wchar_t*>(itemPath));

                        _bstr_t itemPath2 = wbook->GetFirstTreeItem(itemPath);
                        if (itemPath2.length() && level < 5)
                        {
                                stack[level++] = itemPath;
                                itemPath = itemPath2;
                                continue;
                        }
                }
                else
                {
```

```
                        ::wprintf(L"\n%zu. Worksheet: %s", ++counter,
static_cast<wchar_t*>(itemPath));
                        //
                        //... e.g. wbook->SetActiveWorksheet(itemPath)...
                        //
                }
                itemPath = wbook->GetNextTreeItem(itemPath);
                if (!itemPath.length() && level)
                        itemPath = wbook->GetNextTreeItem(stack[--level]);
        }

        assert(counter == static_cast<size_t>(wbook->GetWorksheetCount()) +
wbook->GetFolderCount());

        wbook->Close();
}
catch(_com_error error)
{
        // ...
}
```

---

**Inserting, copying and removing worksheets and folders**

```
#include <stdio.h>
#include <stdio.h>
#include <comdef.h>

#import "E:\gscalc\gscalc.exe"

using namespace GSCALCLib;

struct StartOle
{
        StartOle() { ::CoInitialize(NULL); }
        ~StartOle() { ::CoUninitialize(); }
} _startOle;


try
{
        IApplicationPtr app;

        app.CreateInstance(L"GSCalc.Application");

        //open the sample.gsc workbook
        IWorkbookPtr wbook = app->OpenWorkbook(_bstr_t(L"e:\\sample.gsc"),
(BSTR)NULL);

        //copy/duplicate the "data types" worksheet within the main folder;
        //the worksheet will be inserted as "data types(1)"
        wbook->CopyTreeItem(_bstr_t(L"data types"), (BSTR)NULL);

        //create a new folder in the main/root folder;
```

```
        //the InsertFolder()/InsertWorskeet() functions require names without
paths as parameters
        //and folders/sheets are inserted in the active folder
        wbook->SetActiveFolder((BSTR)NULL);
        wbook->(_bstr_t(L"new-folder"));

        //copy the worksheet "data types" from the main folder to the folder
"new-folder";
        wbook->CopyTreeItem(_bstr_t(L"data types"), _bstr_t(L"new-folder"));

        //copy it again - the worksheet name will be "data types(1)"
        wbook->CopyTreeItem(_bstr_t(L"data types"), _bstr_t(L"new-folder"));

        //create a new nested folder in the folder "new-folder";
        //the InsertFolder()/InsertWorskeet() functions require names without
paths as parameters
        //and folders/sheets are inserted in the active folder
        wbook->SetActiveFolder(_bstr_t(L"new-folder"));
        wbook->InsertFolder(_bstr_t(L"new-nested-folder"));

        //move "data types(1)" to "new-nested-folder"
        wbook->MoveTreeItem(_bstr_t(L"new-folder\\data types(1)"),
_bstr_t(L"new-folder\\new-nested-folder"));


        //create a new folder in the main/root folder
        wbook->SetActiveFolder((BSTR)NULL);
        wbook->InsertFolder(_bstr_t(L"new-folder2"));

        //copy the folder "2d charts" from the main folder to "new-folder2"
        wbook->CopyTreeItem(_bstr_t(L"2D charts"), _bstr_t(L"new-folder2"));

        //delete the active folder "2d charts" along with its worksheets;
        //the DeleteTreeItem() functions concerns the currently selected
worksheet or folder
        wbook->SetActiveFolder(_bstr_t(L"2d charts"));
        wbook->DeleteTreeItem();

        //move the "3d charts" folder to "new-folder2";
        wbook->MoveTreeItem(_bstr_t(L"3d charts"), _bstr_t(L"new-folder2"));

        //some time to check the results
        ::Sleep(10000);

        //close without saving
        wbook->Close();
}
catch(_com_error error)
{
        // ...
}
```

---

**Password protection and encryption**

```
#include <stdio.h>
#include <stdio.h>
#include <comdef.h>

#import "E:\gscalc\gscalc.exe"

using namespace GSCALCLib;

struct StartOle
{
	StartOle() { ::CoInitialize(NULL); }
	~StartOle() { ::CoUninitialize(); }
} _startOle;


try
{
	IApplicationPtr app;

	app.CreateInstance(L"GSCalc.Application");

	//open the sample.gsc workbook
	IWorkbookPtr wbook = app->OpenWorkbook(_bstr_t(L"e:\\sample.gsc"),
(BSTR)NULL);

	wbook->SetFilePassword(true, _bstr_t(L"blowfish"), _bstr_t(""),
_bstr_t("Rk4@m7"));

	wbook->SaveAs(_bstr_t(L"e:\\sample_p.gsc"));
	wbook->Close();

	//open the protected sample.gsc workbook
	wbook = app->OpenWorkbook(_bstr_t(L"e:\\sample_p.gsc"),
_bstr_t("Rk4@m7"));

	//
	//...
	//

	wbook->Close();
}
catch(_com_error error)
{
	// ...
}
```

---

**Saving and editing text files**

```
#include <stdio.h>
#include <stdio.h>
```

```
#include <comdef.h>

#import "E:\gscalc\gscalc.exe"

using namespace GSCALCLib;

struct StartOle
{
	StartOle() { ::CoInitialize(NULL); }
	~StartOle() { ::CoUninitialize(); }
} _startOle;


try
{
	IApplicationPtr app;

	app.CreateInstance(L"GSCalc.Application");

	//open the sample.gsc workbook
	IWorkbookPtr wbook = app->OpenWorkbook(_bstr_t(L"e:\\sample.gsc"),
(BSTR)NULL);

	ITextParamsPtr txt = app->CreateTextParams();
	txt->put_separator(_bstr_t("\t"));

	wbook->SetActiveWorksheet(_bstr_t(L"Pivot tables\\customers"));

	//save the "customers" worksheet as a text file;
	//don't display dialog boxes with options
	wbook->SaveAsTextFile(_bstr_t("e:\\sample_123.txt"), false, txt);

	//close the text file
	wbook->Close();


	//open the "e:\\sample_123.txt" text file;
	//show the dialog box with text options
	wbook = app->OpenTextFile(_bstr_t("e:\\sample_123.txt"), true, txt);

	//
	// ... edit the file
	//

	//to save the file with modified parameters, the SaveAsTextFile()
function must be used
	txt->put_separator(_bstr_t(","));
	//set the utf16 encoding (the default one was "utf8")
	txt->put_encoding(_bstr_t("utf16"));
	wbook->SaveAsTextFile(_bstr_t("e:\\sample_123.txt"), false, txt);

	//to save it using the originally used "txt" parameters, the Save()
function can be used
	//wbook->Save();

	wbook->Close();
}
```

```
catch(_com_error error)
{
        // ...
}
```

---

## Printing

```
#include <stdio.h>
#include <stdio.h>
#include <comdef.h>

#import "E:\gscalc\gscalc.exe"

using namespace GSCALCLib;

struct StartOle
{
        StartOle() { ::CoInitialize(NULL); }
        ~StartOle() { ::CoUninitialize(); }
} _startOle;


try
{
        IApplicationPtr app;

        app.CreateInstance(L"GSCalc.Application");

        //open the sample.gsc workbook
        IWorkbookPtr wbook = app->OpenWorkbook(_bstr_t(L"e:\\sample.gsc"),
(BSTR)NULL);

        //set the default active worksheet;
        //once it's set and the file is saved it'll remain the active one
after re-opening;
        if (::_wcsicmp(wbook->GetActiveWorksheetPath(), _bstr_t(L"data
types")))
                wbook->SetActiveWorksheet(_bstr_t(L"data types"));

        IWorksheetPtr wsheet = wbook->GetActiveWorksheet();

        IPrintSettingsPtr prn = wsheet->CreatePrintSettings();

        //get the default print settings for this worksheet
        wsheet->GetPrintSettings(prn);

        //modify some options
        prn->put_autoFit(true);
        prn->put_marginTop(15/*mm*/);

        //save back the modified settings
        wsheet->SetPrintSettings(prn);
```

```
        //print the worksheet; don't display the "Print" dialog box
        wsheet->Print(false);

        wbook->Close();
}
catch(_com_error error)
{
        // ...
}
```

---

### Saving ranges as *.png images

```
#include <stdio.h>
#include <stdio.h>
#include <comdef.h>

#import "E:\gscalc\gscalc.exe"

using namespace GSCALCLib;

struct StartOle
{
        StartOle() { ::CoInitialize(NULL); }
        ~StartOle() { ::CoUninitialize(); }
} _startOle;


try
{
        IApplicationPtr app;

        app.CreateInstance(L"GSCalc.Application");

        //open the sample.gsc workbook
        IWorkbookPtr wbook = app->OpenWorkbook(_bstr_t(L"e:\\sample.gsc"),
(BSTR)NULL);

        //set the default active worksheet;
        //once it's set and the file is saved it'll remain the active one
after re-opening;
        if (::_wcsicmp(wbook->GetActiveWorksheetPath(), _bstr_t(L"data
types")))
                wbook->SetActiveWorksheet(_bstr_t(L"data types"));

        IWorksheetPtr wsheet = wbook->GetActiveWorksheet();

        wsheet->put_selectedRange(_bstr_t(L"c1:c5"));

        wsheet->SaveSelectionAsImage(_bstr_t(L"e:\\image_c1c5.png"));

        wbook->Close();
}
catch(_com_error error)
```

```
{
        // ...
}
```

---

# Support

## Support

If you have any questions, comments or suggestions, please visit the support forum or e-mail Citadel5 directly at info@citadel5.com.

# Copyrights

## Copyrights

## Mathematical Functions

---

**abs(x)**

Returns the absolute value of x.

=abs(-1)  returns 1

---

**acos(x)**

Returns the arccosine of x.

=acos(-1)  returns 3.14159265358979 (=pi())

---

**acosh(x)**

Returns the inverse hyperbolic cosine of x.

=acosh(10)  returns 2.99322284612638

---

**asin(x)**

Returns the arcsine of x.

=asin(1)  returns 1.5707963267949 (=pi()/2)

---

**asinh(x)**

Returns the inverse hyperbolic sine of x.

=asinh(10)  returns 2.99822295029797

---

**atan(x)**

Returns the arctangent of x (-PI/2, PI/2).

=atan(1)  returns 0.78539816339745 (=pi()/4)

---

**atan2(x, y)**

Returns the arctangent of y/x (-PI, PI). If x is 0 or if both x and y are zero, atan2 returns 0.

=atan2(1, 1)  returns 0.78539816339745 (=pi()/4)

---

**atanh(x)**

Returns the inverse hyperbolic tangent of x.

=atanh(0.76159415595576)  returns 0.9(9)

---

**ceiling(x, m)**

Rounds up x to the nearest integer or the nearest multiplicity of m.

=ceiling(5.1, 1)  returns 6

=ceiling(-5.1, 1)  returns -6.

---

**CEILING.MATH(number, [significance], [mode])**

See the description online.

---

**CEILING.PRECISE(number, [significance])**

See the description online.

---

**combin(n, k)**

Returns the number of k-element combinations for a k-element set.

=combin(8, 2)  returns 28

---

**combin2(n, k)**

Returns the number of k-element combinations with repeating elements for a n-element set.

=combin2(8, 2)  returns 36

---

**cos(x)**

Returns the cosine of x. The argument is in radians.

=cos(pi())  returns -1

---

**cosh(x)**

Returns the hyperbolic cosine of x.

=cosh(1)  returns 1.54308063481524

---

**degress(x)**

Converts radians to degrees.

=degrees(pi()/2)  returns 90

---

**even(x)**

Rounds x up (away from zero) to the nearest even integer.

=even(1.5)  returns 2

=even(-1)  returns -2

---

**exp(x)**

Raises e to the power of x.

=exp(1)  returns 2.71828182845905

---

**fact(n)**

Returns the factorial of n. The argument must be in the range <1,170>.

=fact(6)  returns 720

---

**factDouble(n)**

If n is an even number, factDouble returns the product n(n-2)(n-4)...(4)(2).
If n is an odd number, factDouble returns the product n(n-2)(n-4)...(3)(1).
The argument must be in the range <1,170>.

=factDouble(6)  returns 48

---

**floor(x, m)**

Rounds (down) x to the nearest integer or the nearest multiplicity of m.

=floor(5.1, 1)  returns 5

=floor(-5.1, 1)  returns -5.

---

**FLOOR.MATH(number, [significance], [mode])**

See the description online.

---

**FLOOR.PRECISE(number, [significance])**

See the description online.

---

**gcd(v1, v2, ...)**

Returns the greatest common divisor for a given list of arguments (numbers/arrays of numbers).

=gcd(32, 24)  returns 8.

=gcd({24,64,32}, 128)  returns 8.

---

**int(x)**

Rounds (down) x to the nearest integer.

=int(5.1)  returns 5

=int(-5.1)  returns -6.

---

**ISO.CEILING(number)**

See the description online.

---

**lcm(v1, v2, ...)**

Returns the least common multiple for a given list of arguments (numbers/array of numbers).

=lcm(8, 7)  returns 56

=lcm({16, 44, 32}, 11)  returns 352.

---

**ln(x)**

Returns the natural logarithm of x.

=ln(2.71828182845905)  returns 1

---

**log(x, [y])**

Returns the logarithm of x to the base y. If y is omitted, it's assumed to be 10.

=log(10,)  returns 1

---

**log10(x)**

Returns the base-10 logarithm of x.

=log10(100)  returns 2

---

**mDeterm(m)**

Returns the determinant of the square matrix m.

=mDeterm({8,5,-2; 2,3,1; 3,-1,-3})  returns 3

---

**mInverse(m)**

Returns the inverse of the square matrix m.

=mInverse({8,5,-2; 2,3,1; 3,-1,-3})  returns {-2.66666666666667, 5.66666666666667, 3.66666666666667; 3, -6, -4; -3.66666666666667, 7.66666666666667, 4.66666666666667}

---

**mMult(v1, v2, ...)**

Multiplies the specified matrices.

=mMult({1,2,1;0,2,1}, {1;1;2}, {2,1,1}*2)  returns {20, 10, 10; 16, 8, 8}

---

**mod(x, y)**

Returns the floating point remainder of x/y.

=mod(10, 3)  returns 1

=mod(-3, 2)  returns -1

=mod(3, -2)  returns 1

=mod(-3, -2)  returns -1

---

**mRound(x, m)**

Rounds x to the multiplicity of m, where x and m have the same sign.

=mRound(10, 3)  returns 9

=mRound(1.3, 0.2)  returns 1.4

---

**multinomial(number1, number2, ...)**

Returns the ratio (number1+number2+...)!/(number1!number2!...).

=multinomial(2, 3, 4)  returns 1260.

---

**odd(x)**

Rounds x up (away from zero) to the nearest odd integer.

=odd(6)  returns 7

=odd(-4)  returns -5

---

**pi()**

Returns the value of PI (3.14159265358979).

---

**power(x, y)**

Returns x raised to the power of y.

---

**product(v1, v2, ...)**

Multiplies the specified arguments which can be numbers, text representations of numbers or arrays/ranges. Empty cells are ignored. If there are no valid numbers, the function returns the #N/A! error value.

=product(3, 4, {1, 5})  returns 60

---

**productIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Multiplies numbers that meet the specified criteria. All text strings in the specified 'data_range' array/range are ignored. All arrays/ranges must have the same number of columns and rows. If there are no valid numbers (to multiple), the function returns the #N/A! error value.
The criteria can be one of the following:

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

=productIfs({1,2;3,2}, {4,5;2,7}, 2)  returns 3

=productIfs({1,2;3,4}, {4,5;2,7}, ">2", {4,5;2,7}, "<7")  returns 2

=productIfs({2, 4; 3, 1}, {"abcde","def";"abc","a"}, "?bc*")  returns 6

=productIfs($a$1:$a$1000, $d$1:$d$1000, ">"&f2, $c$1:$c$1000, "<="&f3)  multiplies values greater than value from the cell F2 and not greater than the F3 cell value.

---

**quotient(x, y)**

Returns the integer part of x/y.

=quotient(5, 2)  returns 2

=quotient(-10, 3)  returns -3

---

**radians(x)**

Converts degrees to radians.

=radians(180)  returns 3.14159265358979 (=pi())

---

**rand()**

Returns an evenly distributed number in the range <0, 1). A new number is returned each time the workbook is updated.

See the mtxRand function.

---

**randBetween(x1, x2)**

Returns an evenly distributed number in the range <x1, x2). A new number is returned each time the workbook is updated.

---

**round(x, [digits])**

round(x)

Rounds x to the specified number of digits. If the 'digits' argument is greater than 0, then x is rounded to the specified number of decimal places. If it's 0, then x is rounded to the nearest integer. If it's less than 0, then x is rounded to the left of the decimal point. The default value for 'digits' is 0.

=round(2.6,)  returns 3

=round(2.15, 1)  returns 2.2

=round(2.149, 1)  returns 2.1

=round(-1.475, 2)  returns -1.48

=round(123.5, -2)  returns 100

=round(2.6)  returns 3

=round(2.15)  returns 2

---

**roundDown(x, [digits])**

Rounds x down (towards 0) to the specified number of digits. If the 'digits' argument is greater than 0, then x is rounded to the specified number of decimal places. If it's 0, then x is rounded to the nearest integer. If it's less than 0, then x is rounded to the left of the decimal point. The default value for 'digits' is 0.

=roundDown(2.6,)  returns 2

=roundDown(2.15, 1)  returns 2.1

=roundDown(2.7, 0)  returns 2

=roundDown(-2.7, 0)  returns -2

---

**roundUp(x, [digits])**

Rounds x up (away from zero) to the specified number of digits. If the 'digits' argument is greater than 0, then x is rounded to the specified number of decimal places. If it's 0, then x is rounded to the nearest integer. If it's less than 0, then x is rounded to the left of the decimal point. The default value for 'digits' is 0.

=roundUp(2.6,)  returns 3

=roundUp(2.15, 1)  returns 2.2

=roundUp(2.7, 0)  returns 3

=roundUp(-2.7, 0)  returns -3

---

**seriesSum(x, n, m, a)**

For the given numbers x, n, m and array a, seriesSum returns the value of the power series:

a(1)x^n + a(2)x^(n+m) + a(3)x^(n+2m) + ... + a(l)x^(n+(l-1)m)

where l is the number of cells in the array a and a(i) is the i-th cell (row-wise) in the array a.

The polynomial:
5 + 2*x + 3*x^3

=seriesSum(2, 0, 1, {5, 2, 0, 3})  returns 33

---

**sign(x)**

Returns 1 if x is positive, 0 if x equals 0 and -1 if x is negative.

=sign(-5)  returns -1

=sign(5)  returns 5

---

**sin(x)**

Returns the sin of x. The argument is in radians.

=sin(pi()/2)  returns 1

---

**sinh(x)**

Returns the hyperbolic sine of x.

=sinh(1)  returns 1.1752011936438

---

**sqrt(x)**

Returns the square root of x.

=sqrt(81)  returns 9

---

**sqrtPi(x)**

Returns the square root of x*PI.

=sqrtPi(1)  returns 1.77245385090552

---

**subTotal(v1, v2, ...)**

Calculate a subtotal for the specified arguments which can be numbers or arrays of numbers. Cells containing formulas with another "subTotal" function references are ignored.

---

**sum(v1, v2, ...)**

Adds the specified arguments. Arguments that are either text strings which can't be converted to numbers or errors cause an error.

Text strings in arrays or cell/range references are ignored. To include the text representations of numbers, use the sumA() function.

=sum(3, 4, {1, 5})  returns 13

=sum(3, 4, {1, "5"})  returns 8

---

**sumA(v1, v2, ...)**

Adds the specified arguments. Arguments that are either text strings which can't be converted to numbers or errors cause an error.

Text representations of numbers in arrays or cell/range references are included. Other text strings are ignored. To exclude text strings entirely, use the sum() function.

=sumA(3, 4, {1, 5})  returns 13

=sumA(3, 4, {1, "5"})  returns 13

---

**sumIf(if_range, criteria, [data_range])**

Adds numbers that meet the specified criteria. All text strings are ignored. If the 'data_range' range is omitted, cells from the 'if_range' range are used. The 'sum_range' and 'if_range' ranges must have the same number of columns and rows. The criteria can be one of the following:

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

=sumIf({1,2;3,2}, 2,)  returns 4

=sumIf({1,2;3,4}, ">2",)  returns 7

=sumIf({1,2;3,4}, ">"&b4,)  returns a sum based on the b4 cell value

=sumIf({"abcde","def";"abc","a"}, "?bc*", {2, 4; 3, 1})  returns 5

=sumIf({"abcde","def";"bc","a"}, "bc", {2, 4; 3, 1})  returns 3

=sumIf({"ABcde","def";"Bc","a"}, ">=bc", {2, 4; 3, 1})  returns 7

---

**sumAIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Adds numbers from the 'data_range' that meet a number of criteria. Text representations of numbers are included. All other text strings are ignored.
All ranges must have the same number of columns and rows. The criteria can be one of the following:

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

=sumAIfs({1,2;3,2}, {4,5;2,7}, 2)  returns 3

=sumAIfs({1,2;3,4}, {4,5;2,7}, ">2", {4,5;2,7}, "<7")  returns 3

=sumIfs({2, 4; 3, 1}, {"abcde","def";"abc","a"}, "?bc*")  returns 5

=sumAIfs($a$1:$a$1000, $d$1:$d$1000, ">"&f2, $c$1:$c$1000, "<="&f3)  sums values greater than value from the cell F2 and not greater than the F3 cell value.

---

**sumIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Adds numbers from the 'data_range' array/ that meet a number of criteria. All text strings 'data_range' are ignored. All ranges must have the same number of columns and rows. The criteria can be one of the following:

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

=sumIfs({1,2;3,2}, {4,5;2,7}, 2)  returns 3

=sumIfs({1,2;3,4}, {4,5;2,7}, ">2", {4,5;2,7}, "<7")  returns 3

=sumIfs({2, 4; 3, 1}, {"abcde","def";"abc","a"}, "?bc*")  returns 5

=sumIfs($a$1:$a$1000, $d$1:$d$1000, ">"&f2, $c$1:$c$1000, "<="&f3)  sums values greater than value from the cell F2 and not greater than the F3 cell value.

---

**sumProduct(v1, v2, ...)**

Multiplies numbers from the specified 'v1','v2',... arrays and add the obtained products. Empty cells are ignored.

=sumProduct({1,3}, {2,4})  returns 14

---

**sumSq(v1, v2, ...)**

Adds squares of all the specified arguments which can be numbers or arrays of numbers.

=sumSq(3, 4, {1, 5})  returns 51

=sumSq(true, 2)  returns 5

---

**sumX2mY2(x_array, y_array)**

Returns the sum of the difference of squares of numbers from two arrays. Empty cells are treated as 0 values.

=sumX2mY2({2,5}, {6,1})  returns -8

---

**sumX2pY2(x_array, y_array)**

Returns the sum of the sum of squares of numbers from two arrays. Empty cells are treated as 0 values.

=sumX2pY2({2,5}, {6,1})  returns 66

---

**sumXmY2(x_array, y_array)**

Returns the sum of squares of differences of numbers from two arrays. Empty cells are treated as 0 values.

=sumXmY2({2,5}, {6,1})  returns 32

---

**tan(x)**

Returns the tangent of x. The argument is in radians.

=tan(pi()/4)  returns 1

---

**tanh(x)**

Returns the hyperbolic tangent of x.

=tanh(1)  returns 0.76159415595576

---

**trunc(x, [digits])**

trunc(x)

Removes the fractional part of x to the precision specified by the number of digits to the left of

the decimal point. The default value of the 'digit' argument is 0.

=trunc(8.9,)  returns 8

=trunc(-8.9,)  returns -8

=trunc(1.256, 2)  returns 1.25

=trunc(8.9)  returns 8

=trunc(1.256)  returns 1

## Text Functions

---

**char(n)**

Returns the character specified by n.

=char(65)  returns "A"

---

**clean(text)**

Removes non-printable characters from text.

=clean(char(7) & "abc")  returns "abc".

---

**code(text)**

Returns the code of the first character in text.

=code("ABC")  returns 65

---

**concatenate(v1, v2, ...)**

Merges all strings specified by the arguments which can be text strings, numbers or arrays. Empty cells in arrays are ignored.

=concatenate("abc", 1, "abc", 2)  returns "abc1abc2"

---

**dollar(x, [decimals])**

Converts x to a text string using the default currency format and the specified precision. If the 'digits' argument is greater than 0, then x is rounded to the specified number of decimal places. If it's 0, then x is rounded to the nearest integer. If it's less than 0, then x is rounded to the left of the decimal point. The default value of 'digits' is 2.

=dollar(10.95,)  returns "$10.95"

=dollar(10.9487, 3)  returns "$10.949"

---

**exact(text1, text2)**

Returns 1 if text1 and text2 are the same, 0 otherwise. The comparison is case-sensitive.

=exact("Abc", "abc")  returns 0

=exact("Abc", "Abc")  returns 1

---

**find(pattern, subject, [start])**

find(pattern, subject, [start], [flags])

Searches 'subject' for 'pattern' starting at the position 'start' and returns the position of the first found occurrence (except for the "RegExStr" flag). If 'pattern' is not found or if 'start' is out of the valid range, it returns the #VALUE! error value.

The first version accepts a plain text string as 'pattern' and the comparison is case-sensitive. The second version also accepts regular expressions as 'pattern' and can perform both case-sensitive and caseless comparisons. The 'flags' parameter can be any combination of the following options:
SEARCH::RegEx (or 8192) - 'pattern' is a regular expression,
SEARCH::RegExStr (or 131072) - 'pattern' is a regular expression but the matching substring is returned instead of its position,
SEARCH::CaseSensitive (or 128) - use case-sensitive string comparison (regular expressions can overwrite this).

=find("text", "sample text", 1)  returns 8
=find("e*\d", "abcdef abcee abcde5", 1, SEARCH::RegEx)  returns 18

---

**fixed(x, [digits], [no_separators])**

Rounds x to the specified number of digits and formats it using the general format. If the 'digits' argument is greater than 0, then x is rounded to the specified number of decimal places. If it's 0, then x is rounded to the nearest integer. If it's less than 0, then x is rounded to the left of the decimal point. If no_separators is 0, the number is formatted using thousand separators. If 'digits' is omitted, it's assumed to be 2. The default value of 'no_separators' is 0.

=fixed(10.956,,)  returns "10.96"

=fixed(1234.89, -2, 0)  returns "1,200"

---

**left(text, [n])**

Returns the first n characters from text. If n is omitted, it's assumed to be 1. If n is greater than the number of characters in text, the entire text is returned.

=left("abc",)  returns "a"

---

**len(text)**

Returns the number of characters in text.

=len("text")  returns 4

---

**lenb(text)**

Returns the number of bytes in text. Since internally text are stored as UTF-8 strings, the number of bytes will be larger than then number of characters for strings containing non-ASCII characters (characters with codes above 127).

=lenb("örtlich")  returns 8

---

**lower(text)**

Converts all letters in text to lowercase.

=lower("TEXT")  returns "text"

---

**mid(text, n1, n2)**

Returns (up to) n2 characters from text starting at the position n1. If n1 is greater than the number of characters in text, it returns an empty string. If n1 is less than 1 or if n2 is negative, mid returns the #VALUE! error value.

=mid("some text", 3, 2)  returns "me"

---

**proper(text)**

Converts the first character of each word in text to uppercase and all other characters to lowercase.

=proper("some teXT")  returns "Some Text"

---

**replace(subject, n1, n2, replace)**

replace(pattern, subject, start, replace, [flags])

The first version removes 'n2' characters from 'subject' at the position 'n1' and replaces them with 'replace'.

The second version searches 'subject' for 'pattern' starting at the position 'start' and replaces all found occurrences with 'replace'.The 'pattern' parameter can be either a plain text string or a regular expression. The 'flags' parameter can be any combination of the following options:
SEARCH::RegEx (or 8192) - 'pattern' is a regular expression,
SEARCH::CaseSensitive (or 128) - use case-sensitive string comparison (regular expressions can overwrite this).

If SEARCH::RegEx is specified, the 'replace' argument can contain:
(1) absolute references (by number) to capturing subpatterns, eg. \1, \2...\999
(2) \l, \L, \u, \U literals to (binary) switch upper- and lower-case conversion,
(3) \r, \n - 'line feed' and 'new line' literals (by default, pressing Ctrl+Enter when editing cells inserts the \r\n sequence).

=replace("abcd", 3, 2, "*")  returns "ab*"
=replace("(.)a+\d{1,3}", "abc aa0102", 1, "\1", SEARCH::RegEx)  returns 'abc 2'
=replace("(ab)", "abcdef ghijk abb123", 1, "\u\1", SEARCH::RegEx)  returns "ABcdef ghijk ABb123"

---

**rept(text, n)**

Duplicates text n times.

=rept("abc", 3)  returns "abcabcabc"

---

**right(text, [n])**

Returns the last n characters from text. If n is omitted, it's assumed to be 1. If n is greater than the number of characters in text, the entire text is returned.

=right("abc",)  returns "c"

---

**search(searchText, withinText, [n])**

Finds the first occurrence of searchText in withinText and returns its position. It starts searching at the position n. If n is omitted, it's assumed to be 1 (the 1st character).The seachText argument can represent a search pattern containing special characters '?' (any character) or '*' (any string, including an empty string). To search for '?' or '*' place a tilde (~) before them.

=search("text", "some text", 1)  returns 6

=search("?bc?e*", "abc abcd abcde ab",)  returns 10

---

**substitute(searchText, withinText, replaceText, [n]**

Finds the n-th occurrence of searchText in withinText and substitutes searchText for replaceText. To replace all occurrences of searchText, type 0 as the n parameter. The default value of n is 0.

=substitute("a1b1c1d", "1", "-", 0)  returns "a-b-c-d"

=substitute("abc abcd abcde ab", "?bc?e*a", "x", 1)  returns "abc abcd xb"

---

**t(value)**

If the 'value' parameter is a text string, the 't' function returns that string. Otherwise it returns an empty string.

---

**text(value, format)**

Format a given number or text using the specified format code.

If the function is used to format a date, the "value" argument has to be either a date/time serial number or a generic date/time string.
For more information about format codes, please see the "Formatting: Style" help topic.

=text(-2345.4, "\$#,##0.00")  returns -$2,345.40

=text(-2345.4, "#,##0.00;\(#,##0.00\)")  returns (2,345.40)

=text("bc", "\a@\d")  returns "abcd"

=text(date(2009,10,21), "mmmm\ d\,\ yyyy")  returns October 21, 2009

=text("2009-10-21", "mmmm\ d\,\ yyyy")  returns October 21, 2009

---

**trim(text)**

Removes all whitespaces from text except single spaces between words.

=trim(" abc def  ghi")  returns "abc def ghi"

---

**trimE(text)**

Removes leading and trailing spaces from a given text string. As opposed to "trim()", this function doesn't compress the inner spaces.

=trimE("  abc def  ")  returns "abc def"

---

**trimL(text)**

Removes leading spaces from a given text string.

=trimL("  abc def  ")  returns "abc def  "

---

**trimR(text)**

Removes trailing spaces from a given text string.

=trimR("  abc def   ")    returns "  abc def"

---

**upper(text)**

Converts all letters in text to uppercase.

=upper("text")    returns "TEXT"

---

**value(text)**

Converts text to a number.

=value("123.45")    returns 123.45

=value("10 4/5")    returns 10.8

=value("$1,000")    returns 1000

## Date/Time Functions

---

**date(year, month, day)**

Returns a date serial number.

A date/time serial number is a floating-point value representing a date between January 1, 100 and December 31, 9999. The integer part of that value represents days and the fractional part represents the time of the day. The value 1.0 represents December 31, 1899. Negative numbers represent dates prior to December 30, 1899.

=date(2009,10,10)   returns 40096

=date(1899,12,30)  returns 0

=date(100,1,1)  returns -657434

=date(1800,1,1) - time(1,1,1)  returns 36522.0423726852 (1/1/1800 1:01)

---

**dateDiff(startDateString, endDateString)**

Returns a period string representing the difference between two dates.

The date parameters must be specified as generic date/time strings.

A generic date/time/period string is a string that conforms to the www.w3.org specification for date/time/period data types, for example:
2006-01-31
2006-01-31T13:10:55
2006-01-31T13:10:55.123
2006-01-31T23:30:00+02:00
13:10:00
13:10:55.123
13:10:55-00:30
P1Y2M3DT10H30M50S.000 (which means a period of: 1 year, 2 months, 3 days, 10 hours, 30 minutes, 50 seconds, 0 milliseconds)
-P120D (which means a period of minus 120 days)
PT63H (which means 63 hours)

=dateDiff("2005-01-01", "2005-12-31")  returns "P364D"

=dateDiff("2005-01-01T18:15:10", "2005-01-04T12:10:00")  returns "P2DT17H50M50S"

=dateDiff(dateText(2005, 1, 1), dateText(2005, 12, 31))  returns "P364D"

---

**dateText(year, month, day)**

Returns a generic date string representing a given date.

A generic date/time string is a string that conforms to the www.w3.org specification for date/time/period data types, for example:

2006-01-31
2006-01-31T13:10:55
2006-01-31T13:10:55.123
2006-01-31T23:30:00+02:00
13:10:00
13:10:55.123
13:10:55-00:30

=dateText(2010, 7, 15)  returns "2010-07-15"

---

**dateValue(text)**

Converts formatted date/time text to a date serial number string. If the text doesn't contain the date year, the current year will be used.

A date/time serial number is a floating-point value representing a date between January 1, 100 and December 31, 9999. The integer part of that value represents days and the fractional part represents the time of the day. The value 1.0 represents December 31, 1899. Negative numbers represent dates prior to December 30, 1899.

=dateValue("8/11/2005")  returns 38575

=dateValue("Apr-15")  returns 39918

=dateValue("Sep, 2008")  returns 40422

---

**day(dateNumber)**

day(dateString)

Returns an integer <1, 31> representing the day of the month of a given date.

The date parameter can be specified either as a date serial number or as a generic date/time/period string.

A date/time serial number is a floating point value representing a date between January 1, 100 and December 31, 9999. The integer part of that value represents days and the fractional part represents the time of the day. The value 1.0 represents December 31, 1899. Negative numbers represent dates prior to December 30, 1899.

A generic date/time string is a string that conforms to the www.w3.org specification for date/time/period data types, for example:
2006-01-31
2006-01-31T13:10:55
2006-01-31T13:10:55.123
2006-01-31T23:30:00+02:00
13:10:00
13:10:55.123
13:10:55-00:30

=day(date(2010,2,9))  returns 9

=day(dateValue("Apr-15"))  returns 15

=day("2010-02-09")  returns 9

---

**eDate(dateNumber, n)**

eDate(dateString, period)

The date parameter can be specified either as a date serial number or as a generic date/time/period string.

The first version returns a date serial number representing a date n months after (n>0) or before (n<0) a given date.
The second version returns a generic date/time string representing a date after or before - depending on the period - a given date

A date/time serial number is a floating point value representing a date between January 1, 100 and December 31, 9999. The integer part of that value represents days and the fractional part represents the time of the day. The value 1.0 represents December 31, 1899. Negative numbers represent dates prior to December 30, 1899.

A generic date/time/period string is a string that conforms to the www.w3.org specification for date/time/period data types, for example:
2006-01-31
2006-01-31T13:10:55
2006-01-31T13:10:55.123
2006-01-31T23:30:00+02:00
13:10:00
13:10:55.123
13:10:55-00:30

=eDate(date(2005,2,15), 4)  returns  38518

=eDate(38382, 1)  returns 38411

=eDate("2005-01-01", "P2D")  returns "2005-01-03"

=eDate("2005-02-01", "P30D")  returns "2005-03-03"

=eDate("2005-03-03", "-P30D")  returns "2005-02-01"

=eDate("13:34:22.33", "PT1H7M")  returns "14:41:22.330

=eDate("2005-08-10T15:27:36.10", "P10DT3H7M12.95S")  returns "2005-08-20T18:34:49.050"

---

**eoMonth(dateNumber, n)**

eoMonth(dateString, period)

The date parameter can be specified either as a date serial number or as a generic date/time/period string.

The first version returns a date serial number representing a date of the last day of the month n months after (n>0) or before (n<0) a given date.

The second version returns a generic date/time string representing a date of the last day of the month after or before a given period.

A date/time serial number is a floating point value representing a date between January 1, 100 and December 31, 9999. The integer part of that value represents days and the fractional part represents the time of the day. The value 1.0 represents December 31, 1899. Negative numbers represent dates prior to December 30, 1899.

A generic date/time/period string is a string that conforms to the www.w3.org specification for date/time/period data types, for example:
2006-01-31
2006-01-31T13:10:55
2006-01-31T13:10:55.123
2006-01-31T23:30:00+02:00
13:10:00
13:10:55.123
13:10:55-00:30

=eoMonth(date(2005,1,1), 1)  returns 38411

=eoMonth(date(2010,1,1), -1)  returns 40178

=eoMonth("2005-01-01", "-P1M")  returns "2004-12-31"

=eoMonth("2010-01-01", "-P250D")  returns "2009-04-30"

---

**hour(timeNumber)**

hour(timeString)

Returns an integer <0, 23> representing the hour of the time.

The time parameter can be specified either as a date/time serial number or as a generic date/time string.

A date/time serial number is a floating point value representing a date between January 1, 100 and December 31, 9999. The integer part of that value represents days and the fractional part represents the time of the day. The value 1.0 represents December 31, 1899. Negative numbers represent dates prior to December 30, 1899.

A generic date/time string is a string that conforms to the www.w3.org specification for date/time/period data types, for example:
2006-01-31
2006-01-31T13:10:55
2006-01-31T13:10:55.123
2006-01-31T23:30:00+02:00
13:10:00
13:10:55.123
13:10:55-00:30

=hour(time(12,59,11))  returns 12

=hour(date(2010,8,8) + time(20,0,1.01))  returns 20

=hour("12:59:11")  returns 12

=hour("2010-08-08T20:00:01.01")  returns 20

---

**minute(timeNumber)**

minute(timeString)

Returns an integer <0, 59> representing the minute of the time.

The time parameter can be specified either as a date/time serial number or as a generic date/time string.

A date/time serial number is a floating point value representing a date between January 1, 100 and December 31, 9999. The integer part of that value represents days and the fractional part represents the time of the day. The value 1.0 represents December 31, 1899. Negative numbers represent dates prior to December 30, 1899.

A generic date/time string is a string that conforms to the www.w3.org specification for date/time/period data types, for example:
2006-01-31
2006-01-31T13:10:55
2006-01-31T13:10:55.123
2006-01-31T23:30:00+02:00
13:10:00
13:10:55.123
13:10:55-00:30

=minute(time(12,59,11))  returns 59

=minute(date(2010,8,8) + time(20,0,1.01))  returns 0

=minute("12:59:11")  returns 59

=minute("2005-08-08T20:00:01.01")  returns 0

---

**month(dateNumber)**

month(dateString)

Returns an integer <1, 12> representing the month of a given date.

The date parameter can be specified either as a date/time serial number or as a generic date/time string.

A date/time serial number is a floating point value representing a date between January 1, 100 and December 31, 9999. The integer part of that value represents days and the fractional part represents the time of the day. The value 1.0 represents December 31, 1899. Negative numbers represent dates prior to December 30, 1899.

A generic date/time string is a string that conforms to the www.w3.org specification for date/time/period data types, for example:

2006-01-31
2006-01-31T13:10:55
2006-01-31T13:10:55.123
2006-01-31T23:30:00+02:00
13:10:00
13:10:55.123
13:10:55-00:30

=month(date(2010,1,1))  returns 1

=month("2005-01-01")  returns 1

---

**networkDays(startDateNumber1, endDateNumber2, [offDates])**

networkDays2(startDateString1, endDateString2, offDates)

Returns the number of working days between two dates (inclusive) excluding weekends and dates specified in the offDates array.

The date parameters can be specified either as date serial numbers or as generic date/time strings.

A date/time serial number is a floating point value representing a date between January 1, 100 and December 31, 9999. The integer part of that value represents days and the fractional part represents the time of the day. The value 1.0 represents December 31, 1899. Negative numbers represent dates prior to December 30, 1899.

A generic date/time string is a string that conforms to the www.w3.org specification for date/time/period data types, for example:
2006-01-31
2006-01-31T13:10:55
2006-01-31T13:10:55.123
2006-01-31T23:30:00+02:00
13:10:00
13:10:55.123
13:10:55-00:30

=networkDays(date(2010,1,1), date(2010,1,31),)  returns 21

=networkDays(date(2010,1,1), date(2010,1,31), {40190, 40197})  returns 19

=networkDays("2010-01-01", "2010-01-31",)  returns 21

=networkDays("2010-01-01", "2010-01-31", {"2010-01-12", "2010-01-19"})  returns 19

---

**NETWORKDAYS.INTL(startDateNumber1, endDateNumber2, [offDates], [weekends])**

See the description online.

---

**now()**

Returns the current date and time as a date/time serial number.

=now()  returns 40116.6196296296

---

**nowText()**

Returns the current date and time as a generic date/time string.

=nowText()  returns "2009-10-30T14:52:16"

---

**second(timeNumber)**

second(timeString)

Returns an integer <0, 59> representing the second of the time value.

The time parameter can be specified either as a date/time serial number or as a generic date/time string.

A date/time serial number is a floating point value representing a date between January 1, 100 and December 31, 9999. The integer part of that value represents days and the fractional part represents the time of the day. The value 1.0 represents December 31, 1899. Negative numbers represent dates prior to December 30, 1899.

A generic date/time string is a string that conforms to the www.w3.org specification for date/time/period data types, for example:
2006-01-31
2006-01-31T13:10:55

2006-01-31T13:10:55.123
2006-01-31T23:30:00+02:00
13:10:00
13:10:55.123
13:10:55-00:30

=second(time(12,59,11))  returns 11

=second(date(2010,8,8) + time(20,0,1.01))  returns 1

=second("12:59:11")  returns 11

=second("2010-08-08T20:00:01.01")  returns 1

---

**time(hour, minute, second)**

Returns a date/time serial number representing a given time.

A date/time serial number is a floating-point value representing a date between January 1, 100 and December 31, 9999. The integer part of that value represents days and the fractional part represents the time of the day. The value 1.0 represents December 31, 1899. Negative numbers represent dates prior to December 30, 1899.

=time(16, 20, 15)  returns 0.68072916667006

---

**timeText(hour, minute, second)**

Returns a generic time string representing a given time.

A generic date/time string is a string that conforms to the www.w3.org specification for date/time/period data types, for example:
2006-01-31
2006-01-31T13:10:55
2006-01-31T13:10:55.123
2006-01-31T23:30:00+02:00
13:10:00
13:10:55.123
13:10:55-00:30

=timeText(16, 20, 15)  returns "16:20:15"

---

**timeValue(text)**

Converts formatted date/time text to a time serial number.

A date/time serial number is a floating-point value representing a date between January 1, 100 and December 31, 9999. The integer part of that value represents days and the fractional part represents the time of the day. The value 1.0 represents December 31, 1899. Negative numbers represent dates prior to December 30, 1899.

=timeValue("6:24 PM")  returns 0.76666666667006

---

**today()**

Returns a date serial number representing the current date.

A date/time serial number is a floating-point value representing a date between January 1, 100 and December 31, 9999. The integer part of that value represents days and the fractional part represents the time of the day. The value 1.0 represents December 31, 1899. Negative numbers represent dates prior to December 30, 1899.

=today()  returns 40055

---

**todayText()**

Returns a generic date string representing the current date.

A generic date/time string is a string that conforms to the www.w3.org specification for date/time/period data types, for example:
2006-01-31
2006-01-31T13:10:55
2006-01-31T13:10:55.123
2006-01-31T23:30:00+02:00
13:10:00

13:10:55.123
13:10:55-00:30

=todayText()  returns "2009-08-30"

---

**weekDay(dateNumber, [numbering])**

weekDay(dateString, [numbering])

Returns the day of the week for a given date. The numbering argument specifies which day numbering variant should be used:
1 - Sunday=1,...,Saturday=7
2 - Monday=1,...,Sunday=7
3 - Monday=0,...,Sunday=6.
If numbering is omitted, it's assumed to be 1.

The date parameter can be specified either as a date/time serial number or as a generic date/time string.

A date/time serial number is a floating point value representing a date between January 1, 100 and December 31, 9999. The integer part of that value represents days and the fractional part represents the time of the day. The value 1.0 represents December 31, 1899. Negative numbers represent dates prior to December 30, 1899.

A generic date/time string is a string that conforms to the www.w3.org specification for date/time/period data types, for example:
2006-01-31
2006-01-31T13:10:55
2006-01-31T13:10:55.123
2006-01-31T23:30:00+02:00
13:10:00
13:10:55.123
13:10:55-00:30

=weekDay(date(2010,1,1),)  returns 6

=weekDay("2010-01-01",)  returns 6

---

**weekNum(dateNumber, [numbering])**

weekNum(dateString, [numbering])

Returns the number of the week for a given date. The numbering argument specifies on what day the week should begin:
1 - on Sunday
2 - on Monday.
If numbering is omitted, it's assumed to be 1.

The date parameter can be specified either as a date/time serial number or as a generic date/time string.

A date/time serial number is a floating point value representing a date between January 1, 100 and December 31, 9999. The integer part of that value represents days and the fractional part represents the time of the day. The value 1.0 represents December 31, 1899. Negative numbers represent dates prior to December 30, 1899.

A generic date/time string is a string that conforms to the www.w3.org specification for date/time/period data types, for example:
2006-01-31
2006-01-31T13:10:55
2006-01-31T13:10:55.123
2006-01-31T23:30:00+02:00
13:10:00
13:10:55.123
13:10:55-00:30

=weekNum(date(2010,1,21),)  returns 4

=weekNum("2010-01-21",)  returns 4

---

**workDay(dateNumber, n, [offDateList])**

workDay(dateString, n, [offDateList])

The date parameters can be specified either as date/time serial numbers or as generic date/time strings.

The first version returns a date serial number representing a date n working days before (n<0) or after (n>0) a given day excluding weekends. The optional offDates array contains additional dates that should be excluded. All arguments must be in the form of date/time serial numbers.

The second version returns a generic date/time string and all the input dates must be generic date/time strings.

A date/time serial number is a floating point value representing a date between January 1, 100

and December 31, 9999. The integer part of that value represents days and the fractional part represents the time of the day. The value 1.0 represents December 31, 1899. Negative numbers represent dates prior to December 30, 1899.

A generic date/time string is a string that conforms to the www.w3.org specification for date/time/period data types, for example:
2006-01-31
2006-01-31T13:10:55
2006-01-31T13:10:55.123
2006-01-31T23:30:00+02:00
13:10:00
13:10:55.123
13:10:55-00:30

=workDay(date(2010,1,1), 21,)  returns 40210

=workDay(date(2010,1,1), 19, {40190, 40197})  returns 40210

=workDay("2010-01-01", 21,)  returns "2010-02-01"

=workDay("2010-01-01", 19, {"2010-01-12", "2010-01-19"})  returns "2010-02-01"

---

**WORKDAY.INTL(start_date, days, [weekend], [holidays])**

See the description online.

---

**year(dateNumber)**

year(dateString)

Returns an integer <100, 9999> representing the year of a given date.

The date parameter can be specified either as a date/time serial number or as a generic date/time string.

A date/time serial number is a floating point value representing a date between January 1, 100 and December 31, 9999. The integer part of that value represents days and the fractional part represents the time of the day. The value 1.0 represents December 31, 1899. Negative numbers represent dates prior to December 30, 1899.

A generic date/time string is a string that conforms to the www.w3.org specification for

date/time/period data types, for example:
2006-01-31
2006-01-31T13:10:55
2006-01-31T13:10:55.123
2006-01-31T23:30:00+02:00
13:10:00
13:10:55.123
13:10:55-00:30

=year(date(2010,1,1))  returns 2010

=year(date(9999,1,1) + time(15,10,00))  returns 9999

=year("2010-01-01")  returns 2010

=year("40000-01-01T15:10:00")  returns 40000

---

**yearDay(dateNumber)**

yearDay(dateString)

Returns the year day number for a given date.

The date parameter can be specified either as a date/time serial number or as a generic date/time string.

A date/time serial number is a floating point value representing a date between January 1, 100 and December 31, 9999. The integer part of that value represents days and the fractional part represents the time of the day. The value 1.0 represents December 31, 1899. Negative numbers represent dates prior to December 30, 1899.

A generic date/time string is a string that conforms to the www.w3.org specification for date/time/period data types, for example:
2006-01-31
2006-01-31T13:10:55
2006-01-31T13:10:55.123
2006-01-31T23:30:00+02:00
13:10:00
13:10:55.123
13:10:55-00:30

=yearDay(date(2010,1,1))  returns 1

=yearDay(date(9999,1,1) + time(15,10,00))  1

=yearDay("2010-07-04")  returns 185

## Financial Functions

---

**cumIPMT(rate, nper, pv, n1, n2)**

Returns the cumulative interest paid on a loan between the periods 'startPeriod' and 'endPeriod', based on a given interest rate (relative to one period), the total number of periods 'nper', the present value 'pv' and payments made in arrears.

=cumIPMT(0.0075, 360, 125000, 13, 24)  returns -11135.2321307508

---

**cumPRINC(rate, period, pv, n1, n2)**

Returns the cumulative principal paid on a loan between the periods 'startPeriod' and 'endPeriod', based on a given interest rate (relative to one period), the total number of periods 'nper', the present value 'pv' and payments made in arrears.

=cumPRINC(0.0075, 360, 125000, 13, 24)  returns  -934.107123420876

---

**db(cost, salvage, life, period, [month])**

Returns the depreciation of an asset for a given period using the fixed-declining balance method, where 'cost' is the initial cost of an asset, 'salvage' is the value at the end of the depreciation, 'life' is the total number of depreciation periods, 'period' is the period for which the depreciation is calculated, 'month' is the number of months in the first year. The default value of 'month' is 12.

=db(1000000, 100000, 6, 1, 7)  returns 185,912.959716189

=db(1000000, 100000, 6, 7, 7)  returns 15,867.888384391

---

**ddb(cost, salvage, life, period, [factor])**

Returns the depreciation of an asset for a given period using the double-declining balance method or other method determined by the optional 'factor' declining rate, where 'cost' is the initial cost of an asset, 'salvage' is the value at the end of the depreciation, 'life' is the total number of depreciation periods, 'period' is the period for which the depreciation is calculated, 'factor' is the rate at which the balance declines. The default value of 'factor' is 2.

=ddb(2400, 300, 3650, 1,)  returns 1.31506849315068

=ddb(2400, 300, 10, 2, 1.5)  returns 306

---

**dollarDe(x, denominator)**

Converts a dollar price represented by a fraction with a given denominator value to a decimal number.

=dollarDe(1.02, 16)  returns 1.125 (=1 2/16)

---

**dollarFr(x, denominator)**

Converts a decimal number to a dollar price represented by a fraction with a given denominator value.

=dollarFr(1.125, 16)  returns 1.02 (=1 2/16)

---

**effect(rate, nper)**

Calculates and returns the annual effective interest rate based on the nominal annual interest rate and the number of compounding periods per year 'nper'.

=effect(5.25%, 4)  returns 0.05354266737076

---

**fv(rate, pmt, [pv], [type])**

Returns the future value of an investment based on constant periodic payments and interest rate where 'rate' is the interest rate (relative to one period), 'pmt' is the periodic payment, 'pv' is the present value of the investment and and 'type' specifies the payment type: 0 - at the end of each period, 1 - at the beginning of each period. If the 'pmt' argument is omitted, the 'pv' value must be specified. If the 'pv' argument is omitted, it's assumed to be 0 and the 'pmt' value must be specified. The default value of 'type' is 0.

=fv(0.5%, 10, -200, -500, 1)  returns 2581.40337406014

---

**fvSchedule(pv, v1, v2, ...)**

Returns the future value of an initial principal after the specified number of periods at a variable interest rate in each period. The 'v_' arguments are numbers or array(s) of numbers representing the subsequent interest rates. Empty cells are treated as zeros.

=fvSchedule(1, {0.09, 0.11, 0.1})  returns 1.33089

---

**ipmt(rate, period, nper, pv, [fv])**

Returns the interest payment for a given period based on constant periodic payments and a constant interest rate where 'nper' is the total number of periods, 'pv' is the present value or the current value of the future payments, 'fv' is the future value (after the last payment is made) and payments occur at the end of each period. If 'fv' is omitted, it's assumed to be 0.

=ipmt(0.1/12, 1, 36, 8000,)  returns -66.6666666666667

---

**irr(pv1, pv2, ..., guess)**

Returns the internal interest rate for a series of (negative and positive) cash flows occurring at regular intervals. The pv(n) arguments can numbers or arrays of numbers, 'guess' is the value of

the rate that irr will use as the initial approximation of the actual rate. Empty cells in arrays are ignored.

=irr(-70000, 12000, 15000, 18000, 21000, 26000, 0%)  returns 0.08663094803653

---

**mirr(flowsArray, rate)**

Returns the modified internal rate of return for a series of (negative and positive) cash flows assuming that the obtained cash can be reinvested at a given interest rate (relative to one period). Empty cells in arrays are ignored. To evaluate an investment, compare that value with the interest rate paid on the capital.

=mirr({-120000, 39000, 30000, 21000, 37000, 46000}, 12%)  returns 0.12609413036591

---

**nominal(rate, nper)**

Calculates and returns the annual nominal interest rate, based on a given effective rate and the number of compounding periods per year.

=nominal(5.3543%, 4)  returns 0.05250031986836

---

**nper(rate, pmt, pv, [fv], [type])**

Returns the total number of periods for an investment based on constant periodic payments and a constant interest rate. The 'pmt' argument is the periodic payment, 'pv' is the present value, 'fv' is a cash balance after the last payment is made and 'type' specifies the payment type: 0 - at the end of each period, 1 - at the beginning of each period. If 'fv' is omitted, it's assumed to be 0. The default value of 'type' is 0.

=nper(12%/12, -100, -1000, 10000, 1)  returns 59.6738656742946

=nper(1%, -100, 1000,,)  returns 10.5886444594232

---

**npv(rate, v1, v2, ...)**

Returns the net present value of an investment, given the 'rate' discount rate and a series of future payments and income occurring at regular intervals in arrears. The 'rate' argument is the discount rate (relative to one period), the 'v_' values are numbers or arrays of numbers representing payments or income. Empty cells are ignored.

=npv(10%, -10000, 3000, 4200, 6800)  returns 1188.44341233522

---

**pmt(rate, nper, pv, [fv], [type])**

Returns the payment for a loan based on constant payments and constant interest rate. The 'rate' argument is the interest rate, 'nper' is the total number of payment periods, 'pv' the present value, 'fv' is a cash balance after the last payment is made and 'type' specifies the payment type: 0 - at the end of each period, 1 - at the beginning of each period. If 'fv' is omitted, it's assumed to be 0. The default value of 'type' is 0.

=pmt(8%/12, 10, 10000,,)  returns -1037.03208935916

=pmt(8%/12, 10, 10000, 0, 1)  returns -1030.16432717798

---

**ppmt(rate, period, nper, pv, [fv], [type])**

Returns the principal portion of a given payment. The 'rate' argument is the interest rate (relative to one period), 'period' is the period that the payment refers to, 'nper' is the total number of periods, 'pv' is the present value, 'fv' is a cash balance after the last payment is made and 'type' specifies the payment type: 0 - at the end of each period, 1 - at the beginning of each period. If 'fv' is omitted, it's assumed to be 0. The default value of 'type' is 0.

=ppmt(10%/12, 1, 24, 2000,,)  returns -75.6231860083666

---

**pv(rate, nper, pmt, [fv], [type])**

Returns the present value of an investment where 'rate' is the interest rate (relative to one period), 'nper' is the total number of compounding periods, 'pmt' is the periodic payment, 'fv' is the future value and 'type' specifies the payment type: 0 - at the end of each period, 1 - at the beginning of

each period. If 'fv' is omitted, it's assumed to be 0. The default value of 'type' is 0.

=pv(0.08/12, 12*20, 500, , 0)  returns -59777.1458511878

---

**rate(nper. pmt, pv, [fv], [type])**

Returns the average interest rate per period. The 'nper' argument is the total number of periods, 'pmt' is the periodic payment, 'pv' is the present value, 'fv' is the future value, 'type' is the payment type: 0 - at the end of each period, 1 - at the beginning of each period, 'guess' is the value of the rate that irr will use as the initial approximation of the actual rate. If 'fv' is omitted, it's assumed to be 0. The default value of 'type' is 0.

=rate(48, -200, 8000,,,)  returns 0.0077014724882

---

**sln(cost, salvage, life)**

Returns the straight-line depreciation of an asset for one period. The 'cost' argument is initial values of an asset, 'salvage' is the value at the end of the depreciation, 'life' is the total number of depreciation periods.

=sln(30000, 7500, 10)  returns 2250

---

**syd(cost, salvage, life, period)**

Returns the sum-of-years' digits depreciation of an asset for a given period. The 'cost' argument is the initial value of an asset, 'salvage' is the value at the end of the depreciation period, 'life' is the total number of depreciation periods, 'period' is the period for which the SYD value is calculated.

=syd(30000,7500,10,1)  returns 4090.909090909

---

**xirr(flows, dates, [guess])**

Returns the internal rate of return for a schedule of cash flows without the periodicity constraint. The 'flows' argument is an array of numbers representing cash flows and the 'dates' argument is an array of generic date/time strings indicating when the corresponding cash flows occur. The first element of the 'flows' array is optional and represents the initial cost/investment. The first element of the 'dates' array specifies the beginning of the schedule; the remaining dates can be placed in any other but they must be later than the first one. The 'guess' argument is the value of the rate that xirr will use as the initial approximation of the actual rate. The default value of 'guess' is 10%.

=xirr({-10000, 2750, 4250, 3250, 2750}, {"1992-01-01", "1992-03-01", "1992-10-30", "1993-02-15", "1993-04-01"}, 0.1)  returns  0.37336253351883

---

**xnpv(rate, flows, dates)**

Returns the net present value of an investment for a schedule of cash flows without the periodicity constraint. The 'rate' argument is the discount rate, 'flows' is an array of numbers representing cash flows and the 'dates' argument is an array of generic date/time strings indicating when the corresponding cash flows occur. The first element of the 'flows' array is optional and represents the initial cost/investment. The first element of the 'dates' array specifies the beginning of the schedule; the remaining dates can be placed in any other but they must be later than the first one.

=xnpv(0.09, {-10000, 2750, 4250, 3250, 2750}, {"1992-01-01", "1992-03-01", "1992-10-30", "1993-02-15", "1993-04-01"})  returns 2086.64760205153

## Lookup & Reference Functions

---

**address([row], [column], [type], [mode], [path])**

address([row], [column])

Creates and returns a string representing a cell address for the specified column and/or row numbers.

If either of the two numbers is omitted, a cell range is returned.

The 'type' argument specifies the reference type for the row/column pair:
1 - absolute,
2 - absolute/relative,
3 - relative/absolute,
4 - relative.
If it's omitted, it's assumed to be 1.

The 'mode' argument specifies the address notation:
0 - the RC notation,
1 - the A1 notation.
The default value of 'mode' is 1.

The optional 'path' argument specifies the (tree) path of the worksheet that the created address refers to.

=address(1, 2,,,)  returns $B$1

=address(1, 2)  returns $B$1

=address(1,)  returns $1:$1

=address(,2)  returns $B:$B

=address(1, 2, 4, 0, "folder1\sheet2")  returns "folder1\sheet2"!B1

=address(2,,1,0,)  returns R2 (the entire 2nd row)

---

**areas(v1, v2, ...)**

Returns the number of separated ranges that the argument list points to.

---

**choose(n, v1, v2, ...)**

Returns the n-th element from the list of the 'v_' arguments.

=choose(2, A1:A10, B1:B10, C1:C10)  returns B1:B10

---

**column(reference)**

Returns the column number of 'reference'.

=column(B1)  returns 2

=column(A1:C5)  returns {1, 2, 3}

---

**columns(array)**

Returns the number of columns in 'array'.

=columns({1, 2, 3; 4, 5, 6})  returns 3

---

**filter(search-range, filters, hyperlink-path, options[, empty-string])**

The FILTER() function enables you to filter data with the number of filters up to the max.number of columns in a worksheet, sort the results using up to six keys and pre - or post - filter the data to find compound duplicates consisting of up to 100 keys / columns.

For details, please see the "FILTERING TABLES" help topic.

The "search-range" argument represents a range(or an array) with rows to filter.It can be of any size up to the maximum number of columns in a worksheet minus one(as the 1st output column may additionally contain hyperlinks).
If there are any rows found in the search range, FILTER() returns an array with the corresponding number of columns.

The "filters" argument represents a range or an array with filtering expressions for the subsequent columns in the "search-range" range. Thus the number of cells in "filters" must not exceed the number of columns in the "search-range" range.The n - th cell position in this range / array is a filter expression for the the n - th column in the "search-range" data range.
If there is no filter for a given column, the corresponding filter cell can be left empty.

Entering filters in worksheet cells is easy : you can use the Format > Search Filter format style for the desirable cells.

The "hyperlink-path" argument represents a worksheet/workbook path that will be be used along with cell references if the "include hyperlinks" options is specified(see below).
For example : sheet1, folder1\sheet1, c:\documents\[sample.gsc]sheet1
It can be left empty if both the data and the FILTER() functions are in the same worksheet.

The "option" argument is a number and can be 0 or a combination(a sum from 1 to 7) of the following:
1 - include hyperlinks to the original search - range data rows in the first column in of the FILTER() function results;
2 - perform searching for duplicates AFTER all other filters are applied; by default the searches for duplicates are performed first;
4 - as FILTER() is supposed to offer the best possible speed when handling very large tables, by default it doesn't create calculation chains with formulas placed in the "search - range" range; if e.g. some column in the "search - range" consists of formulas and you want to make sure FILTER() is executed after they're updated, add 4 to options. Please note that this will result in significantly slower filtering.

The optional "empty-string" argument represents a number or a string that the FILTER() function is to return if no matching rows are found for the specified filters. If it's omitted, FILTER() returns the "#N/A!" error code in such a case.

---

**hLookUp(v, array, n, [type])**

hLookUp(v, array, n, [type], [start], [occurrence])

Searches the top row of 'array' for 'v' and - if found - returns a value from the same column and the n-th row of 'array'.

The 2nd hLookUp() variant uses two additional parameters:
'start' - positive numbers specify from which cell the searching should start and negative numbers specify where it should end. For the first top-left cell of the searched range 'start'=1, for the 2nd one 'start'=2 etc. The bottom-right cell has the index of -1, for the preceding cell 'start'=-2 etc.
'occurrence' - specifies which occurrence of the matching/found value should be returned. Positive values indicate top-down searching and counting. Negative values indicate bottom-up searching and counting. For the first match 'occurrence'=1, for the 2nd 'occurrence'=2 etc. For the last match 'occurrence'=-1, for the preceding match 'occurrence'=-2 etc. The number of occurrences is counted from the 'start' index.

The 'type' argument specifies how the searching procedure should be performed. It can be either one of the three values 0, -1, 1 or a combination (sum) of various 'SEARCH::' flags:

0 - hLookUp searches a given range linearly for an exact match; 'v' can be a search pattern containing '?' (any single character) and '*' (any string, including an empty string); to search for '?' or '*' place a tilde (~) before them,

1 - if an exact match is not found, hLookUp will search for the largest value that is not greater than 'v'; no pattern matching is performed; the searched range must be sorted in the ascending order,

-1 - if an exact match is not found, hLookUp will search for the smallest value than is not smaller than 'v'; no pattern matching is performed; the searched range must be sorted in the descending order,

SEARCH::MatchNotGreater (or 2) - if an exact match is not found, the function will search for the largest value that is not greater than 'v',

SEARCH::MatchNotSmaller (or 4) - if an exact match is not found, the function will search for the smallest value than is not smaller than 'v',

SEARCH::SortAscending (or 8) - perform a fast search for a range that is sorted in the ascending order,

SEARCH::SortDescending (or 16) - perform a fast search for a range that is sorted in the descending order,

SEARCH::CaseSensitive (or 128) - use case sensitive string comparison,

SEARCH::FirstMatch (or 256) - find the first match,

SEARCH::LastMatch (or 512) - find the last match,

SEARCH::MixedData (or 2048) - the searched range contains both text and numbers,

SEARCH::RegEx (or 8192) - the "v" parameter is a regular expression; can't be used with fast binary searching,

SEARCH::IgnorePunctuation (or 16384) - use the word sort order (ignoring certain punctuation marks),

SEARCH::NeutralSortOrder (or 32768) - use neutral, language independent string comparison instead of the default language specific comparison,

SEARCH::Pattern (or 65536) - the "v" parameter is a simple (wildcard) pattern; can't be used with fast binary searching.

-- For more detailed information, please see the respective help topic. --

The '1' value is an equivalent to (SEARCH::SortAscending + SEARCH::MatchNotGreater).

The '-1' value is an equivalent to (SEARCH::SortDescending + SEARCH::MatchNotSmaller).

The '0' value is an equivalent to (0).

The SEARCH::Pattern and SEARCH::RegEx flags can't be used with SEARCH::MatchNotGreater, SEARCH::MatchNotSmaller, SEARCH::StringSort, SEARCH::CaseSensitive, SEARCH::SortAscending, SEARCH::SortDescending.

The SEARCH::MatchNotGreater and SEARCH::MatchNotSmaller flags can not be used with the SEARCH::FirstMatch and SEARCH::LastMatch flags.

If neither SEARCH::FirstMatch nor SEARCH::LastMatch is specified, the linear search returns the first match and the fast search may return any of the existing matches.

If SEARCH::SortAscending or SEARCH::SortDescending is specified, the searched range either should not contain any formulas or the formulas should not break the sort order during the recalculation. Additionally, in such a case, no circular reference will be reported for cells other than the result cell.

For better performance, when specifying the above options one can use the resulting numeric code instead of the individual option names.

If the match is not found, it returns the #N/A! error value.

If 'type' is omitted, it's assumed to be 1.

=hLookUp(2, {1, 2, 3; "a", "b", "c"}, 2, 0)  returns "b"

=hLookUp(2.5, {1, 2, 3; "a", "b", "c"}, 2, -1)  returns "c"

=hLookUp(2.5, {1, 2, 3; "a", "b", "c"}, 2, 1)  returns "b"

=hLookUp("*bc??", {"abc", "abcde", "ac"; 1, 2, 3}, 2, SEARCH::Pattern)  returns 2

=hLookUp("bc\d", {"abc", "abcde", "abc10"; 1, 2, 3}, 2, SEARCH::RegEx,,)  returns 3

---

**index(x, m, n)**

If 'x' is a reference (a cell or a range of cells) and 'm' and 'n' are numbers, the function returns the reference of the cell from the m-th row and the n-th column of 'x'. If either 'm' or 'n' is omitted or has an explicit value of 0, the function returns a reference respectively of the n-th column or the

m-th row.

If 'x' is an array, the function returns respectively a cell value or an array instead of references.

If 'm' or 'n' are arrays or ranges, the functions returns an array containing respectively the specified rows or columns of 'x'. If one of these parameters is an array/range, the other must be a non-zero column/row index or #REF! error codes will be returned.

If 'x' is omitted, it's assumed to be a range representing the whole worksheet containing that function. In this case 'm' and 'n' must be greater than 0.

If 'm' or 'n' are out of the valid ranges or if they are 0 which wasn't entered as a constant value, the function returns the #REF! error.

=index(a1:d4, 2, 2)  returns b2

=index(, 2, 2)  returns b2

=index(a1:b5, 0, 2)  returns b1:b5

=index(a1:b5, 3, )  returns a3:b3

=index({1, 3, 5; 2, 4, 6}, 2, 2)  returns 4

=index({1, 3, 5; 2, 4, 6},, 2)  returns {3; 4}

=index({1, "a"; 2, "b"; 3, "c"}, {1; 2; 3}, 2)  returns {"a"; "b"; "c"}

=index({1, "a"; 2, "b"; 3, "c"}, 3, 0)  returns {3, "c"}

=index({1, "a"; 2, "b"; 3, "c"}, 0, 2)  returns {"a"; "b"; "c"}

=index({1, 3, 5; 2, 4, 6}, {2, 1}, 1)  returns {2, 1}

---

**indirect(text, [mode])**

indirect(text)

Converts text to a cell reference. The 'mode' argument specifies the address notation used in 'text': 0 - the 'RC' notation, 1 - the 'A1' notation. The default value of 'mode' is 1.

=indirect("A1", 1)  returns (value of) A1

=indirect("A:A", 1)  returns the 1st column

=indirect("1:3")  returns rows 1 to 3

=indirect("C1", 0)  returns the 1st column

---

**lookUp(v, array)**

lookUp(v, searchArray, resultArray)

The first version searches either (1) the top row of 'array' (if 'array' has more columns than rows) or (2) the leftmost column of 'array' (if 'array' has more rows than columns) and returns a value respectively from either (1) the same column and the last row or (2) the same row and the last column.

The second version of the 'lookUp' function uses the 'searchArray' and 'resultArray' arguments that are one-column or one-row arrays with the same number of cells. After finding 'v' in 'searchArray', the corresponding value from 'resultValue' is returned.

If an exact match can't be found, the largest value not greater than 'v' is returned. On failure, both versions return the #N/A error value.

=lookUp(3, {1, 5, 3, 4; "a", "b", "c", "d"})  returns "c"

=lookUp(3, {1, 5, 3, 4}, {"a", "b", "c", "d"})  returns "c"

---

**match(v, array, [options])**

match(v, array, [options], [start], [occurrence])

Returns the relative position of a number or text 'v' in 'array'.

The 2nd match() variant uses two additional parameters:
'start' - positive numbers specify from which cell the searching should start and negative numbers specify where it should end. For the first top-left cell of the searched range 'start'=1, for the 2nd one 'start'=2 etc. The bottom-right cell has the index of -1, for the preceding cell 'start'=-2 etc.
'occurrence' - specifies which occurrence of the matching/found value should be returned. Positive values indicate top-down searching and counting. Negative values indicate bottom-up searching and counting. For the first match 'occurrence'=1, for the 2nd 'occurrence'=2 etc. For the last match 'occurrence'=-1, for the preceding match 'occurrence'=-2 etc. The number of occurrences is counted from the 'start' index.

The 'options' argument specifies how the searching procedure should be performed. It can be either one of the three values 0, -1, 1 or a combination (sum) of various 'SEARCH::' flags:

0 - the function searches a given range linearly for an exact match; 'v' can be a search pattern containing '?' (any single character) and '*' (any string, including an empty string); to search for '?' or '*' place a tilde (~) before them,

1 - if an exact match is not found, the function will search for the largest value that is not greater than 'v'; no pattern matching is performed; the searched range must be sorted in the ascending order,

-1 - if an exact match is not found, the function will search for the smallest value than is not smaller than 'v'; no pattern matching is performed; the searched range must be sorted in the descending order,

SEARCH::MatchNotGreater (or 2) - if an exact match is not found, the function will search for the largest value that is not greater than 'v',

SEARCH::MatchNotSmaller (or 4) - if an exact match is not found, the function will search for the smallest value than is not smaller than 'v',

SEARCH::SortAscending (or 8) - perform a fast search for a range that is sorted in the ascending order, the searched range must have the form of a one-column or one-row vector,

SEARCH::SortDescending (or 16) - perform a fast search for a range that is sorted in the descending order, the searched range must have the form of a one-column or one-row vector,

SEARCH::CaseSensitive (or 128) - use case sensitive string comparison,

SEARCH::FirstMatch (or 256) - find the first match,

SEARCH::LastMatch (or 512) - find the last match,

SEARCH::AutoSort (or 1024) - perform background sorting automatically then use the fast binary search. The searched range must have the form of a one-column vector. In this case the 'startFrom' parameter refers to the internally sorted range.

SEARCH::MixedData (or 2048) - the searched range contains both text and numbers.

SEARCH::SortIndex (or 4096) - can only be used with SEARCH::AutoSort; if it's specified, Match() will return the index related to the internally sorted searched range, not to the actual un-sorted range in the worksheet,

SEARCH::RegEx (or 8192) - the 'v' parameter is a regular expression,

SEARCH::IgnorePunctuation (or 16384) - use the word sort order (ignoring certain punctuation marks),

SEARCH::NeutralSortOrder (or 32768) - use neutral, language independent string comparison instead of the default language specific comparison,

SEARCH::Pattern (or 65536) - the "v" parameter is a simple (wildcard) pattern; can't be used with fast binary searching.

-- For more detailed information, please see the respective help topic. --

The '1' value is an equivalent to (SEARCH::SortAscending + SEARCH::MatchNotGreater).

The '-1' value is an equivalent to (SEARCH::SortDescending + SEARCH::MatchNotSmaller).

The '0' value is an equivalent to (0).

The SEARCH::Pattern and SEARCH::RegEx flags can't be used with SEARCH::MatchNotGreater, SEARCH::MatchNotSmaller, SEARCH::StringSort, SEARCH::CaseSensitive, SEARCH::SortAscending, SEARCH::SortDescending.

The SEARCH::MatchNotGreater and SEARCH::MatchNotSmaller flags can not be used with the SEARCH::FirstMatch and SEARCH::LastMatch flags.

If neither SEARCH::FirstMatch nor SEARCH::LastMatch is specified, the linear search returns the first match and the fast search may return any of the existing matches.

If SEARCH::SortAscending or SEARCH::SortDescending is specified, the searched range either must not contain any formulas or the formulas must not break the sort order during the recalculation. Additionally, in such a case, no circular reference will be reported for cells other than the result cell.

If SEARCH::AutoSort is specified, GS-Calc will be creating and maintaining sort indices for the referenced searched ranges containing unsorted data. Thanks to those internally created indices it's possible to use fast binary searches for data that doesn't have to be sorted manually by the user.
The SEARCH::AutoSort flag must be used with SEARCH::SortAscending or SEARCH::SortDescending. If the searched data is already partially sorted, it's recommended that you use the sort order flag that matches that partial sorting the best.

For better performance, when specifying the above options one can use the resulting numeric code instead of the individual option names.

If the 'options' parameter is omitted, it's assumed to be 1.

If the searched value is not found, 'match' returns the #N/A! error value.

=match(2, {1, 2, 3}, 0)  returns 2

=match("b", {1, 2, 3; "a", "b", "c"}, 0)  returns 5

=match("*bc??", {"abc", "abcde", "ac"}, SEARCH::Pattern)  returns 2

=match("bc\d", {"abc", "abcde", "abc10"}, SEARCH::RegEx)  returns 3

---

**offset(reference, m, n, [height], [width])**

Returns a reference that starts 'm' rows below or - depending on the sign of 'm' - over the first row of 'source_reference' and 'n' columns after or before the first column of 'source_reference'. The 'with' and 'height' arguments specify the dimensions of the returned reference. If any of the two argument is omitted, it's assumed to have the same value as 'source_reference'.

=offset(A1:B5, 2, 1, 2, 2)  returns B3:C4

---

**pivotData(source, rows, columns, data, functions, options [, field1, filter1, field2, filter2, ...])**

Creates and returns a pivot table for the 'source' data range.

The 'rows', 'columns' and 'data' parameters represent arrays/ranges containing the respective pivot field indices. The indices are relative to the top-left corner of the 'source' range and the numbering starts from 1. For example:
{1, 5}, {4}, {1}
If some of the indices are incorrect, pivotData returns the #VALUE! error. In the current GS-Calc version the 'columns' array can contain only one element. If the 'data' are omitted, the last specified row field and the default pivot function (set in the 'Options' dialog) will be used (for the last row field).

The 'functions' argument is an array of predefined functions IDs associated with the specified data fields. The possible values are:
PIVOT::Sum
PIVOT::SumPositive
PIVOT::SumNegative
PIVOT::SumSquares
PIVOT::Count
PIVOT::CountPositive
PIVOT::CountNegative

PIVOT::CountZeroes
PIVOT::Mean
PIVOT::Geomean
PIVOT::Harmean
PIVOT::Min
PIVOT::Max
PIVOT::Quartile1
PIVOT::Median
PIVOT::Quartile3
PIVOT::Var
PIVOT::VarP
PIVOT::Stdev
PIVOT::StdevP
PIVOT::Skew
PIVOT::Kurt
PIVOT::Mode

For example: {PIVOT::Sum, PIVOT::Count}

The 'options' parameter can be a sum of the following constants:

PIVOT::ColumnGrandTotals - display column grandtotals in the last column,

PIVOT::RowGrandTotals - include row grandtotals in the last row,

PIVOT::SubTotals - include subtotals (for pivot tables with 2 or more row fields),

PIVOT::ShowZeroes - show zeroes for empty data fields; without this option 'empty' subtotals will be displayed as the #N/A! error code,

PIVOT::RepeatRowFields - if multiple row fields are specified, display all their values, even duplicated ones,

PIVOT::CaseSensitive - if filters are specified, use case sensitive comparison,

PIVOT::SortRowsDescending - present the output rows using the descending sort order,

PIVOT::SortColumnsDescending - present the output columns using the descending sort order,

PIVOT::NoSourceFieldNames - the source range contains no field names in the first row; use the 'Field n.' names instead; for example: {PIVOT::ColumnGrandTotals + PIVOT::RowGrandTotals + PIVOT::SubTotals}

PIVOT::IgnorePunctuation (or 16384) - use the word sort order (ignoring certain punctuation marks) when sorting and filtering,

PIVOT::NeutralSortOrder (or 32768) - use neutral, language independent string comparison instead of the default language specific comparison when sorting and filtering,

PIVOT::Pattern (or 65536) - treat filters that don't start with (>, >=, <, <=, =) operator as simple (wildcard) patterns.

You must specify either the PIVOT::ColumnGrandTotals or at least one column field.

The optional [field, filter] pairs specify the 1-based field index and the condition. Only source data meeting all the specified conditions will be included in the pivot table. Filters can be patterns or plain numbers and text strings with optional leading =,>,<,=>,=< operators. Conditions can have the following form:
(1) a text string beginning with the =,>,>=,<,<=,<> operators,
(2) a number or a search pattern: a text string containing special characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

=pivotData(C3:H19, {1},,, {PIVOT::Sum}, PIVOT::ColumnGrandTotals)

=pivotData(sheet1!C3:H19, {1}, {2}, {3, 4}, {PIVOT::Sum, PIVOT::Count}, PIVOT::RowGrandTotals + PIVOT::ColumnGrandTotals + PIVOT::SubTotals)

=pivotData(sheet1!B3:F8, {5, 1}, {3},, {PIVOT::Sum}, PIVOT::RowGrandTotals + PIVOT::ColumnGrandTotals + PIVOT::SubTotals, 2, "Jones", 4, ">2010-01-01")

---

**row(reference)**

Returns the row number of 'reference'.

=row(B1)  returns 1

=row(A1:C5)  returns {1; 2; 3; 4; 5}

---

**rows(array)**

Returns the number of rows in 'array'.

=rows({1, 2, 3; 4, 5, 6})  returns 2

---

**rtd(prog_id, machine, v1, v2, ...)**

Receives data from a COM (automation) server identified by its registered 'prog_id' name.

The COM server application can be run on a remote computer or - if the 'machine' argument is omitted - locally.

The optional 'v_' arguments specify the names of the requested properties. Each property can return a variant of the following types:
(1) VT_EMPTY (empty cells)
(2) VT_UI1 (error codes)
(3) VT_R8 (numbers)
(4) VT_BSTR (text strings)
(5) SAFEARRAY: an array of any of the above.

If there are two or more names specified, the function returns a vector (a one column array) of values. If a given property returns an array, it must be the only 'v_' argument.

=rtd("MSComCtl2.MonthView.2",, "Day")

---

**sort(array, column1, order1 [, column2, order2, column3, order3])**

Sorts 'array' based on the specified 'column/order' pairs and returns the result as a vector of relative rows indices.

The 'column(n)' arguments specified the relative position of the columns that should be used as the sort keys and the 'order(n)' argument specifies the sort order:
0 - descending order,
1 - ascending order.

Use the 'index' function to obtain the final sorted array.

=sort({3; 1; 5; 2; 4}, 1, 1)  returns {2; 4; 1; 5; 3}

=sort({3; 1; 5; 2; 4}, 1, 0)  returns {3; 5; 1; 4; 2}

=sort({2, "b"; 2, "a"; 3, "d"; 3, "c"; 1, "e"}, 1, 1, 2, 1)  returns { 5; 2; 1; 4; 3}

=index({2, "b"; 2, "a"; 3, "d"; 3, "c"; 1, "e"}, {5; 2; 1; 4; 3}, 1)  returns {1; 2; 2; 3; 3}  where {5; 2; 1; 4; 3} is the result of the above sorting.

---

**transpose(array)**

Transpose a given array.

=transpose({1, 2, 3; 4, 5, 6})  returns {1, 4; 2, 5; 3, 6}

---

**udf(prog_id, machine, method, v1, v2, ...)**

Executes a method on a COM (automation) server identified by its registered 'prog_id' name.

The COM server application be run on a remote computer or - if the 'machine' argument is omitted - locally.

The optional 'v_' arguments are passed via the variant structures of the following types:
(1) VT_EMPTY (empty cells)
(2) VT_UI1 (error codes)
(3) VT_R8 (numbers)
(4) VT_BSTR (text strings)
(5) SAFEARRAY: an array of any of the above (when ranges/arrays are passed).

The value returned by the requested method can be a number, a text string or an array of VARIANT structures.

=udf("",, "test", 1, 1, 0)

---

**unique(vector, [type])**

Searches 'search' (a one-column array) for unique values.

If 'type' is 0, the 'unique' function returns a vector of indices to the subsequent unique values found in 'vector'.

If 'type' is 1, the 'unique' function returns a vector of unique values found in 'vector'.

If 'type' is 2, the 'unique' function returns a two-column array.
The first column contains unique values, the second one: the number of occurrences of the

corresponding value.

If 'type' is omitted, it's assumed to be 1.

=unique({2; 3; 1; 2; 5; 3; 1}, 1)  returns {1; 2; 3; 5}

=unique({"a"; "b"; "cd"; "a"; "cd"; "b"; "d"}, 0)  returns {1; 2; 3; 5}

=unique({"a"; "b"; "cd"; "a"; "cd"; "b"; "d"}, 2)  returns { "a", 2; "b", 2; "cd", 2; "d", 1}

---

**vLookUp(v, array, n, [type])**

vLookUp(v, array, n, [type], [startFrom], [occurrence])

Searches for 'v' in the leftmost column of 'array' and - if found - returns a value from the same row and the n-th column of 'array'.

The 2nd vLookUp() variant uses two additional parameters:
'startFrom' - positive numbers specify where the searching should start and negative numbers specify where it should end. For the first, top-left cell of the searched range 'startFrom'=1, for the 2nd one 'startFrom'=2 etc. For the last, bottom-right cell of the searched range 'startFrom'=-1, for the preceding cell 'startFrom'=-2 etc.
'occurrence' - specifies which occurrence of the matching/found value should be used. Positive values indicate top-down searching and counting. Negative values indicate bottom-up searching and counting. For the first match 'occurrence'=1, for the 2nd 'occurrence'=2 etc. For the last match 'occurrence'=-1, for the preceding match 'occurrence'=-2 etc. The number of occurrences is counted from the 'startFrom' index.

The 'type' argument specifies how the searching procedure should be performed. It can be either one of the three values 0, -1, 1 or a combination (sum) of various 'SEARCH::' flags:

0 - hLookUp searches a given range linearly for an exact match; 'v' can be a search pattern containing '?' (any single character) and '*' (any string, including an empty string); to search for '?' or '*' place a tilde (~) before them,

1 - if an exact match is not found, hLookUp will search for the largest value that is not greater than 'v'; no pattern matching is performed; the searched range must be sorted in the ascending order,

-1 - if an exact match is not found, hLookUp will search for the smallest value than is not smaller than 'v'; no pattern matching is performed; the searched range must be sorted in the descending order,

SEARCH::MatchNotGreater (or 2) - if an exact match is not found, the function will search for

the largest value that is not greater than 'v',

SEARCH::MatchNotSmaller (or 4) - if an exact match is not found, the function will search for the smallest value than is not smaller than 'v',

SEARCH::SortAscending (or 8) - perform a fast search for a range that is sorted in the ascending order,

SEARCH::SortDescending (or 16) - perform a fast search for a range that is sorted in the descending order,

SEARCH::CaseSensitive (or 128) - use case sensitive string comparison,

SEARCH::FirstMatch (or 256) - find the first match,

SEARCH::LastMatch (or 512) - find the last match,

SEARCH::AutoSort (or 1024) - perform background sorting automatically then use the fast binary search. The searched range must have the form of a one-column vector. In this case the 'startFrom' parameter refers to the internally sorted range.

SEARCH::MixedData (or 2048) - the searched range contains both text and numbers.

SEARCH::SortIndex (or 4096) - can only be used with SEARCH::AutoSort; if it's specified, Match() will return the index related to the internally sorted searched range, not to the actual un-sorted range in the worksheet,

SEARCH::RegEx (or 8192) - the 'v' parameter is a regular expression,

SEARCH::IgnorePunctuation (or 16384) - use the word sort order (ignoring certain punctuation marks),

SEARCH::NeutralSortOrder (or 32768) - use neutral, language independent string comparison instead of the default language specific comparison,

SEARCH::Pattern (or 65536) - the "v" parameter is a simple (wildcard) pattern; can't be used with fast binary searching.

-- For more detailed information, please see the respective help topic. --

The '1' value is an equivalent to (SEARCH::SortAscending + SEARCH::MatchNotGreater).

The '-1' value is an equivalent to (SEARCH::SortDescending + SEARCH::MatchNotSmaller).

The '0' value is an equivalent to (0).

The SEARCH::Pattern and SEARCH::RegEx flags can't be used with SEARCH::MatchNotGreater, SEARCH::MatchNotSmaller, SEARCH::StringSort, SEARCH::CaseSensitive, SEARCH::SortAscending, SEARCH::SortDescending.

The SEARCH::MatchNotGreater and SEARCH::MatchNotSmaller flags can not be used with the SEARCH::FirstMatch and SEARCH::LastMatch flags.

If neither SEARCH::FirstMatch nor SEARCH::LastMatch is specified, the linear search returns the first match and the fast search may return any of the existing matches.

If SEARCH::SortAscending or SEARCH::SortDescending is specified, the searched range either must not contain any formulas or the formulas must not break the sort order during the recalculation. Additionally, in such a case, no circular reference will be reported for cells other than the result cell.

If SEARCH::AutoSort is specified, GS-Calc will be creating and maintaining sort indices for the referenced searched ranges containing unsorted data. Thanks to those internally created indices it's possible to use fast binary searches for data that doesn't have to be sorted manually by the user.
The SEARCH::AutoSort flag must be used with SEARCH::SortAscending or SEARCH::SortDescending. If the searched data is already partially sorted, it's recommended that you use the sort order flag that matches that partial sorting the best.

For better performance, when specifying the above options one can use the resulting numeric code instead of the individual option names.

If 'type' is omitted, it's assumed to be 1.

If the match is not found, the function returns the #N/A! error value.

=vLookUp(2, {1, "a"; 2, "b"; 3, "c"}, 2, 0)  returns "b"

=vLookUp(2.5, {1, "a"; 2, "b"; 3, "c"}, 2, -1)  returns "c"

=vLookUp(2.5, {1, "a"; 2, "b"; 3, "c"}, 2, 1)  returns "b"

=vLookUp("*bc??", {"abc", 1; "abcde", 2; "ac", 3}, 2, SEARCH::Pattern)  returns 2

=vLookUp("bc\d", {"abc", 1; "abcde", 2; "abc10", 3}, 2, SEARCH::RegEx)  returns 3

## Logical Functions

---

**and(v1, v2, ...)**

Performs a bitwise AND on two or more numeric values. If the specified parameters are expressions that evaluate to logical True (1) or False (0), the function returns 1 or 0 and is an equivalent to plain logical AND. All numbers are rounded to the nearest integers with up to 15 digits. Text strings are converted to numbers. Text strings that can't be converted to numbers cause #NUM! errors. Empty cells are ignored. If there is no non-empty cells, the functions returns the #N/A! error value.

=and(1, 1, true, "1")  returns 1

=and(a1 >= 0, a1 < 0)  returns 0

=and({true, true, 1, 1}, {true, false})  returns 0

=and(345, 178)  returns 16

=and(1, 2, 4, 8)  returns 0

---

**false()**

Returns 0.

---

**if(v, if-true-expression, if-false-expression)**

If 'v' is 'true' or simply evaluates to a positive integer number, the function returns the 'if-true-expression'. Otherwise 'if-false-expression' is returned. The returned expressions can be of any type: numbers, text, arrays, references or error codes.

If 'v' is a text string or a floating-point number, the corresponding conversion to an integer number is performed. Floating point 'v' values are rounded to the nearest integer.

If 'v' is an array or a range, the if() function becomes an array formula returning an array of the 'v' size. As specified in the 'Using Array Formulas' help topic the rules of using array arguments require that 'if-true-expression' and 'if-false-expression' must be either a single (scalar) value or an array/range of the same size.

=if((1 > 0)*(2 > 0), {1, 2}, {4, 5})  returns {1, 2}

=if("ab" > "a", 1, #N/A!)  returns 1

=if(b1="abc", c1:c5, d1:d10)  returns one of the two references c1:c5 or d1:d10 depending on the value of b1.

=if(b1:c5 > 100, "ok", b1:c5)  returns an array of b1:c5 values with the 'ok' text replacing values > 100.

=if(isError(b1:c5), errorType(b1:c5)=0, b1:c5)  returns an array of b1:c5 values with errors converted to 0.

=if(isError(b1:c5), if(errorType(b1:c5)=0, "", ""), b1:c5)  the same as above except that empty "" strings are displayed instead of zeroes.

---

**not(v, bits)**

not(v)

Returns the bitwise negation of "v" using the specified number of bits. The "bits" parameter must be in the range <1, 49>

The second variant uses only one bit as the default value of "bits" and returns simple logical negation (1 or 0) of a logical expression.

=not(1,8)  returns 254

=not(1,16)  returns 65534

=not(34,8)  returns 221

=not(true)  returns 0

---

**or(v1, v2, ...)**

Performs a bitwise OR on two or more numeric values. If the specified parameters are expressions that evaluate to logical True (1) or False (0), the function returns 1 or 0 and is an equivalent to plain logical OR. All numbers are rounded to the nearest integers with up to 15

digits. Text strings are converted to numbers. Text strings that can't be converted to numbers cause #NUM! errors. Empty cells are ignored. If there is no non-empty cells, the functions returns the #N/A! error value.

=or(1, 1, true, "1")  returns 1

=or(a1 >= 0, a1 < 0)  returns 1

=or({true, true, 1, 1}, {true, false})  returns 1

=or(345, 178)  returns 507

=or(1, 2, 4, 8)  returns 15

---

**true()**

Returns 1.

---

**xor(v1, v2, ...)**

Performs a bitwise exclusive OR on two or more numeric values. If the specified parameters are expressions that evaluate to logical True (1) or False (0), the function returns 1 or 0.All numbers are rounded to the nearest integers with up to 15 digits. Text strings are converted to numbers. Text strings that can't be converted to numbers cause #NUM! errors. Empty cells are ignored. If there is no non-empty cells, the functions returns the #N/A! error value.

=xor(1, 1)  returns 0

=xor(a1 >= 0, a1 < 0)  returns 1

=xor({true, true, 1, 1}, {true, false})  returns 1

=xor(345, 178)  returns 491

=xor(1, 2, 4, 8)  returns 15

## Informational Functions

---

**cell(type, [reference])**

Depending on the "type" value the function returns:

"address" : the specified reference as text

"col" : column number of the specified cell

"row" : row number of the specified cell

"color" : 1 if the current cell format is set to display negative numbers in red; 0 otherwise

"contents" : value of the specified cell

"filepath" : filename and full path of current document

"format" : symbol of the format code of the specified cell, as listed below

"parentheses" : 1 if the current cell format is set to display negative numbers in parentheses; 0 otherwise

"prefix" : single quotation mark (') if the cell contents is left-aligned, double quotation mark (") if the cell contents is right-aligned, caret (^) if the cell contents is centered, empty text in all other cases

"protect" : 1 if the specified cell is protected

"type" : "b" if the cell is empty, "l" if the cell contains text, "v" in all other cases

"width" : number of characters in the default font size fitting in the specified column

General "G"
0 "F0"
#,##0 ",0"
0.00 "F2"
#,##0.00 ",2"

$#,##0_);($#,##0)  "C0"
$#,##0_);[Red]($#,##0)  "C0-"
$#,##0.00_);($#,##0.00)  "C2"
$#,##0.00_);[Red]($#,##0.00)  "C2-"
0%  "P0"
0.00%  "P2"
0.00E+00  "S2"
# ?/? or # ??/??  "G"
m/d/yy or m/d/yy h:mm or mm/dd/yy  "D4"
d-mmm-yy or dd-mmm-yy  "D1"
d-mmm or dd-mmm  "D2"
mmm-yy  "D3"
mm/dd  "D5"
h:mm AM/PM  "D7"
h:mm:ss AM/PM  "D6"
h:mm  "D9"
h:mm:ss  "D8"

=cell("address", sheet1!$a$2)  returns "sheet1!$A$2"

=cell("address", sheet1!a2)  returns "sheet1!A2"

=cell("width", b10)  returns 9

---

**countBlank(v1, v2, ...)**

Counts empty cells and cells containing empty strings for the specified list of arguments.

=countBlank({1, 2, 3,,}, {""},,)  returns 5

---

**errorType(error)**

Returns an integer representing a given error value.

For the complete list of error codes, please see the 'Data types' help topic.

---

**isBlank(x)**

Returns 1 if 'x' refers to an empty cell or an empty string, 0 otherwise.

=isBlank("")  returns 1

---

**isErr(x)**

Returns 1 if 'x' is an error value except #N/A!, 0 otherwise.

=isErr(1/0)  returns 1
=isErr(#SYNTAX!)  returns 1

---

**isError(x)**

Returns 1 if 'x' is an error value, 0 otherwise.

=isError(#N/A!)  returns 1

---

**isEven(n)**

Returns 1 if 'n' is even, 0 otherwise. All numbers are rounded to the nearest integers.

=isEven(12)  returns 1

=isEven(12.6)  returns 0

---

**isLogical(n)**

Returns 1 if 'n' is 1 or 0, 0 otherwise.

=isLogical(1)  returns 1

---

**isNA(error)**

Returns 1 if 'error' refers to the #N/A value, 0 otherwise.

=isNA(#N/A!)  returns 1

---

**isNonText(x)**

Returns 1 if 'x' refers to any value that is not a text string, 0 otherwise.

=isNonText("")  returns 0
=isNonText(1)  returns 1

---

**isNumber(x)**

Returns 1 if 'x' represents a number (which also includes a string that can be converted to a number), 0 otherwise.

=isNumber(9)  returns 1
=isNumber("9")  returns 1

---

**isOdd(n)**

Returns 1 if 'n' is odd, 0 otherwise. All numbers are rounded to the nearest integers.

=isOdd(12)  returns 0

=isOdd(12.6)  returns 1

---

**isRef(x)**

Returns 1 if 'x' is a reference, 0 otherwise.

=isRef(a1)  returns 1
=isRef({1,2,3})  returns 0

---

**isText(x)**

Returns 1 if 'x' represents a text string, 0 otherwise.

=isText("a")  returns 1

---

**n(x)**

Converts x to a number.

=n("2005")  returns 2005

---

**na()**

Returns the #N/A error value.

=na()  returns #N/A!

---

**type(x)**

Returns the type of x:
1 Number
2 Text
16 Error value
64 Array

=type(1)  returns 1

=type({1, 2, 3})  returns 64

## Engineering Functions

---

**bessellJ(x, n)**

Returns the Bessel function of the n-th order for a given value 'x'.

=besselJ(1.9, 2)  returns 0.32992572769239

---

**besselY(x, n)**

Returns the Bessel/Weber function of the n-th order for a given value 'x'.

=besselY(2.5, 1)  returns 0.14591813796679

---

**bin2dec(n)**

Converts 'n' representing a binary number to a decimal number. The 'n' argument can contain up to 10 characters. The most significant bit in 'n' determines its sign. Negative numbers are represented in two's complement format.

=bin2dec("1100100")  returns 100

=bin2dec("1111111111")  returns -1

---

**bin2hex(n, [l])**

Converts 'n' representing a binary number to a string representing a hexadecimal number consisting of 'l' characters. The 'n' argument can contain up to 10 characters. The most significant bit in 'n' determines its sign. Negative numbers are represented in two's complement format. The 'l' argument must be in the range [0, 10]. If it's omitted, it's assumed to be 10.

=bin2hex(11111011, 4)  returns "00FB"

=bin2hex("1111111111", )  returns "FFFFFFFFFF"

---

**bin2oct(n, [l])**

Converts 'n' representing a binary number to a string representing an octal number consisting of 'l' characters. The 'n' argument can contain up to 10 characters. The most significant bit in 'n' determines its sign. Negative numbers are represented in two's complement format. The 'l' argument must be in the range [0, 10]. If it's omitted, it's assumed to be 10.

=bin2oct(1001, 3)  returns "011"

=bin2oct(1111111111, )  returns "7777777777"

---

**complex(x, y, [i])**

Converts real and imaginary coefficients into a string representing a complex number. The optional 'i' argument specifies the suffix of the imaginary 'y' coefficient. The default value of 'i' is "i".

=complex(1, 2,)  returns "1 + 2i"
=complex(3.5, 4.6, "j")  returns "3.5 + 4.6j"

---

**dec2bin(n, [l])**

Converts a given number to a string representing a binary number consisting of 'l' characters. The

'n' argument must be in the range [-512, 511]. The most significant bit in 'n' determines its sign. Negative numbers are represented in two's complement format. The 'l' argument must be in the range [0, 10]. If it's omitted, it's assumed to be 10.

=dec2bin(9, 4)  returns "1001"

=dec2bin(-1,)  returns "1111111111"

---

**dec2hex(n, [l])**

Converts a given number to a string representing a hexadecimal number consisting of 'l' characters. The most significant bit in 'n' determines its sign. Negative numbers are represented in two's complement format. The 'l' argument must be in the range [0, 10]. If it's omitted, it's assumed to be 10.

=dec2hex(100, 4)  returns "064"

=dec2hex(-1,)  returns "FFFFFFFFFF"

---

**dec2oct(n, [l])**

Converts a given number to a string representing an octal number consisting of 'l' characters. The 'number' argument must be in the range [-536870912, 536870911]. The most significant bit in 'n' determines its sign. Negative numbers are represented in two's complement format. The 'l' argument must be in the range [0, 10]. If it's omitted, it's assumed to be 10.

=dec2oct(58, 3)  returns "1001"

=dec2oct(-1,)  returns "7777777777"

---

**delta(x, [y])**

Returns 1 if 'x' = 'y', 0 otherwise. If 'y' is omitted, it's assumed to be 0.

=delta(12, 11)  returns 0

=delta(0,)  returns 1

---

**erf(x1, [x2])**

Returns the value of the error function integrated between the specified limits. If the 'x2' argument is omitted, the integration is performed between 0 and 'x1'.

=erf(0.745, )  returns 0.70792891807065

---

**ERF.PRECISE(x1, x2)**

See the description online.

---

**erfc(x)**

Returns the value of the complementary error function integrated between x and the infinity.

=erfc(1)  returns 0.15722921001143

---

**ERFC.PRECISE(x1, x2)**

See the description online.

---

**getStep(x, [y])**

Returns 1 if x > y, 0 otherwise. The default value of y is 0.

=getStep(1.4,)  returns 1

---

**hex2bin(n, [l])**

Converts 'n' representing a hexadecimal number to a string representing a binary number consisting of 'l' characters. The 'n' argument can contain up to 10 characters and it must be in the range [-512, 511]. The most significant bit in 'n' determines its sign. Negative numbers are represented in two's complement format. If 'l' is omitted, it's assumed to be 10.

=hex2bin("F", 8)  returns "00001111"

=hex2bin("FFFFFFFFFF", )  returns "1111111111"

---

**hex2dec(n)**

Converts 'n' representing a hexadecimal number to a decimal number. The 'n' argument can contain up to 10 characters. The most significant bit in 'n' determines its sign. Negative numbers are represented in two's complement format.

=hex2dec("A5")  returns 165

=hex2dec("FFFFFFFFFF")  returns -1

---

**hex2oct(n, [l])**

Converts 'n' representing a hexadecimal number to a string representing an octal number consisting of 'l' characters. The 'n' argument can contain up to 10 characters and must be in the range [-536870912, 536870911]. The most significant bit in 'n' determines its sign. Negative numbers are represented in two's complement format. If 'l' is omitted, it's assumed to be 10.

=hex2oct("F", 3)  returns "017"

=hex2oct("FFFFFFFFFF", )  returns "7777777777"

---

**imAbs(z)**

Returns the absolute value of a given complex number. The 'z' argument represents a complex number in the form: "x + yi", x, or "yi".

=imAbs("5 + 12i")  returns 13

---

**imaginary(z)**

Returns the imaginary coefficient of a given imaginary number. The 'z' argument represents a complex number in the form: "x + yi", x, or "yi".

=imaginary("5 + 12i")  returns 12.

---

**imArgument(z)**

Returns an angle, such that
z = |z|(cos(t) + sin(t)i)
The 'z' argument represents a complex number in the form: "x + yi", x, or "yi".

=imArgument("3 + 4i")  returns 0.92727521800161

---

**imConjugate(z)**

Returns the complex conjugate of 'z'. The 'z' argument represents a complex number in the form: "x + yi", x, or "yi".

=imConjugate("3 + 4i")  returns "3 - 4i".

---

**imCos(z)**

Returns the cosine of 'z'. The 'z' argument represents a complex number in the form: "x + yi", x, or "yi".

=imCos("1 + 1i")  returns "0.83373002513115 + 0.98889770576287i"

---

**imDiv(z1, z2)**

Devides 'z1' by 'z2' and returns the result. The 'z1' and 'z2' arguments represent complex numbers in the form: "x + yi", x, or "yi".

=imDiv("1 + 2i", "2 + 1i")  returns "0.8 + 0.6i"

---

**imExp(z)**

Raises 'e' to the power of 'z'. The 'z' argument represents a complex number in the form: "x + yi", x, or "yi".

=imExp("1 + 1i")  returns "1.43869393991589 + 2.28735528717884i"

---

**imLn(z)**

Returns the natural logarithm of 'z'. The 'z' argument represents a complex number in the form: "x + yi", x, or "yi".

=imLn("3 + 4i")  returns "1.6094379124341 + 0.92729521800161i"

---

**imLog10(z)**

Returns the base-10 logarithm of 'z'. The 'z' argument represents a complex number in the form: "x + yi", x, or "yi".

=imLog10("3+4i")  returns "0.69897000433601 + 0.40271919627337i"

---

**imLog2(z)**

Returns the base-2 logarithm of 'z'. The 'z' argument represents a complex number in the form: "x + yi", x, or "yi".

=imLog2("3 + 4i")  returns "2.23192809488732 + 1.33780421245095i"

---

**imPower(z, n)**

Raises 'z' to the power of 'n' and returns the result. The 'z' argument represents a complex number in the form: "x + yi", x, or "yi".

=imPower("2+3i", 3)  returns "-46 + 9i"

---

**imProduct(z1, z2, ...)**

Multiplies all the specified arguments which can be complex numbers or arrays of complex numbers in the form: "x + yi", x, or "yi". Empty cells in arrays are skipped.

=imProduct("3+4i", "5-3i")  returns "27 + 11i"

---

**imReal(z)**

Returns the real coefficient of 'z'. The 'z' argument represents a complex number in the form: "x + yi", x, or "yi".

=imReal("3 + 4i")  returns 3

---

**imSin(z)**

Returns the sine of 'z'. The 'z' argument represents a complex number in the form: "x + yi", x, or "yi".

=imSin("3 + 4i")  returns  "3.85373803791938 - 27.0168132580039i"

---

**imSqrt(z)**

Returns the squre root of 'z'. The 'z' argument represents a complex number in the form: "x + yi", x, or "yi".

=imSqrt("1 + i")  returns 1.09868411346781 + 0.45508986056223i"

---

**imSub(z1, z2)**

Returns the difference between of 'z1' and 'z2'. The 'z1' and 'z2' arguments represent complex numbers in the form: "x + yi", x, or "yi".

=imSub("10 + 5i", "9 + 4i")  returns "1 + 1i"

---

**imSum(v1, v2, ...)**

Adds all the specified arguments which can be complex numbers or arrays of complex numbers in the form: "x + yi", x, or "yi".

=imSum("1 + 1i", "2 + 2i", {"1 + 1i", "1 + 1i"}, 1, "1i")  returns "6 + 6i"

---

**oct2bin(n, [l])**

Converts 'n' representing an octal number to a string representing a binary number consisting of 'l' characters. The 'n' argument can contain up to 10 characters and it must be in the range [-512, 511]. The most significant bit in 'n' determines its sign. Negative numbers are represented in two's complement format. The 'l' argument must be in the range [0, 10]. If it's omitted, it's

assumed to be 10.

=oct2bin(3, 3)  returns "011"

=oct2bin(7777777000, )  returns "1000000000"

---

**oct2dec(n)**

Converts 'n' representing an octal number to a decimal number. The 'n' argument can contain up to 10 characters. The most significant bit in 'n' determines its sign. Negative numbers are represented in two's complement format.

=oct2dec(54)  returns 44

=oct2dec(7777777777)  returns -1

---

**oct2hex(n, [l])**

Converts 'n' representing an octal number to a string representing a hexadecimal number consisting of 'l' characters. The 'n' argument can contain up to 10 characters. The most significant bit in 'n' determines its sign. Negative numbers are represented in two's complement format. The 'l' argument must be in the range [0, 10]. If it's omitted, it's assumed to be 10.

=oct2hex(100, 4)  returns "0040"

=oct2hex(7777777777, )  returns "FFFFFFFFFF"

## Statistical Functions

---

**aveDev(v1, v2, ...)**

Returns the average of the absolute deviations of the specified numbers from their mean.

=aveDev(4, 5, 6, 3, 5, 4)  returns 0.83333333333333

---

**average(v1, v2, ...)**

Returns the arithmetic mean for the specified arguments. Arguments that are either text strings which can't be converted to numbers or errors cause an error.

For arguments that are arrays or cell/range references, text strings are not converted to numbers. To include the text representations of numbers, use the averageA() function. Empty cells are ignored.

=average(4, 5, 6, 3, 5, 4)  returns 4.5

=average(4, 5, 6, 3, {1, "1"})  returns 3.8

---

**averageA(v1, v2, ...)**

Returns the arithmetic mean for the specified arguments. Arguments that are either text strings which can't be converted to numbers or errors cause an error.

For arguments that are arrays or cell/range references, both numbers and text representations of numbers are included in the calculation. To exclude text strings entirely, use the average() function. Empty cells are ignored.

=averageA(4, 5, 6, 3, 5, 4)  returns 4.5

=averageA(4, 5, 6, 3, {1, "1"})  returns 3.3(3)

---

**averageAIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the arithmetic mean for numbers in the 'data_range' array/range. The numbers must meet the specified criteria.
Text representations of numbers in the 'data_range' array/range are included. All other text strings are ignored.

All ranges must have the same number of columns and rows.The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=averageAIfs({1, 3, "6", 2}, {3, 3, 3, 0.1}, ">0")  returns 3

=averageAIfs({1, 3, "6", 2}, {3, 3, 3, 0.1}, ">"&c5)  returns mean based on the c5 cell value

---

**averageIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the arithmetic mean for numbers in the 'data_range' array/range. The numbers must meet the specified criteria.
All text strings in the 'data_range' array/range are ignored.
All ranges must have the same number of columns and rows.The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=averageIfs({1, 3, "6", 2}, {3, 3, 3, 0.1}, ">0")  returns 2

=averageIfs({1, 3, "6", 2}, {3, 3, 3, 0.1}, ">"&c5)  returns mean based on the c5 cell value

---

**BETA.DIST(x, alpha, beta, cumulative, [A], [B])**

See the description online.

---

**BINOM.DIST(successes, trials, probability, cumulative)**

See the description online.

---

**binomDist(s, t, p, c)**

Returns the binomial distribution probability.

The 's' argument represents the number of successes, the 't' argument is the number of trials and the 'p' argument is the probability of success on each trial.

If the 'c' argument is 1 (true), it returns the cumulative distribution function. Otherwise, the probability mass function is returned.

=binomDist(6,10,0.5,FALSE)  returns 0.205078125

---

**chiDist(x, f_degrees)**

Returns the one-tailed probability of the chi-squared distribution. The returned value is calculated as P(X < x).
The 'f_degrees' argument specifies the number of degrees of freedom.

=chiDist(18.307, 10)  returns 0.9499994109077

---

**chiInv(probability, f_degrees)**

Returns the inverse of the chiDist function for a given probability.

=chiInv(0.9499994109077, 10)  returns 18.307

---

**CHISQ.DIST(x,deg_freedom,cumulative)**

See the description online.

---

**CHISQ.DIST.RT(x,deg_freedom)**

See the description online.

---

**CHISQ.INV(probability,deg_freedom)**

See the description online.

---

**CHISQ.INV.RT(probability,deg_freedom)**

See the description online.

---

**CHISQ.TEST(actual_range,expected_range)**

See the description online.

---

**chiTest(observed, expected)**

Performs the test for independence of two discrete variables x (x[1], ..., x[k]) and y (y[1], ..., y[l]).
The 'observed' argument specifies the k x l contingence table with elements representing the observed numbers of experiments in which the x[i] & y[j] pairs were obtained.
The returned probability is calculated as P(X > c^2) where c^2 is the obtained chi-squared statistic.

The 'expected' argument specifies a respective table with the expected numbers of subsequent experiments.

=chiTest({58, 35; 11, 25; 10, 23}, {45.35, 47.65; 17.56, 18.44; 16.09, 16.91}) returns 0.99969180798302

---

**chiTest2(observed, expected)**

Returns the chi2-statistic fot two discrete variables x (x[1], ..., x[k]) and y (y[1], ..., y[l]).
The 'observed' argument specifies the k x l contingence table with elements representing the observed numbers of experiments in which the x[i] & y[j] pairs were obtained.
The returned value can be compared against the critical chi2 values obtained with the 'chi2Inv' function.

The 'expected' argument specifies a respective table with the expected numbers of subsequent experiments.

=chiTest2({58, 35; 11, 25; 10, 23}, {45.35, 47.65; 17.56, 18.44; 16.09, 16.91}) returns 16.1695750747969

---

**confidence(alpha, deviation, size)**

Returns the confidence interval for a population mean. The 'alpha' parameter specifies the significance level (which equals 1 - 'confidence level').
The 'deviation' argument specifies the (known) population standard deviation.
The 'size' argument specifies the sample size.

=confidence(0.05, 2.5, 50) returns 0.6929519802241

---

**CONFIDENCE.NORM(alpha,standard_dev,size)**

See the description online.

---

**CONFIDENCE.T(alpha,standard_dev,size)**

See the description online.

---

**correl(array1, array2)**

Returns the correlation coefficient of 'array1' and 'array2'. Both arguments can be arrays or cell ranges having the same dimensions.

=correl({3, 2, 4, 5, 6}, {9, 7, 12, 15, 17})  returns 0.99705448550158

---

**count(v1, v2, ...)**

Counts numbers. The 'v_' arguments can be any data types including arrays and references. All text strings in array and ranges are ignored.

=count({1, 3, "abc"}, {"4.5", 5.6})  returns 3

---

**countA(v1, v2, ...)**

Counts non-empty arguments and non-empty cells in arrays/ranges. The 'v_' arguments can be any data types including arrays and references.

=countA({1, 3, "abc"}, {"4.5", 5.6,,})  returns 5

---

**countIf(range, criteria)**

Counts numbers that meet the specified criteria. The criteria can be one of the following:

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

(4) an error code.

=countIf({1,2;3,4}, 2)  returns 1

=countIf({1,2;3,4}, ">2")  returns 2

=countIf({"abcde","def";"abc","a"}, "?bc*")  returns 2

=countIf(a1:d5, "*")  returns the number of non-empty cells within a1:d5.

=countIf(c1:d100, ">="&c101)  counts and returns values not less than value in the c101 cell.

=countIf({"ABcde","def";"Bc","a"}, ">=bc")  returns 2

=countIf({1,2;"a",#N/A!;3,#N/A!}, #N/A!)  returns 2

---

**countIfs(range1, criteria1 [, range2, criteria2, ...])**

Counts numbers from the 'range1' range/array that meet the specified 'criteria1' and any other criteria applied to other ranges. All the ranges must have the same number of columns and rows. The criteria can be one of the following:

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

(4) an error code.

=countIfs({1,2;3,2}, 2, {1,4;3,5}, 4)  returns 1

=countIfs({1,2;3,2}, ">=2", {1,4;3,5}, "<=4")  returns 2

=countIfs({"abcde","def";"abc","a"}, "?bc*")  returns 2

=countIfs(a1:d5, "*", e1:h5, "*")  returns the number of times when corresponding cells from a1:d5 and e1:h5 are (both) not empty.

=countIfs(c1:d100, "*se*", a1:b100, ">="&F1)  returns the number of times when a value from C1:D100 contains the "se" substring and a value from A1:B100 is greater than the F1 cell value.

=countIfs({1,2;"a",#N/A!;3,#N/A!}, #N/A!, a1:b3, 2009)  returns 2 if all cells in a1:b3 contains 2009 values.

---

**covar(array1, array2)**

Returns covariance of 'array1' and 'array2'. Both arguments can be arrays or cell ranges having the same dimensions.

=covar({3, 2, 4, 5, 6}, {9, 7, 12, 15, 17})  returns 5.2

---

**COVARIANCE.P(array1, array2)**

See the description online.

---

**COVARIANCE.S(array1, array2)**

See the description online.

---

**critBinom(trials, probability, alpha)**

Returns the smallest value (the number of successes) for which the cumulative binomial distribution function is greater or equal to 'alpha' for a given number of trials and the probability of success in each trial.

=critBinom(6,0.5,0.75)  returns 4

---

**devSq(v1, v2, ...)**

Returns the sum of squares of deviations of the specified numbers from their mean.
The 'v_' arguments can be any numbers, arrays or references.

=devSq(4, 5, 8, 7, 11, 4, 3)  returns 48

---

**EXPON.DIST(x, lambda, cumulative)**

See the description online.

---

**exponDist(x, lambda, cumulative)**

Returns the probability of the exponential distribution. The returned value is calculated as P(X < x).
The 'lambda' argument specifies distribution parameter.
If the 'cumulative' argument is 1/TRUE, it returns the cumulative distribution function.
Otherwise the probability density function is returned.

=exponDist(0.2, 10, 1)  returns 0.86466471676339

---

**F.DIST(x,deg_freedom1,deg_freedom2,cumulative)**

See the description online.

---

**F.DIST.RT(x,deg_freedom1,deg_freedom2)**

See the description online.

---

**F.INV(probability,deg_freedom1,deg_freedom2)**

See the description online.

---

**F.INV.RT(probability,deg_freedom1,deg_freedom2)**

See the description online.

---

**F.TEST(array1, array2)**

See the description online.

---

**fDist(x, f_degrees1, f_degrees2)**

Returns the F probability distribution function. The 'f_degrees1' and 'f_degrees2' arguments specifies the numbers of degrees of freedom used in the F expression respectively as the numerator and denominator values. The returned value is calculated as P(F < x).

=fDist(15.20675, 6, 4)  returns 0.99729988597273

---

**fInv(x, f_degrees1, f_degrees2)**

Returns the inverse of the 'fDist' probability distribution function. The 'f_degrees1' and 'f_degrees2' arguments specifies the numbers of degrees of freedom used in the F expression respectively as the numerator and denominator values.

=fInv(0.99729988597273, 6, 4)  returns 15.206750002102

---

**fisher(x)**

Returns the Fisher transformation for a given x. The returned value is calculated as z = 0.5*ln((1 - x)/(1 + x)).

=fisher(0.75)  returns 0.97295507452766

---

**fisherInv(y)**

Returns the inverse of the 'fisher' function. The returned value is calculated as x = exp(2*y) - 1)/(exp(2y) + 1).

=fisherInv(0.97295507452766)  returns 0.75

---

**forecast(x, array_y, array_x)**

Finds a best-fit regression line for the known x- and y-values and calculates the future value using the existing 'x' value.

If the procedure is not convergent, the function returns #NUM!.

=forecast(30, {6, 7, 9, 15, 21}, {20, 28, 31, 38, 40})  returns 10.6072530864198

---

**frequency(data_array, bins_array)**

Counts how often values in 'data_array' occur in the ranges specified by 'bins_array'.
The returned value is a vector (a one-column array) of numbers. If 'bins_array' has 'n' elements, the returned array has n + 1 elements. Each 'bins_array' element define a range of values smaller or equal to that element. The last range includes all values greater than the last 'bins_array' element.

=frequency({79, 85, 78, 85, 83, 81, 95, 88, 97}, {70, 79, 89})  returns {0; 2; 5; 2}

---

**fTest(array1, array2)**

Performs the F-test and return the one-tail probability that variances in the 'array1' and 'array2' data sets are significantly different.

=fTest({6, 7, 9, 15, 21}, {20, 28, 31, 38, 40})  returns 0.67584107664634

---

**GAMMA(number)**

See the description online.

---

**GAMMALN(x)**

See the description online.

---

**GAMMA.DIST(x, alpha, beta, cumulative)**

See the description online.

---

**GAMMALN.PRECISE(x)**

See the description online.

---

**GAUSS(x)**

See the description online.

---

**geoMean(v1, v2, ...)**

Returns the geometric mean for the specified positive values.

The 'v_' arguments can be numbers, array of numbers or references.

=geoMean(4, 5, 8, 7, 11, 4, 3)  returns 5.47698696965696

---

**geoMeanAIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the geometric mean for numbers in the 'data_range' array/range. The numbers must meet the specified criteria.
Text representations of numbers in the 'data_range' array/range are included. All other text strings are ignored.
All ranges must have the same number of columns and rows. The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=geoMeanAIfs({1, 3, "6", 2}, {3, 3, 3, 0.1}, ">0")  returns 2.44948974278318

=geoMeanAIfs({1, 3, "6", 2}, {3, 3, 3, 0.1}, ">" & b5)  returns mean based on the b5 cell value

---

**geoMeanIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the geometric mean for numbers in the 'data_range' array/range. The numbers must meet the specified criteria.
All text strings in the 'data_range' array/range are ignored.
All ranges must have the same number of columns and rows. The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=geoMeanIfs({1, 3, "6", 2}, {3, 3, 3, 0.1}, ">0")  returns 1.81712059283214

=geoMeanIfs({1, 3, "6", 2}, {3, 3, 3, 0.1}, ">" & b5)  returns mean based on the b5 cell value

---

**harMean(v1, v2, ...)**

Returns the harmonic for the specified values.

The 'v_' arguments can be numbers, array of numbers or references.

=harMean(4, 5, 8, 7, 11, 4, 3)  returns 5.02837596206173

---

**harMeanAIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the harmonic mean for numbers in the 'data_range' array/range. The numbers must meet the specified criteria.
Text representations of numbers in the 'data_range' array/range are included. All other text strings are ignored.
All ranges must have the same number of columns and rows. The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=harMeanAIfs({1, 3, "6", 2}, {3, 3, 3, 0.1}, ">0")  returns 2

=harMeanAIfs({1, 3, "6", 2}, {3, 3, 3, 0.1}, ">" & b5)  returns mean based on the b5 cell value

---

**harMeanIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the harmonic mean for numbers in the 'data_range' array/range. The numbers must meet the specified criteria.
All text strings in the 'data_range' array/range are ignored.
All ranges must have the same number of columns and rows.The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=harMeanIfs({1, 3, "6", 2}, {3, 3, 3, 0.1}, ">0")  returns 1.63636363636364

=harMeanIfs({1, 3, "6", 2}, {3, 3, 3, 0.1}, ">" & b5)  returns mean based on the b5 cell value

---

**HYPGEOM.DIST(sample_s,number_sample,population_s,number_pop,cumulative)**

See the description online.

---

**hypGeomDist(sample_s, sample_size, population_s, population_size)**

Returns the hypergeometric distribution function: the probability of 'sample_s' successes in an 'sample_size'-element sample with the known 'population_s' number of successes in the 'population_size'-element population.

=hypGeomDist(1, 4, 8, 20)  returns 0.36326109391125

---

**intercept(array_y, array_x)**

Finds a best-fit regression line for the known x- and y-values and returns the point of intersection of the y-axis.

The 'array_y' and 'array_x' arrays must have the same dimensions.

If the procedure is not convergent, the function returns #NUM!.

=intercept({2, 3, 9, 1, 8}, {6, 5, 11, 7, 5})  returns 0.04838709677419

---

**kurt(v1, v2, ...)**

Returns the kurtosis of the specified numbers.

The 'v_' arguments can be any numbers, arrays or references.

=kurt(3, 4, 5, 2, 3, 4, 5, 6, 4, 7)  returns -0.1517993720842

---

**large(array, k)**

Returns the k-th largest value in the specified array.

=large({3, 4, 5, 2, 3, 4, 5, 6, 4, 7}, 3)  returns 5

---

**linEst(array_y, [array_x], [const_b])**

Finds a best-fit regression line for the known x- and y-values and returns the a two-element vector containing the 'a' and 'b' coefficients of the y=a*x + b line.

If the 'const_b' argument is 0 (false), the 'b' parameter is assumed to be 0. If 'const_b' is omitted, it's assumed to be 1.

If the 'array_x' argument is omitted, it's assumed to be an array containing integers from 1 to the number of elements in 'array_y'.

The 'array_y' and 'array_x' arrays must have the same dimensions.

If the procedure is not convergent, the function returns the #NUM!

=linEst({2, 3, 9, 1, 8}, {6, 5, 11, 7, 5},)  returns {0.04838709677419; 0.66935483870968}

---

**LOGNORM.DIST(x, mean, standard_dev, cumulative)**

See the description online.

---

**LOGNORM.INV(probability, mean, standard_dev)**

See the description online.

---

**max(v1, v2, ...)**

Returns the largest value. Arguments that are either text strings which can't be converted to numbers or errors cause an error.

For arguments that are arrays or cell/range references, text strings are not converted to numbers. To include the text representations of numbers, use the maxA() function.

=max(4, 5, 8, 7, 11, 4, 3)  returns 11

=max(4, 5, 8, 7, {"11", 4})  returns 8

---

**maxa(v1, v2, ...)**

Returns the largest value. Arguments that are either text strings which can't be converted to numbers or errors cause an error.

For arguments that are arrays or cell/range references both numbers and text representations of numbers are included in the comparison. To exclude the text representations of numbers, use the maxA() function.

=maxA(4, 5, 8, 7, 11, 4, 3)  returns 11

=maxA(4, 5, 8, 7, {"11", 4})  returns 11

---

**maxAIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the largest number in the 'data_range' array/range. The numbers must meet the specified criteria.
Text representations of numbers in the 'data_range' array/range are included. All other text strings are ignored.
All ranges must have the same number of columns and rows. The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=maxAIfs({1, 3, "6", 7}, {3, 3, 3, 0.1}, ">1")  returns 6

=maxAIfs({1, 3, "6", 7}, {3, 3, 3, 0.1}, ">" & b5)  returns the largest number based on the b5 cell value

---

**maxAIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the largest number in the 'data_range' array/range. The numbers must meet the specified criteria.
All text strings in the 'data_range' array/range are ignored.
All ranges must have the same number of columns and rows.The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=maxIfs({1, 3, "6", 7}, {3, 3, 3, 0.1}, ">1")  returns 3

=maxIfs({1, 3, "6", 7}, {3, 3, 3, 0.1}, ">" & b5)  returns the largest number based on the b5 cell value

---

**median(v1, v2, ...)**

Returns the median of the specified numbers. The median is such a number that half the numbers are greater and half the numbers are smaller than the median. The 'v_' arguments can be numbers, array of numbers or references.

=median(1, 2, 3, 4, 5, 6)  returns 3.5

---

**medianAIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the median for numbers in the 'data_range' array/range. The numbers must meet the specified criteria.
Text representations of numbers in the 'data_range' array/range are included. All other text strings are ignored.
All ranges must have the same number of columns and rows. The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=medianAIfs({1, 2, 3, 4, "5", 6}, {3, 3, 3, 0.1}, ">0")  returns 3.5

=medianAIfs({1, 2, 3, 4, "5", 6}, {3, 3, 3, 0.1}, ">"&b5)  returns the median based on the b5 cell value

---

**medianIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the median for numbers in the 'data_range' array/range. The numbers must meet the specified criteria.
All text strings in the 'data_range' array/range are ignored.
All ranges must have the same number of columns and rows. The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=medianIfs({1, 2, 3, 4, "5", 6}, {3, 3, 3, 0.1}, ">0")  returns 3

=medianIfs({1, 2, 3, 4, "5", 6}, {3, 3, 3, 0.1}, ">"&b5)  returns the median based on the b5 cell value

---

**min(v1, v2, ...)**

Returns the smallest value. Arguments that are either text strings which can't be converted to numbers or errors cause an error.

For arguments that are arrays or cell/range references, text strings are not converted to numbers. To include the text representations of numbers, use the minA() function.

=min(4, 5, 8, 7, 11, 4, 3)  returns 3

=min(4, 5, 8, 7, 11, 4, {"1", "3"})  returns 4

---

**minA(v1, v2, ...)**

Returns the smallest value. Arguments that are either text strings which can't be converted to numbers or errors cause an error.

For arguments that are arrays or cell/range references both numbers and text representations of numbers are included in the comparison. To exclude the text representations of numbers, use the minA() function.

=minA(4, 5, 8, 7, 11, 4, 3)  returns 3

=minA(4, 5, 8, 7, 11, 4, {"1", "3"})  returns 1

---

**minAIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the smallest number in the 'data_range' array/range. The numbers must meet the specified criteria.
Text representations of numbers in the 'data_range' array/range are included. All other text strings are ignored.
All ranges must have the same number of columns and rows. The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=minAIFS({1, 3, "6", 7}, {3, 3, 0.1, 0.1}, "<1")  returns 6

=minAIfs({1, 3, "6", 7}, {3, 3, 3, 0.1}, ">"&b5)  returns the smallest number based on the b5 cell value

---

**minIFS(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the smallest number in the 'data_range' array/range. The numbers must meet the specified criteria.
All text strings in the 'data_range' array/range are ignored.
All ranges must have the same number of columns and rows. The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=minIfs({1, 3, "6", 7}, {3, 3, 0.1, 0.1}, "<1")  returns 7

=minIfs({1, 3, "6", 7}, {3, 3, 3, 0.1}, ">"&b5)  returns the smallest number based on the b5 cell value

---

**mode(v1, v2, ...)**

Returns the most frequently occurring number in the specified numbers. The 'v_' arguments can be numbers, array of numbers or references.

=mode({5.6, 4, 4, 3, 2, 4})  returns 4

---

**MODE.MULT(data1, data2, ...)**

See the description online.

---

**MODE.SNGL(data1, data2, ...)**

See the description online.

---

**modeAIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the most frequently occurring number in the 'data_range' array/range. The included numbers must meet the specified criteria.
Text representations of numbers in the 'data_range' array/range are included. All other text strings are ignored.
All ranges must have the same number of columns and rows. The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=modeAIfs({5.6, 4, 4, "3", "3", 4}, {3, 3, 3, 0.1, 0.1, 0.1}, "<1")  returns 3

=modeAIfs({5.6, 4, 4, "3", "3", 4}, {3, 3, 3, 0.1, 0.1, 0.1}, "<"&b5)  returns the most frequently occurring number based on the b5 cell value

---

**modeIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the most frequently occurring number in the 'data_range' array/range. The included numbers must meet the specified criteria.
All text strings in the 'data_range' array/range are ignored.
All ranges must have the same number of columns and rows.The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=modeIfs({5.6, 4, 4, "3", "3", 4}, {3, 3, 3, 0.1, 0.1, 0.1}, "<1")  returns 4

=modeIfs({5.6, 4, 4, "3", "3", 4}, {3, 3, 3, 0.1, 0.1, 0.1}, "<"&b5)  returns the most frequently occurring number based on the b5 cell value

---

**NORM.DIST(x, mean, standard_dev, cumulative)**

See the description online.

---

**NORM.INV(probability, mean, standard_dev)**

See the description online.

---

**NORM.S.DIST(z, cumulative)**

See the description online.

---

**NORM.S.INV(probability)**

See the description online.

---

**normDist(x, mean, deviation, cumulative)**

Returns the normal distribution for a given mean and standard deviation.
If the 'cumulative' argument is 1 (true), the 'normDist' function returns the cumulative distribution function. Otherwise, the probability mass function is returned.

=normDist(42, 40, 1.5, true)  returns 0.90878877482188

---

**normsDist(x)**

Returns the standard normal cumulative distribution (where the mean is 0 and the standard deviation is 1).

=normsDist(1.33333333432674)  returns 0.90878877498481

---

**normInv(probability, mean, deviation)**

Returns the inverse of the normal distribution for the given mean and standard deviation.

=normInv(0.90878877482188, 40, 1.5)  returns 42.0000000014901

---

**normsInv(probability)**

Returns the inverse of the standard normal cumulative distribution (where the mean is 0 and the standard deviation is 1).

=normsInv(0.90878877482188)  returns 1.33333333432674

---

**pearson(array1, array2)**

Returns the Pearson correlation coefficient for numbers in 'array1' and 'array2'. Both arrays must have the same dimensions.

=pearson({9,7,5,3,1},{10,6,1,5,3})  returns 0.69937860618024

---

**percentile(range, k)**

Returns the k-th percentile for numbers in the array/range. The 'k' argument must be in the range <0, 1>. If it's not a multiple of 1/(n - 1), the returned value is interpolated.

=percentile({1, 2, 3, 4}, 0.3)  returns 1.9

---

**PERCENTILE.EXC(array, k)**

See the description online.

---

**PERCENTILE.INC(array, k)**

See the description online.

---

**percentileAIfs(data_range, k, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the k-th percentile for numbers in the 'data_range' array/range. The numbers must meet the specified criteria. The 'k' argument must be in the range <0, 1>. If it's not a multiple of 1/(n - 1), the returned value is interpolated.
Text representations of numbers in the 'data_range' array/range are included. All text strings are ignored.
All ranges must have the same number of columns and rows. The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=percentileAIfs({1, 2, 3, 4}, 0.3, {3, 3, 3, 0.1}, ">0")  returns 1.9

=percentileAIfs({1, 2, 3, 4}, 0.3, {3, 3, 3, 0.1}, ">" & b5)  returns the k-th percentile based on the b5 cell value

---

**percentileIfs(data_range, k, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the k-th percentile for numbers in the 'data_range' array/range. The numbers must meet the specified criteria. The 'k' argument must be in the range <0, 1>. If it's not a multiple of 1/(n - 1), the returned value is interpolated.
All text strings in the 'range' array/range are ignored.
All ranges must have the same number of columns and rows.The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=percentileIfs({1, 2, "3", 4}, 0.3, {3, 3, 3, 0.1}, ">0")  returns 1.6
=percentileIfs({1, 2, 3, 4}, 0.3, {3, 3, 3, 0.1}, ">" & b5)  returns the k-th percentile based on the b5 cell value

---

**percentRank(array, x, [significance])**

Returns the rank of the value 'x' in the specified array of numbers. The returned percentage value represents the relative standing of x within 'array'.

The 'significance' argument specifies the number of decimal places in the result. If it's omitted, it's assumed to be 3.
If 'array' doesn't include 'x', the function returns an interpolated value.

=percentRank({1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, 4,)  returns 0.333

---

**PERCENTILE.EXC(array, k,[significance])**

See the description online.

---

**PERCENTILE.INC(array, k, [significance]))**

See the description online.

---

**permut(n, k)**

Returns the number of k-element permutations of the n-element set.

=permut(100, 3)  returns 970200

---

**PHI(x)**

See the description online.

---

**poisson(x, mean, cumulative)**

Returns the Poisson distribution for a given number of events 'x' where 'mean' is the distribution parameter.

If the 'cumulative' argument is 1 (true), the 'Poisson' function returns the cumulative distribution function. Otherwise, the probability mass function is returned.

=poisson(2, 5, false)  returns 0.08422433748857

---

**POISSON.DIST(x, mean, cumulative)**

See the description online.

---

**polyPoints(x, B, chi2, n, r, t, p)**

The 'polyPoints' uses results returned by 'polyReg' to calculate values lying on the specified polynomial line along with and their confidence limits for a given probability 'p'.

The 'x', 'chi2' vectors and the 'B' matrix are values returned by the 'polyReg' functions.

The 'n' argument specifies the number of elements t[i] passed to the 'polyReg' functions (the 't' argument).

The 'r' argument specifies the r-1 degree of the polynomial that the calculation is performed for. It must be greater or equal to 1 and not greater than the 'r' value passed to the 'polyReg' function.

The 't' argument is a column vector (a one-column array) containing N new values that the polynomial is calculated for.

The 'p' argument specifies the probability used to calculate the confidence limits for a given point lying on the polynomial line.

The functions returns a two-column matrix; the first column contains N values y'[i] lying on the polynomial line and the second column contains the corresponding N confidence limits eta[i] for each y'[i], such that the true value is expected to be in the range <y'[i] - eta[i], y'[i] + eta[i]> at the specified probability 'p'

Input argument used for 'polyReg'
t = {1; 2; 3; 4; 5; 6; 7}
y = {3.1; 1.2; 1.3; 2.4; 3.5; 6.5; 7.5}
sy = {0.2; 1.8; 0.2; 0.2; 1.9; 2.8; 3.8}
r = 7
Arguments returned by 'polyReg':
x[1...7]
B[1...7, 1...7]
chi2[1...7]

Calculating polynomial values for the following new points:
t = {1.5; 2.5; 3.5; 8}

=polyPoints(x, B, chi2, 7, 5, t, 0.85)  returns {1.901, 1.176; 1.112, 0.681; 1.754, 0.320; 9.930, 22.740}

=index(polyPoints(x, B, chi2, 7, 5, t, 0.85),,1)  returns {1.901; 1.112; 1.754; 9.930}

---

**polyReg(t, y, sy, r, type)**

Performs polynomial regression using orthogonal polynomials. If w(t) is a polynomial of the controlled variable 't', the 'polyReg' functions calculates the coefficients of the family of r orthogonal polynomials (from the 0 degree to the r-1 degree) and suggests the most suitable (minimal) degree of w(t).

The polynomial of the (r - 1)-th degree can be expressed as:
w(t) = x[1]*f_1(t) + x[2]*f_2(t) + ... + x[r]*f_r(t)
where the 'f_i' functions belong to a family of 'r' orthogonal polynomials defined by the lower triangular matrix B[r,r] so that for each 'f_i' which is a polynomial of the j - 1 degree:
f_i(t) = Sum<k=1, j>( B[j, k] * t^(k - 1) )
and
Sum<i=1, N>( g(i) * f_j(t[i]) * f_k(t[i]) ) = { 1 if i=j, 0 if i!=j }
for given measurement weights g(i).

The 't' argument is a column vector (a one-column array) containing N values t[i] of the controlled variable 't'.

The 'y' argument is a column vector containing N measurements y[i] of the w(t) variable.

The 'sy' argument is a column vector containing N values of normally distributed measurement errors.

The 'r' argument specifies the number of the coefficients to find (r - 1 = the maximum degree of the searched polynomial); r <= N.

If 'type'=1, the function returns the 'x' column vector of the calculated x[1...r] coefficients. By definition, all x[i] have the same standard deviation: 1. Thus the last x[i] coefficient significantly different than 0 determines the degree of the polynomial that should be used to describe w(t).

If 'type'=2, the function returns the B[r,r] matrix.

If 'type'=3, the functions returns a column vector of chi2[1...r] values that can be used to verify the goodness-of-fit for each analyzed polynomial of the i-th degree and N-i-1 degrees of freedom.

=polyReg({1; 2; 3; 4; 5}, {1.1; 1.2; 1.3; 1.4; 1.5}, {0.2; 0.2; 0.2; 0.2; 0.23}, 4, 1) returns {14.0688; 1.4979; 0; 0} - which means that the data can be described with the polynomial of the 1st degree (a line).

---

**prob(array_x, array_p, lower_limit, [upper_limit])**

Returns the cumulative probability for values in the range <lower_limit, upper_limit> based on the intervals specified by the 'array_x' and the corresponding probabilities 'array_b'.

Each 'array_b' element must be in the range <0, 1> and their sum must equal 1.
If the 'upper_limit' argument is omitted, the function returns the probability for the 'lower_limit' value.

=prob({0, 1, 2, 3}, {0.2, 0.3, 0.1, 0.4}, 1, 3)  returns 0.8

---

**quartile(array, n)**

Returns the quartile of the data set.

The 'n' argument specifies which value should be returned:
0  minimum value
1  1st quartile
2  2nd quartile
3  3rd quartile
4  maximum value

=quartile({1, 2, 4, 7, 8, 9, 10, 12}, 1)  returns 3.5

---

**QUARTILE.EXC(array, quart)**

See the description online.

---

**QUARTILE.INC(array, quart)**

See the description online.

---

**rank(x, array, [order])**

Returns the rank of a given number in the data set specified by the 'array' argument.
The rank of a number is its position obtained after sorting the data set. Equal numbers have the

same rank.

If the 'order' argument is 1/TRUE, the returned rank value corresponds to the ascending order. Otherwise, the descending order is used. If it's omitted, it's assumed to be 0.

=rank(3.5, {7, 3.5, 3.5, 1, 2}, 1)  returns 3

---

**RANK.AVG(number, ref, [order])**

See the description online.

---

**RANK.EQ(number, ref, [order])**

See the description online.

---

**rsq(array1, array2)**

Returns the square of the Pearson correlation coefficient for numbers in 'array1' and 'array2'. Both arrays must have the same dimensions.

=rsq({9, 7, 5, 3, 1}, {10, 6, 1, 5, 3})  returns 0.48913043478261

---

**skew(data1, [data2], ...)**

Returns the skewness of (the distribution consisting of) the specified numbers.

The 'v' arguments can be any numbers, arrays or references.

=skew(3, 4, 5, 2, 3, 4, 5, 6, 4, 7)  returns 0.3595430714068

---

**SKEW.P(data1, [data2], ...**

See the description online.

---

**slope(array_y, array_x)**

Finds a best-fit regression line for the known x- and y-values and returns the slope of that line.

The 'array_y' and 'array_x' arrays must have the same dimensions.

=slope({2, 3, 9, 1, 8}, {6, 5, 11, 7, 5})  returns 0.66935483870968

---

**small(array, k)**

Returns the k-th smallest value in the specified array.

=small({3, 4, 5, 2, 3, 4, 5, 6, 4, 7}, 3)  returns 3

---

**standarize(x, mean, deviation)**

Returns a normalized value 'z' for the standard normal distribution such that normDist(x, mean, deviation) = normsDist(z).

=standardize(42, 40, 1.5)  returns 1.3(3)

---

**stdev(v1, v2, ...)**

Returns the standard deviation based on a given sample.

The 'v_' arguments can be any numbers, arrays or references.

=stdev(1345, 1301, 1368, 1322, 1310, 1370, 1318, 1350, 1303, 1299)  returns 27.4639157198435

---

**STDEV.P(data1, data2, ...)**

See the description online.

---

**STDEV.S(data1, data2, ...)**

See the description online.

---

**stdevA(v1, v2, ...)**

Returns the standard deviation based on a given sample.

The 'v_' arguments can be any numbers, arrays or references.

=stdevA(1345, 1301, 1368, 1322, 1310, 1370, 1318, 1350, 1303, 1299)  returns 27.4639157198435

---

**stdevAIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the standard deviation based on a given sample from the 'data_range' array/range. The included numbers must meet the specified criteria.
Text representations of numbers in the 'data_range' array/range are included. All other text strings are ignored.
All ranges must have the same number of columns and rows.The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=stdevAIfs({5.6, 4, 4, "3", "3", 4}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<1")  returns 0.57735026918963

=stdevAIfs({5.6, 4, 4, "3", "3", 4}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<"&b5)  returns the standard deviation based on the b5 cell value

---

**stdevIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the standard deviation based on a given sample from the 'data_range' array/range. The included numbers must meet the specified criteria.
All text strings in the 'data_range' array/range are ignored.
All ranges must have the same number of columns and rows. The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=stdevIfs({5.6, 4, 4, "3", "3", 4}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<1")  returns 0

=stdevIfs({5.6, 4, 4, "3", "3", 4}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<"&b5)  returns the standard deviation based on the b5 cell value

---

**stdevP(v1, v2, ...)**

Returns the standard deviation based on the entire population.

The 'v_' arguments can be any numbers, arrays or references.

=stdevP(1345, 1301, 1368, 1322, 1310, 1370, 1318, 1350, 1303, 1299)  returns 26.0545581424825

---

**stdevPA(v1, v2, ...)**

Returns the standard deviation based on the entire population.

The 'v_' arguments can be any numbers, arrays or references.

=stdevPA(1345, 1301, 1368, 1322, 1310, 1370, 1318, 1350, 1303, 1299)  returns 26.0545581424825

---

**stdevPAIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the standard deviation based on the entire population from the 'data_range' array/range. The included numbers must meet the specified criteria.
Text representations of numbers in the 'data_range' array/range are included. All other text strings are ignored.
All ranges must have the same number of columns and rows. The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=stdevPAIfs({5.6, 4, 4, "3", "3", 3}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<1")  returns 0.43301270189222

=stdevPAIfs({5.6, 4, 4, "3", "3", 4}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<"&b5)  returns the standard deviation based on the b5 cell value

---

**stdevPIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the standard deviation based on the entire population from the 'data_range' array/range.

The included numbers must meet the specified criteria.
All text strings in the 'data_range' array/range are ignored.
All ranges must have the same number of columns and rows.The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=stdevPIfs({5.6, 4, 4, "3", "3", 3}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<1")  returns 0.5

=stdevPIfs({5.6, 4, 4, "3", "3", 3}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<"&b5)  returns the standard deviation based on the b5 cell value

---

**stEyx(array_y, array_x)**

Finds a best-fit regression line for the known x- and y-values and returns the standard error of the predicted y-values lying on that line.

=stEyx({2, 3, 9, 1, 8}, {6, 5, 11, 7, 5})  returns 3.74560674557182

---

**T.DIST(x, deg_freedom, cumulative)**

See the description online.

---

**T.DIST.2T(x, deg_freedom)**

See the description online.

---

**T.DIST.RT(x, deg_freedom)**

See the description online.

---

**T.INV(probability, deg_freedom)**

See the description online.

---

**T.INV.2T(probability, deg_freedom)**

See the description online.

---

**T.TEST(array1, array2, tails, type)**

See the description online.

---

**tDist(x, f_degrees)**

Returns the Student's t-distribution function.

The 'x' argument is the numeric value for which the distribution is calculated.
The 'f_degrees' argument is the number of degrees of freedom.

The returned value is calculated as P(X < x).

=tDist(1.96, 60)  returns 0.97267753526449

---

**timeSeries(y, l, k, p)**

Analyzes a time series: a set of measurements y[i] performed at regular intervals t[i]. The actual relationship between real y[i] values and the t[i] values is unknown. Normally distributed measurement errors are unknown. To find a trend line, the function uses moving averages and

assumes that the relationship is linear (polynomial) in the neighborhood of each y[i].

The 'y' argument is a column vector containing N measurements y[i].

The 'l' argument specifies the degree of the polynomial that should be used to approximate the trend line.

The 'k' argument specifies - for each y[i] - the range <i - k, i + k> of points that will be used to calculate the moving average and the polynomial coefficients. For points lying within the first and the last compartment, such that i <= k or i > n - k, the function extents the trend line using the coefficients calculated respectively for the points i = k + 1 and i = n - k.

The 'p' argument specifies the probability used to calculate the confidence limits for the points lying on the trend line.

The functions returns a two-column array; the first column contains N + 2k values y'[i] lying on the trend line and second column contains N + 2k values eta[i] defining the confidence limits for each y'[i], such that the true value is expected to be in the range <y'[i] - eta[i], y'[i] + eta[i]> at the specified probability 'p'.

=timeSeries({11; 8; 7; 2; 1; 3; 8; 4}, 3, 2, 0.8)
returns {7.800, 71.396; 10.800; 10.800; 8.800; 5.800; 2.771; 0.914; 4.029; 7.314; 4.171; 11.4; 45.400)

=timeSeries({11; 8; 7; 2; 1; 3; 8; 4}, 2, 3, 0.8)
returns {35.071, 10.531; 25.928, 7.154; 18.285, 4.447; 12.142, 2.490; 7.500, 1.525; 2.714, 1.647; 2.761, 2.363; 2.928, 2.188; 4.017, 2.188; 6.190, 3.573; 9.285, 6.380; 13.357, 10.264; 18.404, 15.109}

=index(timeSeries({11; 8; 7; 2; 1; 3; 8; 4}, 2, 3, 0.8),,1)
returns {30.071; 25.928; 18.285; 12.142; 7.500; 2.714; 2.761; 2.928; 4.017; 6.190; 9.285; 13.357; 18.404}

---

**tInv(p, f_degrees)**

Returns the inverse of the Student's (one-tailed) t-distribution function for a given probability value.

=tInv(0.97267753526449, 60)  returns 1.95999983312891

---

**trend(array_y, [array_x], [array_n], [const_b])**

Finds a best-fit regression line for the known x- and y-values and returns a vector (a one-column array) of predicted y-values for the new x-values specified in the 'array_n' array.

If the 'const_b' argument is 0/FALSE, the 'b' parameter is assumed to be 0. The default value of 'const_b' is 1.

If the 'array_x' argument is omitted, it's assumed to be an array containing integers from 1 to the number of elements in 'array_y'.
If the 'array_n' argument is omitted, it's assumed to be the same as 'array_x'.

The 'array_y' and 'array_x' arrays must have the same dimensions.

If the procedure is not convergent, the function returns #NUM!.

=trend({2, 3, 9, 1, 8}, {6, 5, 11, 7, 5}, {12, 13, 14}, 1)  returns {8.08064516129032; 8.75; 9.41935483970968 }

---

**trimMean(array, percent)**

Returns the mean of the specified data set after excluding a given percentage of numbers from the top and bottom of that data set.

The 'percent' argument must be in the range <0, 1>. It's rounded to the nearest multiple of 2 so that the data set elements can be exclude symmetrically.

=trimMean({4, 5, 6, 7, 2, 3, 4, 5, 1, 2, 3}, 0.2)  returns 3.77777777777778

---

**tTest(array1, array2, tails)**

Performs the T-test and returns the probability that the means in the 'array1' and 'array2' data sets are significantly different.

The 'tails' argument specifies the number of distribution tails: 1 or 2.

=tTest({3, 4, 5, 8, 9, 1, 2, 4, 5}, {6, 19, 3, 2, 14, 4, 5, 17, 1}, 2)  returns 0.1919958849411

---

**tTest2(array1, array2)**

Returns the t-statistic for the 'array1' and 'array2' data sets.

The returned value can be compared against critical t-statistic values obtained with the 'tInv' function.

=tTest2({3, 4, 5, 8, 9, 1, 2, 4, 5}, {6, 19, 3, 2, 14, 4, 5, 17, 1})  returns -1.3622298275595

---

**var(v1, v2, ...)**

Returns variance based on a given sample.

The 'v_' arguments can be any numbers, arrays or references.

=var(1345, 1301, 1368, 1322, 1310, 1370, 1318, 1350, 1303, 1299)  returns 754.266666666667

---

**VAR.P(data1, data2, ...)**

See the description online.

---

**VAR.S(data1, data2, ...)**

See the description online.

---

**varA(v1, v2, ...)**

Returns variance based on a given sample.

The 'v_' arguments can be any numbers, arrays or references.

=varA(1345, 1301, 1368, 1322, 1310, 1370, 1318, 1350, 1303, 1299)  returns 754.266666666667

---

**varAIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the variance based on a given sample from the 'data_range' array/range. The included numbers must meet the specified criteria.
Text representations of numbers in the 'data_range' array/range are included. All other text strings are ignored.
All ranges must have the same number of columns and rows.The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=varAIfs({5.6, 4, 3, "3", "3", 4}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<1")  returns 0.25

=varAIfs({5.6, 4, 3, "3", "3", 4}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<"&b5)  returns the variance based on the b5 cell value

---

**varIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the variance based on a given sample from the 'data_range' array/range. The included numbers must meet the specified criteria.
All text strings in the 'data_range' array/range are ignored.
All ranges must have the same number of columns and rows. The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=varIfs({5.6, 4, 3, "3", "3", 4}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<1")  returns 0.5

=varIfs({5.6, 4, 3, "3", "3", 4}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<"&b5)  returns the variance based on the b5 cell value

---

**varP(v1, v2, ...)**

Returns variance based on the entire population.

The 'v_' arguments can be any numbers, arrays or references.

=varP(1345, 1301, 1368, 1322, 1310, 1370, 1318, 1350, 1303, 1299)  returns 678.84

---

**varPA(v1, v2, ...)**

Returns variance based on the entire population.

The 'v_' arguments can be any numbers, arrays or references.

=varPA(1345, 1301, 1368, 1322, 1310, 1370, 1318, 1350, 1303, 1299)  returns 678.84

---

**varPAIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the variance based on the entire population from the 'data_range' array/range. The included numbers must meet the specified criteria.
Text representations of numbers in the 'data_range' array/range are included. All other text strings are ignored.
All ranges must have the same number of columns and rows. The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=varPAIfs({5.6, 4, 3, "3", "3", 4}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<1")  returns 0.1875

=varPAIfs({5.6, 4, 3, "3", "3", 4}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<"&b5)  returns the variance based on the b5 cell value

---

**varPIfs(data_range, if_range1, criteria1 [, if_range2, criteria2, ...])**

Returns the variance based on the entire population from the 'data_range' array/range. The included numbers must meet the specified criteria.
All text strings in the 'data_range' array/range are ignored.
All ranges must have the same number of columns and rows. The criteria can be one of the following :

(1) a number,

(2) a text string beginning with the =,>,>=,<,<=,<> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD) or the & followed by a single cell address,

(3) a text string optionally containing wildcard characters '?' (any character) or '*' (any string, including an empty string). To search for ? or * place a tilde (~) before them.

If there are no values meeting the specified criteria, the #N/A! error code is returned.

=varPIfs({5.6, 4, 3, "3", "3", 4}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<1")  returns 0.25

=varPIfs({5.6, 4, 3, "3", "3", 4}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<"&b5)  returns the variance based on the b5 cell value

---

**weibull(x, alpha, beta, cumulative)**

Returns the Weibull distribution for a geiven 'x'.

The 'alpha' and 'beta' arguments are the distribution parameters.

If the 'cumulative' argument is 1 (true), 'weibull' returns the cumulative distribution function. Otherwise, the probability mass function is returned.

=weibull(105, 20, 100, true)  returns 0.92958139006928

---

**WEIBULL.DIST(x, alpha, beta, cumulative)**

See the description online.

---

**Z.TEST(array, x, [sigma])**

See the description online.

---

**zTest(array, x, [sigma])**

Performs Z-test and returns the two-tailed probability that 'x' doesn't belong to the population represented by the n-element 'array' sample.

The returned value is calculated as
1 - normsDist((m - x)/deviation/sqrt(n)) where 'm' is the sample mean.

The 'deviation' argument is the known population standard deviation. If it's omitted, the sample deviation is used.

=zTest({3, 6, 7, 8, 6, 5, 4, 2, 1, 9}, 4,)  returns 0.09057420261958

---

**zTest2(array, x, [sigma])**

Returns the z-statistic for a given variable and a given data set.
The population is represented by the n-element 'array' sample.

The returned value is calculated as
z = (m - x)/deviation/sqrt(n) where 'm' is the sample mean.

The 'deviation' argument is the known population standard deviation. If it's omitted, the sample deviation is used.

=zTest2({3, 6, 7, 8, 6, 5, 4, 2, 1, 9}, 4,)  returns 1.33722748248063

## Optimization/Solver Functions

---

**LProg(A, s, b, c, vector, [epsilon], [m])**

Finds one or more solutions for a given linear programming problem for 'm' constraints of the type
(1) sum<j=1, n>(x[j]*a[i,j]) <= b[i]
(2) sum<j=1, n>(x[j]*a[i,j]) = b[i]
(3) sum<j=1, n>(x[j]*a[i,j]) >= b[i]
and a maximized objective function sum<j=1, n>(x[j]*c[j]).

The 'A', 'b' and 'c' arguments contains the a[], b[] and c[] elements as specified above; 'A' is a 'm' x 'n' matrix, 'b' is a one-column, m-element vector, 'c' is a one-column n-element vector.

The 's' argument is a one-column, m-element vector specifying used constraint types:
(1) s[i]=1 (<=)
(2) s[i]=0 (=)
(3) s[i]=-1 (>=)

The 'vector' argument specifies what should be returned:
0 - the number of optimum vectors found
i > 0 - the i-th optimum vector.

The 'epsilon' argument specifies the precision used during the calculation: intermediate coefficients smaller than 'epsilon' will be treated as 0. If it's omitted, it's assumed to be 1e-08.

The 'm' argument specifies some number significantly bigger than other initial coefficients. It's used as a coefficient for artificial variables. If it's omitted, the function will guess the proper value based on the input data.

All the returned x[i] values can be any non-negative numbers. If the problem allows negative values for some x[i] variables, each such variable must be replaced by a pair of new non-negative variables.

To minimized the objective function, change its sign and use the default maximizing procedure.

If the objective function is unconstrained, LProg returns the #N/A! error value.

If the constraints are inconsistent and the solution can't be found, LProg returns the #NULL! error value.

The initial constraints:
2*x1 + x2 + 2*x3 <= 20
3*x1 - x2 + x3 = 10
the objective function: 3*x1 + x2 - x3 -> max

=LProg({2, 1, 1; 1, 1, 3}, {1; 0}, {5; 10}, {4; 2; 2},1,,)  returns {1; 0; 3}

---

**LProgBin(A, s, b, c, bin, vector, [epsilon], [m])**

Finds one or more solutions for a given linear programming problem for 'm' constraints of the type
(1) sum<j=1, n>(x[j]*a[i,j]) <= b[i]
(2) sum<j=1, n>(x[j]*a[i,j]) = b[i]
(3) sum<j=1, n>(x[j]*a[i,j]) >= b[i]
and a maximized objective function sum<j=1, n>(x[j]*c[j])
where some or all of the x[i] values are required to be 0 or 1.

The 'A', 'b' and 'c' arguments contains the a[], b[] and c[] elements as specified above; 'A' is a 'm' x 'n' matrix, 'b' is a one-column, m-element vector, 'c' is a one-column n-element vector.

The 's' argument is a one-column, m-element vector specifying used constraint types:
(1) s[i]=1 (<=)
(2) s[i]=0 (=)
(3) s[i]=-1 (>=)

The 'bin' argument is a one-column m-element vector defining the type of each x[i] variables. If bin[i] is 1, the x[i] variable can be only 0 or 1, otherwise bin[i] equals 0.

The 'vector' argument specifies what should be returned:
0 - the number of optimum vectors found
i > 0 - the i-th optimum vector.

The 'epsilon' argument specifies the precision used during the calculation: intermediate coefficients smaller than 'epsilon' will be treated as 0. If it's omitted, it's assumed to be 1e-08.

The 'm' argument specifies some number significantly bigger than other initial values. It's used as a coefficient for artificial variables. If it's omitted, the function will guess the proper value based on the input data.

All the returned x[i] values can be any non-negative numbers. If the problem allows negative values for some x[i] variables, each such variable must be replaced by a pair of new non-negative variables.

To minimized the objective function, change its sign and use the default maximizing procedure.

If the objective function is unconstrained, LProgBin returns the #N/A! error value.

If the constraints are inconsistent and the solution can't be found, LProgBin returns the #NULL! error value.

The initial constraints: 2.5*x1 + 1.5*x2 + x3 <= 15
x1 + 1.5*x2 + 3*x3 = 10
the objective function: x1 + x2 + 2*x3 -> max

=LProgBin({2.5, 1.5, 1; 1, 1.5, 3}, {1; 0}, {15; 10}, {1; 1; 2}, {1; 0; 1}, 1,,)  returns {1; 4; 1}

---

**LProgInt(A, s, b, c, int, vector, [epsilon], [m])**

Finds one or more solutions for a given linear programming problem for 'm' constraints of the type
(1) sum<j=1, n>(x[j]*a[i,j]) <= b[i]
(2) sum<j=1, n>(x[j]*a[i,j]) = b[i]
(3) sum<j=1, n>(x[j]*a[i,j]) >= b[i]
and a maximized objective function sum<j=1, n>(x[j]*c[j])
where some or all of the x[i] values are required to be integer.

The 'A', 'b' and 'c' arguments contains the a[], b[] and c[] elements as specified above; 'A' is a 'm' x 'n' matrix, 'b' is a one-column, m-element vector, 'c' is a one-column n-element vector.

The 's' argument is a one-column, m-element vector specifying used constraint types:
(1)  s[i]=1 (<=)

(2) s[i]=0 (=)
(3) s[i]=-1 (>=)

The 'int' argument is a one-column m-element vector defining the type of each x[i] variables. If int[i] is 1, the x[i] variable must be integer, otherwise int[i] equals 0.

The 'vector' argument specifies what should be returned:
0 - the number of optimum vectors found
i > 0 - the i-th optimum vector.

The 'epsilon' argument specifies the precision used during the calculation: intermediate coefficients smaller than 'epsilon' will be treated as 0. If it's omitted, it's assumed to be 1e-08.

The 'm' argument specifies some number significantly bigger than other initial values. It's used as a coefficient for artificial variables. If it's omitted, the function will guess the proper value based on the input data.

All the returned x[i] values can be any non-negative numbers. If the problem allows negative values for some x[i] variables, each such variable must be replaced by a pair of new non-negative variables.

To minimized the objective function, change its sign and use the default maximizing procedure.

If the objective function is unconstrained, LProgInt returns the #N/A! error value.

If the constraints are inconsistent and the solution can't be found, LProgInt returns the #NULL! error value.

The initial constraints: 2.5*x1 + 1.5*x2 + x3 <= 15
x1 + 1.5*x2 + 3*x3 = 10
the objective function: x1 + x2 + 2*x3 -> max

=LProgInt({2.5, 1.5, 1; 1, 1.5, 3}, {1; 0}, {15; 10}, {1; 1; 2}, {1; 0; 1}, 1,,)  returns {4; 0; 2}

---

**minMC(f(x), x, v, from, to, points)**

Perform n-dimensional minimization of a given function using the MonteCarlo method. The function generates a number of random points in n-dimensions and returns the point for which the value of the function is the smallest. This method can be used to obtain the initial vector for the 'minSimplex' function.

The 'f(x)' argument is a reference to a cell containing a desirable function (a numeric formula) with 'n' variables.

The 'x' argument is a reference to a one-column vector containing 'n' cells (numbers or formula returning numbers) that represent variables in f(x).

The 'v' argument is a one-column vector indicating how to use a given variable: if v[i] is 0, the corresponding i-th variable has a fixed value and won't be modified, if v[i] is 0, the i-th variable can be modified.

The 'from' and 'to' arguments specify the search limits for all x[i].

The 'points' argument specifies how many sample n-dimensional points should be generated.

Note: Since the points are generated randomly for each calculation, the results change after each update.

If 'f(x)' is not a reference to a cell containing a valid numeric formula or if 'x' is not a reference to 'n' not empty cells containing numbers (or numeric formulas), 'minMC' returns the #REF! error.

Using the RC notation:

The r2c4 cell contains:
=exp((r3c4-1)*r3c4 + (r4c3-2)*r4c3 + (r5c3-3)*r5c3) - 10*exp(-((r3c4-3)*(r3c4-3) + (r4c4-3)*(r4c4-3) + (r5c4-3)*(r5c4-3)))

The r3c4:r5c4 range contains any numbers.

=minMC(r2c4, r3c4:r5c4, {1;1;1}, -5, 5, 10000)  returns {0.675, 2.725, 2.960}

Using the A1 notation:

The D2 cell contains:
=exp((D3-1)*D3 + (C4-2)*C4 + (C5-3)*C5) - 10*exp(-((D3-3)*(D3-3) + (D4-3)*(D4-3) + (D5-3)*(D5-3)))

The D3:D5 range contains any numbers.

=minMC(D2, D3:D5, {1; 1; 1}, -5, 5, 10000)  returns {0.554969, 2.661842, 3.279377}

---

**minSimplex(f(x), x, v, [start], [epsilon], [max_steps], result)**

Performs n-dimensional minimization of a given function using the downhill simplex method.

The 'f(x)' argument is a reference to a cell containing a desirable function (a numeric formula) with 'n' variables.

The 'x' argument is a reference to a one-column vector containing 'n' cells (numbers or formula returning numbers) that represent variables in f(x). The initial values of 'x' can be obtained with the 'minMC()' function.

The 'v' argument is a one-column vector indicating how to use a given variable: if v[i] is 0, the corresponding i-th variable has a fixed value and won't be modified, if v[i] is 1, the i-th variable will be modified.

The 'start' argument is the initial simplex value. If it's 0 or if it's omitted, it's assumed to be 1e-05.

The 'epsilon' argument is the precision used to evaluate the changes of the function value. If it's 0 or if it's omitted, it's assumed to be 1.0e-25.

The 'max_steps' argument specifies the maximum allowable number of steps the function can perform. The default value is 2000.

The 'result' argument specifies what should be returned.

If 'result'=1, the function returns the 'x' vector for the found minimum 'f(x)' value.
If 'result'=2, the function returns the found minimum f(x) value.
If 'result'=3, the function returns the number of steps that it performed.

If 'f(x)' is not a reference to a cell containing a valid numeric formula or if 'x' is not a reference to 'n' not empty cells containing numbers (or numeric formulas), 'minSimplex' returns the #REF! error. If the procedure is not convergent after performing 'max_steps' steps, the function returns the #NUM! error code.

Using the RC notation:

The r2c4 cell contains:
=exp((r3c4-1)*r3c4 + (r4c3-2)*r4c3 + (r5c3-3)*r5c3) - 10*exp(-((r3c4-3)*(r3c4-3) + (r4c4-3)*(r4c4-3) + (r5c4-3)*(r5c4-3)))

The r3c4:r5c4 range contains:
{0.5; 3; 2}

=minSimplex(r2c4, r3c4:r5c4, {1;1;1},,,,1)  returns {0.59358762694774; 3.00000004102304; 2.9999999389697}

=minSimplex(r2c4, r3c4:r5c4, {1;1;1},,,,2)  returns 0.75509

=minSimplex(r2c4, r3c4:r5c4, {1;1;1},,,,3)  returns 275

Using the A1 notation:

The D2 cell contains:
=exp((D3-1)*D3 + (C4-2)*C4 + (C5-3)*C5) - 10*exp(-((D3-3)*(D3-3) + (D4-3)*(D4-3) + (D5-3)*(D5-3)))

The D3:D5 range contains:
{0.5; 3; 2}

=minSimplex(D2, D3:D5, {1;1;1},,,,1)  returns {0.59358762694774; 3.00000004102304; 2.9999999389697}

=minSimplex(D2, D3:D5, {1;1;1},,,,2)  returns 0.75509

=minSimplex(D2, D3:D5, {1;1;1},,,,3)  returns 275

---

**QProg(A, s, b, p, C, vector, [epsilon])**

Finds one or more solutions for a given quadratic programming problem for 'm' constraints of the following types:
(1) sum<j=1, n>(x[j]*a[i,j]) <= b[i]
(2) sum<j=1, n>(x[j]*a[i,j]) = b[i]
(3) sum<j=1, n>(x[j]*a[i,j]) >= b[i]
and a maximized objective function

p^T*x - x^T*C*x

where 'p' is a (column) n-element vector and 'C' is a positively determined 'n' x 'n' matrix representing a quadratic form.

The 'A', 'b', 'p' and 'C' arguments contains the a[], b[], p[] and c[] elements as specified above; 'A' is a 'm' x 'n' matrix, 'b' is a (column) m-element vector, 'p' is a (column) n-element vector, 'C' is a 'n' x 'n' matrix.

The 's' argument is a column m-element vector that determines constraint types:
(1) s[i]=1 (<=)
(2) s[i]=0 (=)
(3) s[i]=-1 (>=)

The 'vector' argument specifies what should be returned:
0 - the number of optimum vectors found
i > 0 - the i-th optimum vector.

The 'epsilon' argument specifies the precision used during the calculation: intermediate coefficients smaller than 'epsilon' will be treated as 0. If it's omitted, it's assumed to be 1e-08.

All x[i] can be any non-negative numbers. If the problem allows negative values for some x[i] variables, each such variable must be replaced by a pair of new non-negative variables.To minimized the objective function, change its sign and use the default maximizing procedure.

If the objective function is unconstrained, QProg returns the #N/A! error value.

If the constraints are inconsistent and the solution can't be found, QProg returns the #NULL! error value.

Example:

x1 + 2*x2 <= 10
x1 + x2 <= 9
x1 >= 0, x2 >= 0
f(x1, x2) = 10*x1 + 25*x2 - 10x1^2 - x2^2 - 4*x1*x2 -> max

A = {1, 2; 1, 1}
s = {1, 1}
b = {10; 9}
p = {10; 25}
C = {10, 2; 2, 1}

=QProg({1, 2; 1, 1}, {1; 1}, {10; 9}, {10; 25}, {10, 2; 2, 1}, 1,)  returns {0; 5}

---

**root(f(x), x, x0, x1, [epsilon])**

Finds a root in one dimension for a given function. The equation has the form f(x) = 0. The function uses modified secant method and it's always convergent for 'f(x)' that is continuous between 'x0' and 'x1'.

The 'f(x)' argument is a reference to a cell containing a desirable function/formula with one variable.

The 'x' argument is a single cell reference that represent the variable.

The 'x0' and 'x1' arguments specify the initial search limits, such that f(x0)*f(x1) < 0.

The 'epsilon' argument specifies the precision used to evaluate the changes of the function value. If it's 0 or if it's omitted, it's assumed to be 1.0e-15.

If 'f(x)' is not a reference to a function or if 'x' is not a reference to a numeric cell, 'root' returns the #REF! error.

The r2c3 cell contains:
=r3c3*r3c3 + sin(r3c3) - 5

=root(r2c3, r3c3, 0, 3, ) returns 2.02521163744482

=root(r2c3, r3c3, -3, 0, ) returns -2.38467666014657

## Matrix, Eq. Systems Functions

---

**mtxBkSubst(U, B)**

Performs back substitution for a given upper triangular 'n' x 'n' 'U' matrix and a 'n' x 'l' 'B' matrix containing 'l' vectors of right-hand values.

The 'mtxBkSubst' function returns 'l' column vectors that are - typically - solutions of the equation L*U*x=B.

=mtxBkSubst({1, 0.5, 0.3333333; 0, 0.28867507685978, 0.2886752500649; 0, 0, 0.07453530110679}, {0.58333; -0.25979035258533; -0.03740870232733}) returns {0.94965127674073; -0.39804764314169; -0.50189241569889}

---

**mtxChl(A, B, [iterations], result)**

Performs Cholesky decomposition of a given symmetric, positively determined 'n' x 'n' 'A' matrix (x^T*A*x > 0 for each x <> 0). The 'mtxChl' function can be used to:
(1) solve a linear equation set A*x=B, where 'A' is a 'n' x 'n' matrix, 'x' is an n-element column vector and 'B' is a 'n' x 'l' matrix,
(2) find the upper triangular 'U' matrix such that U^T*U = A,
(3) find the inverse and determinant of 'A'.

The 'A' and 'B' arguments specify the matrices used in the A*x = B equation. The 'B' argument is

used only when 'result'=1. If 'B' has 'l' columns (l > 1), the procedure will use each column as different right-hand values and it will returns 'l' solution (column) vectors.

The 'iteration' argument specifies the number of additional improving iterations that should be performed when finding the 'x' vector(s). The default value is 0.

The 'result' argument specifies what should be returned.

If 'result'=1, the functions returns 'l' solution vectors of the A*x=B equation system.

If 'result'=2, the function returns an upper triangular 'n' x 'n' 'U' matrix such that U^T*U=A.

If 'result'=3, the function returns a transposed upper triangular 'n' x 'n' 'U^T' matrix such that U^T*U=A.

If 'result'=4, the function returns the inverse of 'A'.

If 'result'=5, the function returns the determinant of 'A'.

=mtxChl({1, 0.5, 0.3333333; 0.5, 0.3333333, 0.25; 0.33333333, 0.25, 0.2}, {0.58333; 0.21667; 0.11666}, 3, 1)  returns {0.9496504220468; -0.3980425149853; -0.5018975438522}

=mtxChl({1, 0.5, 0.3333333; 0.5, 0.3333333, 0.25; 0.33333333, 0.25, 0.2},,,2)  returns {1, 0.5, 0.3333333; 0, 0.28867507685978, 0.2886752500649; 0, 0, 0.07453530110679}

---

**mtxDiag(n, x)**

mtxDiag(n, A)

Creates and returns a 'n' x 'n' diagonal matrix. The first version of the function uses the 'x' value for all diagonal elements. The second version uses the subsequent elements of the 'A' vector/matrix to set the diagonal elements in the returned matrix.

=mtxDiag(3, 1)  returns {1, 0, 0; 0, 1, 0; 0, 0, 1}

=mtxDiag(3, {1, 2, 3})  returns {1, 0, 0; 0, 2, 0; 0, 0, 3}

---

**mtxFwSubst(L, B)**

Performs forward substitution for a given lower triangular 'n' x 'n' 'L' matrix and a 'n' x 'l' 'B'

matrix containing 'l' vectors of right-hand values.

The 'mtxFwSubst' function returns 'l' (column) vectors.

=mtxFwSubst({1, 0, 0; 0.5, 0.28867507685978, 0; 0.3333333, 0.2886752500649, 0.07453530110679}, {0.58333; 0.21667; 0.11666}) returns {0.58333; -0.25979035258533; -0.03740870232733}

---

**mtxGauss(A, B, [iterations], [scaling], result)**

Performs Gauss transformation with partial pivoting, optional improving iterations and scaling. The 'mtxGauss' function can be used to:
(1) solve a linear equation set A*x=B, where 'A' is a 'n' x 'n' matrix, 'x' is a (n-row) column vector and 'B' is a 'n' x 'l' matrix,
(2) find the LU decomposition of 'A'
(3) find the inverse of 'A',
(4) obtain the determinant of 'A'.

The 'A' and 'B' arguments specify the matrices used in the A*x = B equation. The 'B' argument is used only when 'result'=1. If 'B' has 'l' columns (l > 1), the procedure will use each column as different right-hand values and it will return 'l' solution vectors.

The 'iterations' argument specifies the number of improving iterations that should be performed when finding the 'x' vector(s). The default value is 0.

The 'scaling' argument specifies whether the initial coefficients should be scaled (to minimize roundoff problems if the equations are not well balanced). If 'scaling' equals 1, 'mtxGauss' will scale the coefficients. If 'scaling'=0, no scaling will be performed. The default value is 0.

The 'result' argument specifies what should be returned:

If 'result'=1, the function solves A*x=B and returns 'l' solution vectors, where 'l' is the number of columns in 'B'.

If 'result' is '2', the function returns a 'n' x 'n' lower triangular matrix 'L' such that L*U=A' where A' is a row-wise permutation of A.

If 'result'=3, the function returns a 'n' x 'n' upper triangular matrix 'U' such that L*U=A' where A' is a row-wise permutation of A.

If 'result'=4, the function returns a 'n' x 'n' permutation matrix 'P' such that P*A=A' where 'A" is obtained from 'A' by exchanging rows in the partial pivoting procedure.

If 'result'=5, the function returns the inverse of 'A'

If 'result'=6, the function returns the determinant of 'A'

The initial equation system:
2*x1 - x2 + 3*x3 = 6
1*x1 + 3*x2 + x3 = 8
4*x1 + x2 + x3 = 2

=mtxGauss({2, -1, 3; 1, 3, 1; 4, 1, 1}, {6; 8; 2},,,1)  returns {-0.75; 1.875; 3.125}

=mtxGauss({2, -1, 3; 1, 3, 1; 4, 1, 1},,,,6)  returns -32

=mtxGauss({2, -1, 3; 1, 3, 1; 4, 1, 1},,,,5)  returns {-0.0625, -0.125, 0.3125; -0.09375, 0.3125, -0.03125; 0.34375, 0.1875, -0.21875}

---

**mtxLSC(A, b, E, d, [minMaxRatio], result)**

Solves a given over-determined equation set A*x~=b with constraints in the form of E*x=d, where E is a 'l' x 'n' matrix and l < n.

The 'A' argument is a 'm' x 'n' matrix and the 'b' argument is a column vector with 'm' rows. The 'E' argument is a 'l' x 'n' matrix and the 'd' argument is a column vector with 'l' rows.

The 'minMaxRatio' argument specifies the minimum allowable ratio between the minimum and maximum singular value below which a given singular value will be treated as 0.

The 'result' argument specifies what should be returned.

If 'result' is 1, the function returns the n-row column vector 'x'.

If 'result'=2, the function returns the rest value 'r' such that (A*x - b)^T*(A*x - b) = r.

If the SVD decomposition is not convergent, the #NUM! error value is returned.

=mtxLSC({1, 2; 2, 3; 7, 1}, {1; 2; 3}, {1, 3},{2},,1)  returns {0.2; 0.6}

=mtxLSC({1, 2; 2, 3; 7, 1}, {1; 2; 3}, {1, 3},{2},,2)  returns 0.319512195121195

---

**mtxLSW(A, b, G, [minMaxRatio], result)**

Solves a given over-determined equation set A*x~=b with weights specified by the 'G' matrix so that the minimized function has the form of r = (A*x - b)^T*G*(A*x - b) -> min.

The 'A' argument is a 'm' x 'n' matrix and the 'b' argument is a column vector with 'm' rows.

The 'G' argument is a 'n' x 'n' matrix with weights. If 'G'='I', the function is an equivalent of 'mtxSVD'.

The 'minMaxRatio' argument specifies the minimum allowable ratio between the minimum and maximum singular value below which a given singular value will be treated as 0.

The 'result' argument specifies what should be returned.

If 'result' is 1, the function returns the n-row column vector 'x' .

If 'result'=2, the function returns the rest value 'r' such that (A*x - b)^T*G*(A*x - b) = r.

If the SVD decomposition is not convergent, the #NUM! error value is returned.

=mtxLSW({1, 2; 2, 3; 7, 1}, {1; 2; 3}, {1,0,0;0,1,0;0,0,1},,1)  returns {0.37476459510358; 0.38418079096045}

=mtxLSW({1, 2; 2, 3; 7, 1}, {1; 2; 3}, {1, 0.5 , 0.2; 0.5, 1, 0.7; 0.2, 0.7, 1},,1)  returns  {0.38927943760984; 0.3585237258348}

---

**mtxMlt(v1, v2, ...)**

Multiplies the subsequent specified matrices. The number of rows in each next matrix and the number of columns in the previous result must be the same.

=mtxMlt({1;2;3}, {1, 2, 3}, {2; 2; 2})  returns {12; 24; 36}

---

**mtxRand(n, [seed1], [seed2])**

Generates a column vector (a one-column matrix) of 'n' random floating point numbers from the range (0, 1).

The function uses two combined MLCG generators to create a series of about 2.3*10^18 numbers.

The 'n' argument specifies the size of the returned vector.

The 'seed1' and 'seed2' arguments determine the starting numbers for the first and the second generator.
If both those values are omitted, the first call to 'mtxRand' will always starts from the same hard-coded values of 'seed1' and 'seed2' and subsequent calls will use the previously generated values returning partial series occurring one after another.
If both 'seed1' and 'seed2' are specified, 'mtxRand' will always be generating one and the same partial series.

To generate two or more independent partial series that do not occur one after another, specify a fixed 'seed1' value and skip the 'seed2' argument.

=mtxRand(4, 10000, 25000)  returns {0.71261246808435; 0.28457538373252; 0.42105025462307; 0.93072516522912}

---

**mtxRand2(n, [seed1], [seed2], [type], [v1], [v2])**

Generates a column vector (a one-column matrix) of random floating point numbers for the specified distribution type.

The 'n' argument specifies the size of the returned vector.

The 'seed1' and 'seed2' arguments determine the starting numbers for the first and the second MLCG generator.
If both those values are omitted, the first call to 'mtxRand2' will always starts from the same hard-coded values of 'seed1' and 'seed2' and subsequent calls will use the previously generated values returning partial series occurring one after another.
If both 'seed1' and 'seed2' are specified, 'mtxRand2' will always be generating one and the same partial series.

To generate two or more independent partial series that do not occur one after another, specify a fixed 'seed1' value and skip the 'seed2' argument.

The 'type' argument specifies the distribution type:

0 - uniform (0, 1)
1 - normal with the 'v1' mean and the 'v2' standard deviation
2 - exponential with the 'v1' mean
3 - Poisson with the 'v1' mean
4 - Bernoulli with the 'v1' probability

5 - geometric with the 'v1' probability

If 'type' is omitted, it's assumed to be 0.

If a given distribution type doesn't require the 'v1' and/or 'v2' arguments, they should be omitted.

=mtxRand2(4, 10000, 25000,,,)  returns {0.71261246808435; 0.28457538373252; 0.42105025462307; 0.93072516522912}

---

**mtxSeries(n, x, [step])**

Generates a column vector (a one-column matrix) of 'n' subsequent numbers or generic date/time strings.

The 'x' argument specifies the first element of the series. This can be a number or date/time string.

The 'step' argument specifies the value by which the subsequent numbers/dates are incremented. If it's omitted, it's assumed to be 1 for numeric series and "P1D" (one day) for date series. The general form of the date/time period is:
[+|-]PnYnMnDTnHnMnS.nnn .

For example:
P1Y2M10DT11H5M4S.355 represents a period of 1 year, 2 months, 10 days, 11 hours, 5 minutes, 4 seconds, 355 thousandths.
PT12H7M represents a period of twelve hours and seven minutes

=mtxSeries(5, 1, 0.1)  returns {1; 1.1; 1.2; 1.3; 1.4}

=mtxSeries(5, "2005-12-01", "P1D")  returns {"2005-12-01"; "2005-12-02"; "2005-12-03"; "2005-12-04"; "2005-12-05"}

=mtxSeries(5, "12:50:00", "PT2M")  returns {"12:50:00"; "12:50:02"; "12:50:04"; "12:50:06"; "12:50:08"}

---

**mtxSVD(A, B, [minMaxRatio], result)**

Performs SVD decomposition of a given matrix A[m,n]. The 'mtxSVD' function can be used to:
(1) solve an over-determined equation system A*x~=B where 'B' is a 'm' x 'l' matrix containing 'l' right-hand values vectors,
(2) find the number of non-zero singular values

(3) find 'l' rest values r[i] such that (A*x - Bi)^T*(A*x - Bi) = r[i] where 'Bi' is the i-th column of 'B'
(4) find the (pseudo-)inverse 'A+' of 'A'
(5) find the U[m,n] matrix, S[m,n] matrix and V[n,n] matrix such that U*S*V^T=A

The 'A' and 'B' arguments are matrices used in the equation system A*x~=B. The 'B' argument is ignored for 'result' value other than 1 or 3.

The 'minMaxRatio' argument specifies the minimum allowable ratio between the minimum and maximum singular value below which a given singular value will be treated as 0. The default value (and the smallest possible) value is 1.0e-15.

The 'result' argument specifies what should be returned.

If 'result'=1, the function returns 'l' n-row column vectors that are solutions of A*x~=B.

If 'result'=2, the function returns a column vectors of 'l' rest values.

If 'result'=3, the function returns the number of non-zero singular values.

If 'result' is 4, the function the inverse 'A+' of 'A'.

If 'result' is 5, 6 or 7, the function returns respectively the 'U', 'S' and 'V' matrices described above.

If the SVD decomposition is not convergent, the #NUM! error value is returned.

=mtxSVD({1, 2; 2, 3; 7, 1}, {1; 2; 3},,1)  returns {0.37476459510358; 0.38418079096045}

=mtxSVD({1, 2; 2, 3; 7, 1}, {1; 2; 3},,3)  returns 0.030131826742

=mtxSVD({1, 2; 2, 3; 7, 1},,,6)  returns {7.68114574786861, 0; 0, 3; 0, 0}

---

**mtxT(A)**

Transpose a given matrix and returns the result.

=mtxT({1, 2, 3; 4, 5, 6})  returns {1, 4; 2, 5; 3, 6}