

<https://www.citadel5.com>

GS-Calc 24

1. Introducing GS-Calc
2. Notational conventions (numbers, formulas, dates)
3. **Performance and compatibility**
 1. Ensuring high performance
 2. GS-Calc and ODF file format - compatibility with other spreadsheets
 3. Formula conversions to/from other spreadsheets
4. **Entering data**
 1. Editing and filling cells
 2. Copying cells vs duplicating copied cells
 3. Using AutoScroll Range to automatically move between cells
 4. Entering numbers, text strings, date/time strings and error codes
 5. Using cell references
 6. Merging and splitting cells
 7. Entering formulas
 8. Using formula composer
 9. Using array formulas
 10. Update options
 11. Replacing formulas automatically in worksheets
 12. Entering Unicode characters
 13. Selecting cells, navigation, scrolling, key shortcuts
 14. Performing multiple selections to combine their values
 15. Inserting series
 16. Inserting sequences and series using formulas
 17. Drop-down lists
 18. Using the Increment, Decrement and Operations commands
 19. Spell-checking and adding Hunspell dictionaries
 20. Generating and inserting passwords

5. Worksheet views

1. Splitting worksheet views
2. Row and column synchronization options
3. Selecting functions to display within the table toolbar and related options
4. Hiding view elements / maximizing the table view display areas
5. Cross-highlighting columns/rows for better readability
6. Using the Dark Mode

6. Pivot Tables

1. Using pivot tables
2. Creating Pivot Table Reports
3. Pivot Table Functions

7. Monte Carlo Simulations

1. Using Monte Carlo simulations

8. Lookup functions

1. vLookUp() function
2. hLookUp() function
3. Match() function
4. Unique() function
5. Sort() function

9. Output functions

1. `imageIf()`, `messageIf()`, `playSoundIf()`, `setFormatIf()`

10. Solver functions

1. Solver functions

11. All function categories

1. Mathematical Functions
2. Statistical Functions
3. Text Functions
4. Date/Time Functions
5. Logical Functions
6. Informational Functions
7. Lookup & Reference Functions
8. Solver Functions

- 9. Financial Functions
- 10. Output Functions
- 11. Engineering Functions
- 12. Matrix and Equation System Functions

12. Regular Expressions

- 1. Using regular expressions in formulas
- 2. Find & Replace
- 3. Syntax

13. Using the FILTER() function

- 1. Filtering data tables
- 2. Creating the Filtering Expressions

14. Using Python in GS-Calc

- 1. Using Python functions as UDF()
- 2. Installing Python and GS-Calc integration

15. Inspecting cells and finding certain cell types

- 1. Using the Inspect Cells pane

16. Comparing workbooks, worksheets and cell ranges

- 1. Comparing workbooks, worksheets and cell ranges

17. Formatting

- 1. Data styles and numeric formats
- 2. Using cell style palettes
- 3. Using custom cell styles
- 4. Changing Default Fonts, Data Styles, and Colors
- 5. Conditional formatting
- 6. Using hyperlinks
- 7. Lists of images in cells
- 8. Charts in cells

18. Charts and images

- 1. Creating charts and selecting chart data
- 2. Creating mini-charts in cells
- 3. Inserting images, files in worksheets

4. Images and lists of images in cells

19. Password protection

1. Protecting and encrypting workbooks
2. Protecting workbook structure
3. Protecting individual cells

20. Opening and saving files

1. Content and statistics
2. Managing the list of recently used files and folders
3. GS-Calc *.gsc and ODF *.ods files
4. Excel (*.xlsx, *.xml, *.xls) files
5. GS-Base (*.gsb) file
6. PDF files
7. Text files
8. Text *.zip archives
9. Saving cell ranges as HTML/CSS code
10. dBase/Clipper/FoxPro files
11. MySQL files
12. SQLite (*.db) files

21. Printing workbooks

1. Printing and print previewing

22. Settings

1. Modifying program settings
2. Legacy compatibility

23. Scripting

1. Methods, properties and sample usage
2. Samples

24. Custom DLL libraries

1. Adding functions in DLL libraries

25. COM Programming

1. Interfaces, methods and properties
2. C++ samples

26. **Support**

1. Contacting technical support

27. **Copyrights**

1. Copyrights

Introducing GS-Calc

GS-Calc is a modern spreadsheet with unique capabilities for handling large datasets. You can edit large CSV files with millions of rows, and handle XLSX files that are automatically split and merged. With GS-Calc you can build complex data models containing, for example, millions of binary look-up formulas in a file that is opened, saved and updated almost instantly.

- 32 million rows x 16,384 columns. Loaded text/CSV files that exceed the 32M/16K limits are automatically split into multiple sheets and saved together as a zipped collection of text files.
- Unlimited workbook file size and best possible loading/saving speed. A workbook file with 500 million random numbers can be opened in 30 seconds on an old, slow PC with 16GB RAM.
- Efficient data compression allowing you to save, for example, 20 million formulas as a 5KB file.
- Organizing worksheets in a tree form and creating 3D array cell references for worksheets in a given subfolder.
- Over 450 built-in functions which can automatically detect array arguments and return dynamic 2D array results and images.
- Up to 64 processor cores that can be used to perform calculations with no restrictions for all types of formulas.
- During recalculation, you can reference any number of closed workbooks of any size, using any formulas.
- You can extend the above set of functions in any way you want by calling your own functions from Python modules in GS-Calc formulas as User Defined Functions (UDF). Python functions can return to GS-Calc any type of data, including single values, matrices, dynamic images or even entire CSV files.

- If you need the best performance, instead of Python, you can also add your functions written in C++/C in DLL libraries. Such functions can even replace the built-in ones, ensuring the same performance, dynamic 2D array results and multithreaded updates.
- VLOOKUP, HLOOKUP, MATCH, UNIQUE, FILTER functions using regex and fast binary searching to ensure the best performance for worksheets with millions of rows. For example, one million VLOOKUP() functions in a table with a few million rows can be updated almost instantly, even on a slow, old PC with 8GB RAM.
- The FILTER() function to quickly filter millions of rows (and/or to perform multi-key sorting and searches for duplicates).
- Built-in solver functions (formulas) with no limits for the number of variables.
- Linear programming, including integer, binary and mixed programming, and quadratic programming.
- Various optimization functions. Equation sets. Time series analysis.
- Built-in fast Monte Carlo simulations with detailed reports for models of any complexity.
- Fully configurable cross-highlighting and dark modes to ensure the best screen readability and to fit personal preferences.
- Each single worksheet can be separately split into up to 100 panes to display various regions of that worksheet or other worksheets, and you can password-protect such view layouts.
- Mass modifying/adding/removing parameters/arguments used by functions in large workbooks without the risk of making mistakes in long formulas, no matter how complex they are. ([see how >>](#))
- Unlimited number of hyperlinks and unlimited number of conditional formatting functions in sheets.
- Fast multicolumn drop-down lists with up to 32 million items.
- Fast pivot tables with up to 32 million rows, built-in reports, full regex filtering, statistics.
- 25 pivot table data functions (vs. 11 in Excel).

- Pivot tables can be used as formulas or as separate documents in dedicated view panes.
- Charts easily handling millions of data points with custom (grid-)lines and iso-lines.
- Up to 16 independently scaled axes on each chart.
- Unlimited number of fully parametrized (Google Sheets compatible) sparkline() functions in sheets.
- Unlimited number of in-cell charts as formatting styles.
- Loading/editing/saving CSV, text, XLS, XLSX, dBase, Clipper, FoxPro, MySQL, SQLite files with up to 32 million rows (and up to 1 million columns for text files).
- All opened files are edited in-place: if you click "Save" they are saved back in their original format, with their original structure and parameters.
- Compare files in any format, workbooks, sheets, ranges and generate reports with (hyperlinks to) differences.
- Verify data integrity with SHA256 checksums when switching between file formats. You can calculate control sums separately for several data categories.

[Back](#)

Notational Conventions (Numbers, Formulas, Dates)

1. Overview

All examples of numbers, formulas, dates, and lists in this Help file use the International/English language settings:

Decimal point: .

Thousand separator: ,

List (formula parameter) separator: ,

2. Example Using English Notation

Formulas and numbers must be entered as, for example:

=SUM(5589.7, 56.55, field1)

3. Adapting to Your Windows Language Settings

If your Windows language settings use different separators, you must adapt the notation accordingly. For example, if your settings use:

Decimal point: ,

Thousand separator: [space]

List separator: ;

then the formula becomes:

=SUM(5589,7; 56,55; field1)

4. Locale Settings in GS-Calc

By default, GS-Calc uses the current user's Windows Language/International Settings.

You can override this by selecting:

Settings > Locales > Generic (Fixed)

This forces GS-Calc to use International/English notation regardless of the Windows language version.

Ensuring High Performance

To achieve the best recalculation performance for very large worksheets (millions of cells, large numbers of formulas), follow these guidelines:

1. Avoid formulas that return arrays, such as:

=b10:d2000000 * 5 + 1

These prevent GS-Calc from fully utilizing multicore calculations.

2. Avoid using arrays for intermediate results, such as:

=match(v, b10:d2000000 <> 1, 0)

Instead, fill helper columns (e.g., e10:g2000000) with:

=b10 <> 1, =c10 <> 1, etc., then use:

=match(v, e10:e2000000, 0)

This makes a dramatic difference when thousands of match() formulas are present.

3. The **rand()** function is single-threaded. If used heavily in large worksheets, it forces other cores to wait. For large random datasets, generate static random values using **Insert > Random Data** to allow multicore execution.

4. Ensure GS-Calc is allowed to use the maximum number of threads/cores via:

Settings > Options > Calculation

or per-workbook overrides in:

Tools > Update Options

5. If using **vlookup()** or **match()** on large datasets, choose one of the following:

- Use the default:

Tools > Update Options > VLOOKUP and MATCH mode > Fast with binary searches

GS-Calc performs internal background sorting during the first update, then maintains sorted indices. First update may be slower; subsequent updates are dramatically faster.

Note: Regular expressions cannot be used in binary search mode.

- Use extended versions of these functions with explicit binary-search parameters.

Binary search reduces lookup steps from **$N/2$** to **$\log_2(N)$** . For 10 million rows, this is over **200,000×** fewer steps.

6. **sumif()** and **countif()** automatically use fast binary searching for column ranges with ≥ 16384 rows. You can adjust this threshold in **settings.xml** by modifying **if-calc-treshold**.

Typical location:

C:\Users\your-name\AppData\Local\GS-Calc

In portable setups, the file resides in the chosen installation folder.

7. Deleting very large blocks of cells does not immediately free memory — the deleted data remains in the Undo buffer. Memory is released when the buffer cycles or when you manually reset it. Undo level is configured in **Settings > Options**.

8. When using **Edit > Find/Replace** on extremely large tables (hundreds of millions of cells), you may switch between:

- regular expression mode

- plain text partial/full matching

depending on speed vs functionality. Even very large searches typically complete within seconds.

1. Overview

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

2. GS-Calc (*.gsc)

GS-Calc *.gsc files are typically several times smaller than equivalent ODF files. Thanks to specialized compression techniques, the difference can reach hundreds of times or more. Loading and saving operations can also be tens of times faster.

GS-Calc *.gsc files can be very large — they may exceed 4 GB. The only practical limit is that a single worksheet may use up to ~500 GB of RAM when loaded.

GS-Calc files are plain ZIP archives containing several streams: workbook data, images, file lists (with optional encryption), version metadata, and a thumbnail image stored at **Thumbnail/thumbnails.png**.

If the file is encrypted, both workbook data and all images are encrypted.

To associate the *.gsc file type with GS-Calc in Windows, run GS-Calc as an administrator and use:

Settings > Register GS-Calc *.gsc File Type

To remove the association, use:

Settings > Unregister GS-Calc *.gsc File Type

3. Open Document Format (*.ods)

GS-Calc uses the Open Document Format (ODF) 1.2/1.3 specification, a standard used by several major applications.

Some GS-Calc-specific features cannot be saved in ODF files. When saving, GS-Calc checks which features will be omitted and displays a summary.

Before saving an *.ods file, you can choose whether it should be compatible with Google Sheets or Excel. This is important because dozens of Excel formulas are saved/loaded differently in Google Sheets.

Common differences include:

- GS-Calc requires array strings to be constant, e.g.:

```
=SPARKLINE(D17,  
{ "charttype", "column"; "ymin", 0; "ymax", 2000; "firstcolor", "#334960" })
```

Google Sheets allows embedded references:

```
=SPARKLINE(D17,  
{ "charttype","column";"ymin",0;"ymax",MAX(D17:E17);"firstcolor","#334960" })
```

To address this, you may:

- enter constant values,
 - use a sheet range reference (e.g., **=SPARKLINE(D17,F1:G4)**),
 - use **VSTACK()** or **HSTACK()** to build arrays dynamically.
- Google Sheets omits the attributes **table:number-matrix-columns-spanned** and **table:number-matrix-rows-spanned** for formulas returning arrays/matrices.

After opening such files in GS-Calc (e.g., the last sheet of the “Annual Budget” template), you must:

- update the sheet (F9),
- delete all cells in the spilled area showing **#FILL** errors,

so GS-Calc can recreate the spilled cells correctly.

4. Related Topics

[GS-Calc *.gsc and ODF *.ods files](#)

[Excel \(*.xlsx, *.xml\) files](#)

[PDF files](#)

[Text files \(*.csv, *.txt, *.tab ...\)](#)

[Text *.zip archives](#)

[GS-Base \(*.gsb\) files](#)

[dBase/Clipper/FoxPro files](#)

[SQLite \(*.db\) files](#)

[MySQL \(*.sql\) files](#)

Formula Compatibility Between GS-Calc and Other Spreadsheets

1. Overview

If you encounter compatibility or syntax differences between functions used in formulas in GS-Calc and other spreadsheet applications, you can adjust your formulas automatically using:

Tools → Convert Functions in Formulas

This command allows you to modify formulas during import or export so they match the syntax expected by the target spreadsheet program.

2. Example: Converting DATEDIF / DATEDIFF

In Excel, the function:

DATEDIF(date_number1, date_number2, format)

has a different form in GS-Calc:

DATEDIFF(date_time_string1, date_time_string2)

which returns a standard date/time period string. For example:

=dateDiff(dateText(2005, 1, 1), dateText(2005, 12, 31)) returns **"P364D"**.

2.1 Converting DATEDIF → DATEDIFF

To convert all DATEDIF() functions in formulas to DATEDIFF() (or vice versa), select the desired cell range or the entire worksheet, then open:

Tools → Convert Functions

In the dialog box, enter:

Original function:

datedif(a,b,c)

Replacement:

datediff(dateText(year(a), month(a), day(a)), dateText(year(b), month(b), day(b)))

or

datediff(dateText(year(a), month(a), day(a)), todayText())

or (to return the number of years)

year(b) - year(a)

3. Multiple Conversion Rules

If multiple conversions are required, you can define permanent multi-rule conversion scripts using:

Tools → Function Conversion Scripts

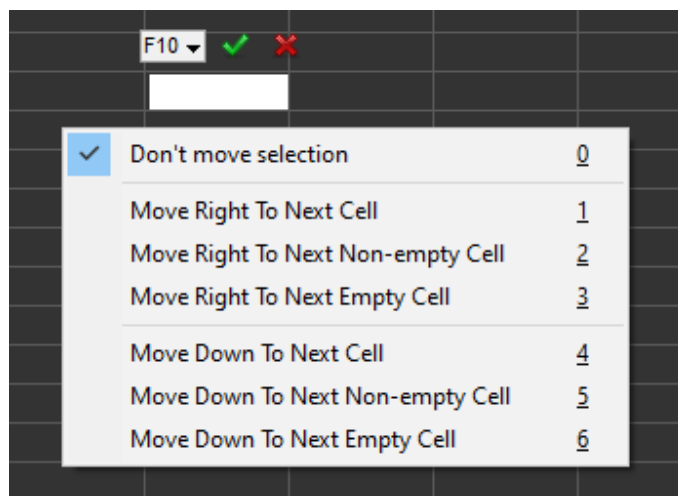
See also: [Replacing formulas automatically in worksheets](#)

Entering New Cell Contents

Do one of the following:

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

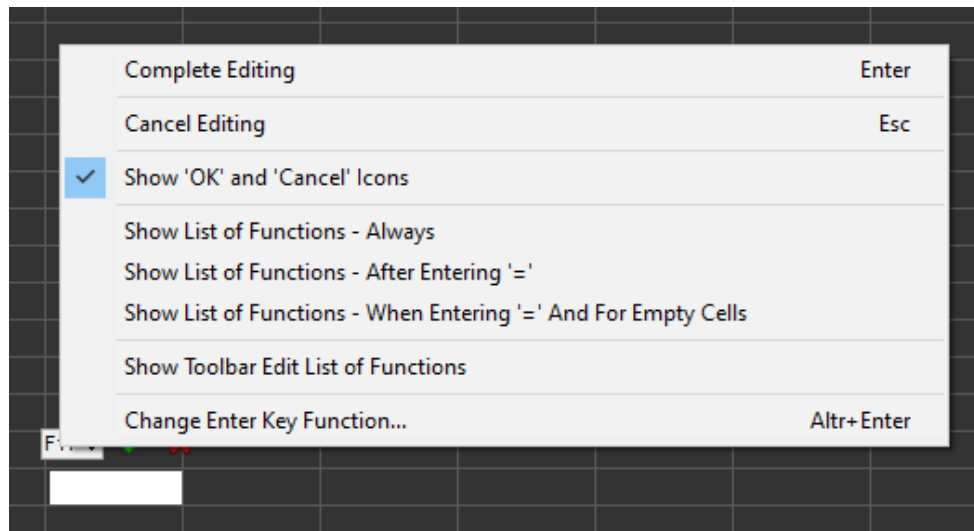
The **Settings > Options > Editing > First Enter starts editing** option can override Enter behavior globally. The **AutoScroll Range** toolbar button can override Enter behavior for specific ranges. The shortcut **Alt+Enter** displays the corresponding context menu which changes global settings or local AutoScroll settings if active in the current location:



You can enter up to 1024 characters. To accept changes, press **Enter**, any cursor key, or click another cell. To insert a new line, press **Ctrl+Enter**. To cancel changes, press **Esc**. If a cell has the Date format, the Date and Time Picker control is activated instead of the usual plain edit control.

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:

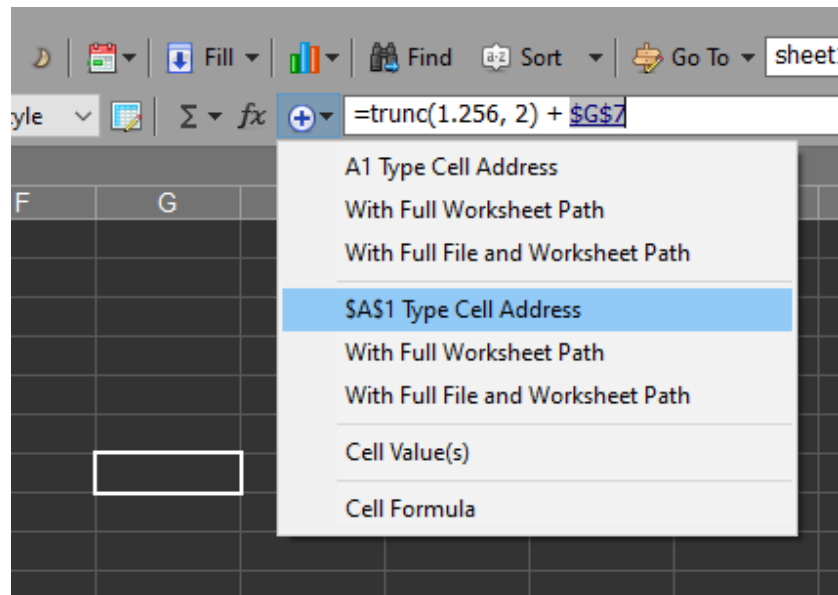


If the edited cell contains the leading = indicating a formula, clicking or selecting other cells causes adding their addresses to the edited text or replacing the address pointed by the current cursor position within the edited cell.



To hide the **List of Functions** permanently, clear 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 ".

Selecting 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 Enter. The list of recently entered selections is attached to that Enter toolbar button.

Deleting Cell Contents and 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 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.

Moving or Copying Cell Contents

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.

Pasting from Recent Copy 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.

Filling a Range with Values

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](#).

Using Drop-Down Lists

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.

Using the Date and Time Picker

The control is displayed by default for each cell that is formatted as Date and does not have any drop-down list attached at the same time. This feature can be turned off by clearing the **Settings > Options > Use Date Picker** checkbox.

The following shortcut keys can be used with that control:

Arrow 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

Copying Cells vs Duplicating Copied Cells

1. Overview

The following **Copy** commands are available on the **Edit** menu.

2. Copy Commands

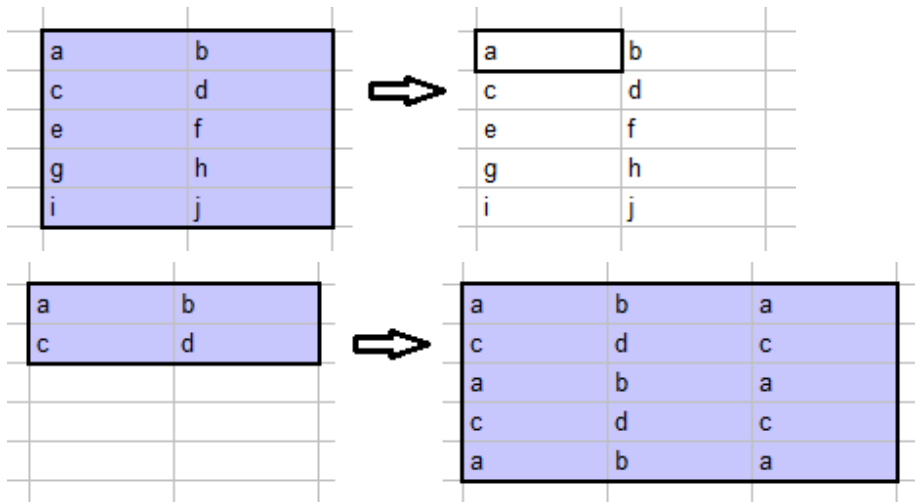
2.1 Edit > Copy

This is the standard copy command. It copies:

- native worksheet data to the cyclic GS-Calc copy/paste buffer,
- plain text to the Clipboard,
- an image/PNG representation of the copied range or objects (e.g., charts).

Subsequent copied native data can be accessed via the menu attached to the **Paste** button. The copy/paste level can be changed in **Settings > Options**; the default buffer size is **5**.

When pasting, you only need to point to the top-left cell of the target location. If you select a larger range before pasting, GS-Calc attempts to **duplicate** the copied source data within that selection only:



2.2 Edit > Copy Values

Same as **Copy**, except that for cells containing formulas it copies their **values** instead of formulas.

2.3 Edit > Copy Width Options

Extends **Copy** with parameters typically used for text data/files:

- separators,
- fixed column widths,
- quoting symbol,
- whether copied data should be formatted,
- text encoding.

You can save specific settings as predefined named parameter sets in the dialog box and select them with a single click.

2.4 Edit > Copy As HTML

Copies a range of cells as HTML code. The copied HTML can be pasted directly into HTML content in other applications as one continuous block.

In web browsers, the HTML looks the same as in the worksheet window, including styles and formatting, except for text rotation. For the “accounting” numeric style, you must use a fixed-width font (e.g., **Courier New**) in GS-Calc for the related cells.

2.5 Edit > Copy As Image To

Copies selected data to an image file (jpg/png/gif/bmp/tiff).

2.6 Edit > Copy To PDF

Copies the selection (range or object/chart) to a PDF file.

2.7 Edit > Copy As Location

Creates and copies (as text) a reference to the current cell/selection and automatically adds it to:

- the list of references used in various **Insert > ...** formula composers,
- the **Go To** list on the toolbar (for navigation or selection of ranges).

2.8 Edit > Copy As Worksheet Hyperlink

Creates and copies (as text) a hyperlink to the current cell/selection. It prefixes the text with the hyperlink protocol name and instructs GS-Calc to set the **hyperlink** cell format for the cells where you paste it.

Clicking such a hyperlink cell jumps to and selects the specified cell/range. To move back to the clicked cell, use the toolbar bookmark **Back** button.

3. Paste Commands

3.1 Edit > Paste

Pastes copied data. If both native data and Clipboard data are available, copied native data take precedence.

3.2 Edit > Paste without Styles

Pastes copied native data without styles or formatting.

3.3 Edit > Paste As Text

Pastes available data (including numbers and formulas) as text strings.

3.4 Edit > Paste with Options

Pastes text data using definable parameters typically used for text data/files:

- separators,
- fixed column widths,
- quoting symbol,
- escaping symbol,
- text conversion types,
- text encoding,
- saving cell styles detected in formatted numbers and dates.

3.5 Edit > Paste Image

If image data are available, this command pastes that image.

3.6 Edit > Insert Copied Cells

Works similarly to **Paste**, except that:

- columns in the target area are shifted **left**,
- rows in the target area are shifted **down**.

Using AutoScroll Ranges

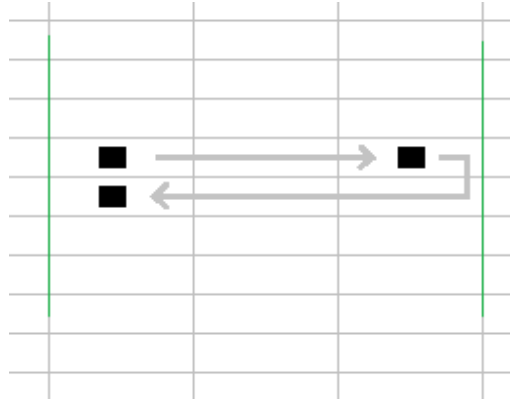
1. Overview

AutoScroll ranges allow you to define regions in which the current (selected) cell moves differently than in the rest of the worksheet when you press **Enter** or use the **Left/Right** or **Up/Down** arrow keys.

2. Horizontal Scrolling

Within an AutoScroll range, horizontal movement can be configured as follows:

- Pressing the **Left/Right** arrow keys moves the current cell to the next or previous cell in the row — either the next empty cell or the next non-empty cell — within the boundaries of the region.
After reaching the left or right boundary, the selection moves to the end or start of the previous or next row.
- Pressing **Enter** can:
 - Move the current cell to the next or previous cell in the row (empty or non-empty), within the region boundaries.
 - Jump to the opposite left or right edge of the region. If the current cell is already at the right edge, the selection jumps to the opposite edge in the next row. After reaching the last row, it returns to the first top-left cell of the region.



3. Vertical Scrolling

Vertical scrolling works similarly:

- Pressing the **Up/Down** arrow keys moves the current cell to the next or previous cell in the column — either the next empty cell or the next non-empty cell — within the region boundaries.
After reaching the top or bottom boundary, the selection moves to the end or start of the previous or next column.
- Pressing **Enter** can:
 - Move the current cell to the next or previous cell in the column (empty or non-empty), within the region boundaries.
 - Jump to the opposite top or bottom edge of the region. If the current cell is already at the bottom edge, the selection jumps to the top edge in the next column. After reaching the last column, it returns to the first top-left cell of the region.

4. Additional Options

Press **Alt+Enter** to quickly switch the active scrolling mode.

You can define multiple AutoScroll ranges and switch between them. To create and activate a range, select a cell range and choose **Edit** → **Activate Autoscroll Range**, or click the **AutoScroll** label on the worksheet toolbar. To deactivate the range, use the corresponding command.

You can also configure AutoScroll behavior globally for the entire worksheet in **Settings** → **Options**.

1. Overview

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

Type	Examples	Comments
Numbers	123 -123 123.09 1.23e+02	The actual decimal and thousand separators depend on your system regional settings and the current Settings > Locales menu selection. Max positive value: 1.7976931348623158e+308 Min positive value: 2.2250738585072014e-308 Precision: up to 15 digits
Text strings	abc abc def 123	Use the Edit > Convert commands to convert numbers to text strings, and vice versa. To force GS-Calc to treat entered data as a text label, press and hold SHIFT before pressing ENTER. GS-Calc automatically converts text strings to numbers in formulas requiring numeric arguments, but only if they represent unformatted numbers using a period (.) as the decimal separator. Example: ="1.5" + 1 -> valid ="1,500.00" + 1 -> #NUM!
Date/Time strings Period strings	Date/Time: 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	Date/time/period values can be specified either as serial numbers or as generic date/time/period 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 represents

	Periods: P1Y2M3DT10H30M60S.000 P2M4D -P120D PT23H34M	days; the fractional part represents time of day. 1.0 represents December 31, 1899. Negative numbers represent dates prior to December 30, 1899. A generic date/time string must conform to the www.w3.org specification. For more information, see the Formula Composer window and included samples. Note: To maintain maximum portability, you should use date serial numbers.										
Arrays	A four element 2x2 array: {1, 2; "abc", "def"} {1.00e+10; 2.00e+10; #N/A!; true}	Array elements may be unformatted numbers, text strings, Boolean values, or error codes. With Settings > Locales > Generic: Columns separated by commas (,) Rows separated by semicolons (;) With system regional settings using comma as decimal separator: Columns separated by semicolons (;) Rows separated by backslashes (\)										
Boolean values	true (= 1) false (= 0)											
Error codes	<table><tr><td>#DIV/0!</td><td>Division by 0</td></tr><tr><td>#NAME?</td><td>Invalid name in formula</td></tr><tr><td>#NULL!</td><td>Empty intersection or empty array</td></tr><tr><td>#N/A!</td><td>No data/result available</td></tr><tr><td>#NUM!</td><td>Invalid number, overflow or underflow</td></tr></table>	#DIV/0!	Division by 0	#NAME?	Invalid name in formula	#NULL!	Empty intersection or empty array	#N/A!	No data/result available	#NUM!	Invalid number, overflow or underflow	Error codes may be returned by formulas or entered directly.
#DIV/0!	Division by 0											
#NAME?	Invalid name in formula											
#NULL!	Empty intersection or empty array											
#N/A!	No data/result available											
#NUM!	Invalid number, overflow or underflow											

	#REF!	Invalid cell/range reference: (+) column or row number out of range (+) array exceeding worksheet dimensions (+) non existing worksheet or workbook (+) inaccessible file/workbook
	#VALUE!	Invalid argument type or too long text string
	#IMPL!	Function not yet implemented; name reserved
	#SYNTAX!	Syntax error: incorrect number of arguments, missing parenthesis, etc.

Entering Data: Using Cell References

1. Overview

Cell references can be written in either **A1 notation** or **RC notation**. The table below shows equivalent forms and explains what each reference points to.

A1 notation	RC notation	Refers To
\$A\$1	R1C1	A cell in row 1, column 1. This is an absolute reference : inserting/deleting rows

		or columns and copying/pasting does not change the reference.
A1	R[0]C[0] R[1]C[2] R[-1]C[-2]	A cell in row 1, column 1. This is a relative reference: inserting/deleting rows or columns and copying/pasting may adjust the reference to preserve correct relationships. Examples: • Copying =B2 from A1 to C3 becomes =D4. • Copying =B\$2 from A1 to C3 becomes =D\$2. • Copying =\$B2 from A1 to C3 becomes =\$B4.
\$1:\$1 1:1	R1	The entire first row.
\$A:\$A A:A	C1	The entire first column.
\$1:\$5	R1:R5	Rows 1 through 5.
\$A:\$E	C1:C5	Columns 1 through 5.
\$A\$1:\$E\$5	R1C1:R5C5	A rectangular range from A1 (top-left) to E5 (bottom-right).
Sheet1!\$A\$1 'Sheet1'!\$A\$1	Sheet1!R1C1 'Sheet1'!R1C1	The A1 cell from the "Sheet1" worksheet in

		the same workbook. Note: • Quotation marks may be omitted if the sheet name contains only alphanumeric characters and no spaces. • If the name contains quotation marks, enclose it in quotes and double the internal marks.
'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 cell (returned as one column) from all worksheets in the "Folder1" folder.
'Folder1'!\$A\$1:\$E\$5	"Folder1"!R1C1:R5C5	All A1:E5 ranges (returned as one column) from all worksheets in the "Folder1" folder.
'd:\documents \ [sample.gsc]Sheet1'!\$A\$1 'd:\documents \ [sample.gsc]Sheet1'!A1	"d:\documents \ [sample.gsc]Sheet1"!R1C1	The A1 cell from "Sheet1" in the sample.gsc workbook. Note: • File names must be enclosed in

		square brackets. • Paths may be absolute or relative. • Relative paths (including empty paths) are resolved based on the path of the document containing the reference, or — if unsaved — the last opened document or the default document folder.
"d: \[sample.gsc]Folder1"!\$A\$1:\$E\$5	"d: \[sample.gsc]Folder1"!R1C1:R5C5	All A1:E5 ranges (returned as one column) from all worksheets in the "Folder1" folder of the sample.gsc workbook.

Entering Data: Merging and Splitting Cells

1. Merging Cells

You can merge adjoining cells using the **Format > Merge Cells...** command. When cells are merged, the contents of the top-left cell are displayed across the entire merged area. All other cells in the merged group are treated as covered and are not displayed.

Merging does not change how formulas reference the merged cells, nor how they are copied, edited, or formatted. To edit or format the displayed content, you must select the top-left cell of the merged group.

Merged ranges may be of any size, but they cannot overlap.

2. Merge Cells Dialog Options

Merge selected cell contents and place them in the top-left cell

The contents of all cells in the merged group are concatenated and copied into the top-left cell. Cells are separated by a single space, and rows are separated by newline characters. All merged cells are then cleared.

Alignment options

You can adjust alignment settings for the top-left cell. To modify alignment later, select the top-left merged cell and use any formatting command.

Save As Defaults / Reset Defaults

Click **Save As Defaults** to store the current alignment settings for future merge operations. If defaults already exist, the button changes to **Reset Defaults**, which restores the dialog's settings to match the current format of the top-left cell.

3. Splitting Cells

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

4. Finding Merged Cells

To locate all merged cell groups, open the **Inspect Cells** pane using **View > Inspect Cells**, click **Find & Add**, and choose **Merged Cells**.

Entering Data: Entering Formulas

To Enter/Edit a Formula

[Enter](#) a desired expression, including the leading "=".

For example, to sum 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 by either single or double quotation marks.

Note: Even if **Settings > Options > Calculations > Automatic update mode** is OFF, it's typically sufficient just to enter a given formula to display its up-to-date value. The exceptions are formulas referring to functions that perform searching or filtering, and the "...If" actions that require internal cell sorting and can cause some delays. In this case, if automatic update mode is OFF, you have to run one of the **Tools > Update** commands to obtain the current value of that formula after you enter or edit it.

Formulas can contain circular references. For example, if cell A1 contains `'=B2 + 1'`, B2 contains `'=C3 + 1'` and C3 contains `'=A1'`, then each time you update the workbook, GS- Calc will perform 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 is 2 or higher. 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, which is displayed when editing formulas or after clicking the **Formula** edit field 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 settings use commas as decimal separators, use semicolons (;) to separate function arguments, or, if you prefer locale-independent (US) settings, select the generic locale via the **Settings > Locales > Generic** command.

Available operators:

Operator	Operation	Comments	Precedence
=	Equal	Compares numbers or text strings (the comparison is not case-sensitive). Example: A1=4 , B2="abc"	9
<	Less than	Compares numbers or text strings (the comparison is not case-sensitive). Example: A1<4 , B2<"abc"	9
>	Greater than	Compares numbers or text strings (the comparison is not case-sensitive). Example: A1>4 , B2>"abc"	9
<=	Less than or equal	Compares numbers or text strings (the comparison is not case-sensitive). Example: A1<=4 , B2<="abc"	9
>=	Greater than or equal	Compares numbers or text strings (the comparison is not case-sensitive). Example: A1>=4 , B2>="abc"	9
<>	Not equal	Compares numbers or text strings (the comparison is not case-sensitive). Example: A1<>4 , B2<>"abc"	9
+	Addition	Adds numbers.	8
-	Subtraction	Subtracts numbers.	8
&	String concatenation	Merges text strings. Example: A1 & "abc" , "a" & "b"	8
*	Multiplication	Multiplies numbers.	7
/	Division	Divides numbers.	7
^	To the power of	Calculates the power of.	6
-	Negative	Changes the sign of a number. Example: -A1	5
%	Percent	Specifies a number entered as a percentage. Example: 12%	4
_	Intersection	Specifies a (reference to an) intersection of two ranges. Example: 5:5_b:b , a1:a5_a5:a8	3

<i>Operator</i>	<i>Operation</i>	<i>Comments</i>	<i>Precedence</i>
!	Worksheet reference	Specifies a reference to another worksheet/folder. Example: sheet2!(5:5_b:b)	2
:	Range of cells	Creates a range. Example: r1c1:r5c5 , A1:B5	1

To Recalculate Formulas

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

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 second **status bar** pane.

or

1. Select a cell range as shown below:

10	15
20	25
30	35
40	45

10	15	
20	25	
30	35	
40	45	

10	15		
20	25		
30	35		
40	45		

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

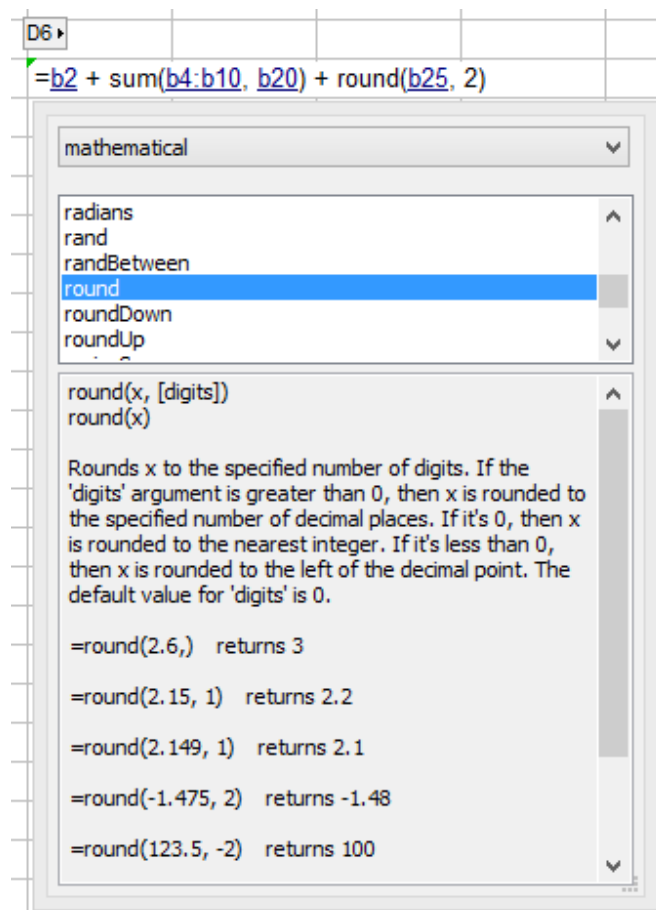
Entering Data: Formula Composer

1. Overview

The **List of Functions** dialog box helps you enter built-in functions more easily when editing formulas.

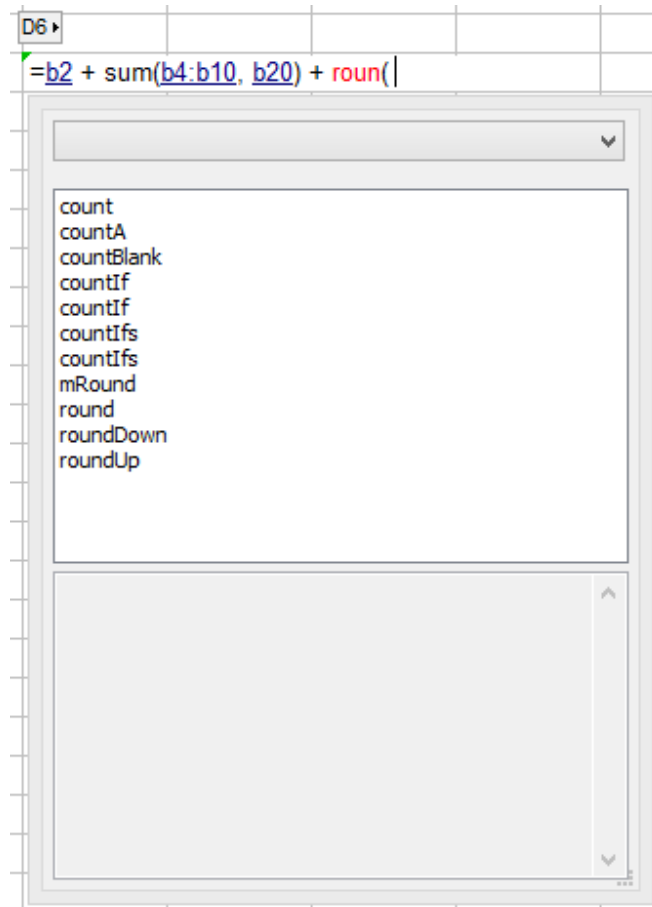
This dialog box is displayed automatically when editing a cell or when activating the **Formula** edit field on the toolbar. The **Insert > Formula** command simply starts editing the current cell.

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



2. Function Name Assistance

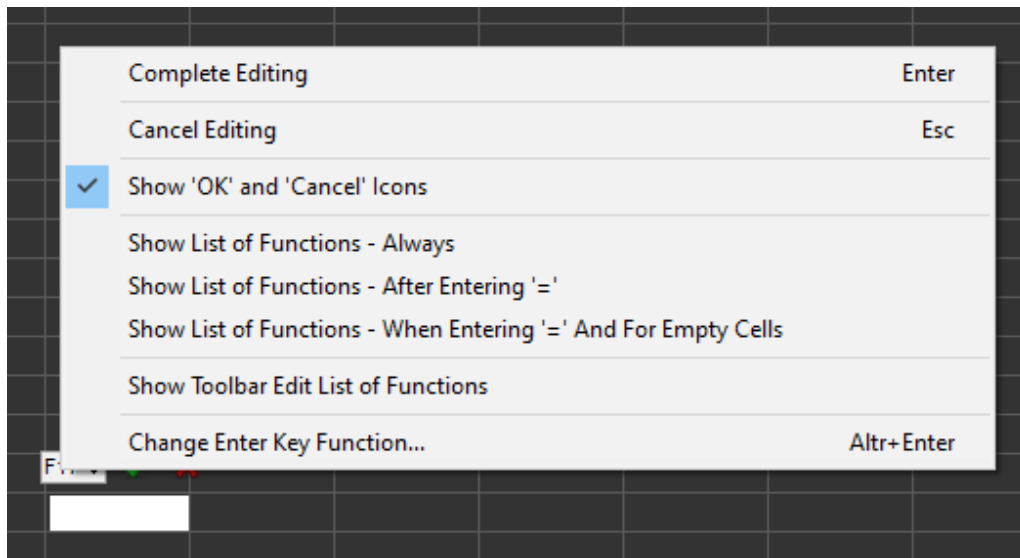
If you enter an incorrect function name (including the opening parenthesis), the dialog box will display possible correct functions.



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

3. Display Options

The **List of Functions** dialog box is displayed automatically if the edited cell begins with "=" indicating a formula. You can also choose other display options:



Entering Data: Using Array Formulas

1. Overview

An **array formula** returns an array of values — numbers, text, Booleans, or errors — instead of a single value. Any non-array formula becomes an array formula when at least one of its arguments is replaced with a range or an explicit array.

Examples:

=A1	→	=A1:A5
=sqrt(E5) + 1	→	=sqrt(E5) + {1; 2; 3; 4; 5}
=sqrt(E5) + 1	→	=sqrt(E5:J10) + 1

When a formula argument is expanded into a range or array, the formula evaluates element-by-element and returns an array whose dimensions match those of the array arguments.

2. Matching Dimensions

If a formula contains multiple array or range arguments, all of them must have identical dimensions — the same number of rows and columns.

Correct:

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

Incorrect (returns **#VALUE!**):

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

The returned array always matches the dimensions of the array arguments. If any argument has mismatched dimensions, the entire formula produces a **#VALUE!** error.

3. Typical Uses

Array formulas are useful for generating sequences (e.g., chart data), performing element-wise operations, or simplifying complex logical and arithmetic expressions.

Counting values in ranges

Count numbers in A1:B100 that are either:

- greater than 100 and smaller than 200, or
- greater than 250 and smaller than 300

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

This works because logical comparisons return arrays of 0s and 1s, and multiplication acts as a logical AND while addition acts as a logical OR.

Summing values that meet conditions

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

The conditional mask is multiplied element-wise by the numeric range, producing an array of values that meet the conditions.

Converting errors to empty strings

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

Note: When ranges are used, all **if()** arguments must have identical dimensions.

Summing values while ignoring errors

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

Here, **errorType()** is used to distinguish real errors from non-errors (**errorType** = 0). Only valid numeric values are included in the final sum.

1. Overview

GS-Calc allows you to configure how worksheet updates and calculations are performed. You can specify:

- the maximum number of threads/cores used (1–64)
- the update mode
- additional update-related options

These settings can be controlled in two places:

Settings → Options → Calculation

Defines the global default update options for all workbooks.

Tools → Update Options

Allows overriding the global settings for the currently opened workbook only.

This flexibility makes it possible to optimize performance depending on the size and complexity of each workbook, or to apply different update strategies across multiple projects.

Replacing Formulas Automatically in Worksheets

1. Overview

To replace any function in all formulas instantly, use the **Tools > Convert Function in Formulas** command.

You can replace a function with any other function or expression, reuse some or all of the original parameters in any order, embed them in other expressions, mix them with cell references, or combine them with constants.

2. Specifying the Function Pattern

In the first field, specify how the function appears in formulas. Replace actual parameters with letters **a...z**. In real formulas, these parameters may be any expressions or nested functions.

Examples:

- VLOOKUP(a, b, c, d, e, f)
- TRIM(a)
- IF(a, b, c)

- MATCH(a, b, c)

3. Specifying the Replacement Expression

In the second field, enter the expression that will replace the function. Examples:

- MY_VLOOKUP(abs(a) + 1, b, b100:b1000, e, d, 2) + 1
- TRIM(a & " of" & d10)
- IF(a, b, IF(c < b, b, c)) * 2
- MATCH(a, b, 0, 1, 1)

After clicking **OK**, all matching formulas are updated. You can choose whether changes apply to:

- the current selection,
- the entire worksheet,
- or all worksheets in the workbook.

If the replacement expression contains new cell references and you want them updated relative to each target cell (as when filling ranges), enable **Update relative cell references**.

Updating is fast — millions of formulas can be modified instantly. GS-Calc displays how many formulas were changed.

4. Converting Excel XMATCH / XLOOKUP

You can also use this feature to convert Excel formulas, for example when opening an Excel worksheet containing **XMATCH** or **XLOOKUP** in older Excel versions.

Replacing XMATCH(a, b, c, d)

- **Vertical binary search** (COLS(b)=1):

MATCH(a, b, c, IF((d)=2, 8, IF((f)=-2, 16, IF((c)=-1, 512, 0))))

- **Classic linear search returning last match:**

MATCH(a, b, c, IF((c)=-1, 512, 0))

- **Classic linear search:**

MATCH(a, b, c)

Replacing XLOOKUP(a, b, c, d, e, f)

- **Vertical binary search** (COLS(b)=1):

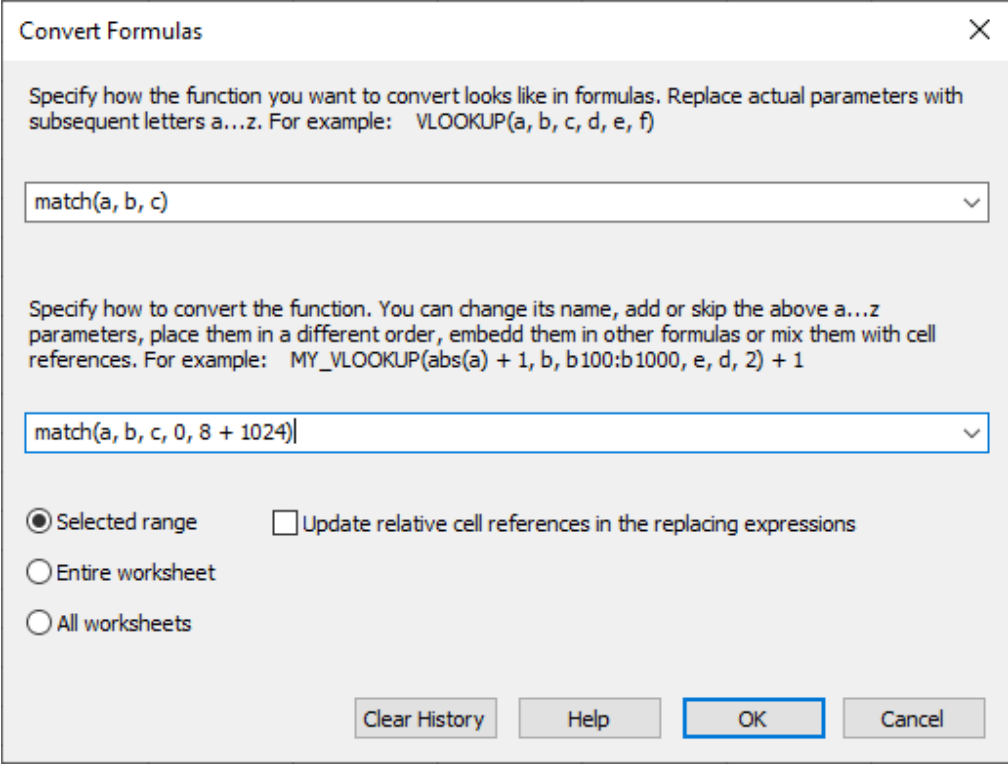
INDEX(c, MATCH(a, b, e, IF((f)=2, 8, IF((f)=-1, 512, 256))),)

5. Notes on GS-Calc Search Functions

Note: GS-Calc's **match()** and **vlookup()** functions offer more parameters and significantly better performance than Excel's equivalents.

Even if a range is not sorted, you can still perform fast binary searching by specifying **SEARCH::AutoSort** (or **1024**) in the search mode. GS-Calc sorts internally during the first update and maintains sorting indices automatically. This can result in thousands-fold speed improvements.

You can also specify which matching occurrence should be returned (first, last, or any specific index).



The screenshot shows the 'Convert Formulas' dialog box. It has a title bar with a close button (X). The main area contains two text boxes for specifying conversion rules. The first text box has the text 'match(a, b, c)' and a dropdown arrow. The second text box has the text 'match(a, b, c, 0, 8 + 1024)' and a dropdown arrow. Below these text boxes are three radio buttons: 'Selected range' (selected), 'Entire worksheet', and 'All worksheets'. There is also a checkbox labeled 'Update relative cell references in the replacing expressions'. At the bottom, there are four buttons: 'Clear History', 'Help', 'OK' (highlighted with a blue border), and 'Cancel'.

Convert Formulas

Specify how the function you want to convert looks like in formulas. Replace actual parameters with subsequent letters a...z. For example: `VLOOKUP(a, b, c, d, e, f)`

match(a, b, c)

Specify how to convert the function. You can change its name, add or skip the above a...z parameters, place them in a different order, embed them in other formulas or mix them with cell references. For example: `MY_VLOOKUP(abs(a) + 1, b, b100:b1000, e, d, 2) + 1`

match(a, b, c, 0, 8 + 1024)

☒ Selected range ☐ Update relative cell references in the replacing expressions

☐ Entire worksheet

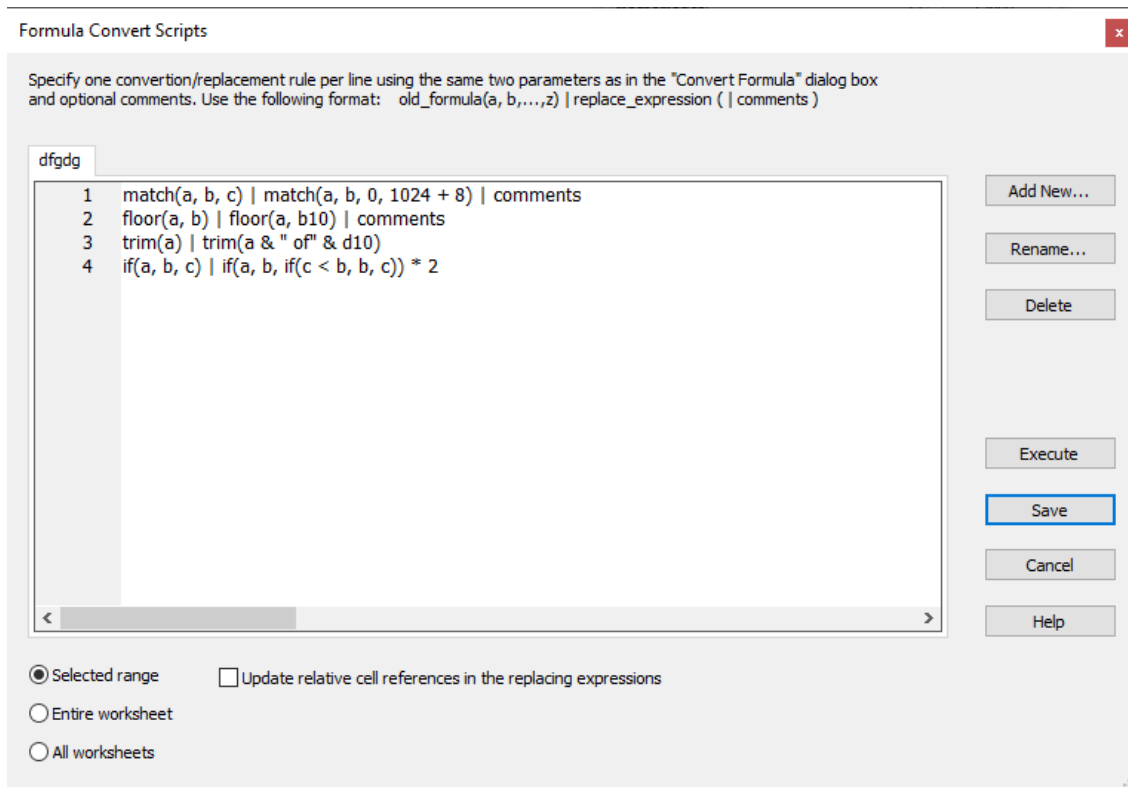
☐ All worksheets

Clear History Help OK Cancel

6. Conversion Scripts

To simplify multiple conversions, use **Tools > Function Conversion Scripts**.

Scripts are named sequences of conversion rules. You can create any number of scripts, each containing any number of conversion definitions.



Entering Unicode Characters

1. Overview

Unicode characters can be entered and edited using several methods. These methods allow entering any Unicode code point using hexadecimal notation. Some methods work only when editing worksheet cells, while others also work in dialog boxes.

2. Methods for Entering Unicode Characters

2.1 Alt+X Method (Cell Editing Only)

Enter a sequence of 1–6 hexadecimal digits and immediately press **Alt+X**. The digits are converted into the corresponding Unicode character.

Important: Any preceding valid hexadecimal digits that are not separated (e.g., by a space) will also be converted.

This method is available only when editing worksheet cells. It is not available in dialog boxes.

2.2 Ctrl+Shift+U Method

Press **Ctrl+Shift+U**, then type a sequence of 1–6 hexadecimal digits, and press **Space** to confirm.

This method works in more contexts than Alt+X and does not require cell-editing mode.

2.3 Alt + Hexadecimal Code (Hex Numpad)

Press and hold **Alt**, press **+** on the numeric keypad, enter the hexadecimal Unicode code, then release **Alt**.

This method may require enabling the Windows Hex Numpad feature by adding the following registry key:

HKEY_CURRENT_USER\Control Panel\Input Method\EnableHexNumpad

Set the value to "1" (type: **REG_SZ**).

Selecting Cells, Navigation, Scrolling, Key Shortcuts

1. Selecting a Range of Cells

Do one of the following:

- Use the mouse cursor.
- Press **Shift** and use any scrolling keys or shortcuts.
- Enter a cell or range in the **Location** field on the toolbar, then click the "Enter" button or press **Enter**.
You can bookmark any location. Click the "*" button to add a bookmark. Use the "<*" and "*>" toolbar buttons to jump between bookmarks, or manage them via **View > Bookmarks**.
- Press **Ctrl+Space** to select the entire current column.
- Press **Shift+Space** to select the entire current row.

2. Workbook Window Navigation Shortcuts

→	sheet1!B2	↵
First Column	Home	
Last Column	End	
First Row	Ctrl+Home	
Last Row	Ctrl+End	
Page Up	PgUp	
Page Down	PgDn	
Page Left	Alt+PgUp	
Page Right	Alt+PgDn	
Data Region Up	Ctrl+Up	
Data Region Down	Ctrl+Down	
Data Region Left	Ctrl+Left	
Data Region Right	Ctrl+Right	
Same Cell Above	Ctrl+Alt+Up	
Different Cell Above		
Same Cell Below	Ctrl+Alt+Down	
Different Cell Below		
Previous Worksheet	Ctrl+PgUp	
Next Worksheet	Ctrl+PgDn	
New Line in Cell	Ctrl+Enter / Altr+Enter	

3. Print Preview Navigation Shortcuts

Use the standard cursor keys: Up, Down, Left, Right, PgUp, PgDn, PgUp+Alt, PgDn+Alt.

Ctrl+PgUp and **Ctrl+PgDn** jump to the previous or next preview page.

4. 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: 1. Worksheet tree (if visible)
-----------	---

	2. Worksheet window 3. Inspect Cells (if visible)
Ctrl+F6	Open/close the worksheet tree pane

5. Jumping to a Cell or Range

- Use the **Location** field as described above.

6. Auto-Scroll

Click inside the worksheet window using the mouse wheel button. Move the mouse to choose scrolling speed and direction relative to the clicked point.

Performing Multiple Selections to Combine Their Values

1. Overview

You can create multiple selections to automatically calculate function values (means, sums, quartiles, etc.) for selected cells.

2. Using the Mouse

- **Add cells or ranges:** Hold **Ctrl** and click a cell or select a range.

The table toolbar displays the label **combined cells** before the function values, and the combined references appear in the composer toolbar's edit field.

0 Combined Cells: sum=429.25 average=26.828125 min=9.2 max=81						
ID	ProductName	UnitPrice	Qty	Total	Date	
3	Aniseed Syrup	\$10.00	2	\$20.00	3/1/2011	
5	Chef Anton's Gumbo Mix	\$21.35	1	\$21.35	2/28/2011	
11	Queso Cabrales	\$21.00	3	\$63.00	3/4/2011	
4	Chef Anton's Cajun Seasoning	\$22.00	2	\$44.00	3/5/2011	
22	Gustaf's Knäckebröd	\$21.00	5	\$105.00	2/27/2011	
20	Sir Rodney's Marmalade	\$81.00	1	\$81.00	3/4/2011	
14	Tofu	\$23.25	1	\$23.25	3/1/2011	
19	Teatime Chocolate Biscuits	\$9.20	3	\$27.60	2/28/2011	
20	Sir Rodney's Marmalade	\$81.00	1	\$81.00	3/4/2011	
14	Tofu	\$23.25	1	\$23.25	1/2/2011	
19	Teatime Chocolate Biscuits	\$9.20	3	\$27.60	2/28/2011	
11	Queso Cabrales	\$21.00	3	\$63.00	3/4/2011	
4	Chef Anton's Cajun Seasoning	\$22.00	2	\$44.00	1/5/2011	
22	Gustaf's Knäckebröd	\$21.00	5	\$105.00	2/23/2011	
11	Queso Cabrales	\$21.00	3	\$63.00	3/8/2011	
4	Chef Anton's Cajun Seasoning	\$22.00	2	\$44.00	3/31/2011	
3	Aniseed Syrup	\$10.00	2	\$20.00	3/1/2011	

To reset the combined list, click anywhere on the sheet without holding Ctrl, or press/release **Shift**.

3. Using the Keyboard

- Press **Ctrl + Numpad0** to add the current selection to the combined list.

Adding the same selection twice exits the selection mode and clears all selections.

4. Copying Calculated Values

To copy the calculated values, right-click the table toolbar. Values can be copied as a single column:

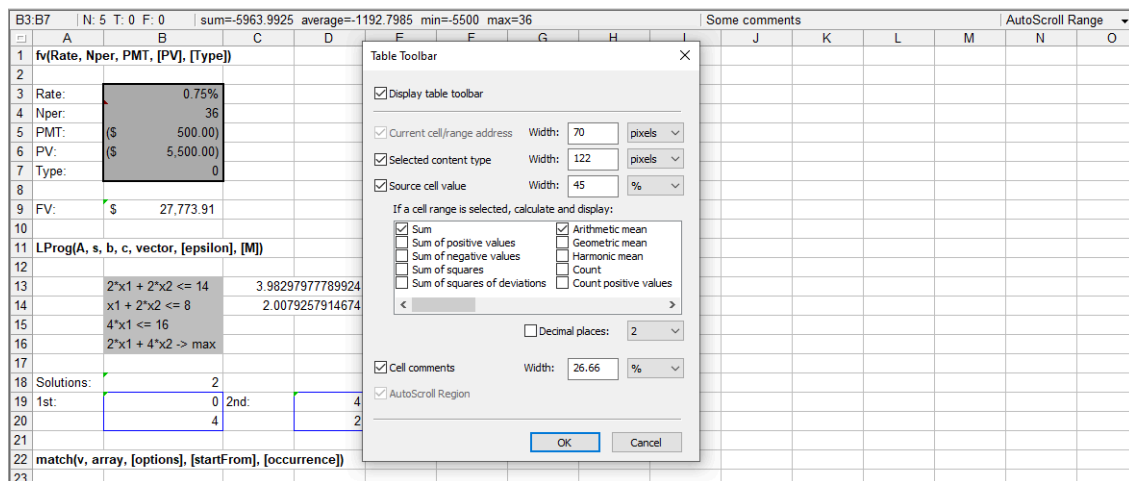
407.25
27.15
9.2
81

or with labels:

sum	407.25
average	27.15
min	9.2
max	81

5. Selecting Functions

To choose which functions are calculated, click the table toolbar to display its options:



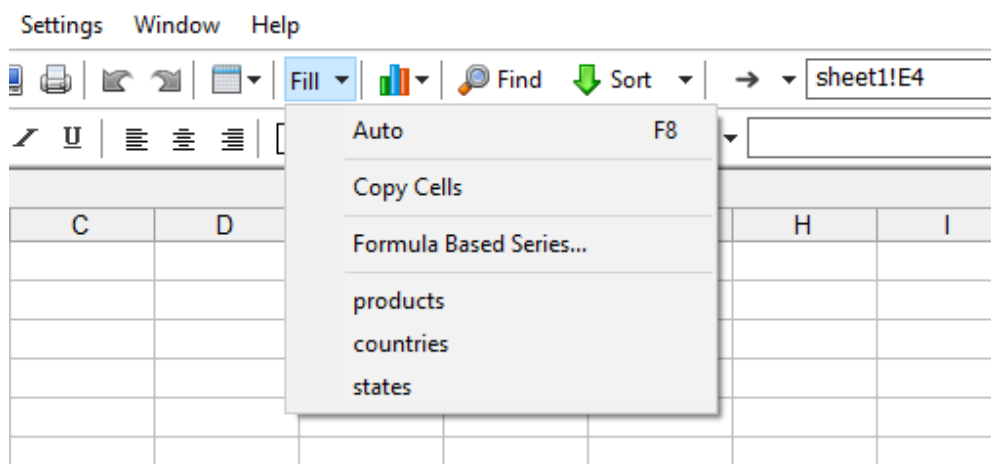
Entering Data: Inserting Series

1. Inserting an Automatic Series (F8)

This fills a selected range with values such as numbers, dates, formulas, built-in lists (day/month names), or custom lists. The series type is determined automatically based on the first one or two cells.

If the initial cells do not match any known series type, a simple copy operation is performed. If the first two cells are separated by empty cells, that gap is repeated throughout the series.

- Built-in list items (day/month names) may appear as substrings inside cell text.
Example: "departure on Monday" → "departure on Tuesday", etc.
- Custom list items may also appear as substrings, but must be inserted explicitly via the list name using the **Fill** button.



2. Built-In Numeric, Date/Time, and Formula Series

If only one element is provided, GS-Calc uses default increments:

- Numbers: +1
- Time values: +1 hour
- Date values: +1 day
- Formulas: +1 column/row reference

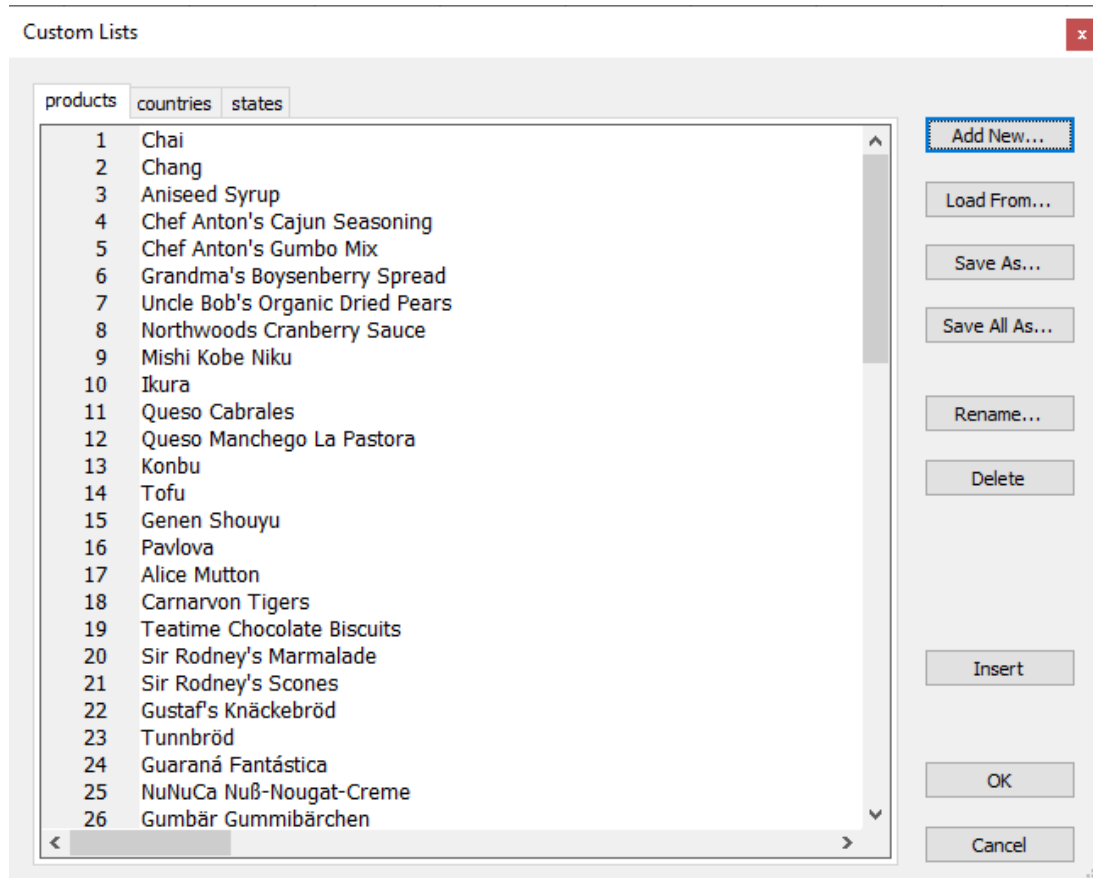
Relative and absolute references may be adjusted depending on settings in **Settings > Options**.

3. Choosing a Specific Custom List

Click the **Fill** button and choose a list, or use the **Manage** dialog and click **Insert**.

If a specific custom list is chosen:

- Custom list items may appear as substrings, but must be inserted explicitly via the list name.
- The target range does not need initial elements; if empty, the series starts from the first list item.

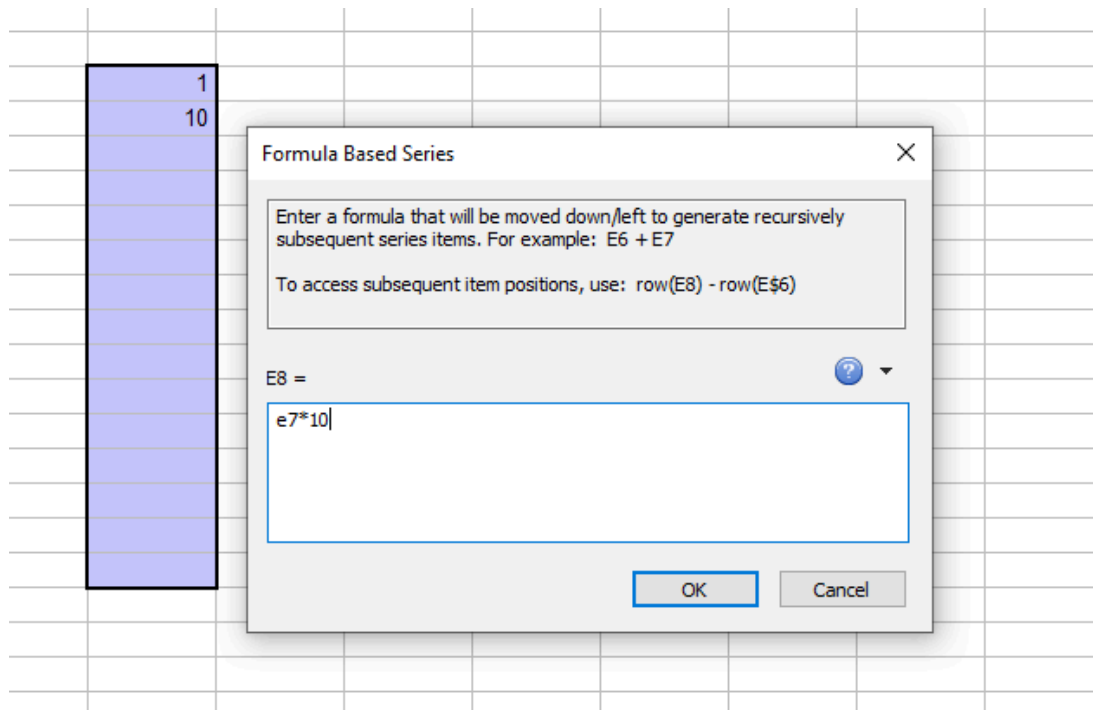


4. Inserting Formula-Based Series

This allows defining any type of series, including those referencing other workbook resources.

Define a formula for the first item. GS-Calc updates references automatically to generate subsequent values. The formula may use the previous value or compute each value directly based on its position.

Used formulas are saved in a “recently used” list and references are updated automatically when reused elsewhere.



5. Inserting Random Series

Several popular distribution types are available. For uniform distribution, you may generate a series of randomly selected custom list items.

Insert Random Data

Distribution

☒ Uniform distribution

☒ From:

0

To:

1

☐ List:

☐ Normal distribution

Mean:

0

Std.dev.:

1

Generator

☒ Software random number generator

☐ Seed 1 (64 bit):

☐ Seed 2 (64 bit):

☐ Hardware random number generator

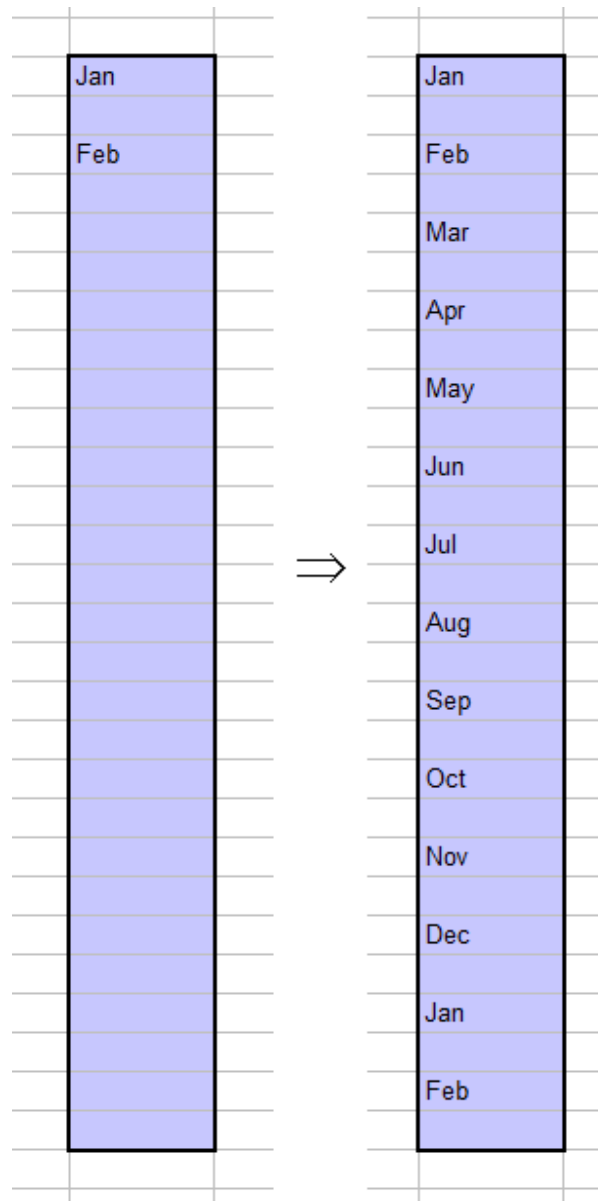
OK

Cancel

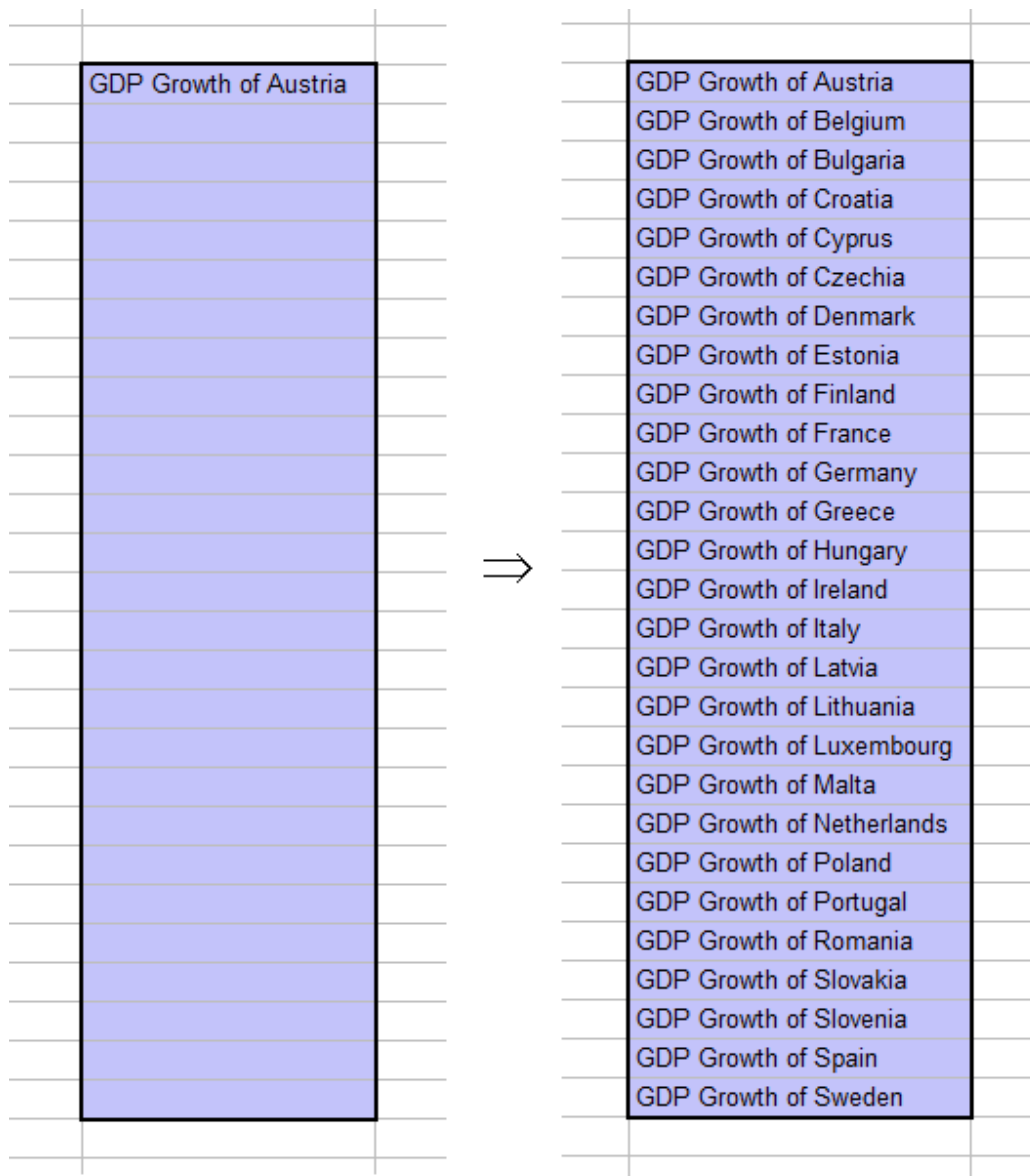
Formula...

6. Examples

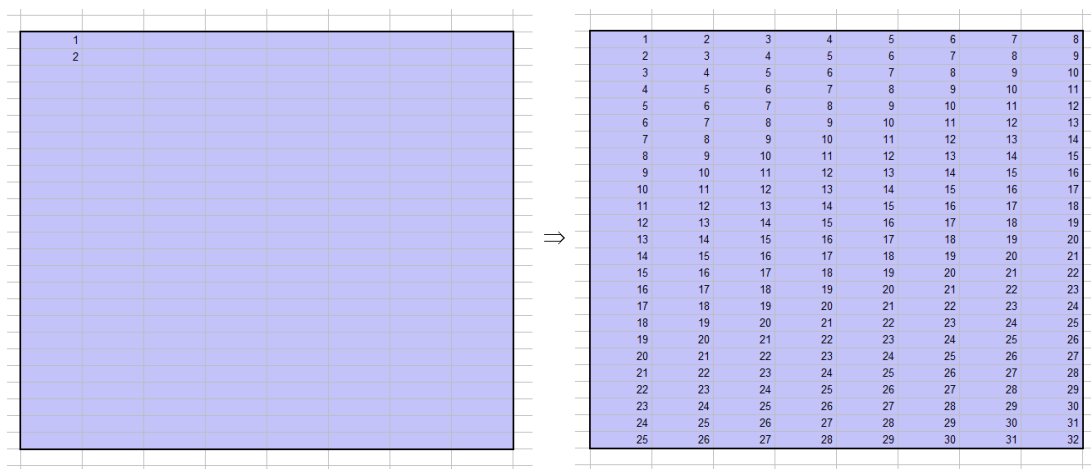
Built-in series of month names (system-dependent):



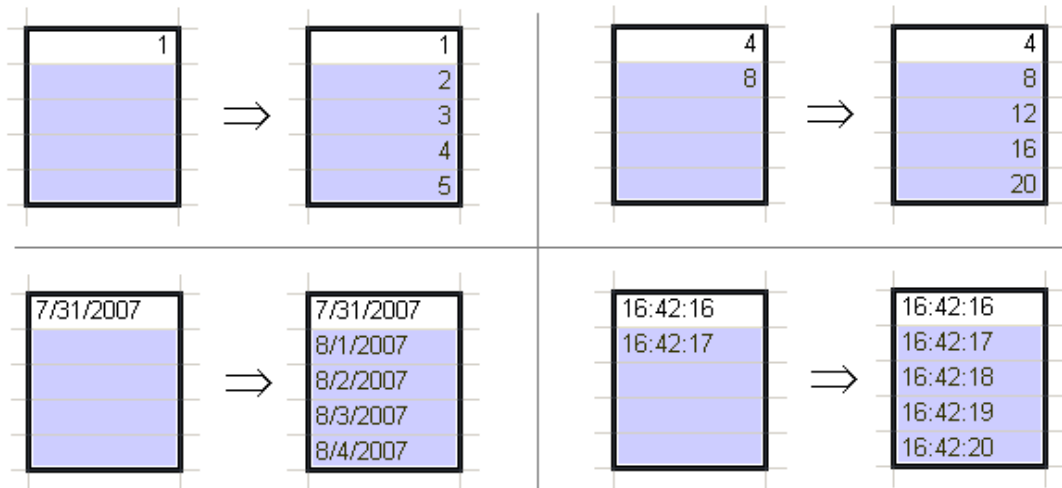
Series with embedded custom list items:



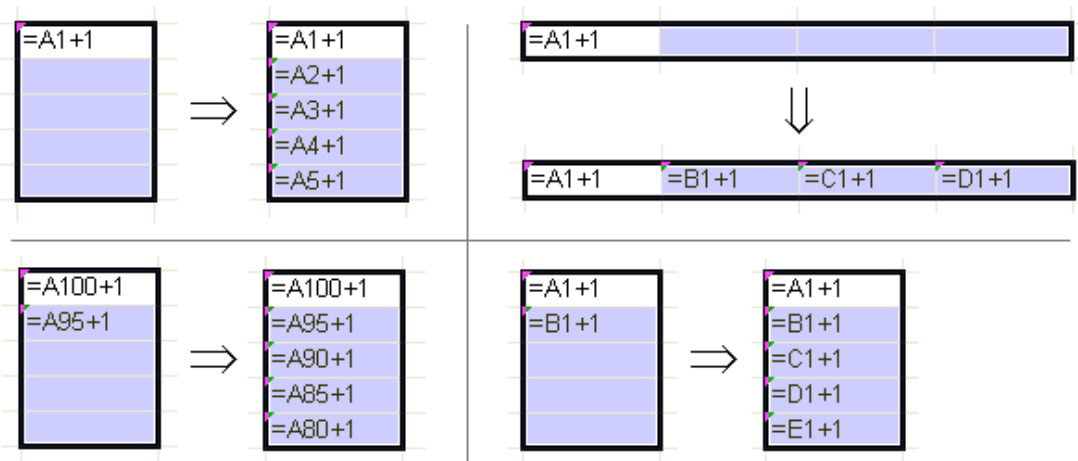
Built-in 2D numeric series:



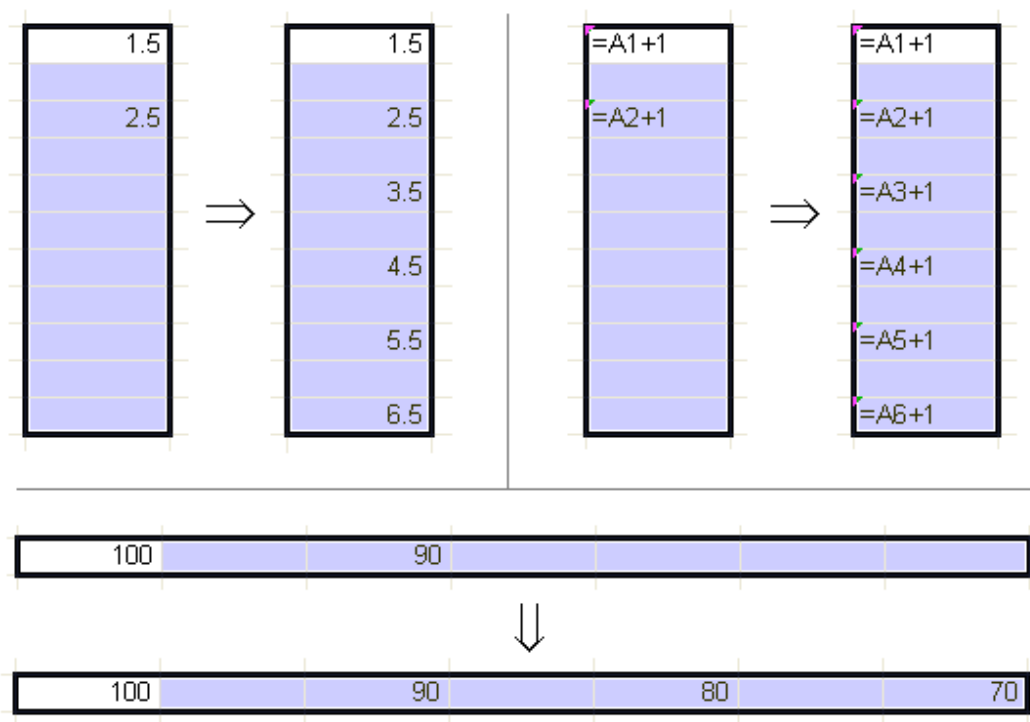
Series of numbers and dates:



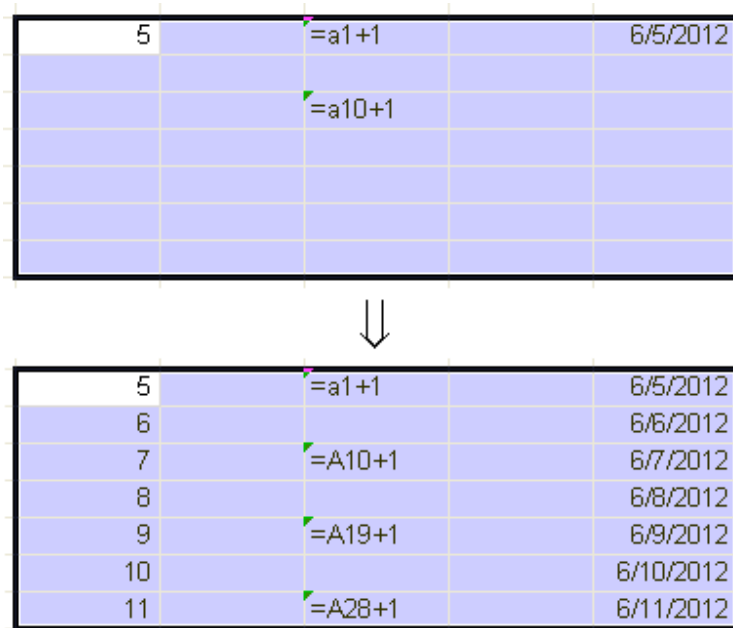
Series of formulas:



Series with elements placed every n-th cell:



Inserting many series at once:



Entering Data: Inserting Sequences and Series Using Formulas

1. Inserting a Sequence of Values

mtxSeries(n, x, [step])

Generates a column vector (a one-column matrix) containing **n** subsequent numbers or date/time strings.

The **x** argument specifies the first element of the series. It may be a number or a date/time string.

The optional **step** argument specifies the increment applied to each subsequent element.

- If omitted for numeric series, **step = 1**.
- If omitted for date/time series, **step = "P1D"** (one day).

Date/Time Period Format

Date/time increments use ISO-8601 duration format:

[+|-]PnYnMnDTnHnMnS.nnn

Examples:

- **P1Y2M10DT11H5M4S.355** → 1 year, 2 months, 10 days, 11 hours, 5 minutes, 4 seconds, 355 milliseconds
- **PT12H7M** → 12 hours and 7 minutes

Examples

=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"}

2. Inserting a Random Series

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

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

2.1 Arguments

- **n** — number of random values to generate.
- **seed1**, **seed2** — initial seeds for the two MLCG generators.
 - If both seeds are omitted, the first call uses fixed internal seeds; subsequent calls continue the sequence.
 - If both seeds are specified, the same random sequence is generated every time.
 - To generate independent sequences, specify **seed1** and omit **seed2**.
- **type** — distribution type:
 - **0** — uniform (0, 1)
 - **1** — normal distribution (mean = v1, stddev = v2)
 - **2** — exponential distribution (mean = v1)
 - **3** — Poisson distribution (mean = v1)
 - **4** — Bernoulli distribution (probability = v1)
 - **5** — geometric distribution (probability = v1)If omitted, **type = 0**.
- **v1**, **v2** — parameters required by some distributions. If a distribution does not require them, they must be omitted.

2.2 Example

`=mtxRand2(4, 10000, 25000,,,) )`

returns `{0.71261246808435; 0.28457538373252; 0.42105025462307; 0.93072516522912}`

Entering Data: Drop-down Lists

1. Overview

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

When editing a cell with an attached drop-down list, GS-Calc opens a window displaying a list of values with checkboxes. This allows you to quickly choose predefined field values instead of re-typing them.

Pressing the cursor keys **Up** (from the first line of the edited field) or **Down** (from the last line) switches focus to the 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.

2. Navigating the List

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** and **Right** to jump between columns.

To check or uncheck a list item, press **Space** or click the checkbox. If the list has multi-selection disabled, checking a new item automatically unchecks the previously selected one.

To accept selected items, press **Enter**, double-click one of the selected items, or click the **OK** button displayed above the edited field.

3. Displaying and Resizing List Columns

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

3	ANTON	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312
4	AROUT	Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.
5	BERGS	Berglunds snabbköp	Christina Berglund	Order Administrator	Berguvsvägen 8
6	BLONP				
7	BOLID	☐ Alfreds Futterkiste	☐ Chop-suey Chinese	☐ France restauration	
8	BONAP	☐ Ana Trujillo Emparedados y helados	☐ Comércio Mineiro	☐ Franchi S.p.A.	
9	BOTTM	☐ Antonio Moreno Taquería	☐ Consolidated Holdings	☐ Frankenversand	
10	BSBEV	☐ Around the Horn	☐ Die Wandermde Kuh	☐ Furia Bacalhau e Frutos do	
11	CACTU	☒ Berglunds snabbköp	☐ Drachenblut Delikatessen	☐ Galeria del gastrónomo	
12	BLAUS	☐ Blondel père et fils	☐ Du monde entier	☐ Godos Cocina Típica	
13	CENTC	☐ Bóldo Comidas preparadas	☐ Eastern Connection	☐ Gourmet Lanchonetes	
14	CHOPS	☐ Bon app'	☐ Ernst Handel	☐ Great Lakes Food Market	
15	COMMI	☐ Bottom-Dollar Markets	☐ Familia Arquibaldo	☐ GROSELLA-Restaurante	
16	CONSH	☐ B's Beverages	☐ FISSA Fabrica Inter. Salchichas S.A.	☐ Hanari Carnes	
17	WANDK	☐ Cactus Comidas para llevar	☐ Folies gourmandes	☐ HILARIÓN-Abastos	
18	DRACD	☐ Centro comercial Moctezuma	☐ Folk och få HB	☐ Hungry Coyote Import Store	
19	DUMON				
20	EASTC				
21	ERNSH				
22	FAMIA				
23	FISSA	FISSA Fabrica Inter. Salchichas S.A.	Diego Roel	Accounting Manager	C/ Moralzarzal, 86
24	FOI IG	Folies gourmandes	Martine Rancé	Assistant Sales Agent	184 chaussée de Tournai

4. Creating and Managing Lists

To create a new list, click the **New List** button. To add items, enter or paste them into the **List Items** field — one item per line.

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

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 select the **AutoComplete** option, the edited cell contents will be automatically completed with the matching text from the list after you type the initial characters.

If the **Sort items** option is not selected, list items appear in the order they were added. Otherwise, items are sorted before displaying.

Drop-Down Lists

List name: Drop-down list 1

List items (one per line):

- Alfreds Futterkiste
- Ana Trujillo Emparedados y helados
- Antonio Moreno Taquería
- Around the Horn
- Berglunds snabbköp
- Blauer See Delikatessen
- Blondel père et fils
- Bólido Comidas preparadas
- Bon app'
- Bottom-Dollar Markets
- B's Beverages
- Cactus Comidas para llevar
- Centro comercial Moctezuma
- Chop-suey Chinese
- Comércio Mineiro
- Consolidated Holdings
- Die Wandernde Kuh
- Drachenblut Delikatessen
- Du monde entier
- Eastern Connection
- Ernst Handel
- Familia Arquibaldo
- FISSA Fabrica Inter. Salchichas S.A.

Columns: 3

☐ Validate cells ☐ AutoComplete

☒ Allow empty cells ☐ Sort list items

☐ AutoAppend

☐ Multiple Selection Separator: comma ,

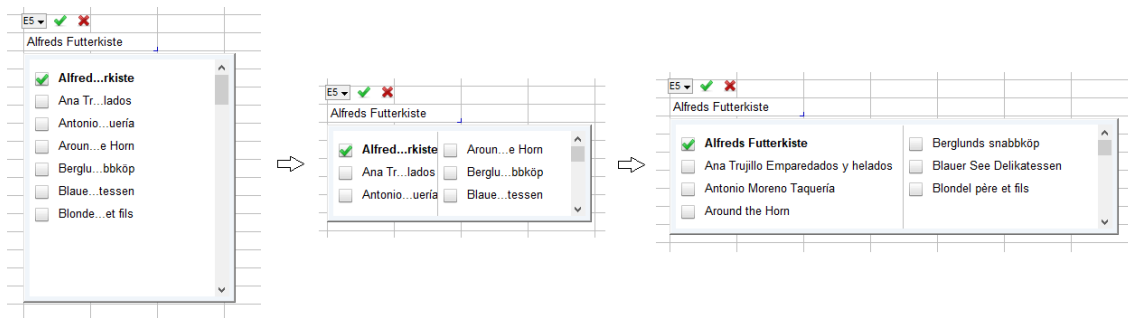
Buttons: New List, Rename, Remove, OK, No List, Cancel, Help

5. Using Lists Across Worksheets

A single list can be used by any number of cells in any worksheet within the workbook. You can create up to **100 lists** per workbook. To move lists between workbooks, copy and paste the data from the **List Items** field.

6. Notes

If you specify more than one column, splitter lines appear only when list items occupy multiple columns. To resize columns, you may need to temporarily shrink the list window, for example:



Using the Increment, Decrement and Operations Commands

1. Automatically Add, Multiply, Divide or Subtract Values

Select a cell containing a numeric value — or any range of such cells — and use **Edit > Increment** or **Edit > Decrement**.

By default, these commands add or subtract **1**.

You can override this behavior using **Edit > Operations** to define more complex value-changing rules.

2. Defining Custom Increment/Decrement Operations

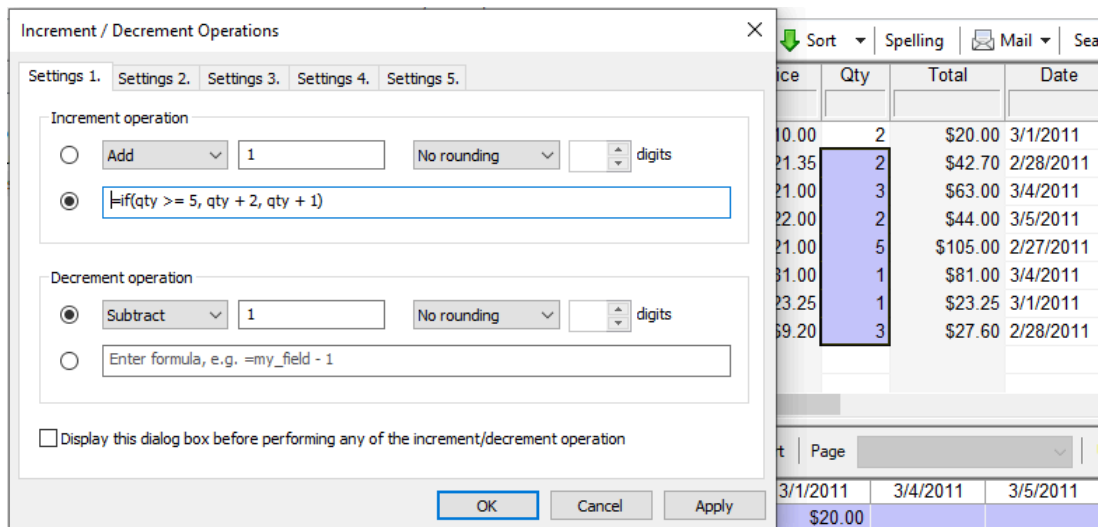
Use **Edit > Operations** to specify how Increment and Decrement should behave. Operations may include:

- adding a fixed value,
- subtracting a fixed value,
- multiplying,
- dividing,
- or applying a formula that refers to the selected cell(s).

There are five independent increment/decrement settings. The active one is determined by the selected tab in the **Operations** dialog.

3. Example

1. Define a formula-based increment (and/or decrement) on the **Settings1** tab and activate it:



2. Use the Increment command (**Ctrl+Alt+="**) or Decrement (**Ctrl+Alt+="-**). Values in selected cells will be modified according to the defined formula:

UnitPrice	Qty	Total	Date
\$10.00	2	\$20.00	3/1/2011
\$21.35	3	\$64.05	2/28/2011
\$21.00	4	\$84.00	3/4/2011
\$22.00	3	\$66.00	3/5/2011
\$21.00	7	\$147.00	2/27/2011
\$81.00	2	\$162.00	3/4/2011
\$23.25	2	\$46.50	3/1/2011
\$9.20	4	\$36.80	2/28/2011

Spell-Checking and Hunspell Dictionaries

1. Spell-Checking Availability

GS-Calc supports two spell-checking engines:

- **Windows Spell Checking** — available on Windows 8.x and later.
- **Hunspell** — available on all supported systems.

You can switch between these engines in:

- **Settings → Options → Spelling**
- **Tools → Spelling**

The selected engine determines which dictionaries are available and how custom words are stored.

2. Adding Hunspell Dictionaries

2.1 Downloading Hunspell Dictionaries

Hunspell dictionaries are distributed as ZIP archives, typically with ***.xpi** or ***.oxz** extensions. You can download them from various sources, for example:

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

After downloading the desired dictionary file, rename it to ***.zip** and extract it.

2.2 Installing Hunspell Dictionaries

Each dictionary archive contains two files:

- ***.aff** — affix rules
- ***.dic** — dictionary word list

Their names typically follow ISO language codes, e.g.:

en-US.aff, en-US.dic

To install a dictionary:

1. Extract the ***.aff** and ***.dic** files.
2. Copy them into a folder of your choice.
3. Specify that folder in **Settings → Options → Spelling**.

GS-Calc will automatically detect all Hunspell dictionaries in that folder and list them in the **Tools → Spelling** dialog.

2.3 Removing Hunspell Dictionaries

To remove a dictionary, simply delete its ***.aff** and ***.dic** files from the folder.

3. Adding Custom Words

3.1 Custom Word Storage

Users can add custom words to existing dictionaries. When using the Hunspell engine, users may also create full custom dictionaries.

Custom words are stored in a text file located in the folder specified in the spelling options.
The file:

- has the extension ***.txt**
- uses the same ISO language code as the dictionary
- must be saved in **UTF-8** encoding

For example, if the dictionary files are:

en-US.aff, en-US.dic

then custom words will be stored in:

en-US.txt

3.2 Adding Words During Spell-Checking

New words can be added directly during spell-checking by clicking the **Add Word** button.

The ***.txt** file can also be edited manually to add many words at once.

4. ISO Language Codes

The following ISO language codes may appear in dictionary file names or custom word files:

- 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

Generating Passwords

1. Overview

The **Insert > Password** command allows you to fill selected cells with unique passwords.

Increasing the password length or expanding the character set increases the probability of uniqueness.

For example, with default settings, generating 8-character passwords for an entire 12-million-row column should produce all unique values.

Passwords are generated using a continuously advancing partial MLCG random series ($\sim 10^{18}/4$ passwords). Clicking **Reset MLCG Generator Seeds** starts a new independent series.

2. Checking for Duplicates

You can verify uniqueness using:

- **F3 / Find > Find All > Find Duplicates**
- **FILTER()** with the “duplicates” filter

Note: Some duplicates may differ only by letter case (lowercase vs uppercase). The Find Duplicates command lists them side by side.

3. Unicode Considerations

Generated passwords may include Unicode characters. However, non-ASCII characters should be used cautiously because some systems — especially recovery systems — may not support them.

4. Using Passwords for Random Data Testing

Unique text strings can also be used to test data models with random data.

To generate random lists of predefined strings:

- Use **Insert > Custom Series** to load or create a list.
- Use **Insert > Random Data** with that list name.

Insert Password

Length: min. 3 max. 8

☐ Prefix:

☐ Suffix:

☒ Use min. 8 max. 8 characters from:
abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789@#\$%&*?-_

☐ Use min. 8 max. 8 characters from:

☐ Use min. 8 max. 8 characters from:

☐ Use min. 8 max. 8 characters from:

Reset MLCG Generator Seeds OK Cancel

1. Overview

Splitting views allow you to display different areas of the same worksheet — or even different worksheets — within a single main worksheet window. This makes it possible to monitor changes in multiple locations simultaneously, edit distant cells without scrolling, and emulate “Freeze Rows/Columns” functionality found in other spreadsheet applications.

Each worksheet can contain up to **100** splitted views, and a workbook may contain an unlimited number of such configurations. Splitting configurations are saved directly into the workbook file.

You can protect the current splitting configuration using **Tools** → **Protect Structure**.

2. Proportional Splitter

The **View** → **Proportional Splitter** option controls how views behave when the main window is resized:

- **On** — all views resize proportionally.
- **Off** — view sizes remain fixed.

3. Pane Layout Templates

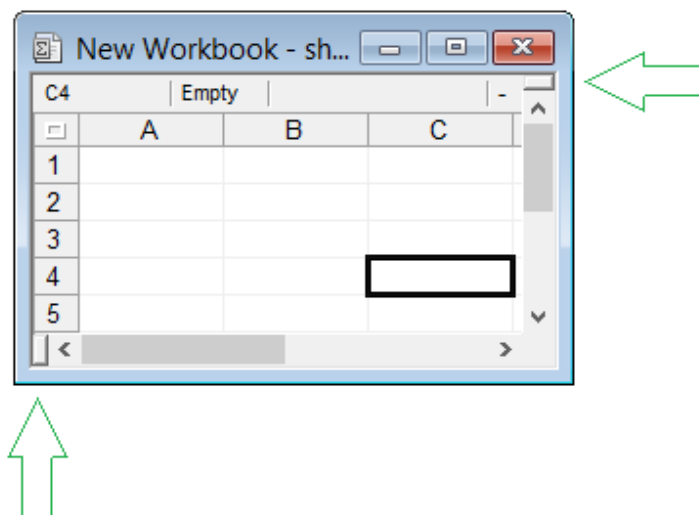
To quickly apply different splitting configurations or copy a layout to other worksheets, use **View** → **Pane Layout Templates**. These templates are stored as global settings and can be reused across worksheets and workbooks.

4. Splitting a Worksheet View

4.1 Using Split Commands or Splitter Bars

You can split a worksheet view using:

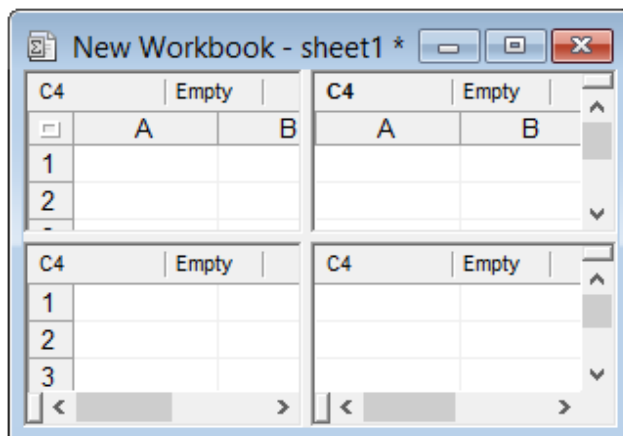
- **View** → **Split View Horizontally**
- **View** → **Split View Vertically**
- Dragging the vertical or horizontal splitter bar to the desired position



4.2 Automatic Splitting Mode

If **View → Automatic Splitting Mode** is active:

- Vertical splitting affects the entire column of views.
- Horizontal splitting affects the entire row of views.
- Views always form a regular grid (e.g., 1×2, 2×1, 2×2, 3×2, ...).



4.3 Synchronization in Automatic Mode

In automatic mode, newly created views are synchronized by default:

- Scrolling columns in one view scrolls columns in all views in the same column.
- Scrolling rows in one view scrolls rows in all views in the same row.
- Synchronization applies to both scrolling movement and the first visible row/column.

By default:

- Row headings are shown only in the left-most views.
- Column headings are shown only in the top-most views.

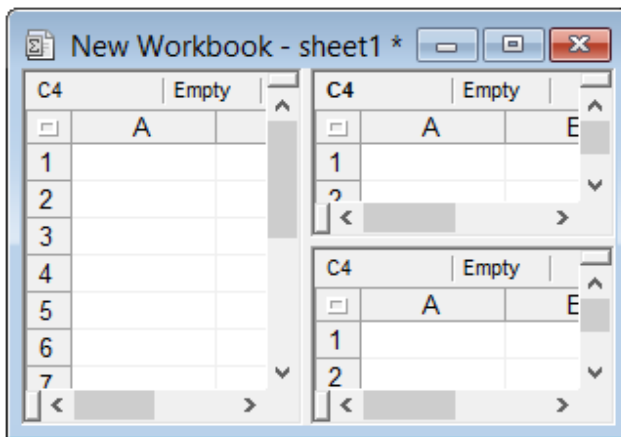
To change headings for a specific view, click the view and use:

- **View → Headings And Scrollbars → Row Headings**
- **View → Headings And Scrollbars → Column Headings**

4.4 Manual Splitting Mode

If **View → Manual Splitting Mode** is active:

- Splitting affects only the selected view.
- Views are not synchronized automatically.



4.5 Synchronization in Manual Mode

You can manually synchronize views using:

- **View → Column Synchronization**
- **View → Row Synchronization**

In manual mode:

- Synchronized views may have different left-most columns.
- Synchronized views may have different top-most rows.

For details, see: [Synchronizing views](#).

4.6 Headings in Manual Mode

By default, row and column headings are displayed in all views. You can change this per view using:

- **View → Headings And Scrollbars → Row Headings**
- **View → Headings And Scrollbars → Column Headings**

4.7 Choosing a Worksheet for a View

By default, all views display the same worksheet. To assign a different worksheet to a specific view, use:

View → Choose Worksheet

5. Removing Splitted Views

To remove a split:

- Use **View → Split View Horizontally** or **View → Split View Vertically** again, or
- Drag the vertical or horizontal splitter bar to any window edge.

Worksheet Views: Row and Column Synchronization Options

1. Overview

Splitted worksheet views can be synchronized so that scrolling columns or rows in one view automatically scrolls the corresponding columns or rows in other synchronized views.

Synchronization commands are available through the [View menu](#).

2. Automatic Splitting Mode

Views created in **Automatic Splitting Mode** are synchronized by default. They form a regular grid of views, and synchronization applies to:

- columns (within each column of views)
- rows (within each row of views)

This default synchronization cannot be disabled in automatic mode.

3. Manual Splitting Mode

Views created in **Manual Splitting Mode** are not synchronized initially. Users must explicitly toggle synchronization using:

- **Column Synchronization**
- **Row Synchronization**

In manual mode, synchronized views do not need to share the same first visible column or row. This allows synchronizing distant worksheet ranges that show different areas of the worksheet.

4. Toggling Synchronization (Manual Mode)

To toggle synchronization:

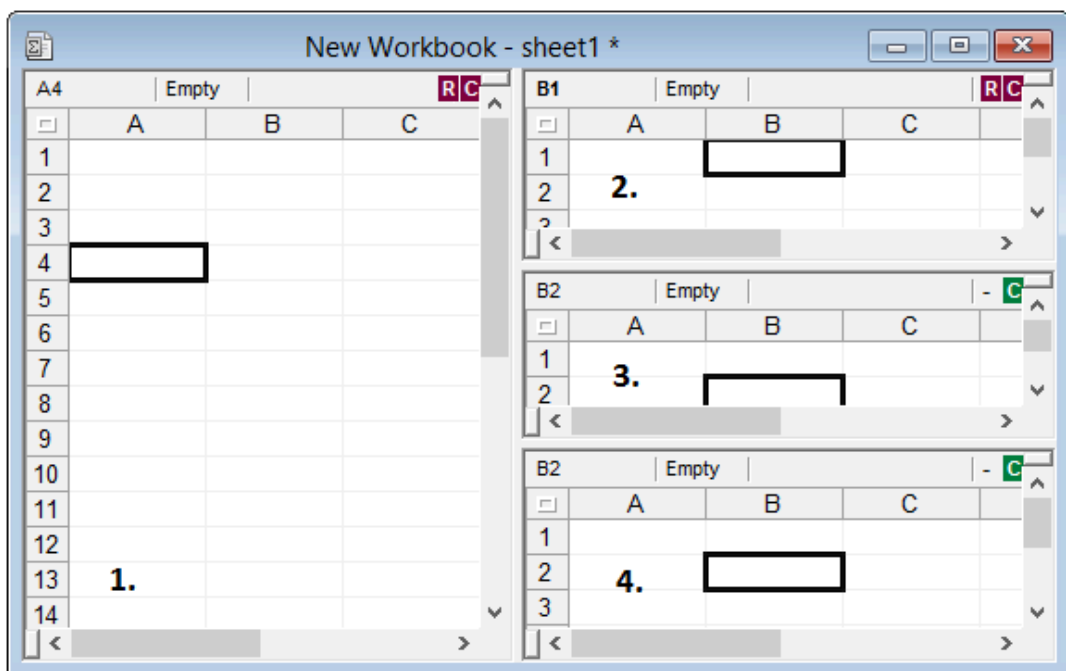
1. Click the first view to synchronize.
2. Use **Column Synchronization** or **Row Synchronization**.
3. A message appears: "Click a view to toggle the column (row) synchronization on or off".
4. Click another view to add it to or remove it from the synchronization chain.

4.1 Synchronization Tags

If the **Synchronization Tags** option is enabled, synchronized views display:

- **R** — row synchronization
- **C** — column synchronization

Views belonging to the same synchronization chain use matching tag colors.



In the example above:

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

5. Synchronizing Cell Range Selections

To synchronize cell selections across views, enable:

Keep Selection Synchronized

Selecting cells in one view will select the same number of rows and/or columns in synchronized views.

To perform a one-time synchronization of selections, use:

Make Same Selection

Note: To quickly return to the origin of a selection, click the selection address in the worksheet/table window toolbar.

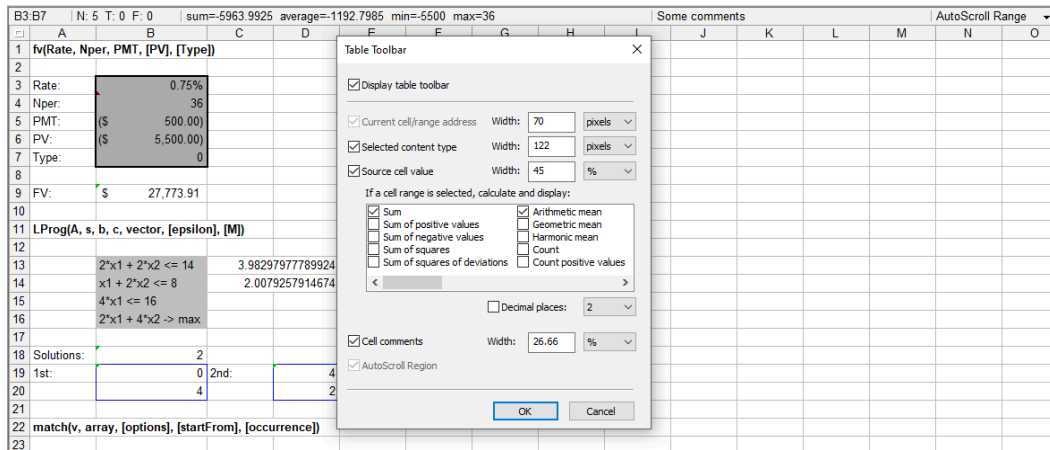
Selecting Functions for the Table Toolbar and Related Options

1. Overview

The worksheet table toolbar is the first line displayed in the worksheet window. It provides quick access to reference navigation, cell type information, automatic calculations, cell comments, and AutoScroll range management.

2. Toolbar Components

- **Reference Navigation** — Clicking the first toolbar section (showing the current reference) jumps back to that location if you previously scrolled away using scrollbars.
- **Cell Type Information** — The second section displays:
 - the current cell type (for a single-cell selection)
 - counts of cell types (for multi-cell selections)
- **Automatic Calculations** — The third section displays several calculation results computed automatically for the selected range. Clicking this section opens the Options dialog box, where you can choose:
 - which functions to display
 - other table toolbar options



Calculations run in the background each time you complete a selection (e.g., releasing the mouse button or the Shift key). During calculation, displayed values are temporarily grayed out.

- **Cell Comments** — The fourth section displays comments for the current cell. Click it to add or edit a comment.
- **AutoScroll Range Indicator** — The fifth section shows whether an AutoScroll range is active. Click it to:
 - create a new AutoScroll range
 - jump to and select an existing range
 - deactivate the current AutoScroll range

Hiding View Elements / Maximizing Table View Display Areas

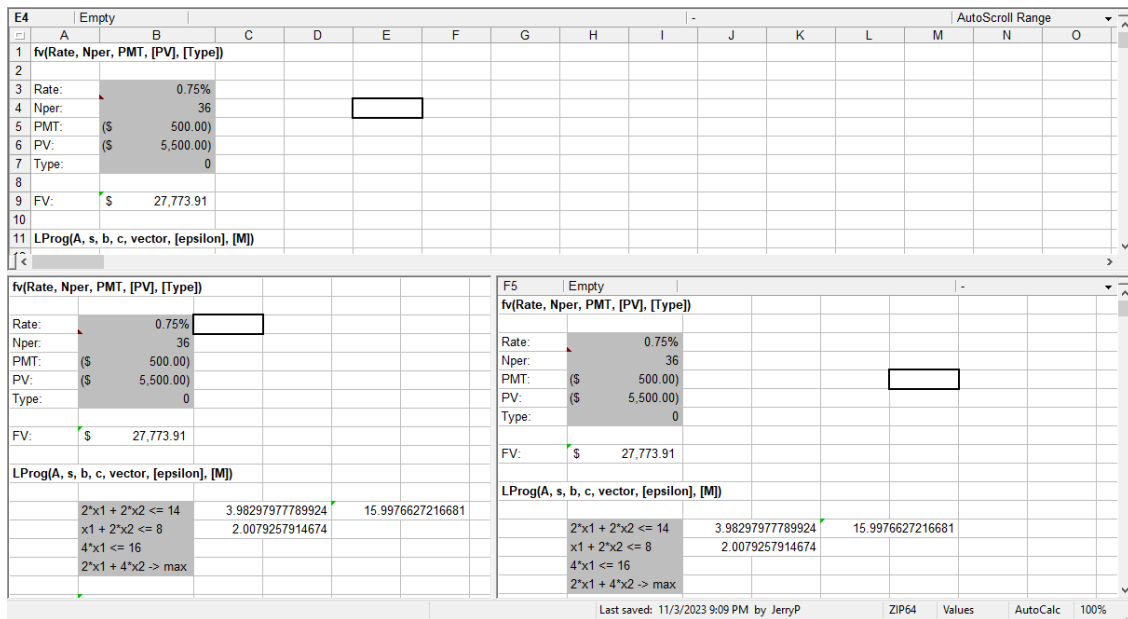
1. Overview

If you want to maximize the worksheet display area to show as many cells as possible — especially after splitting the main view — you can hide all interface elements except the cells themselves.

2. Steps

- Hide the table toolbar using the toolbar button or the **View > Table Toolbar** command.
- Open **View > Headings & Scrollbars** and deselect:
 - Column Headings
 - Row Headings
 - Horizontal Scrollbar
 - Vertical Scrollbar

Each view can be configured independently. The final result may look like the following:



Cross-Highlighting Columns/Rows for Better Readability

1. Overview

Working with wide tables containing textual data can be difficult, especially on large monitors. When you need to control or visually compare values located far apart across many columns or rows, it becomes easy to lose track of the current position.

The **Cross-Highlighting** feature in GS-Calc helps improve readability by highlighting the current row or column. You can optionally highlight additional rows or columns above, below, to the left, or to the right of the active cell. These highlighted areas are displayed with a brighter background.

2. Activating Cross-Highlighting

To enable or disable this feature, use:

View > Toggle Cross-Highlighting On/Off

To adjust the number of highlighted rows or columns and the contrast between highlighted and non-highlighted areas, use:

View > Toggle Cross-Highlighting Options

You can also use the corresponding toolbar button.

B	C	D	E	F	G
CompanyName	ContactName	ContactTitle	Address	City	Region
Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57	Berlin	
Ana Trujillo Emparedados y helados	Ana Trujillo	Owner	Avda. de la Constitución 2222	México D.F.	
Antonio Moreno Taqueria	Antonio Moreno	Owner	Mataderos 2312	México D.F.	
Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.	London	
Berglunds snabbköp	Christina Berglund	Order Administrator	Berguvsvägen 8	Luleå	
Blauer See Delikatessen	Hanna Moos	Sales Representative	Forsterstr. 57	Mannheim	
Blondel père et fils	Frédérique Citeaux	Marketing Manager	24, place Kléber	Strasbourg	
Bólido Comidas preparadas	Martín Sommer	Owner	C/ Araquil, 67	Madrid	
Bon app'	Laurence Lebihan	Owner	12, rue des Bouchers	Marseille	
Bottom-Dollar Markets	Elizabeth Lincoln	Accounting Manager	23 Tsawassen Blvd.	Tsawassen	BC
B's Beverages	Victoria Ashworth	Sales Representative	Fauntleroy Circus	London	
Cactus Comidas para llevar	Patricio Simpson	Sales Representative			
Centro comercial Moctezuma	Francisco Chang	Marketing Representative			
Chop-suey Chinese	Yang Wang	Owner			
Comércio Mineiro	Pedro Afonso	Sales Representative			SP
Consolidated Holdings	Elizabeth Brown	Sales Representative			
Die Wandermöbel	Rita Müller	Sales Representative			
Drachenblut Delikatessen	Sven Ottilieb	Order Administrator			
Du monde entier	Janine Labrune	Owner			
Eastern Connection	Ann Devon	Sales Representative			
Ernst Handel	Roland Mendel	Sales Representative			
Familia Arquibaldo	Aria Cruz	Marketing Representative			SP
FISSA Fabrica Inter. Salchichas S.A.	Diego Roel	Accounting Manager	C/ Moralzarzal, 86	Madrid	
Foies gourmandes	Martine Rancé	Assistant Sales Agent	184, chaussée de Tournai	Lille	

Cross Highlighting Options

☒ Column(s):
 1
 (Proportionally)

☒ Row(s):
 2
 Below selection

Contrast:
 23

OK

Cancel

Using the Dark Mode

1. Overview

If you prefer working with a darker window background to reduce blue-light exposure or to decrease energy usage on a notebook, you can enable the dark mode in GS-Calc.

To turn this mode on or off, use:

View > Dark Mode

You can also use the corresponding toolbar button.

Using Pivot Tables

1. Overview

Pivot tables allow you to quickly compute and visualize statistical summaries from flat source tables. Typical operations include counting, summing, grouping, sorting, and computing minimum/maximum values.

In GS-Calc, pivot tables are created using the **pivotData()** array formula. Because this formula may require several parameters, it is often easier to use the **Insert → Pivot Table** composer.

2. Usage Examples

2.1 Customers Table

Consider a table containing fields such as **Country** and **City**:

	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

Count customers by country

1. Open **Insert** → **Pivot Table**.
2. Select the customers table and click **Source Range**.
3. Select **Country** as the only **Row** field.
4. Select **Count** as the default function.
5. Choose a destination cell and click **Insert**.

	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

Count customers by country and city

1. Open **Insert** → **Pivot Table**.
2. Select the customers table and click **Source Range**.
3. Select **Country** as the first **Row** field.
4. Select **City** as the second **Row** field.
5. Select **Count** as the default function.
6. Choose a destination cell and click **Insert**.

	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

2.2 Products Table

Consider a table containing fields such as **ProductName**, **Date**, **Qty**, and **Total**:

	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
⋮							

Sales volume by day

1. Open **Insert** → **Pivot Table**.
2. Select the products table and click **Source Range**.
3. Select **ProductName** as the only **Row** field.
4. Select **Date** as the only **Column** field.
5. Select **Qty** as the only **Data** field and choose **Sum**.
6. Choose a destination cell and click **Insert**.

	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

Sales volume and quantity by day

1. Open **Insert** → **Pivot Table**.
2. Select the products table and click **Source Range**.
3. Select **ProductName** as the only **Row** field.
4. Select **Date** as the only **Column** field.
5. Select **Total** as the first **Data** field and choose **Sum**.
6. Select **Qty** as the second **Data** field and choose **Sum**.
7. Choose a destination cell and click **Insert**.

	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

3. pivotData() Formula

PIVOT formulas have the following form:

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

This formula creates and returns a pivot table for the data in the **source** range.

3.1 Row, Column, and Data Fields

The **rows**, **columns**, and **data** parameters are arrays containing field indices. Indices are 1-based and relative to the top-left corner of the source range.

Examples:

{1, 5}, {4}, {1}

Incorrect indices cause **#VALUE!**. In GS-Calc 9.0, the **columns** array may contain only one element. If **data** fields are omitted, the last row field and the default pivot function are used.

3.2 Functions

The **functions** argument is an array of predefined constants:

- PIVOT::Sum — sum of values
- PIVOT::SumPositive — sum of positive values only
- PIVOT::SumNegative — sum of negative values only
- PIVOT::SumSquares — sum of squares
- PIVOT::Count — count of non-empty values
- PIVOT::CountPositive — count of positive values
- PIVOT::CountNegative — count of negative values
- PIVOT::CountZeroes — count of zero values
- PIVOT::Mean
- PIVOT::Geomean
- PIVOT::Harmean
- PIVOT::Min — minimum value
- PIVOT::Max — maximum value
- PIVOT::Quartile1
- PIVOT::Median
- PIVOT::Quartile3
- PIVOT::Var
- PIVOT::VarP
- PIVOT::Stdev
- PIVOT::StdevP
- PIVOT::Skew
- PIVOT::Kurt
- PIVOT::Mode

Examples:

{PIVOT::Sum}

{PIVOT::Sum, PIVOT::Count}

3.3 Required Column Field or Grand Totals

You must specify either:

- PIVOT::ColumnGrandTotals
- or at least one column field

Otherwise the function returns **#NUM!**.

3.4 Options (FULL descriptions preserved)

The **options** parameter may be a sum of:

- **PIVOT::ColumnGrandTotals (1)** — display column grand totals in the last column
- **PIVOT::RowGrandTotals (2)** — include row grand totals in the last row
- **PIVOT::SubTotals (4)** — include subtotals (for pivot tables with 2+ row fields)
- **PIVOT::ShowZeroes (8)** — show zeroes for empty data fields; otherwise empty subtotals appear as #N/A!
- **PIVOT::RepeatRowFields (16)** — display all row field values, even duplicates

- **PIVOT::CaseSensitive (218)** — use case-sensitive comparison when filtering
- **PIVOT::SortRowsDescending (256)** — sort output rows in descending order
- **PIVOT::SortColumnsDescending (512)** — sort output columns in descending order
- **PIVOT::NoSourceFieldNames (1024)** — source range contains no field names; use “Field n.” instead
- **PIVOT::IgnorePunctuation (16384)** — ignore certain punctuation marks when sorting/filtering
- **PIVOT::NeutralSortOrder (32768)** — use language-neutral comparison
- **PIVOT::Pattern (65536)** — treat filters not starting with (=, >, >=, <, <=) as wildcard patterns
- **PIVOT::HorzData (131072)** — generate multiple data field values in single rows

Examples:

{PIVOT::ColumnGrandTotals}

{PIVOT::ColumnGrandTotals + PIVOT::RowGrandTotals + PIVOT::SubTotals}

3.5 Filters

Optional **[field, filter]** pairs specify filtering conditions. Only records matching all conditions are included.

Filters may be:

- A text string beginning with =, >, >=, <, <=
- A number
- A wildcard pattern using ? or * (escape with ~? or ~*)

3.6 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")**

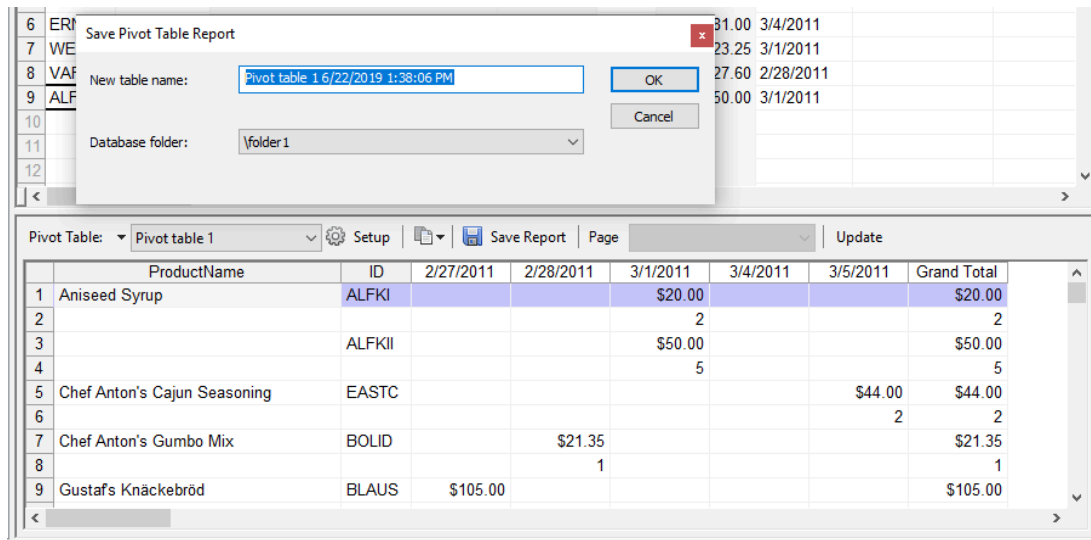
Creating Pivot Table Reports

1. Saving Pivot Table Reports

Clicking the **Save** icon in the Pivot Table View creates a new time-stamp report for the current pivot table.

Procedure:

1.



2.

Tables		Date	ProductName	ID	Sum of Total	Sum of Qty
gs_sample2d						
customers						
products						
orders						
ID						
ProductID						
ProductName						
UnitPrice						
Qty						
Total						
Date						
OrderID						
folder1						
Pivot table 1 6/22/2019 1:37:29 P						

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		

2. Copying Pivot Table Data

You can also copy pivot table data to other programs — either as an image or using the available “Copy” commands from the context menu.

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 occurring
Sum

Grand Totals in columns and rows

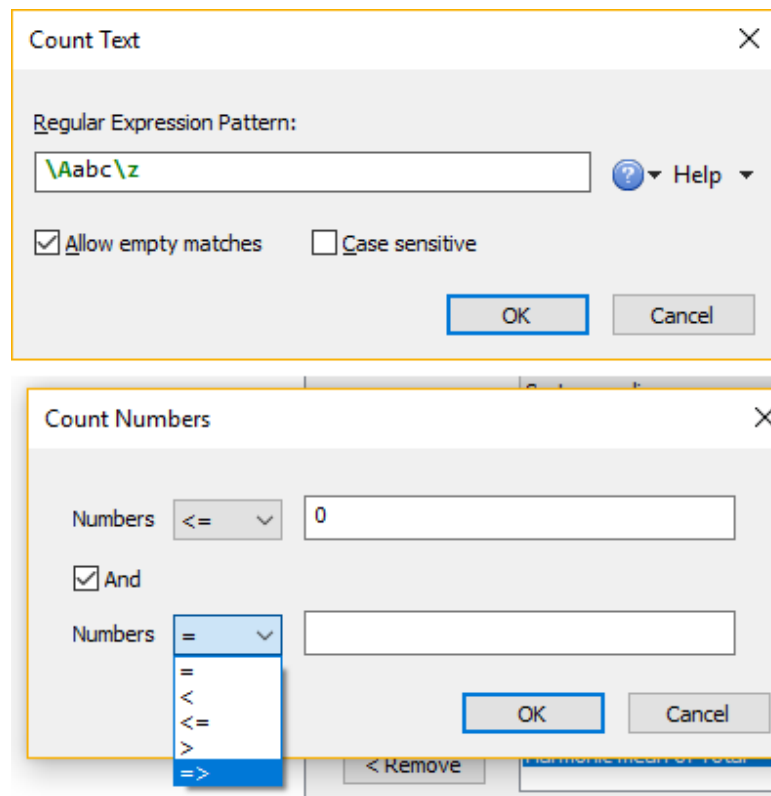
☒ Subtotals ☒ Ignore punctuation

☐ Repeat rows fields ☐ Case sensitive

☐ Treat empty fields as zero values

Page field: ID

OK Cancel Help



Using Monte Carlo Simulations

1. Overview

Monte Carlo simulations use random number generators to analyze formulas, estimate statistical properties, and simulate real-world measurements or observations. They are especially useful for:

- approximating maximum or minimum values of formulas
- simulating noisy or uncertain input data
- evaluating formula behavior under random perturbations
- collecting large numbers of results for statistical analysis

In GS-Calc, simulations operate by repeatedly filling designated **Input** cells with random numbers and evaluating formulas in designated **Output** cells. Running many loops produces statistically meaningful results.

2. Simulation Components

2.1 Input Cells

Input cells must be empty or contain only numbers. During each loop, they are filled with random values generated according to the selected distribution.

Input cells may be:

- single cells
- ranges
- multiple independent regions

2.2 Output Cells

Output cells contain formulas that process the random input values and return numeric results.

Output cells may also include a **rejection rule** — a formula that determines whether the result from a given loop should be included in the statistics.

If the rejection formula returns a non-zero value, the loop is rejected.

2.3 Random Number Generators

Monte Carlo simulations use the same random number generators as the **mtxRand** functions. You may specify generator parameters when creating a simulation.

See the **mtxRand2()** documentation for details on:

- distribution types
- seed behavior
- independent vs. sequential random series

3. Simulation Management

The following commands are available under the **Simulations** menu:

- **Add Simulation** — create a new simulation with input/output cells and optional generator parameters.
- **Duplicate Simulation** — create a copy of an existing simulation.
- **Delete Simulation** — remove the selected simulation.
- **Delete All Simulations** — remove all simulations.

4. Adding Input and Output Cells

4.1 Input Cells

Input cells must be numeric or empty. They are filled with random values during each loop.

4.2 Output Cells

Output cells contain formulas that compute results from the input values. Only numeric results are valid.

4.3 Rejection Rules

A rejection rule determines whether a loop's output should be included in the statistics.

If the rejection formula returns a non-zero value, the loop is rejected.

5. Example I — Linear Programming Approximation

Consider the linear programming problem:

$$2 \cdot x_1 + 2 \cdot x_2 \leq 14$$

$$x_1 + 2 \cdot x_2 \leq 8$$

$$4 \cdot x_1 \leq 16$$

$$\text{Maximize: } 2 \cdot x_1 + 4 \cdot x_2$$

The screenshot shows a spreadsheet with a linear programming problem setup in the top section and a simulation table in the bottom section.

Linear Programming Problem Setup:

Cell	Formula	Value
D13	$2 \cdot x_1 + 2 \cdot x_2 \leq 14$	3.98297977789924
D14	$x_1 + 2 \cdot x_2 \leq 8$	2.0079257914674
D15	$4 \cdot x_1 \leq 16$	
D16	$2 \cdot x_1 + 4 \cdot x_2 \rightarrow \max$	

Simulation Table:

Comments	Cell/Range	Type	Specification	Min	Max	Mean
x1 & x2	Formulas!D13:D14	Input	Uniform: (-0.1, 5)	-0.09985531088323	4.99983776532403	2.4426936476502
max	Formulas!F13	Output	Reject if: $-(2 \cdot d13 + 2 \cdot d14 > 14) + (d13 > 4)$	-0.41603856401671	15.9976627216681	9.40847507709679

5.1 Input Cells

Specify **D13** and **D14** as input cells with uniform distribution $(-0.1, 5)$.

5.2 Output Cell

Specify **E13** as the output cell with:

$$=2 \cdot d13 + 4 \cdot d14$$

5.3 Rejection Rule

Use the following rejection formula:

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

5.4 Results

After 10,000 loops (default generator parameters), the maximum of E13 is approximately:

15.997

with:

- $x1 \approx 3.983$
- $x2 \approx 2.01$

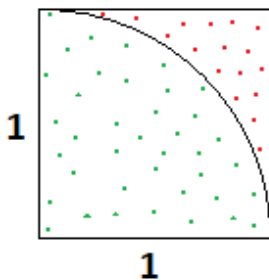
The exact values computed using **LProg()** are:

- Maximum = 16
- $x1 = 4$
- $x2 = 2$

Note: You may add multiple output cells with separate rejection rules instead of merging them.

6. Example II — Estimating π Using Monte Carlo

To estimate π , generate random points inside a unit square and count how many fall inside the quarter circle of radius 1.



6.1 Input Cells

Specify **B2** and **B3** as input cells with uniform distribution (0, 1). These represent random (x, y) pairs.

6.2 Counting Points

The green points lie inside the quarter circle. Thus:

$(\text{green points}) / (\text{all points}) = \pi / 4$

6.3 Adding Control Cells

1. Add the total number of loops to **B5**.
2. Add the number of non-rejected loops (green points) to **B6**.

6.4 Output Formula

Enter the following in **D2**:

=b6/b5*4

Add it as an output value.

6.5 Rejection Rule

Use:

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

This rejects points outside the quarter circle.

6.6 Running the Simulation

Specify the number of loops and click **Start**. After completion, **D2** contains the π estimate.

To save all loop data and track how the estimate evolves, use:

Results → Save All Loop Data

Up to ~3 billion loops can be saved (256 × 12M).

7. Viewing Min/Max

Selecting an item in the Input/Output list and choosing **Data → Show Min/Max** fills the worksheet cells with the input/output values from the loop where the minimum or maximum occurred.

8. Saving Loop Data

After completing the specified number of loops, results may be saved as a new worksheet.

Loops are saved in rows. If the number of input/output cells exceeds 4096 columns, the saved data will be truncated.

Ranges are expanded into individual cells.

9. Running, Cancelling, and Resuming Simulations

Simulations may be stopped and resumed at any time.

During execution, the input/output list updates every 0.5 seconds, showing:

- maximum
- minimum
- mean
- variance
- standard deviation
- counter (excluding rejected loops)

Values in square brackets [] show changes since the previous update.

Starting a simulation pauses pending workbook updates, and vice-versa.

vLookup() Function

1. Overview

The **vLookup()** function searches for a value **v** in the leftmost column of an array. If a match is found, it returns the value from the same row and the **n-th** column of that array.

2. Function Syntax

vLookup(v, array, n, [type])

vLookup(v, array, n, [type], [startIndex], [occurrence])

2.1 Parameters

- **v** — value to search for in the leftmost column.
- **array** — a 2-D array or range; the first column is searched.
- **n** — column index (within the array) from which the result is returned.
- **type** — search mode (exact, approximate, pattern, regex, sorted, etc.).
- **startIndex** — starting position for the search. Examples:
 - 1 = first cell in the searched range
 - 2 = second cell
 - -1 = last cell
 - -2 = second-to-last cell

Any non-zero integer can be used to specify where the search should start or end.

- **occurrence** — which occurrence of the matching value to use. Examples:

- 1 = first match (top-down)
- 2 = second match (top-down)
- -1 = last match (bottom-up)
- -2 = second-to-last match (bottom-up)

Any non-zero integer can be used to specify which occurrence (counted from **startIndex**) should be returned.

If **type** is omitted, it defaults to **1**.

3. Search Types and SEARCH:: Flags

3.1 Basic Types

- **0** — linear search for an exact match; **v** may be a pattern using **?** (any single character) and ***** (any string, including empty). To search for literal **?** or *****, prefix them with **~**.
- **1** — if no exact match is found, returns the largest value that is not greater than **v**; no pattern matching; the searched range must be sorted ascending.
- **-1** — if no exact match is found, returns the smallest value that is not smaller than **v**; no pattern matching; the searched range must be sorted descending.

3.2 SEARCH:: Flags

- **SEARCH::MatchNotGreater (2)** — largest value $\leq v$.
- **SEARCH::MatchNotSmaller (4)** — smallest value $\geq v$.
- **SEARCH::SortAscending (8)** — binary search on ascending sorted range.
- **SEARCH::SortDescending (16)** — binary search on descending sorted range.
- **SEARCH::CaseSensitive (128)** — case-sensitive comparison.
- **SEARCH::FirstMatch (256)** — return the first match.
- **SEARCH::LastMatch (512)** — return the last match.
- **SEARCH::AutoSort (1024)** — automatically sort internally during the first update and then use binary search. In this case, **startIndex** refers to the internally sorted range.
- **SEARCH::MixedData (2048)** — searched range contains both text and numbers.
- **SEARCH::SortIndex (4096)** — only with AutoSort; returns index relative to the internally sorted range.
- **SEARCH::RegEx (8192)** — **v** is a regular expression.
- **SEARCH::IgnorePunctuation (16384)** — ignore certain punctuation marks (word sort order).
- **SEARCH::NeutralSortOrder (32768)** — language-independent comparison.
- **SEARCH::Pattern (65536)** — **v** is a wildcard pattern; cannot be used with fast binary search.

3.3 Equivalences

- **1** = SortAscending + MatchNotGreater
- **-1** = SortDescending + MatchNotSmaller
- **0** = plain linear exact match

3.4 Restrictions

- SEARCH::Pattern and SEARCH::RegEx cannot be used with:
 - SEARCH::MatchNotGreater
 - SEARCH::MatchNotSmaller
 - SEARCH::StringSort
 - SEARCH::CaseSensitive
 - SEARCH::SortAscending
 - SEARCH::SortDescending
- SEARCH::MatchNotGreater and SEARCH::MatchNotSmaller cannot be used with SEARCH::FirstMatch or SEARCH::LastMatch.
- If neither SEARCH::FirstMatch nor SEARCH::LastMatch is specified:
 - linear search returns the first match
 - binary search may return any of the existing matches

3.5 Important Notes

If SEARCH::SortAscending or SEARCH::SortDescending is specified, the searched range must not contain formulas that break the sort order during recalculation. In such cases, circular references are not reported for cells other than the result cell.

Binary searches enabled by SortAscending or SortDescending are typically tens or hundreds of times faster than plain linear searches.

3.6 AutoSort

If SEARCH::AutoSort is specified, GS-Calc creates and maintains sort indices for unsorted ranges. These internal indices allow quick binary searches without requiring manual sorting. All indices are created during the first update and then updated as needed.

The speed gain depends on how many vLookUp() formulas are present, how often ranges are reused, and how large the searched ranges are. For very large worksheets, performance can improve by hundreds of times.

SEARCH::AutoSort must be used together with SEARCH::SortAscending or SEARCH::SortDescending. If the data is already partially sorted, use the sort order flag that best matches that partial sorting.

3.7 MixedData

If SEARCH::MixedData is specified, the searched range is assumed to contain both numeric and text cells:

- If **v** is text, numeric values are internally converted to text before comparison.
- If **v** is numeric, text strings representing numbers are converted to numbers.

If sorting flags are used with MixedData, the range must be sorted using a mixed text/numeric method. With AutoSort, GS-Calc handles this internally, but it may significantly slow down sorting. Otherwise, use the options in **Tools** → **Sort Cell Range**.

3.8 Numeric Codes

For workbooks containing extremely large numbers of vLookup() functions, you can use the numeric sum of flags instead of individual names to specify the **type** argument.

If no match is found, vLookup() returns the **#N/A!** error value.

4. Examples

=vLookup(2, {1, 2, 3; "a", "b", "c"}, 2, 0) → **"b"**

=vLookup(2.5, {1, 2, 3; "a", "b", "c"}, 2, -1) → **"c"**

=vLookup(2.5, {1, 2, 3; "a", "b", "c"}, 2, 1) → **"b"**

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

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

=vLookup("bc\d", {"abc", 1; "abcde", 2; "abc10", 3}, 2, SEARCH::Regex) → **3**

5. Inserting vLookup() via the Menu

You can insert the **vLookup()** function with two clicks using:

Insert → **VLOOKUP()**

The "Insert VLOOKUP()" dialog determines optimal parameters, presets all options, and automatically creates the formula.

×

Insert VLOOKUP()

Source cell range:

'Pivot tables\orders'!A1:G44

Insert VLOOKUP() in:

B258

Lookup value:

Returned column:

1

(A:A)

☒ Standard option:

0 - exact match, linear searching

Returned occurrence:

1

Start/end search pos:

1

☐ SEARCH::MatchNotGreater
☐ SEARCH::SortAscending (binary search)

☐ SEARCH::MatchNotSmaller
☐ SEARCH::SortDescending (binary search)

☐ SEARCH::FirstMatch
☐ SEARCH::AutoSort (binary search)

☐ SEARCH::LastMatch
☐ SEARCH::SortIndex (binary search)

☐ SEARCH::RegEx
☐ SEARCH::CaseSensitive

☐ SEARCH::Pattern
☐ SEARCH::IgnorePunctuation

☐ SEARCH::MixedData
☐ SEARCH::NeutralSortOrder

=vLookUp(, 'Pivot tables\orders'!A1:G44, 1, 0)

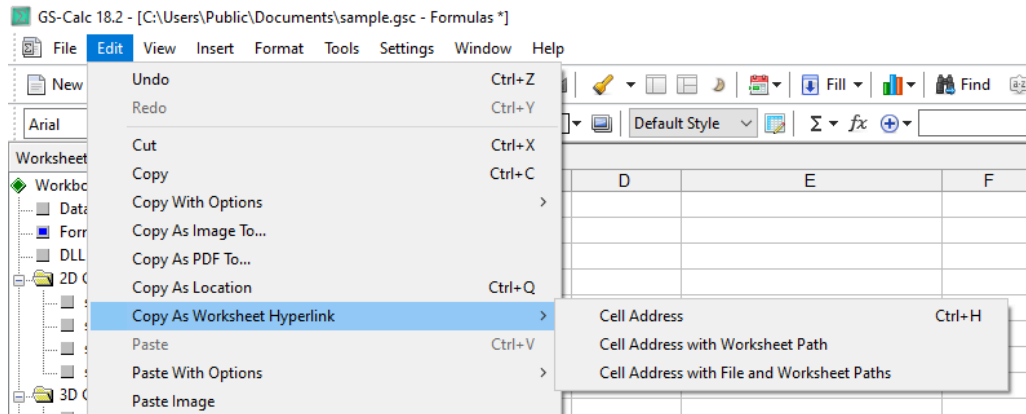
☐ Insert only values returned by formula

OK

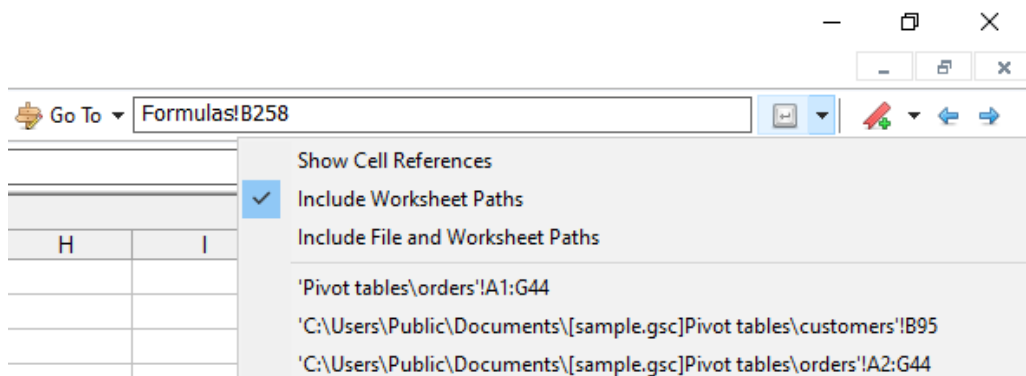
Cancel

The source range 16-element list is stored globally in the settings file. Subsequent source ranges are added in a circular manner using:

Copy as Location (menu command)



Enter (toolbar button



hLookup() Function

1. Function Syntax

hLookup(v, array, n, [type])

hLookup(v, array, n, [type], [startIndex], [occurrence])

The **hLookup()** function searches the top row of **array** for the value **v**. If a match is found, the function returns the value from the same column and the **n-th** row of the array.

2. Extended Variant: startIndex and occurrence

The extended variant adds two parameters:

startIndex — Positive values specify where searching begins (1 = top-left cell, 2 = next cell, etc.). Negative values specify where searching ends (-1 = bottom-right cell, -2 = preceding cell, etc.).

occurrence — Specifies which occurrence of the matching value to return. Positive values count matches top-down (1 = first match, 2 = second match, etc.). Negative values count matches bottom-up (-1 = last match, -2 = second-to-last, etc.). Occurrences are counted relative to **startIndex**.

3. Search Type

The **type** argument determines how the search is performed. It may be one of the basic modes (**0**, **1**, **-1**) or a sum of **SEARCH::** flags.

3.1 Basic Search Modes

- **0 — Exact match (linear search)**

Performs a linear search for an exact match. The search value **v** may contain wildcard patterns:

- **?** — matches any single character
- ***** — matches any sequence of characters (including empty)

To search for literal **?** or *****, prefix them with a tilde:

- **~?** — literal question mark
- **~*** — literal asterisk

Pattern matching is only available in mode **0**.

- **1 — Largest value $\leq v$ (ascending)**

If no exact match is found, returns the largest value that is not greater than **v**. No wildcard matching is performed. The searched range must be sorted in ascending order.

- **-1 — Smallest value $\geq v$ (descending)**

If no exact match is found, returns the smallest value that is not smaller than **v**. No wildcard matching is performed. The searched range must be sorted in descending order.

3.2 SEARCH:: Flags

Instead of using **0**, **1**, or **-1**, you may specify a sum of **SEARCH::** flags:

- **SEARCH::MatchNotGreater (2)** — largest value $\leq v$
- **SEARCH::MatchNotSmaller (4)** — smallest value $\geq v$
- **SEARCH::SortAscending (8)** — binary search, ascending order
- **SEARCH::SortDescending (16)** — binary search, descending order
- **SEARCH::CaseSensitive (128)** — case-sensitive comparison
- **SEARCH::FirstMatch (256)** — return first match
- **SEARCH::LastMatch (512)** — return last match
- **SEARCH::MixedData (2048)** — range contains both text and numbers
- **SEARCH::RegEx (8192)** — **v** is a regular expression
- **SEARCH::IgnorePunctuation (16384)** — ignore certain punctuation marks

- **SEARCH::NeutralSortOrder (32768)** — language-neutral comparison
- **SEARCH::Pattern (65536)** — v is a wildcard pattern (? and *)

3.3 Pattern Matching vs. Regular Expressions

- **SEARCH::Pattern**
Uses wildcard matching with ? and *. Cannot be combined with fast binary searching (SortAscending / SortDescending).
- **SEARCH::RegEx**
Uses regular expressions. Cannot be combined with:
 - SEARCH::MatchNotGreater
 - SEARCH::MatchNotSmaller
 - SEARCH::StringSort
 - SEARCH::CaseSensitive
 - SEARCH::SortAscending
 - SEARCH::SortDescending

3.4 Type Equivalents

1 = SEARCH::SortAscending + SEARCH::MatchNotGreater

-1 = SEARCH::SortDescending + SEARCH::MatchNotSmaller

0 = (0)

If **type** is omitted, it defaults to **1**.

4. Flag Compatibility Rules

SEARCH::Pattern and SEARCH::RegEx cannot be used with:

- SEARCH::MatchNotGreater
- SEARCH::MatchNotSmaller
- SEARCH::StringSort
- SEARCH::CaseSensitive
- SEARCH::SortAscending
- SEARCH::SortDescending

SEARCH::MatchNotGreater and SEARCH::MatchNotSmaller cannot be used with SEARCH::FirstMatch or SEARCH::LastMatch.

If neither SEARCH::FirstMatch nor SEARCH::LastMatch is specified, linear search returns the first match, while binary search may return any match.

5. Sorting Requirements

If SEARCH::SortAscending or SEARCH::SortDescending is used:

- The searched range must be sorted accordingly.
- The range should not contain formulas that break the sort order during recalculation.
- No circular reference warnings will be reported for cells other than the result cell.

Binary searches are typically tens or hundreds of times faster than linear searches.

6. Mixed Data

If SEARCH::MixedData is specified:

- The range may contain both numbers and text.
- If v is text, all values are converted to text before comparison.
- If v is numeric, text values representing numbers are converted to numbers.
- If sorting flags are used, the range must be sorted using mixed numeric/text ordering.

7. Performance Tip

If a workbook contains many hLookup() calls, you may use the numeric sum of flags instead of individual names for faster evaluation.

8. Error Handling

If no match is found, hLookup() returns #N/A!.

9. 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("bcd", {"abc", "abcde", "abc10"; 1, 2, 3}, 2, SEARCH::RegEx,,)

returns 3

10. Insert hLookup() Automatically

You can insert hLookup() with two clicks using **Insert** → **HLOOKUP()**. The dialog box determines optimal parameters, presets options, and builds the formula automatically.

Insert HLOOKUP()

Source cell range: 'Pivot tables\orders'!A1:G44

Insert HLOOKUP() in: B258

Lookup value:

Returned row: 1

☒ Standard option: 0 - exact match, linear searching

Returned occurrence: 1

Start/end search pos: 1

☐ SEARCH::MatchNotGreater ☐ SEARCH::SortAscending (binary search)
☐ SEARCH::MatchNotSmaller ☐ SEARCH::SortDescending (binary search)
☐ SEARCH::FirstMatch ☐ SEARCH::AutoSort (binary search)
☐ SEARCH::LastMatch ☐ SEARCH::SortIndex (binary search)
☐ SEARCH::RegEx ☐ SEARCH::CaseSensitive
☐ SEARCH::Pattern ☐ SEARCH::IgnorePunctuation
☐ SEARCH::MixedData ☐ SEARCH::NeutralSortOrder

=hLookup(, 'Pivot tables\orders'!A1:G44, 1, 0)

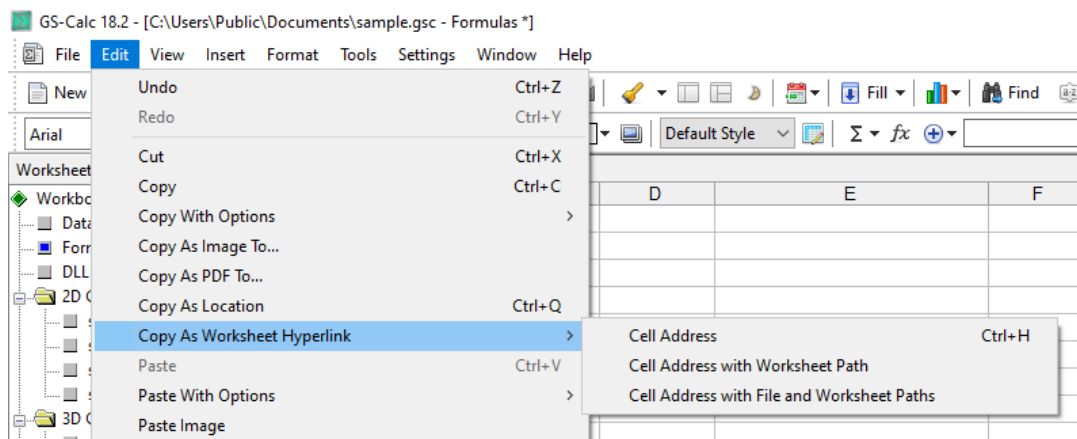
☐ Insert only values returned by formula

OK Cancel

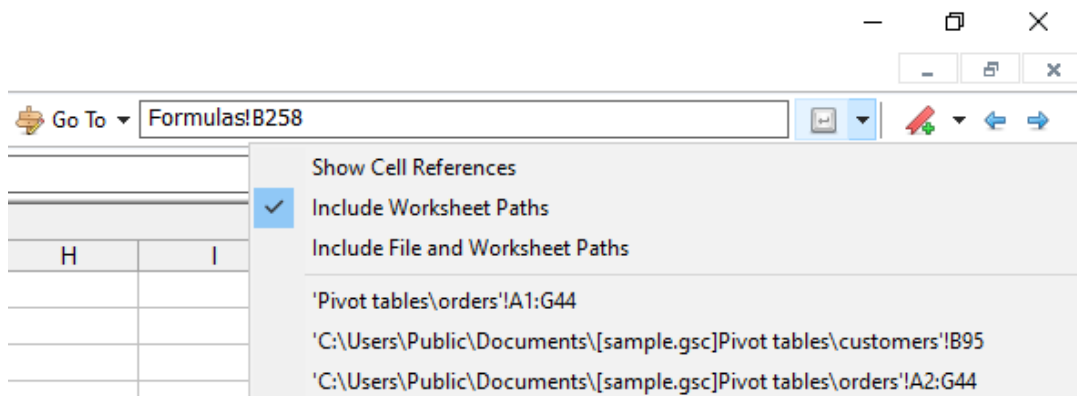
11. Source Range List

The 16-element source range list is global and stored in the settings file. New ranges are added in a circular manner using:

“Copy as Location”



The “Enter” toolbar button



match() Function

1. Function Syntax

match(v, array, [options])

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

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

2. Extended Variant: startIndex and occurrence

startIndex — Positive values specify where searching begins (1 = top-left cell, 2 = next cell, etc.). Negative values specify where searching ends (–1 = bottom-right cell, –2 = preceding

cell, etc.).

occurrence — Specifies which occurrence of the matching value to return. Positive values count matches top-down (1 = first match, 2 = second match, etc.). Negative values count matches bottom-up (-1 = last match, -2 = second-to-last, etc.). Occurrences are counted relative to **startIndex**.

3. Search Options

The **options** argument determines how searching is performed. It may be one of the basic modes (0, 1, -1) or a sum of **SEARCH::** flags.

3.1 Basic Search Modes

- **0 — Exact match (linear search)**

Performs a linear search for an exact match. The search value **v** may contain wildcard patterns:

- **?** — matches any single character
- ***** — matches any sequence of characters (including empty)

To search for literal **?** or *****, prefix them with a tilde:

- **~?** — literal question mark
- **~*** — literal asterisk

Pattern matching is only available in mode **0**.

- **1 — Largest value $\leq v$ (ascending)**

If no exact match is found, returns the largest value that is not greater than **v**. No wildcard matching is performed. The searched range must be sorted in ascending order.

- **-1 — Smallest value $\geq v$ (descending)**

If no exact match is found, returns the smallest value that is not smaller than **v**. No wildcard matching is performed. The searched range must be sorted in descending order.

3.2 SEARCH:: Flags

Instead of using 0, 1, or -1, you may specify a sum of **SEARCH::** flags:

- **SEARCH::MatchNotGreater (2)** — largest value $\leq v$
- **SEARCH::MatchNotSmaller (4)** — smallest value $\geq v$
- **SEARCH::SortAscending (8)** — binary search, ascending order (range must be a one-row or one-column vector)
- **SEARCH::SortDescending (16)** — binary search, descending order (range must be a one-row or one-column vector)
- **SEARCH::CaseSensitive (128)** — case-sensitive comparison
- **SEARCH::FirstMatch (256)** — return first match
- **SEARCH::LastMatch (512)** — return last match
- **SEARCH::AutoSort (1024)** — automatically sort the range internally during the first update
- **SEARCH::MixedData (2048)** — range contains both text and numbers

- **SEARCH::SortIndex (4096)** — return index relative to internally sorted range (requires AutoSort)
- **SEARCH::RegEx (8192)** — v is a regular expression
- **SEARCH::IgnorePunctuation (16384)** — ignore certain punctuation marks
- **SEARCH::NeutralSortOrder (32768)** — language-neutral comparison
- **SEARCH::Pattern (65536)** — v is a wildcard pattern (? and *); cannot be used with binary searching

3.3 Pattern Matching vs. Regular Expressions

- **SEARCH::Pattern**
Uses wildcard matching with ? and *. Cannot be combined with fast binary searching (SortAscending / SortDescending).
- **SEARCH::RegEx**
Uses regular expressions. Cannot be combined with:
 - SEARCH::MatchNotGreater
 - SEARCH::MatchNotSmaller
 - SEARCH::StringSort
 - SEARCH::CaseSensitive
 - SEARCH::SortAscending
 - SEARCH::SortDescending

3.4 Type Equivalents

1 = SEARCH::SortAscending + SEARCH::MatchNotGreater

-1 = SEARCH::SortDescending + SEARCH::MatchNotSmaller

0 = (0)

If **options** is omitted, it defaults to **1**.

4. Compatibility Rules

SEARCH::Pattern and SEARCH::RegEx cannot be used with:

- SEARCH::MatchNotGreater
- SEARCH::MatchNotSmaller
- SEARCH::StringSort
- SEARCH::CaseSensitive
- SEARCH::SortAscending
- SEARCH::SortDescending

SEARCH::MatchNotGreater and SEARCH::MatchNotSmaller cannot be used with SEARCH::FirstMatch or SEARCH::LastMatch.

If neither SEARCH::FirstMatch nor SEARCH::LastMatch is specified, linear search returns the first match, while binary search may return any match.

5. Sorting Requirements

If `SEARCH::SortAscending` or `SEARCH::SortDescending` is specified, the searched range must not contain formulas that break the sort order during recalculation. No circular reference warnings will be reported for cells other than the result cell.

Binary searches are typically tens or hundreds of times faster than linear searches.

6. AutoSort

If `SEARCH::AutoSort` is specified, the program creates and maintains internal sort indices for unsorted ranges. This enables fast binary searching without requiring the user to sort the data manually.

Sort indices are created during the first update after opening the workbook and updated as needed. Performance improvements can be dramatic for large worksheets.

`SEARCH::AutoSort` must be used together with `SEARCH::SortAscending` or `SEARCH::SortDescending`.

7. Mixed Data

If `SEARCH::MixedData` is specified:

- If `v` is text, all values are converted to text before comparison.
- If `v` is numeric, text values representing numbers are converted to numbers.
- If sorting flags are used, the range must be sorted using mixed numeric/text ordering.
- If `AutoSort` is used, internal sorting is performed automatically.

8. Performance Tip

For extremely large numbers of `match()` calls, you may use the numeric sum of flags instead of individual names for faster evaluation.

9. Error Handling

If the searched value is not found, `match()` returns `#N/A!`.

10. 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

11. Insert match() Automatically

You can insert match() with two clicks using **Insert** → **MATCH()**. The dialog box determines optimal parameters, presets options, and builds the formula automatically.

Insert MATCH()

Source cell range:
'Pivot tables\orders'!A1:G44

Insert MATCH() in:
B258

Lookup value:

☒ Standard option:
0 - exact match, linear searching

Returned occurrence:
1

Start/end search pos:
1

☐ SEARCH::MatchNotGreater
☐ SEARCH::SortAscending (binary search)

☐ SEARCH::MatchNotSmaller
☐ SEARCH::SortDescending (binary search)

☐ SEARCH::FirstMatch
☐ SEARCH::AutoSort (binary search)

☐ SEARCH::LastMatch
☐ SEARCH::SortIndex (binary search)

☐ SEARCH::RegEx
☐ SEARCH::CaseSensitive

☐ SEARCH::Pattern
☐ SEARCH::IgnorePunctuation

☐ SEARCH::MixedData
☐ SEARCH::NeutralSortOrder

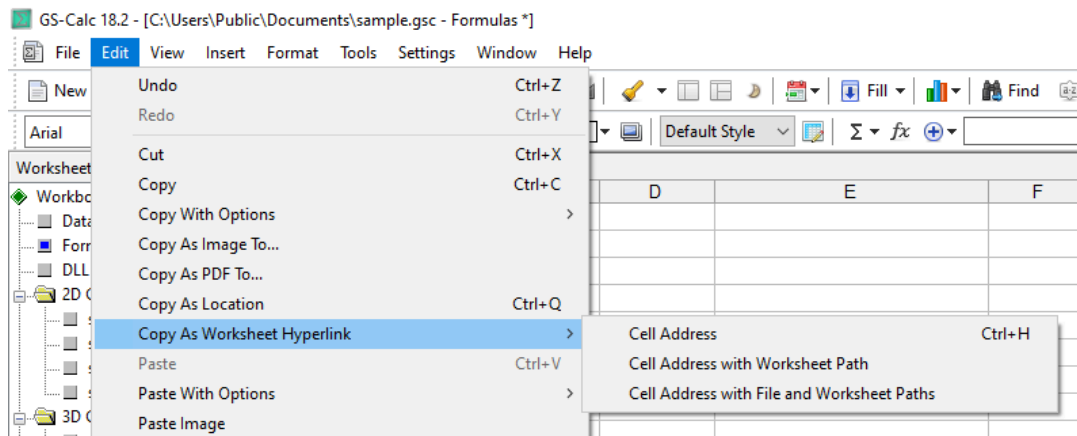
=Match(, 'Pivot tables\orders'!A1:G44, 0)

☐ Insert only values returned by formula
OK
Cancel

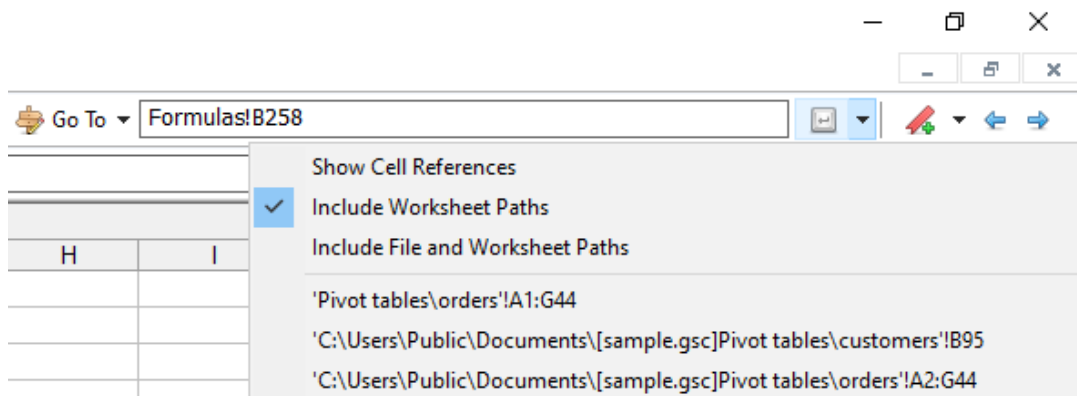
12. Source Range List

The 16-element source range list is global and stored in the settings file. New ranges are added in a circular manner using:

“Copy as Location”



The “Enter” toolbar button



Unique() Function

1. Overview

The **Unique()** function searches a column vector (array or range) and returns unique values or indices depending on the selected mode.

2. Function Syntax

Unique(vector, [type])

2.1 Parameters

- **vector** — a one-column array or range to analyze.
- **type** — optional mode selector:

- **0** — return a vector of row indices of unique values (first occurrences).
- **1** — return a vector of unique values.
- **2** — return a two-column array:
 - column 1 — unique values
 - column 2 — number of occurrences

If **type** is omitted, it defaults to **1**.

3. Examples

3.1 Return unique values

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

returns **{1; 2; 3; 5}**

3.2 Return indices of first occurrences

```
=unique({"a"; "b"; "cd"; "a"; "cd"; "b"; "d"}, 0)
```

returns **{1; 2; 3; 5}**

3.3 Return unique values with counts

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

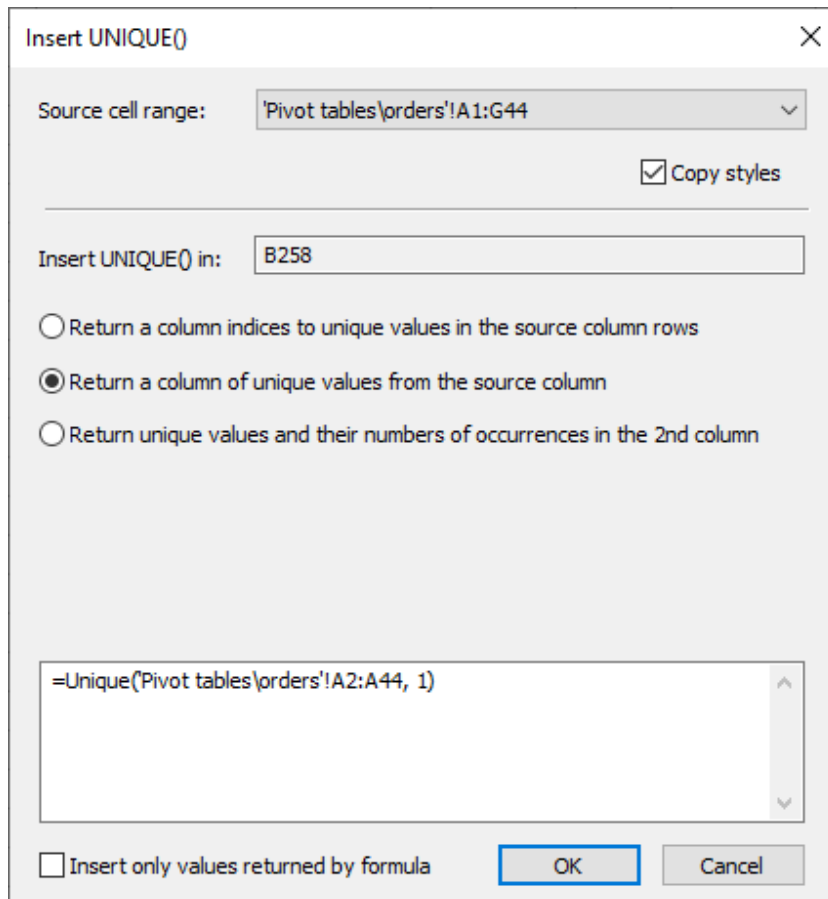
returns **{"a", 2; "b", 2; "cd", 2; "d", 1}**

4. Inserting Unique() via the Menu

You can insert the **Unique()** function with two clicks using:

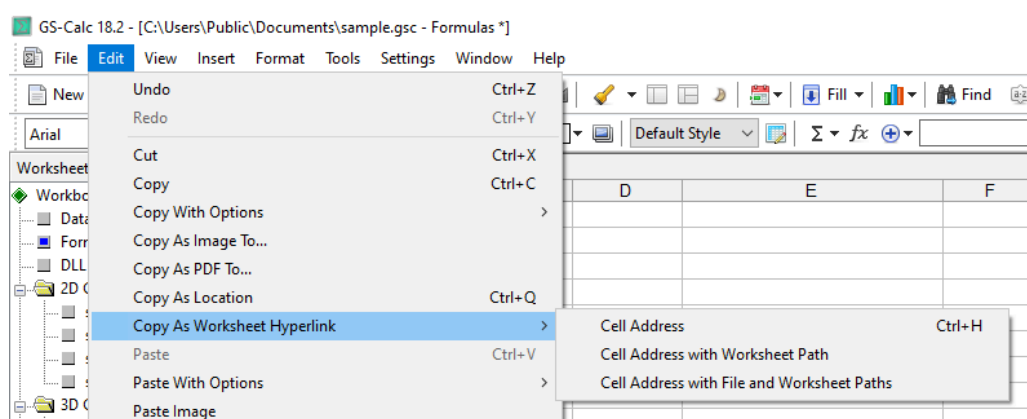
Insert → UNIQUE()

The “Insert UNIQUE()” dialog automatically determines optimal parameters and can copy styles from the selected source range.

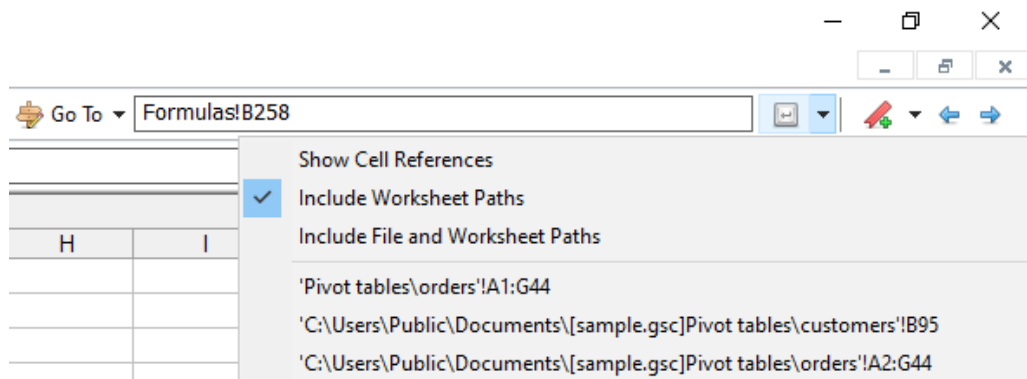


The source range 16-element list is stored globally in the settings file. New source ranges are added in a circular manner using:

Copy as Location (menu command)



Enter (toolbar button)



Sort() function

1. Overview

The **Sort()** function sorts a given array or cell range according to one or more (column, sort order) pairs and returns a vector of row indices describing the sorted order. This index vector can then be used with the **INDEX()** function to obtain the final sorted array or range.

2. Function syntax

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

2.1 Parameters

- **array** — the source array or cell range to be sorted.
- **column1, column2, column3** — 1-based column indices specifying the relative column positions within the source range used for sorting.
- **order1, order2, order3** — sort order for the corresponding column:
 - 0 — descending order
 - 1 — ascending order

2.2 Result

The function returns a column vector of row indices. Each element of this vector is the 1-based index of a row in the original array, in the order determined by the specified sort criteria.

To obtain the sorted data itself, use the returned index vector with the **INDEX()** function.

3. Examples

3.1 Sorting a single-column numeric array

Ascending order:

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

returns **{2; 4; 1; 5; 3}**, which means that the sorted order of rows is: row 2, row 4, row 1, row 5, row 3.

Descending order:

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

returns **{3; 5; 1; 4; 2}**, corresponding to the order: row 3, row 5, row 1, row 4, row 2.

3.2 Sorting a two-column mixed array

Consider the array:

{2, "b"; 2, "a"; 3, "d"; 3, "c"; 1, "e"}

Sort by first column ascending, then second column ascending:

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

returns **{5; 2; 1; 4; 3}**. This means the sorted row order is: row 5, row 2, row 1, row 4, row 3.

Using INDEX() to obtain the sorted array:

=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 index vector produced by the Sort() function above. In practice, you would typically use INDEX() with both row and column indices to reconstruct the fully sorted table.

4. Inserting Sort() via the menu

You can insert the **Sort()** function with just two clicks using the **Insert > SORT()** command. The "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.

Insert SORT()

Source cell range:

'Pivot tables\orders'!A1:G44

☐ Copy styles

☒ 1. Sort Key:

3

Ascending order

☐ 2. Sort Key:

1

Ascending order

☐ 3. Sort Key:

1

Ascending order

☐ 4. Sort Key:

1

Ascending order

☐ 5. Sort Key:

1

Ascending order

☐ 6. Sort Key:

1

Ascending order

☐ 7. Sort Key:

1

Ascending order

☐ 8. Sort Key:

1

Ascending order

☐ 9. Sort Key:

1

Ascending order

☐ 10. Sort Key:

1

Ascending order

☐ Return row indices
 ☒ Return values from column:

3

(C:C)

=Index('Pivot tables\orders'!A1:G44, Sort('Pivot tables\orders'!A1:G44, 3, 1), 3)

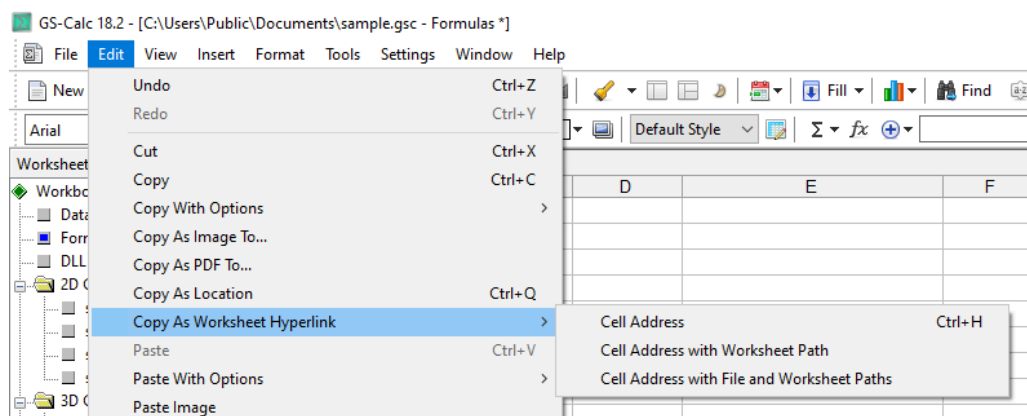
☐ Insert only values returned by formula

OK

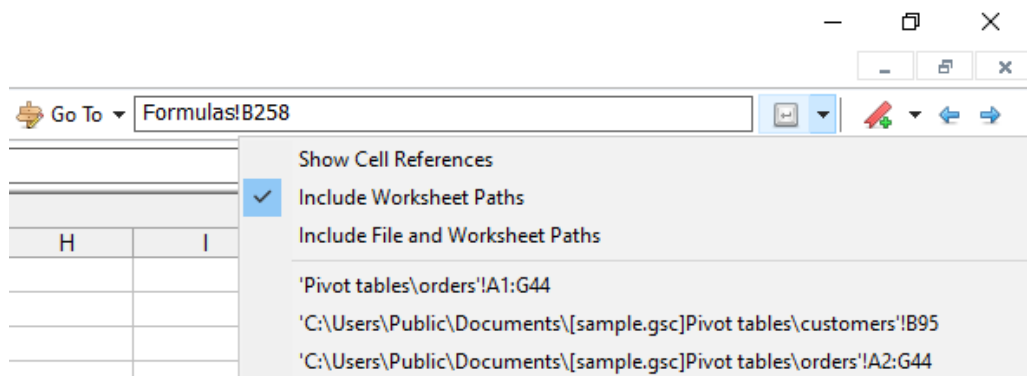
Cancel

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



Output Functions

1. imagelf(if_condition, path_true, path_false, type)

Displays an image in a cell (overflowing cell boundaries if necessary).

if_condition:

If non-zero, **path_true** is displayed; otherwise **path_false**.

path_true / path_false:

Image file paths or names.

- If **type = 1**: paths refer to external disk files (e.g., **e:\image_file1.png**).
- If **type = 0**: names refer to images inserted into the workbook using **Insert > Image**.

The list of inserted images can be managed via the **Set Cell Background Image** toolbar button or the **Cell Style Properties** dialog.

type: Determines how paths are interpreted.

Images are drawn immediately after formula evaluation is complete.

```
=imagelf(P251 > 0, "e:\image_file1.png", "e:\image_file2.png", 1)
```

```
=imagelf(S10:S120 = 0, "folder1\music_file1.jpg", "image_file2.gif", 0)
```

2. messagelf(message, type, if_condition, reference-as-text)

Displays a message box with **OK** and **Cancel**.

message: Text to display.

type:

- 1 — information icon
- 2 — warning icon
- 3 — question icon

if_condition:

Message appears only if the condition evaluates to non-zero.

reference (as text):

Optional full worksheet path. If provided and you click **OK**, GS-Calc jumps to the referenced cell/range.

Messages appear immediately after formula evaluation is complete. You can use **if_condition = 1** to signal the end of updating.

=messageIf("need to inspect this cell", 1, isError(folder2\sheet2!P251), "folder2\sheet2!P251")

=messageIf("need to inspect this cells", 1, isError(S120), "S121:S200")

3. playSoundIf(path, type, if_condition, loops, volume)

Plays MP3 and WAV files.

path:

File path.

- If **type = 1**: external disk file (e.g., **e:\music_file1.mp3**).
- If **type = 0**: full worksheet path of a file inserted via **Insert > File**.

type: Determines how the path is interpreted.

if_condition: File plays only if non-zero.

loops:

Number of times to play. **-1** = endless loop.

volume:

Value from 0 to 1000.

Files play immediately after formula evaluation is complete. Use **if_condition = 1** to signal the end of updating.

If multiple files are queued, they play sequentially.

To stop playback, press **ESC** or use **Tools > Cancel Update**.

=playSoundIf("e:\music_file1.mp3", 1, true, 1, 40)

=playSoundIf("folder1\music_file1.mp3", 0, true, 100, 80)

4. setFormatIf(reference, action, if_condition, format_true, format_false)

See: [Conditional cell formatting](#)

Solver Functions

1. Overview

GS-Calc provides several solver functions for linear programming, binary/integer programming, quadratic programming, Monte-Carlo minimization, simplex minimization, and root finding. These functions allow you to solve constrained optimization problems directly inside worksheets.

All solver functions return either:

- the number of optimum vectors found
- a specific optimum vector
- a scalar result (minimum value, number of steps, root, etc.)

All solver functions use non-negative variables. If a problem requires negative variables, each must be replaced by a pair of non-negative variables.

2. LProg — Linear Programming

LProg(A, s, b, c, vector, [epsilon], [BigM])

2.1 Purpose

Finds one or more solutions for a linear programming problem with **m** constraints of the form:

$$(1) \sum_{j=1..n} (x[j] \cdot a[i,j]) \leq b[i]$$

$$(2) \sum_{j=1..n} (x[j] \cdot a[i,j]) = b[i]$$

$$(3) \sum_{j=1..n} (x[j] \cdot a[i,j]) \geq b[i]$$

and a maximized objective function:

$$\sum_{j=1..n} (x[j] \cdot c[j])$$

2.2 Arguments

- **A** — m×n matrix of coefficients a[i,j]
- **b** — m-element column vector
- **c** — n-element column vector
- **s** — m-element column vector specifying constraint types:

- $s[i] = 1 \rightarrow \leq$
- $s[i] = 0 \rightarrow =$
- $s[i] = -1 \rightarrow \geq$
- **vector** — return mode:
 - 0 → number of optimum vectors found
 - $i > 0 \rightarrow i$ -th optimum vector
- **epsilon** — precision threshold; coefficients with absolute value $< \text{epsilon}$ are treated as 0 (default: $1e-8$)
- **BigM** — large artificial-variable coefficient used in the Big-M method; if omitted, an appropriate Big-M value is chosen automatically

2.3 Behavior

- All returned $x[i]$ values are non-negative.
- If the model allows negative variables, each such variable must be represented by a pair of non-negative variables.
- To minimize the objective function, negate it and use the default maximization.
- If the objective function is unconstrained, LProg returns **#N/A!**.
- If the constraints are inconsistent and no solution exists, LProg returns **#NULL!**.

2.4 Example

Initial constraints:

$$2 \cdot x_1 + x_2 + 2 \cdot x_3 \leq 20$$

$$3 \cdot x_1 - x_2 + x_3 = 10$$

Objective function:

$$3 \cdot x_1 + x_2 - x_3 \rightarrow \max$$

Formula:

`=LProg({2, 1, 1, 1, 1, 3}, {1; 0}, {5; 10}, {4; 2; 2}, 1,,)`

returns **{1; 0; 3}**

3. LProgBin — Binary Linear Programming

LProgBin(A, s, b, c, bin, vector, [epsilon], [BigM])

3.1 Purpose

Finds one or more solutions for a linear programming problem with constraints of the same form as LProg, where some or all variables $x[i]$ are required to be binary (0 or 1).

3.2 Arguments

- **A, b, c, s, epsilon, BigM** — as in LProg.
- **bin** — n-element column vector defining variable types:
 - $\text{bin}[i] = 1 \rightarrow x[i] \in \{0, 1\}$
 - $\text{bin}[i] = 0 \rightarrow x[i] \geq 0$
- **vector** — return mode:
 - 0 → number of optimum vectors found
 - $i > 0 \rightarrow i\text{-th optimum vector}$

3.3 Behavior

- All returned $x[i]$ values are non-negative; binary variables are restricted to 0 or 1.
- To minimize the objective function, negate it and use the maximization procedure.
- If the objective function is unconstrained, LProgBin returns **#N/A!**.
- If the constraints are inconsistent and no solution exists, LProgBin returns **#NULL!**.

3.4 Example

Initial constraints:

$$2.5 \cdot x_1 + 1.5 \cdot x_2 + x_3 \leq 15$$

$$x_1 + 1.5 \cdot x_2 + 3 \cdot x_3 = 10$$

Objective function:

$$x_1 + x_2 + 2 \cdot x_3 \rightarrow \max$$

Formula:

=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}

4. LProgInt — Integer Linear Programming

LProgInt(A, s, b, c, int, vector, [epsilon], [BigM])

4.1 Purpose

Finds one or more solutions for a linear programming problem with constraints of the same form as LProg, where some or all variables $x[i]$ are required to be integers.

4.2 Arguments

- **A, b, c, s, epsilon, BigM** — as in LProg.
- **int** — n-element column vector defining variable types:
 - $\text{int}[i] = 1 \rightarrow x[i]$ must be integer
 - $\text{int}[i] = 0 \rightarrow x[i] \geq 0$
- **vector** — return mode:
 - 0 → number of optimum vectors found
 - $i > 0 \rightarrow i$ -th optimum vector

4.3 Behavior

- All returned $x[i]$ values are non-negative; selected variables must be integers.
- To minimize the objective function, negate it and use the maximization procedure.
- If the objective function is unconstrained, LProgInt returns **#N/A!**.
- If the constraints are inconsistent and no solution exists, LProgInt returns **#NULL!**.

4.4 Example

Initial constraints:

$$2.5 \cdot x_1 + 1.5 \cdot x_2 + x_3 \leq 15$$

$$x_1 + 1.5 \cdot x_2 + 3 \cdot x_3 = 10$$

Objective function:

$$x_1 + x_2 + 2 \cdot x_3 \rightarrow \max$$

Formula:

=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}

5. QProg — Quadratic Programming

QProg(A, s, b, p, C, vector, [epsilon])

5.1 Purpose

Finds one or more solutions for a quadratic programming problem with **m** constraints:

$$(1) \sum_{j=1..n} (x[j] \cdot a[i,j]) \leq b[i]$$

$$(2) \sum_{j=1..n} (x[j] \cdot a[i,j]) = b[i]$$

$$(3) \sum_{j=1..n} (x[j] \cdot a[i,j]) \geq b[i]$$

and a maximized objective function:

$$\mathbf{p}^T \cdot \mathbf{x} - \mathbf{x}^T \cdot \mathbf{C} \cdot \mathbf{x}$$

where $\mathbf{p}$ is an n-element column vector and $\mathbf{C}$ is a positive-definite $n \times n$ matrix representing a quadratic form.

5.2 Arguments

- **A** — $m \times n$ matrix of coefficients $a[i,j]$
- **b** — m-element column vector
- **p** — n-element column vector
- **C** — $n \times n$ positive-definite matrix
- **s** — m-element column vector specifying constraint types:
 - $s[i] = 1 \rightarrow \leq$
 - $s[i] = 0 \rightarrow =$
 - $s[i] = -1 \rightarrow \geq$
- **vector** — return mode:
 - 0 → number of optimum vectors found
 - $i > 0 \rightarrow i$ -th optimum vector
- **epsilon** — precision threshold (default: $1e-8$)

5.3 Behavior

- All $x[i]$ are non-negative.
- If negative variables are needed, each must be represented by a pair of non-negative variables.
- To minimize the objective function, change its sign and use the maximization procedure.
- If the objective function is unconstrained, QProg returns **#N/A!**.
- If the constraints are inconsistent and no solution exists, QProg returns **#NULL!**.

5.4 Example

Constraints:

$$x_1 + 2 \cdot x_2 \leq 10$$

$$x_1 + x_2 \leq 9$$

$$x_1 \geq 0, x_2 \geq 0$$

Objective function:

$$f(x_1, x_2) = 10 \cdot x_1 + 25 \cdot x_2 - 10 \cdot x_1^2 - x_2^2 - 4 \cdot x_1 \cdot x_2 \rightarrow \max$$

Data:

$$A = \{1, 2; 1, 1\}$$

$$s = \{1; 1\}$$

$$b = \{10; 9\}$$

p = {10; 25}
C = {10, 2; 2, 1}

Formula:

=QProg({1, 2; 1, 1}, {1; 1}, {10; 9}, {10; 25}, {10, 2; 2, 1}, 1,)

returns {0; 5}

6. minMC — Monte-Carlo Minimization

minMC(f(x), x, v, from, to, points)

6.1 Purpose

Performs n-dimensional minimization of a given function using the Monte-Carlo method. The function generates a number of random points in n dimensions and returns the point for which f(x) is smallest. This method can be used to obtain an initial vector for **minSimplex()**.

6.2 Arguments

- **f(x)** — reference to a cell containing a numeric formula with n variables.
- **x** — reference to a one-column vector of n cells (numbers or numeric formulas) representing the variables in f(x).
- **v** — one-column vector indicating how to use each variable:
 - v[i] = 0 → i-th variable has a fixed value and is not modified
 - v[i] = 1 → i-th variable is modified
- **from, to** — search limits for all x[i].
- **points** — number of sample n-dimensional points to generate.

6.3 Notes and Errors

- Since points are generated randomly for each calculation, results change after each update.
- If f(x) is not a reference to a cell containing a valid numeric formula, or x is not a reference to n non-empty numeric cells, minMC returns **#REF!**.

6.4 Example (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}

6.5 Example (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}

7. minSimplex — Downhill Simplex Minimization

minSimplex(f(x), x, v, [start], [epsilon], [max_steps], result)

7.1 Purpose

Performs n-dimensional minimization of a given function using the downhill simplex method.

7.2 Arguments

- **f(x)** — reference to a cell containing a numeric formula with n variables.
- **x** — reference to a one-column vector of n cells (numbers or numeric formulas) representing the variables in f(x). Initial values of x can be obtained with **minMC()**.
- **v** — one-column vector indicating how to use each variable:
 - v[i] = 0 → i-th variable has a fixed value and is not modified
 - v[i] = 1 → i-th variable will be modified
- **start** — initial simplex size; if 0 or omitted, default is 1e-5.
- **epsilon** — precision used to evaluate changes in the function value; if 0 or omitted, default is 1e-25.
- **max_steps** — maximum number of steps; default is 2000.
- **result** — return mode:
 - 1 → x vector for the found minimum
 - 2 → minimum f(x) value
 - 3 → number of steps performed

7.3 Errors

- If f(x) is not a reference to a valid numeric formula or x is not a reference to n non-empty numeric cells, minSimplex returns **#REF!**.
- If the procedure is not convergent after max_steps steps, the function returns **#NUM!**.

7.4 Example (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

7.5 Example (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

8. root — One-Dimensional Root Finding

root(f(x), x, x0, x1, [epsilon])

8.1 Purpose

Finds a root of a one-dimensional equation $f(x) = 0$ using a modified secant method. The method is convergent if $f(x)$ is continuous on $[x0, x1]$ and $f(x0) \cdot f(x1) < 0$.

8.2 Arguments

- **f(x)** — reference to a cell containing a numeric formula with one variable.
- **x** — single cell reference representing the variable.
- **x0, x1** — initial search limits such that $f(x0) \cdot f(x1) < 0$.
- **epsilon** — precision used to evaluate changes in the function value; if 0 or omitted, default is $1e-15$.

8.3 Errors

- If f(x) is not a reference to a valid numeric formula or x is not a reference to a numeric cell, root returns #REF!.

8.4 Example

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

[GS-Calc Help](#) > *Calculation Functions* > *Mathematical Functions*

Function Index

abs acos acosh asin asinh atan atan2 atanh ceiling CEILING.MATH
CEILING.PRECISE combin combin2 cos cosh degrees even exp fact factDouble
floor FLOOR.MATH FLOOR.PRECISE gcd int ISO.CEILING lcm ln log log10
mDeterm mInverse mMult mod mRound multinomial odd pi power product
productIfs quotient radians rand randBetween round roundDown roundUp
seriesSum sign sin sinh sqrt sqrtPi subTotal sum sumA sumIf sumAIfs sumIfs
sumProduct sumSq sumX2mY2 sumX2pY2 sumXmY2 tan tanh trunc

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 ($-\pi$, π). 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])

Rounds **number** up to the nearest multiple of **significance**.

If **significance** is omitted, it defaults to 1. If **significance** is negative, its absolute value is used.

If **mode** is omitted or 0, negative numbers are rounded *away from zero*. If **mode** is 1, negative numbers are rounded *toward zero*.

Examples:

CEILING.MATH(5.1, 1) returns 6

CEILING.MATH(-5.1, 1) returns -5

CEILING.MATH(-5.1, 1, 1) returns -6

CEILING.MATH(7.2, 0.5) returns 7.5

CEILING.PRECISE(number, [significance])

Rounds **number** up to the nearest multiple of **significance**

If **significance** is omitted, it defaults to 1. The sign of **significance** is ignored; its absolute value is used.

Examples:

CEILING.PRECISE(5.1) returns 6

CEILING.PRECISE(-5.1) returns -5

CEILING.PRECISE(7.2, 0.5) returns 7.5

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

degrees(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 $n(n-2)(n-4)\dots(4)(2)$.

If n is an odd number, factDouble returns the product $n(n-2)(n-4)\dots(3)(1)$.

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])

Rounds **number** down to the nearest multiple of **significance**.

If **significance** is omitted, it defaults to 1. If **significance** is negative, its absolute value is used.

If **mode** is omitted or 0, negative numbers are rounded *away from zero*. If **mode** is 1, negative numbers are rounded *toward zero*.

Examples:

FLOOR.MATH(5.1, 1) returns 5

FLOOR.MATH(-5.1, 1) returns -6

FLOOR.MATH(-5.1, 1, 1) returns -5

FLOOR.MATH(7.2, 0.5) returns 7.0

FLOOR.PRECISE(number, [significance])

Rounds **number** down to the nearest multiple of **significance**, always *toward zero*.

If **significance** is omitted, it defaults to 1. The sign of **significance** is ignored; its absolute value is used.

Examples:

FLOOR.PRECISE(5.1) returns 5

FLOOR.PRECISE(-5.1) returns -6

FLOOR.PRECISE(7.2, 0.5) returns 7.0

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)

Rounds **number** up to the nearest integer, always *away from zero*, following ISO 605 rounding rules.

Equivalent to CEILING.PRECISE(number, 1).

Examples:

ISO.CEILING(5.1) returns 6

ISO.CEILING(-5.1) returns -5

ISO.CEILING(2.0) returns 2

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 $(\text{number1} + \text{number2} + \dots)! / (\text{number1!} \text{number2!} \dots)$.

=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 mtRand 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:

$$a(1)x^n + a(2)x^{(n+m)} + a(3)x^{(n+2m)} + \dots + a(l)x^{(n+(l-1)m)}$$

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:

$$5 + 2x + 3x^3$$

=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

sumAlfs(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.

=sumAlfs({1,2;3,2}, {4,5;2,7}, 2) returns 3

=sumAlfs({1,2;3,4}, {4,5;2,7}, ">2", {4,5;2,7}, "<7") returns 3

=sumlfs({2, 4; 3, 1}, {"abcde","def";"abc","a"}, "?bc*") returns 5

=sumAlfs(\$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.

sumlfs(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

[GS-Calc Help](#) > [Calculation Functions](#) > [Statistical Functions](#)

Function Index

aveDev average averageA averagelf averageAlfs averagelfs BETA.DIST
BINOM.DIST binomDist chiDist chiInv CHISQ.DIST CHISQ.DIST.RT CHISQ.INV
CHISQ.INV.RT CHISQ.TEST chiTest chiTest2 confidence CONFIDENCE.NORM
CONFIDENCE.T correl count countA countlf countlfs covar COVARIANCE.P
COVARIANCE.S critBinom devSq EXPON.DIST exponDist F.DIST F.DIST.RT
F.INV F.INV.RT F.TEST fDist flnv fisher fisherInv forecast frequency fTest
GAMMA GAMMALN GAMMA.DIST GAMMALN.PRECISE GAUSS geoMean
geoMeanAlfs geoMeanlfs harMean harMeanAlfs harMeanlfs HYPGEOM.DIST
hypGeomDist intercept kurt large linEst LOGNORM.DIST LOGNORM.INV max
maxA maxAlfs maxlfs median medianAlfs medianlfs min minA minAlfs
minlfs mode MODE.MULT MODE.SNGL modeAlfs modelfs NORM.DIST
NORM.INV NORM.S.DIST NORM.S.INV normDist normsDist normInv normsInv
pearson percentile PERCENTILE.EXC PERCENTILE.INC percentileAlfs
percentilelfs percentRank permut PHI poisson POISSON.DIST polyPoints
polyReg prob quartile QUARTILE.EXC QUARTILE.INC rank RANK.AVG
RANK.EQ rsq skew SKEW.P slope small standardize stdev STDEV.P
STDEV.S stdevA stdevAlfs stdevlfs stdevP stdevPA stdevPAIfs stdevPIfs
stEyX T.DIST T.DIST.2T T.DIST.RT T.INV T.INV.2T T.TEST tDist timeSeries
tlnv trend trimMean tTest tTest2 var VAR.P VAR.S varA varAlfs varlfs
varP varPA varPAIfs varPIfs weibull WEIBULL.DIST Z.TEST zTest zTest2

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) → 0.83333333333333

average(v1, v2, ...)

Returns the arithmetic mean for the specified arguments. Arguments that are either text strings which cannot be converted to numbers or error codes will 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 in the calculation, use the `averageA()` function. Empty cells are ignored.

```
=average(4, 5, 6, 3, 5, 4) → 4.5
```

```
=average(4, 5, 6, 3, {1, "1"}) → 3.8
```

averageA(v1, v2, ...)

Returns the arithmetic mean for the specified arguments. Arguments that are either text strings which cannot be converted to numbers or error codes will 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) → 4.5
```

```
=averageA(4, 5, 6, 3, {1, "1"}) → 3.3(3)
```

averageIf(if_range, criteria, [data_range])

Calculates the mean value for numbers that meet the specified criteria. If the `data_range` range is omitted, cells from the `if_range` range are used. The `data_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 `=`, `>`, `>=`, `<`, `<=`, or `<>` operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD), or the `&` operator followed by a single cell address.
3. A text string optionally containing wildcard characters `?` (matching any single character) or `*` (matching any sequence of characters, including an empty string). To search for a literal `?` or `*`, place a tilde (`~`) before them.

```
=averageIf({1, 2; 3, 2}, 2) → 2
```

```
=averageIf({1, 2; 3, 2}, ">=2") → 2.33(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 in the calculation. 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 `=`, `>`, `>=`, `<`, `<=`, or `<>` operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD), or the `&` operator followed by a single cell address.
3. A text string optionally containing wildcard characters `?` or `*`. To search for a literal `?` 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")` → 3

`=averageAifs({1, 3, "6", 2}, {3, 3, 3, 0.1}, ">"&c5)` → returns the mean based on the value in cell C5.

averagelfs(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 `=`, `>`, `>=`, `<`, `<=`, or `<>` operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD), or the `&` operator followed by a single cell address.
3. A text string optionally containing wildcard characters `?` or `*`. To search for a literal `?` 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")` → 2

`=averageIfs({1, 3, "6", 2}, {3, 3, 3, 0.1}, ">"&c5)` → returns the mean based on the value in cell C5.

BETA.DIST(x, alpha, beta, cumulative, [A], [B])

Returns the beta distribution. `x` is the value between `A` and `B` at which to evaluate the function. `alpha` and `beta` are parameters of the distribution. If `cumulative` is TRUE, `BETA.DIST` returns the cumulative distribution function; if FALSE, it returns the probability density function. `A` (lower bound) and `B` (upper bound) are optional; if omitted, 0 and 1 are used respectively.

```
=BETA.DIST(0.5, 8, 10, TRUE) → 0.592735
```

BINOM.DIST(successes, trials, probability, cumulative)

Returns the individual term binomial distribution probability. `successes` is the number of successes in trials, `trials` is the number of independent trials, `probability` is the probability of success on each trial. If `cumulative` is TRUE, it returns the cumulative distribution function; if FALSE, it returns the probability mass function.

```
=BINOM.DIST(6, 10, 0.5, FALSE) → 0.205078125
```

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) → 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) → 0.9499994109077
```

chiInv(probability, f_degrees)

Returns the inverse of the `chiDist` function for a given probability.

```
=chiInv(0.9499994109077, 10) → 18.307
```

CHISQ.DIST(x, deg_freedom, cumulative)

Returns the chi-squared distribution. If `cumulative` is `TRUE`, `CHISQ.DIST` returns the cumulative distribution function; if `FALSE`, it returns the probability density function.

```
=CHISQ.DIST(0.5, 1, TRUE) → 0.520499876
```

CHISQ.DIST.RT(x, deg_freedom)

Returns the right-tailed probability of the chi-squared distribution.

```
=CHISQ.DIST.RT(18.307, 10) → 0.050000589
```

CHISQ.INV(probability, deg_freedom)

Returns the inverse of the left-tailed probability of the chi-squared distribution.

```
=CHISQ.INV(0.95, 10) → 18.30704
```

CHISQ.INV.RT(probability, deg_freedom)

Returns the inverse of the right-tailed probability of the chi-squared distribution.

```
=CHISQ.INV.RT(0.05, 10) → 18.30704
```

CHISQ.TEST(actual_range, expected_range)

Returns the test for independence. `CHISQ.TEST` returns the value from the chi-squared distribution for the statistic and the appropriate degrees of freedom.

```
=CHISQ.TEST({58, 35; 11, 25; 10, 23}, {45.35, 47.65; 17.56, 18.44; 16.09, 16.91}) → 0.999691808
```

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 $k \times l$ contingency 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}) → 0.99969180798302
```

chiTest2(observed, expected)

Returns the chi2-statistic for two discrete variables x ($x[1], \dots, x[k]$) and y ($y[1], \dots, y[l]$).

The `observed` argument specifies the $k \times l$ contingency 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}) → 16.1695750747969
```

confidence(alpha, deviation, size)

Returns the confidence interval for a population mean. The `alpha` parameter specifies the significance level (which equals $1 - \text{confidence level}$).

The `deviation` argument specifies the (known) population standard deviation. The `size` argument specifies the sample size.

```
=confidence(0.05, 2.5, 50) → 0.6929519802241
```

CONFIDENCE.NORM(alpha, standard_dev, size)

Returns the confidence interval for a population mean using a normal distribution. `alpha` is the significance level used to compute the confidence level. `standard_dev` is the population standard deviation for the data range and is assumed to be known. `size` is the sample size.

```
=CONFIDENCE.NORM(0.05, 2.5, 50) → 0.692951912
```

CONFIDENCE.T(alpha, standard_dev, size)

Returns the confidence interval for a population mean using a Student's t-distribution.

`alpha` is the significance level, `standard_dev` is the population standard deviation, and `size` is the sample size.

`=CONFIDENCE.T(0.05, 2.5, 50) → 0.710344445`

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}) → 0.99705448550158`

count(v1, v2, ...)

Counts numbers. The `v_` arguments can be any data types including arrays and references. All text strings in arrays and ranges are ignored.

`=count({1, 3, "abc"}, {"4.5", 5.6}) → 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,,}) → 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 `=`, `>`, `>=`, `<`, `<=`, or `<>` operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD), or the `&` operator followed by a single cell address.
3. A text string optionally containing wildcard characters `?` (matching any single character) or `*` (matching any sequence of characters, including an empty string). To search for a literal `?` or `*`, place a tilde (`~`) before them.
4. An error code.

`=countIf({1, 2; 3, 4}, 2) → 1`

=countIf({1, 2; 3, 4}, ">2") → 2

=countIf({"abcde", "def"; "abc", "a"}, "?bc*") → 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 the value in cell C101.

=countIf({"ABcde", "def"; "Bc", "a"}, ">=bc") → 2

=countIf({1, 2; "a", #N/A!; 3, #N/A!}, #N/A!) → 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 `=`, `>`, `>=`, `<`, `<=`, or `<>` operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD), or the `&` operator followed by a single cell address.
3. A text string optionally containing wildcard characters `?` (any single character) or `*` (any sequence of characters). To search for a literal `?` or `*`, place a tilde (`~`) before them.
4. An error code.

=countifs({1, 2; 3, 2}, 2, {1, 4; 3, 5}, 4) → 1

=countifs({1, 2; 3, 2}, ">=2", {1, 4; 3, 5}, "<=4") → 2

=countifs({"abcde", "def"; "abc", "a"}, "?bc*") → 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 or equal to the value in cell F1.

=countifs({1, 2; "a", #N/A!; 3, #N/A!}, #N/A!, a1:b3, 2009) → returns 2 if all cells in A1:B3 contain 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}) → 5.2
```

COVARIANCE.P(array1, array2)

Returns population covariance, which is the average of the products of deviations for each data point pair in two data sets. Use population covariance to determine the relationship between two data sets.

```
=COVARIANCE.P({3, 2, 4, 5, 6}, {9, 7, 12, 15, 17}) → 5.2
```

COVARIANCE.S(array1, array2)

Returns the sample covariance, the average of the products of deviations for each data point pair in two data sets.

```
=COVARIANCE.S({3, 2, 4, 5, 6}, {9, 7, 12, 15, 17}) → 6.5
```

critBinom(trials, probability, alpha)

Returns the smallest value (the number of successes) for which the cumulative binomial distribution function is greater than or equal to `alpha` for a given number of trials and the probability of success in each trial.

```
=critBinom(6, 0.5, 0.75) → 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) → 48
```

EXPON.DIST(x, lambda, cumulative)

Returns the exponential distribution. `x` is the value of the function, `lambda` is the parameter value, and `cumulative` is a logical value that indicates which form of the exponential function to provide.

`=EXPON.DIST(0.2, 10, TRUE)` → 0.864664717

exponDist(x, lambda, cumulative)

Returns the probability of the exponential distribution. The returned value is calculated as $P(X < x)$. The `lambda` argument specifies the distribution parameter. If the `cumulative` argument is 1 or TRUE, it returns the cumulative distribution function; otherwise, the probability density function is returned.

`=exponDist(0.2, 10, 1)` → 0.86466471676339

F.DIST(x, deg_freedom1, deg_freedom2, cumulative)

Returns the F probability distribution. `x` is the value at which to evaluate the function, `deg_freedom1` is the numerator degrees of freedom, and `deg_freedom2` is the denominator degrees of freedom.

`=F.DIST(15.206, 6, 4, TRUE)` → 0.997299

F.DIST.RT(x, deg_freedom1, deg_freedom2)

Returns the right-tailed F probability distribution.

`=F.DIST.RT(15.206, 6, 4)` → 0.002701

F.INV(probability, deg_freedom1, deg_freedom2)

Returns the inverse of the F probability distribution. If $p = F.DIST(x, \dots)$, then $F.INV(p, \dots) = x$.

`=F.INV(0.9973, 6, 4)` → 15.207

F.INV.RT(probability, deg_freedom1, deg_freedom2)

Returns the inverse of the right-tailed F probability distribution.

`=F.INV.RT(0.01, 6, 4)` → 9.19731

F.TEST(array1, array2)

Returns the result of an F-test. An F-test returns the two-tailed probability that the variances in `array1` and `array2` are not significantly different.

```
=F.TEST({6, 7, 9, 15, 21}, {20, 28, 31, 38, 40}) → 0.675841077
```

fDist(x, f_degrees1, f_degrees2)

Returns the F probability distribution function. The `f_degrees1` and `f_degrees2` arguments specify 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) → 0.99729988597273
```

fInv(x, f_degrees1, f_degrees2)

Returns the inverse of the `fDist` probability distribution function. The `f_degrees1` and `f_degrees2` arguments specify the numbers of degrees of freedom used in the F expression respectively as the numerator and denominator values.

```
=fInv(0.99729988597273, 6, 4) → 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) → 0.97295507452766
```

fisherInv(y)

Returns the inverse of the `fisher` function. The returned value is calculated as $x = \frac{\exp(2*y) - 1}{(\exp(2*y) + 1)}$.

```
=fisherInv(0.97295507452766) → 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}) →  
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 defines 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}) → {0;  
2; 5; 2}
```

fTest(array1, array2)

Performs the F-test and returns 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}) → 0.67584107664634
```

GAMMA(number)

Returns the Gamma function value.

```
=GAMMA(2.5) → 1.329340388
```

GAMMALN(x)

Returns the natural logarithm of the Gamma function, $\Gamma(x)$.

```
=GAMMALN(4) → 1.791759469
```

GAMMA.DIST(x, alpha, beta, cumulative)

Returns the gamma distribution. `x` is the value at which to evaluate the distribution, `alpha` and `beta` are parameters. If `cumulative` is TRUE, it returns the cumulative distribution function; if FALSE, it returns the probability density function.

```
=GAMMA.DIST(10, 9, 2, TRUE) → 0.068093635
```

GAMMALN.PRECISE(x)

Returns the natural logarithm of the Gamma function, $\Gamma(x)$, using a high-precision algorithm.

`=GAMMALN.PRECISE(4)` → 1.791759469

GAUSS(x)

Returns the probability that a member of a standard normal population falls between the mean and x standard deviations from the mean.

`=GAUSS(2)` → 0.477249868

geoMean(v1, v2, ...)

Returns the geometric mean for the specified positive values. The `v_` arguments can be numbers, arrays of numbers or references.

`=geoMean(4, 5, 8, 7, 11, 4, 3)` → 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 `=`, `>`, `>=`, `<`, `<=`, or `<>` operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD), or the `&` operator followed by a single cell address.
3. A text string optionally containing wildcard characters `?` (any single character) or `*` (any string). To search for a literal `?` 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")` → 2.44948974278318

`=geoMeanAifs({1, 3, "6", 2}, {3, 3, 3, 0.1}, ">" & b5)` → returns geometric mean based on the value in cell B5.

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 follow the same rules as defined in `geoMeanAIfs`.

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") → 1.81712059283214
```

```
=geoMeanIfs({1, 3, "6", 2}, {3, 3, 3, 0.1}, ">" & b5) → returns  
geometric mean based on the value in cell B5.
```

harMean(v1, v2, ...)

Returns the harmonic mean for the specified values. The `v_` arguments can be numbers, arrays of numbers or references.

```
=harMean(4, 5, 8, 7, 11, 4, 3) → 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 `=`, `>`, `>=`, `<`, `<=`, or `<>` operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD), or the `&` operator followed by a single cell address.
3. A text string optionally containing wildcard characters `?` or `*`. To search for a literal `?` 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") → 2
```

```
=harMeanAIfs({1, 3, "6", 2}, {3, 3, 3, 0.1}, ">" & b5) → returns  
harmonic mean based on the value in cell B5.
```

harMeanIifs(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 follow the same rules as defined in `harMeanAIfs`.

If there are no values meeting the specified criteria, the `#N/A!` error code is returned.

```
=harMeanIifs({1, 3, "6", 2}, {3, 3, 3, 0.1}, ">0") → 1.63636363636364
```

```
=harMeanIifs({1, 3, "6", 2}, {3, 3, 3, 0.1}, ">" & b5) → returns harmonic mean based on the value in cell B5.
```

HYPGEOM.DIST(sample_s, number_sample, population_s, number_pop, cumulative)

Returns the hypergeometric distribution. `sample_s` is the number of successes in the sample, `number_sample` is the size of the sample, `population_s` is the number of successes in the population, `number_pop` is the population size. If `cumulative` is `TRUE`, it returns the cumulative distribution function; if `FALSE`, it returns the probability mass function.

```
=HYPGEOM.DIST(1, 4, 8, 20, TRUE) → 0.465428277
```

hypGeomDist(sample_s, sample_size, population_s, population_size)

Returns the hypergeometric distribution function: the probability of `sample_s` successes in a `sample_size`-element sample with the known `population_s` number of successes in the `population_size`-element population.

```
=hypGeomDist(1, 4, 8, 20) → 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}) → 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) → -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) → 5
```

linEst(array_y, [array_x], [const_b])

Finds a best-fit regression line for the known x- and y-values and returns a two-element vector containing the `a` and `b` coefficients of the $y = a \cdot x + b$ line.

If the `const_b` argument is 0 (false), the `b` parameter is assumed to be 0. If `const_b` is omitted, it is assumed to be 1. If the `array_x` argument is omitted, it is 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 `#NUM!`.

```
=linEst({2, 3, 9, 1, 8}, {6, 5, 11, 7, 5},) → {0.04838709677419;  
0.66935483870968}
```

LOGNORM.DIST(x, mean, standard_dev, cumulative)

Returns the lognormal distribution of `x`, where $\ln(x)$ is normally distributed with parameters `mean` and `standard_dev`. If `cumulative` is TRUE, it returns the cumulative distribution function; if FALSE, it returns the probability density function.

```
=LOGNORM.DIST(4, 3.5, 1.2, TRUE) → 0.039083556
```

LOGNORM.INV(probability, mean, standard_dev)

Returns the inverse of the lognormal cumulative distribution function.

```
=LOGNORM.INV(0.03908, 3.5, 1.2) → 4.000101032
```

max(v1, v2, ...)

Returns the largest value. Arguments that are either text strings which cannot 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) → 11
```

```
=max(4, 5, 8, 7, {"11", 4}) → 8
```

maxA(v1, v2, ...)

Returns the largest value. Arguments that are either text strings which cannot 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 `max()` function.

```
=maxA(4, 5, 8, 7, 11, 4, 3) → 11
```

```
=maxA(4, 5, 8, 7, {"11", 4}) → 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 `=`, `>`, `>=`, `<`, `<=`, or `<>` operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD), or the `&` operator followed by a single cell address.
3. A text string optionally containing wildcard characters `?` (any single character) or `*` (any sequence of characters). To search for a literal `?` 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") → 6
```

```
=maxAifs({1, 3, "6", 7}, {3, 3, 3, 0.1}, ">" & b5) → returns the largest number based on the value in cell B5.
```

maxIfs(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 `=`, `>`, `>=`, `<`, `<=`, or `<>` operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD), or the `&` operator followed by a single cell address.
3. A text string optionally containing wildcard characters `?` (any single character) or `*` (any sequence of characters). To search for a literal `?` 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") → 3
```

```
=maxIfs({1, 3, "6", 7}, {3, 3, 3, 0.1}, ">" & b5) → returns the largest number based on the value in cell B5.
```

median(v1, v2, ...)

Returns the median of the specified numbers. The median is a number such that half the numbers are greater and half the numbers are smaller than the median. The `v_` arguments can be numbers, arrays of numbers, or references.

```
=median(1, 2, 3, 4, 5, 6) → 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 `=`, `>`, `>=`, `<`, `<=`, or `<>` operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD), or the `&` operator followed by a single cell address.
3. A text string optionally containing wildcard characters `?` or `*`. To search for a literal `?` 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") → 3.5
```

```
=medianAIfs({1, 2, 3, 4, "5", 6}, {3, 3, 3, 0.1}, ">"&b5) → returns the  
median based on the value in cell B5.
```

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 `=`, `>`, `>=`, `<`, `<=`, or `<>` operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD), or the `&` operator followed by a single cell address.
3. A text string optionally containing wildcard characters `?` or `*`. To search for a literal `?` 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") → 3
```

```
=medianIfs({1, 2, 3, 4, "5", 6}, {3, 3, 3, 0.1}, ">"&b5) → returns the  
median based on the value in cell B5.
```

min(v1, v2, ...)

Returns the smallest value. Arguments that are either text strings which cannot 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) → 3
```

```
=min(4, 5, 8, 7, 11, 4, {"1", "3"}) → 4
```

minA(v1, v2, ...)

Returns the smallest value. Arguments that are either text strings which cannot 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 `min()` function.

```
=minA(4, 5, 8, 7, 11, 4, 3) → 3
```

```
=minA(4, 5, 8, 7, 11, 4, {"1", "3"}) → 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 `=`, `>`, `>=`, `<`, `<=`, or `<>` operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD), or the `&` operator followed by a single cell address.
3. A text string optionally containing wildcard characters `?` or `*`. To search for a literal `?` 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") → 6
```

```
=minAifs({1, 3, "6", 7}, {3, 3, 3, 0.1}, ">"&b5) → returns the smallest  
number based on the value in cell B5.
```

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 `=`, `>`, `>=`, `<`, `<=`, or `<>` operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD), or the `&` operator followed by a single cell address.
3. A text string optionally containing wildcard characters `?` or `*`. To search for a literal `?` 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") → 7
```

```
=minIfs({1, 3, "6", 7}, {3, 3, 3, 0.1}, ">"&b5) → returns the smallest  
number based on the value in cell B5.
```

mode(v1, v2, ...)

Returns the most frequently occurring number in the specified numbers. The `v_` arguments can be numbers, arrays of numbers or references.

```
=mode({5.6, 4, 4, 3, 2, 4}) → 4
```

MODE.MULT(data1, data2, ...)

Returns a vertical array of the most frequently occurring or repetitive values in an array or range of data.

```
=MODE.MULT({1, 2, 2, 3, 3, 4}) → {2; 3}
```

MODE.SNGL(data1, data2, ...)

Returns the most frequently occurring or repetitive value in an array or range of data.

```
=MODE.SNGL({5.6, 4, 4, 3, 2, 4}) → 4
```

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. 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 `=`, `>`, `>=`, `<`, `<=`, or `<>` operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD), or the `&` operator followed by a single cell address.
3. A text string optionally containing wildcard characters `?` or `*`. To search for a literal `?` 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")  
→ 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 value in cell B5.
```

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 `=`, `>`, `>=`, `<`, `<=`, or `<>` operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD), or the `&` operator followed by a single cell address.
3. A text string optionally containing wildcard characters `?` or `*`. To search for a literal `?` 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") →  
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 value in cell B5.
```

NORM.DIST(x, mean, standard_dev, cumulative)

Returns the normal distribution for the specified mean and standard deviation. `x` is the value for which you want the distribution. If `cumulative` is `TRUE`, it returns the cumulative distribution function; if `FALSE`, it returns the probability density function.

```
=NORM.DIST(42, 40, 1.5, TRUE) → 0.908788775
```

NORM.INV(probability, mean, standard_dev)

Returns the inverse of the normal cumulative distribution for the specified mean and standard deviation.

```
=NORM.INV(0.90878877482188, 40, 1.5) → 42
```

NORM.S.DIST(z, cumulative)

Returns the standard normal distribution (mean of 0 and standard deviation of 1). `z` is the value for which you want the distribution.

`=NORM.S.DIST(1.333333334, TRUE)` → 0.908788775

NORM.S.INV(probability)

Returns the inverse of the standard normal cumulative distribution.

`=NORM.S.INV(0.9087887748)` → 1.333333334

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)` → 0.90878877482188

normsDist(x)

Returns the standard normal cumulative distribution (where the mean is 0 and the standard deviation is 1).

`=normsDist(1.33333333432674)` → 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)` → 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)` → 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}) → 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 is not a multiple of $1 / (n - 1)$, the returned value is interpolated.

`=percentile({1, 2, 3, 4}, 0.3) → 1.9`

PERCENTILE.EXC(array, k)

Returns the k-th percentile of values in a range, where `k` is in the range (0, 1) exclusive.

`=PERCENTILE.EXC({1, 2, 3, 4}, 0.25) → 1.25`

PERCENTILE.INC(array, k)

Returns the k-th percentile of values in a range, where `k` is in the range [0, 1] inclusive.

`=PERCENTILE.INC({1, 2, 3, 4}, 0.3) → 1.9`

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 is 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 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 `=`, `>`, `>=`, `<`, `<=`, or `<>` operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD), or the `&` operator 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 a literal ? 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") → 1.9
```

```
=percentileAIfs({1, 2, 3, 4}, 0.3, {3, 3, 3, 0.1}, ">" & b5) → returns  
the k-th percentile based on the value in cell B5.
```

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 is 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 =, >, >=, <, <=, or <> operators followed by an unformatted number, a generic date/time string (YYYY-MM-DD), or the & operator 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 a literal ? 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") → 1.6
```

```
=percentileIfs({1, 2, 3, 4}, 0.3, {3, 3, 3, 0.1}, ">" & b5) → returns  
the k-th percentile based on the value in cell B5.
```

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 the array.

The significance argument specifies the number of decimal places in the result. If it is omitted, it is assumed to be 3. If array does not include x, the function returns an interpolated value.

```
=percentRank({1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, 4, ) → 0.333
```

PERCENTILE.EXC(array, k, [significance])

Returns the k-th percentile of values in a range, where **k** is in the range (0, 1) exclusive. The **significance** argument determines the precision of the result.

```
=PERCENTILE.EXC({1, 2, 3, 4, 5}, 0.2, 2) → 1.2
```

PERCENTILE.INC(array, k, [significance])

Returns the k-th percentile of values in a range, where **k** is in the range [0, 1] inclusive. The **significance** argument determines the precision of the result.

```
=PERCENTILE.INC({1, 2, 3, 4, 5}, 0.3, 2) → 2.2
```

permut(n, k)

Returns the number of k-element permutations of the n-element set.

```
=permut(100, 3) → 970200
```

PHI(x)

Returns the value of the density function for a standard normal distribution at value **x**.

```
=PHI(0.75) → 0.301137432
```

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) → 0.08422433748857
```

POISSON.DIST(x, mean, cumulative)

Returns the Poisson distribution for a given number of events **x** where **mean** is the distribution parameter. Parameters function similarly to the **poisson** function.

```
=POISSON.DIST(2, 5, FALSE) → 0.084224337
```

polyPoints(x, B, chi2, n, r, t, p)

The `polyPoints` function uses results returned by `polyReg` to calculate values lying on the specified polynomial line along with their confidence limits for a given probability `p`.

The `x` and `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 than or equal to 1 and not greater than the `r` value passed to the `polyReg` function. The `t` argument is a column vector containing `N` new values that the polynomial is calculated for. The `p` argument specifies the probability used to calculate the confidence limits.

The function 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 $[y'[i] - \text{eta}[i], y'[i] + \text{eta}[i]]$ at the specified probability `p`.

Input arguments 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`

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) → {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) → {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` function calculates the coefficients of the family of `r` orthogonal polynomials (from the 0 degree to the $r - 1$ degree) and suggests the most suitable degree of $w(t)$.

The polynomial of the $(r - 1)$ -th degree can be expressed as:

$$w(t) = x[1]*f_1(t) + x[2]*f_2(t) + \dots + x[r]*f_r(t)$$

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:

$$f_i(t) = \text{Sum}_{k=1}^{j-1} (B[j, k] * t^{(k - 1)})$$

and

$\text{Sum}_{i=1, N} (g(i) * f_j(t[i]) * f_k(t[i])) = \{ 1 \text{ if } i=j, 0 \text{ if } i \neq j \}$

for given measurement weights $g(i)$.

The t argument is a column vector containing N values $t[i]$. The y argument is a column vector containing N measurements $y[i]$. The sy argument is a column vector containing N values of measurement errors. The r argument specifies the number of coefficients to find (where $r - 1$ is the maximum degree); $r \leq N$.

- If $\text{type} = 1$, the function returns the x column vector of calculated coefficients.
- If $\text{type} = 2$, the function returns the $B[r, r]$ matrix.
- If $\text{type} = 3$, the function returns a column vector of $\text{chi2}[1..r]$ values.

`=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) → {14.0688; 1.4979; 0; 0}` (indicating a 1st degree polynomial description).

prob(array_x, array_p, lower_limit, [upper_limit])

Returns the cumulative probability for values in the range $[\text{lower_limit}, \text{upper_limit}]$ based on the intervals specified by array_x and the corresponding probabilities array_p .

Each array_p 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 only.

`=prob({0, 1, 2, 3}, {0.2, 0.3, 0.1, 0.4}, 1, 3) → 0.8`

quartile(array, n)

Returns the quartile of a data set. The n argument specifies which value should be returned:

- 0: minimum value
- 1: 1st quartile
- 2: 2nd quartile (median)
- 3: 3rd quartile
- 4: maximum value

`=quartile({1, 2, 4, 7, 8, 9, 10, 12}, 1) → 3.5`

QUARTILE.EXC(array, quart)

Returns the quartile of the data set, based on percentile values from (0..1), exclusive. `quart` specifies the value to return (1, 2, or 3).

```
=QUARTILE.EXC({1, 2, 3, 4, 5, 6, 7, 8, 9}, 1) → 2.5
```

QUARTILE.INC(array, quart)

Returns the quartile of the data set, based on percentile values from [0..1], inclusive. `quart` specifies the value to return (0, 1, 2, 3, or 4).

```
=QUARTILE.INC({1, 2, 3, 4, 5, 6, 7, 8, 9}, 1) → 3
```

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 or TRUE, the returned rank value corresponds to ascending order. Otherwise, descending order is used. If omitted, it is assumed to be 0.

```
=rank(3.5, {7, 3.5, 3.5, 1, 2}, 1) → 3
```

RANK.AVG(number, ref, [order])

Returns the rank of a number in a list of numbers. If multiple values have the same rank, the average rank is returned.

```
=RANK.AVG(3.5, {7, 3.5, 3.5, 1, 2}, 1) → 3.5
```

RANK.EQ(number, ref, [order])

Returns the rank of a number in a list of numbers. Its size is relative to other values in the list; if multiple values have the same rank, the top rank of that set of values is returned.

```
=RANK.EQ(3.5, {7, 3.5, 3.5, 1, 2}, 1) → 3
```

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}) → 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) → 0.3595430714068
```

SKEW.P(data1, [data2], ...)

Returns the skewness of a distribution based on a population: a characterization of the degree of asymmetry of a distribution around its mean.

```
=SKEW.P(3, 4, 5, 2, 3, 4, 5, 6, 4, 7) → 0.306122449
```

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}) → 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) → 3
```

standardize(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) → 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) →  
27.4639157198435
```

STDEV.P(data1, data2, ...)

Calculates standard deviation based on the entire population given as arguments.

```
=STDEV.P(1345, 1301, 1368, 1322, 1310, 1370, 1318, 1350, 1303, 1299)  
→ 26.05455814
```

STDEV.S(data1, data2, ...)

Estimates standard deviation based on a sample (ignores logical values and text in the sample).

```
=STDEV.S(1345, 1301, 1368, 1322, 1310, 1370, 1318, 1350, 1303, 1299)  
→ 27.46391572
```

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)  
→ 27.4639157198435
```

stdevAlfs(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 comparison operators (=, >, etc.) followed by a number, date string, or cell address.
3. A text string optionally containing wildcard characters `?` or `*`.

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") → 0.57735026918963
```

```
=stdevAIfs({5.6, 4, 4, "3", "3", 4}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<"&b5) → returns standard deviation based on cell B5 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 follow the same rules as `stdevAIfs`.

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") → 0
```

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) → 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) → 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 dimensions. Criteria follow the rules defined in `stdevAIfs`.
If no values meet the criteria, returns `#N/A!`.

```
=stdevPAIfs({5.6, 4, 4, "3", "3", 3}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<1") → 0.43301270189222
```

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 dimensions. Criteria follow the rules defined in `stdevAIfs`.
If no values meet the criteria, returns `#N/A!`.

```
=stdevPIfs({5.6, 4, 4, "3", "3", 3}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<1") → 0.5
```

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}) → 3.74560674557182
```

T.DIST(x, deg_freedom, cumulative)

Returns the Student's t-distribution. `x` is the numeric value to evaluate, `deg_freedom` is an integer indicating degrees of freedom. If `cumulative` is `TRUE`, it returns the cumulative distribution function; if `FALSE`, it returns the probability density function.

```
=T.DIST(1.96, 60, TRUE) → 0.972677535
```

T.DIST.2T(x, deg_freedom)

Returns the two-tailed Student's t-distribution.

```
=T.DIST.2T(1.96, 60) → 0.054644929
```

T.DIST.RT(x, deg_freedom)

Returns the right-tailed Student's t-distribution.

```
=T.DIST.RT(1.96, 60) → 0.027322465
```

T.INV(probability, deg_freedom)

Returns the t-value of the Student's t-distribution as a function of the probability and the degrees of freedom.

```
=T.INV(0.027, 60) → -1.9669
```

T.INV.2T(probability, deg_freedom)

Returns the inverse of the two-tailed Student's t-distribution.

```
=T.INV.2T(0.054, 60) → 1.9654
```

T.TEST(array1, array2, tails, type)

Returns the probability associated with a Student's t-test. Use `T.TEST` to determine whether two samples are likely to have come from the same two underlying populations that have the same mean. `tails` specifies the number of distribution tails (1 or 2). `type` specifies the kind of t-test to perform.

```
=T.TEST({3, 4, 5}, {6, 19, 3}, 2, 2) → 0.385311
```

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) → 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]` and `t[i]` values is 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. The `l` argument specifies the degree of the polynomial to approximate the trend line. The `k` argument specifies the range $[i - k, i + k]$ of points used to calculate the moving average. For points where $i \leq k$ or $i > n - k$, the function extends the trend line using coefficients calculated for the closest points. The `p` argument specifies the probability for the confidence limits.

The function returns a two-column array; the first column contains $N + 2k$ values $y'[i]$ lying on the trend line and the second column contains $N + 2k$ values $\eta[i]$ defining the confidence limits such that the true value is expected to be in the range $[y'[i] - \eta[i], y'[i] + \eta[i]]$ at probability `p`.

```
=timeSeries({11; 8; 7; 2; 1; 3; 8; 4}, 3, 2, 0.8) → {7.800, 71.396;  
10.800; ... ; 45.400}
```

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) → 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 of predicted y-values for the new x-values specified in `array_n`.

If `const_b` is 0 or FALSE, the `b` parameter is assumed to be 0 (default is 1). If `array_x` is omitted, it's assumed to be $[1, 2, \dots, n]$. If `array_n` is omitted, it's assumed to be the same as `array_x`.

```
=trend({2, 3, 9, 1, 8}, {6, 5, 11, 7, 5}, {12, 13, 14}, 1) →  
{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 $[0, 1]$ and is rounded to the nearest multiple of 2 to exclude elements symmetrically.

```
=trimMean({4, 5, 6, 7, 2, 3, 4, 5, 1, 2, 3}, 0.2) → 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) → 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})  
→ -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) →  
754.266666666667
```

VAR.P(data1, data2, ...)

Calculates variance based on the entire population.

```
=VAR.P(1345, 1301, 1368, 1322, 1310, 1370, 1318, 1350, 1303, 1299) →  
678.84
```

VAR.S(data1, data2, ...)

Estimates variance based on a sample (ignores logical values and text in the sample).

```
=VAR.S(1345, 1301, 1368, 1322, 1310, 1370, 1318, 1350, 1303, 1299) →  
754.2666667
```

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) →  
754.266666666667
```

varAlfs(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 comparison operators (=, >, etc.) followed by a number, date string, or cell address.
3. A text string optionally containing wildcard characters ? or *.

If there are no values meeting the specified criteria, the `#N/A!` error code is returned.

```
=varAlfs({5.6, 4, 3, "3", "3", 4}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<1")  
→ 0.25
```

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 follow the rules defined in `varAlfs`.

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")  
→ 0.5
```

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) →  
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) → 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 dimensions. Criteria follow the rules defined in `varAifs`. If no values meet the criteria, returns `#N/A!`.

```
=varPAifs({5.6, 4, 3, "3", "3", 4}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<1") → 0.1875
```

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 dimensions. Criteria follow the rules defined in `varAifs`. If no values meet the criteria, returns `#N/A!`.

```
=varPIfs({5.6, 4, 3, "3", "3", 4}, {3, 3, 0.1, 0.1, 0.1, 0.1}, "<1") → 0.25
```

weibull(x, alpha, beta, cumulative)

Returns the Weibull distribution for a given `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) → 0.92958139006928
```

WEIBULL.DIST(x, alpha, beta, cumulative)

Returns the Weibull distribution. Parameters function similarly to the `weibull` function.

`=WEIBULL.DIST(105, 20, 100, TRUE)` → 0.92958139

Z.TEST(array, x, [sigma])

Returns the one-tailed P-value of a z-test. For a given hypothesized population mean, `x`, `Z.TEST` returns the probability that the sample mean is greater than the average of observations in the `array`.

`=Z.TEST({3, 6, 7, 8, 6, 5, 4, 2, 1, 9}, 4)` → 0.090574203

zTest(array, x, [sigma])

Performs a Z-test and returns the two-tailed probability that `x` does not 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 is omitted, the sample deviation is used.

`=zTest({3, 6, 7, 8, 6, 5, 4, 2, 1, 9}, 4,)` → 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 is omitted, the sample deviation is used.

`=zTest2({3, 6, 7, 8, 6, 5, 4, 2, 1, 9}, 4,)` → 1.33722748248063

[GS-Calc Help](#) > Calculation Functions > Text Functions

Function Index

[char](#) [clean](#) [code](#) [concatenate](#) [dollar](#) [exact](#) [find](#) [fixed](#) [left](#) [len](#) [lenb](#) [lower](#)
[mid](#) [proper](#) [replace](#) [rept](#) [right](#) [search](#) [substitute](#) [t](#) [text](#) [trim](#) [trimE](#) [trimL](#)
[trimR](#) [upper](#) [value](#)

Text Functions

char(n)

Returns the character specified by the code number `n`.

```
=char(65) → "A"
```

clean(text)

Removes all non-printable characters from the specified text.

```
=clean(char(7) & "abc") → "abc"
```

code(text)

Returns a numeric code for the first character in a text string.

```
=code("ABC") → 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) → "abc1abc2"
```

dollar(x, [decimals])

Converts a number `x` to a text string using the default currency format and the specified precision. If the `decimals` argument is greater than 0, then `x` is rounded to the specified number of decimal places. If it is 0, then `x` is rounded to the nearest integer. If it is less than 0, then `x` is rounded to the left of the decimal point. The default value of `decimals` is 2.

```
=dollar(10.95, ) → "$10.95"
```

```
=dollar(10.9487, 3) → "$10.949"
```

exact(text1, text2)

Checks whether two text strings are exactly the same. Returns 1 if they are identical and 0 otherwise. The comparison is case-sensitive.

```
=exact("Abc", "abc") → 0
```

```
=exact("Abc", "Abc") → 1
```

find(pattern, subject, [start])

find(pattern, subject, [start], [flags])

Searches `subject` for a `pattern` starting at the position `start` and returns the position of the first occurrence found (except when using the "RegExStr" flag). If the `pattern` is not found or if `start` is out of the valid range, it returns the #VALUE! error.

The first version accepts a plain text string as the `pattern` and the comparison is case-sensitive.

The second version also accepts regular expressions as the `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; 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) → 8
```

```
=find("e*\d", "abcdef abcee abcde5", 1, SEARCH::RegEx) → 18
```

fixed(x, [digits], [no_separators])

Rounds `x` to the specified number of digits and formats it as a string 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 is 0, `x` is rounded to the nearest integer. If it is less than 0, `x` is rounded to the left of the decimal point. If `no_separators` is 0 (default), the number is formatted using thousand separators. If `digits` is omitted, it is assumed to be 2.

```
=fixed(10.956,,) → "10.96"
```

```
=fixed(1234.89, -2, 0) → "1,200"
```

left(text, [n])

Returns the first `n` characters from the start of a text string. If `n` is omitted, it is assumed to be 1. If `n` is greater than the number of characters in the text, the entire string is returned.

```
=left("abc", ) → "a"
```

len(text)

Returns the number of characters in a text string.

```
=len("text") → 4
```

lenb(text)

Returns the number of bytes in a text string. Since text is internally stored as UTF-8, the number of bytes will be larger than the number of characters for strings containing non-ASCII characters (characters with codes above 127).

```
=lenb("örtlich") → 8
```

lower(text)

Converts all letters in a text string to lowercase.

```
=lower("TEXT") → "text"
```

mid(text, n1, n2)

Returns up to `n2` characters from `text` starting at the position `n1`. If `n1` is greater than the total number of characters, it returns an empty string. If `n1` is less than 1 or if `n2` is negative, it returns the `#VALUE!` error.

```
=mid("some text", 3, 2) → "me"
```

proper(text)

Converts the first character of each word in a text string to uppercase and all other characters to lowercase.

```
=proper("some teXT") → "Some Text"
```

replace(subject, n1, n2, replace)

replace(pattern, subject, start, replace, [flags])

The first version removes `n2` characters from `subject` starting at position `n1` and replaces them with the string `replace`.

The second version searches `subject` for `pattern` starting at 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:

- `SEARCH::Regex` (or 8192) - `pattern` is a regular expression.
- `SEARCH::CaseSensitive` (or 128) - use case-sensitive comparison (regular expressions can overwrite this).

If `SEARCH::Regex` is specified, the `replace` argument can contain:

- (1) Absolute references (by number) to capturing subpatterns, e.g., `\1`, `\2` ... `\999`.
- (2) `\l`, `\L`, `\u`, `\U` literals for binary switching of upper- and lower-case conversion.
- (3) `\r`, `\n` - 'line feed' and 'new line' literals (pressing Ctrl+Enter in a cell typically inserts a `\r\n` sequence).

```
=replace("abcd", 3, 2, "") → "ab*"
```

```
=replace("(.)a+\d{1,3}", "abc aa0102", 1, "\1", SEARCH::Regex) → "abc  
2"
```

```
=replace("(ab)", "abcdef ghijk abb123", 1, "\u\1", SEARCH::Regex) →  
"ABcdef ghijk ABb123"
```

rept(text, n)

Repeats a text string `n` times.

```
=rept("abc", 3) → "abcabcabc"
```

right(text, [n])

Returns the last `n` characters from a text string. If `n` is omitted, it is assumed to be 1. If `n` is greater than the number of characters in the text, the entire string is returned.

```
=right("abc", ) → "c"
```

search(searchText, withinText, [n])

Finds the first occurrence of `searchText` within `withinText` and returns its position. It starts searching at position `n`. If `n` is omitted, it is assumed to be 1 (the first character). The `searchText` argument can contain special characters: `?` (matching any single character) or `*` (matching any string, including an empty string). To search for a literal `?` or `*`, place a tilde (`~`) before them.

```
=search("text", "some text", 1) → 6
```

```
=search("?bc?e*", "abc abcd abcde ab",) → 10
```

substitute(searchText, withinText, replaceText, [n])

Finds the `n`-th occurrence of `searchText` in `withinText` and substitutes `searchText` with `replaceText`. To replace all occurrences, specify 0 for the `n` parameter. The default value of `n` is 0.

```
=substitute("a1b1c1d", "1", "-", 0) → "a-b-c-d"
```

```
=substitute("abc abcd abcde ab", "?bc?e*a", "x", 1) → "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)

Formats a given number or text using the specified format code.

If the function is used to format a date, the `value` argument must 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") → "-$2,345.40"
```

```
=text(-2345.4, "#,##0.00;(#,##0.00)") → "(2,345.40)"
```

```
=text("bc", "\a@d") → "abcd"
```

```
=text(date(2009,10,21), "mmm\ d\, \ yyyy") → "October 21, 2009"
```

```
=text("2009-10-21", "mmm\ d\, \ yyyy") → "October 21, 2009"
```

trim(text)

Removes all whitespace from text except for single spaces between words.


```
=trim(" abc def ghi") → "abc def ghi"
```

trimE(text)

Removes leading and trailing spaces from a given text string. Unlike the `trim()` function, `trimE` does not compress inner spaces.

```
=trimE(" abc def ") → "abc def"
```

trimL(text)

Removes only the leading spaces from a given text string.

```
=trimL(" abc def ") → "abc def "
```

trimR(text)

Removes only the trailing spaces from a given text string.

```
=trimR(" abc def ") → "abc def"
```

upper(text)

Converts all letters in a text string to uppercase.

```
=upper("text") → "TEXT"
```

value(text)

Converts a text string that represents a number to a numeric value.

```
=value("123.45") → 123.45
```

```
=value("10 4/5") → 10.8
```

```
=value("$1,000") → 1000
```

Function Index

[date](#) [dateDiff](#) [dateText](#) [dateValue](#) [day](#) [eDate](#) [eoMonth](#) [hour](#) [minute](#) [month](#)
[networkDays](#) [NETWORKDAYS.INTL](#) [now](#) [nowText](#) [second](#) [time](#) [timeText](#)
[timeValue](#) [today](#) [todayText](#) [weekDay](#) [weekNum](#) [workDay](#) [WORKDAY.INTL](#) [year](#)
[yearDay](#)

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, 1900 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, 1900 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, [weekend], [holidays])

Returns the number of working days between **startDateNumber1** and **endDateNumber2**, excluding weekends and any dates listed in **holidays**.

The **weekend** argument defines which days of the week are treated as weekend days. It can be either:

- a numeric code (1–17), or
- a 7-character string of 0/1 flags (Monday → Sunday), where 1 marks a weekend day.

If **weekend** is omitted, the default weekend is Saturday–Sunday.

If **holidays** is omitted, no holiday dates are excluded.

Examples:

NETWORKDAYS.INTL("2025-01-01", "2025-01-31") returns the number of business days in January 2025 (Sat–Sun weekend).

NETWORKDAYS.INTL("2025-01-01", "2025-01-31", "0000011") treats Friday–Saturday as weekend.

NETWORKDAYS.INTL("2025-01-01", "2025-01-31", 7, { "2025-01-06", "2025-01-15" }) excludes two holiday dates.

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])

Returns the date that is **days** working days before or after **start_date**, excluding weekends and any dates listed in **holidays**.

The **weekend** argument defines which days of the week are treated as weekend days. It can be either:

- a numeric code (1–17), or
- a 7-character string of 0/1 flags (Monday → Sunday), where 1 marks a weekend day.

If **days** is positive, the result is a future working date. If **days** is negative, the result is a past working date.

If **weekend** is omitted, the default weekend is Saturday–Sunday.

If **holidays** is omitted, no holiday dates are excluded.

Examples:

WORKDAY.INTL("2025-01-01", 10) returns the date 10 workdays after Jan 1 (Sat–Sun weekend).

WORKDAY.INTL("2025-01-01", 10, "0000011") treats Friday–Saturday as weekend.

WORKDAY.INTL("2025-01-01", -5, 1, { "2025-01-06" }) returns the date 5 workdays before Jan 1, excluding a holiday.

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)) returns 1

=yearDay("2010-07-04") returns 185

[GS-Calc Help](#) > *Calculation Functions* > *Logical Functions*

Function Index

[and](#) [bitand](#) [bitnot](#) [bitor](#) [bitxor](#) [false](#) [if](#) [ifError](#) [ifNA](#) [ifs](#) [not](#) [or](#) [switch](#)
[true](#) [xor](#) [cell](#) [countBlank](#) [errorType](#) [isBlank](#) [isErr](#) [isError](#) [isEven](#) [isLogical](#)
[isNA](#) [isNonText](#) [isNumber](#) [isOdd](#) [isRef](#) [isText](#) [n](#) [na](#) [type](#)

Logical Functions

and(expression1, expression2, ...)

Returns 1/true if all the supplied expressions evaluate to non-zero values and 0/false otherwise.

Text strings are converted to numbers. Text strings that can't be converted to numbers cause #NUM! errors. Empty cells are ignored. If there are no non-empty cells, the function returns the #N/A! error value.

```
=and(1, 1, true, "1") returns 1  
=and(a1 >= 0, a1 < 0)  
=and({true, true, 1, 1}, {true, true, 1, 0}) returns 0  
=and(345, 178) returns 1  
=and(1, 2, 4, 8) returns 1
```

bitand(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 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 are no non-empty cells, the function returns the #N/A! error value.

```
=bitand(1, 1, true, "1") returns 1  
=bitand(a1 >= 0, a1 < 0) returns 0  
=bitand({true, true, 1, 1}, {true, false}) returns 0  
=bitand(345, 178) returns 16  
=bitand(1, 2, 4, 8) returns 0
```

bitnot(v, bits)

bitnot(v)

Returns the bitwise negation of v using the specified number of bits. The bits parameter must be in the range <1, 49>. If it is omitted, the default value is 16.

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.

```
=bitnot(1, 8) returns 254  
=bitnot(1, 16) returns 65534  
=bitnot(34, 8) returns 221  
=bitnot(true) returns 0
```

bitor(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 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 are no non-empty cells, the function returns the #N/A! error value.

=bitor(1, 1, true, 1) returns 1
=bitor(a1 >= 0, a1 < 0) returns 1
=bitor(345, 178) returns 507
=bitor(1, 2, 4, 8) returns 15

bitxor(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 are no non-empty cells, the function returns the #N/A! error value.

=bitxor(1, 1) returns 0
=bitxor(a1 >= 0, a1 < 0) returns 1
=bitxor({true, true, 1, 1}, {true, false}) returns 1
=bitxor(345, 178) returns 491
=bitxor(1, 2, 4, 8) returns 15

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.

ifError(data [, replacement])

If the data parameter evaluates to any error value, the replacement is returned (or an empty cell if replacement is omitted).

Data can be any expression: a value, reference, or array.

ifNA(data [, replacement])

If the data parameter is the #N/A! error value, the replacement is returned (or an empty cell if replacement is omitted).

Data can be any expression: a value, reference, or array.

ifs(condition1, value1, [condition2, value2, ...])

Evaluates multiple conditions condition1, condition2, ... and returns a value that corresponds to the first true condition. The evaluation is performed from left to right. If conditions are arrays/ranges, all must have the same size.

=ifs(A1 > 0.75, "above 75%", A1 > 0.5, "above 50%", A1 > 0.25, "above 25%")

=ifs({1, 2, 3, 4, 5} > 2, ">2", {1, 2, 3, 4, 5}=1, "equal 1") returns {"equal 1", #N/A!, ">2", ">2", ">2"}

not(v)

Returns the negation of a logical expression (as 1 or 0).

=not(1) returns 0
=not(1.1) returns 0
=not(0) returns 1
=not(true) returns 0

or(v1, v2, ...)

Returns 1/true if at least one of the specified expressions evaluates to a non-zero value and 0/false otherwise.

Text strings are converted to numbers. Text strings that can't be converted to numbers cause #NUM! errors. Empty cells are ignored. If there are no non-empty cells, the function returns the #N/A! error value.

=or(1, 1, true, 1) returns 1
=or(a1 >= 0, a1 < 0)
=or(345, 178) returns 1
=or(1, 2, 4, 8) returns 1

switch(expression, case1, value1, [case2, value2, ...], [default])

Tests the value of the expression against the case1, case2, ... values and, if a matching case value is found, returns the corresponding value. If no matching case value is found, the #N/A! error value is returned (or the optional default value if specified).

=SWITCH(A5, "dog", "mammal", "falcon", "bird")
=SWITCH(A1:A10, 1, "one", 2, "two", 3, "three", "etc.")

true()

Returns 1.

xor(expression1, expression2, ...)

Returns 1/true if there is an odd number of logical expressions that evaluate to 1/true and 0/false otherwise.

Text strings are converted to numbers. Text strings that can't be converted to numbers cause #NUM! errors. Empty cells are ignored. If there are no non-empty cells, the function returns the #N/A! error value.

=xor(1, 1, true) returns 1
=xor(a1 >= 0, a1 < 0)

=xor({true, true, 1, 1}, {true, 1}) returns 0
=xor(345, 178) returns 0
=xor(1, 2, 3) returns 1

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

[GS-Calc Help](#) > Calculation Functions > Informational Functions

Function Index

cell countBlank errorType isBlank isErr isError isEven isLogical isNA isNonText
isNumber isOdd isRef isText n na type

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

GS-Calc Help > Calculation Functions > Lookup & Reference Functions

Function Index

[address](#) [areas](#) [choose](#) [column](#) [columns](#) [filter](#) [hLookUp](#) [hstack](#) [index](#) [indirect](#)
[lookUp](#) [match](#) [offset](#) [pivotData](#) [row](#) [rows](#) [rtd](#) [sort](#) [sparkLine](#) [transpose](#) [udf](#)
[unique](#) [vLookUp](#) [vstack](#)

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 maximum 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 first 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 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 desired cells.

The "hyperlink-path" argument represents a worksheet/workbook path that will be used along with cell references if the "include hyperlinks" option 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 "options" 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 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, for example, 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], [startIndex], [occurrence])

The **hLookup()** function searches the top row of **array** for the value **v**. If a match is found, the function returns the value from the same column and the **n-th** row of the array.

1. Extended Variant: startIndex and occurrence

The extended variant adds two parameters:

startIndex — Positive values specify where searching begins (1 = top-left cell, 2 = next cell, etc.). Negative values specify where searching ends (-1 = bottom-right cell, -2 = preceding cell, etc.).

occurrence — Specifies which occurrence of the matching value to return. Positive values count matches top-down (1 = first match, 2 = second match, etc.). Negative values count matches bottom-up (-1 = last match, -2 = second-to-last, etc.). Occurrences are counted relative to **startIndex**.

2. Search Type

The **type** argument determines how the search is performed. It may be one of the basic modes (0, 1, -1) or a sum of **SEARCH::** flags.

2.1 Basic Search Modes

- **0 — Exact match (linear search)**

Performs a linear search for an exact match. The search value **v** may contain wildcard patterns:

- **?** — matches any single character
- ***** — matches any sequence of characters (including empty)

To search for literal **?** or *****, prefix them with a tilde:

- **~?** — literal question mark
- **~*** — literal asterisk

Pattern matching is only available in mode **0**.

- **1 — Largest value ≤ v (ascending)**

If no exact match is found, returns the largest value that is not greater than **v**. No wildcard matching is performed. The searched range must be sorted in ascending order.

- **-1 — Smallest value ≥ v (descending)**

If no exact match is found, returns the smallest value that is not smaller than **v**. No wildcard matching is performed. The searched range must be sorted in descending order.

2.2 SEARCH:: Flags

Instead of using 0, 1, or -1, you may specify a sum of **SEARCH::** flags:

- **SEARCH::MatchNotGreater (2)** — largest value ≤ v
- **SEARCH::MatchNotSmaller (4)** — smallest value ≥ v
- **SEARCH::SortAscending (8)** — binary search, ascending order
- **SEARCH::SortDescending (16)** — binary search, descending order

- **SEARCH::CaseSensitive (128)** — case-sensitive comparison
- **SEARCH::FirstMatch (256)** — return first match
- **SEARCH::LastMatch (512)** — return last match
- **SEARCH::MixedData (2048)** — range contains both text and numbers
- **SEARCH::RegEx (8192)** — v is a regular expression
- **SEARCH::IgnorePunctuation (16384)** — ignore certain punctuation marks
- **SEARCH::NeutralSortOrder (32768)** — language-neutral comparison
- **SEARCH::Pattern (65536)** — v is a wildcard pattern (? and *)

2.3 Pattern Matching vs. Regular Expressions

- **SEARCH::Pattern**
Uses wildcard matching with ? and *. Cannot be combined with fast binary searching (SortAscending / SortDescending).
- **SEARCH::RegEx**
Uses regular expressions. Cannot be combined with:
 - SEARCH::MatchNotGreater
 - SEARCH::MatchNotSmaller
 - SEARCH::StringSort
 - SEARCH::CaseSensitive
 - SEARCH::SortAscending
 - SEARCH::SortDescending

2.4 Type Equivalents

1 = SEARCH::SortAscending + SEARCH::MatchNotGreater

-1 = SEARCH::SortDescending + SEARCH::MatchNotSmaller

0 = (0)

If **type** is omitted, it defaults to **1**.

3. Flag Compatibility Rules

SEARCH::Pattern and SEARCH::RegEx cannot be used with:

- SEARCH::MatchNotGreater
- SEARCH::MatchNotSmaller
- SEARCH::StringSort
- SEARCH::CaseSensitive
- SEARCH::SortAscending
- SEARCH::SortDescending

SEARCH::MatchNotGreater and SEARCH::MatchNotSmaller cannot be used with SEARCH::FirstMatch or SEARCH::LastMatch.

If neither SEARCH::FirstMatch nor SEARCH::LastMatch is specified, linear search returns the first match, while binary search may return any match.

4. Sorting Requirements

If SEARCH::SortAscending or SEARCH::SortDescending is used:

- The searched range must be sorted accordingly.
- The range should not contain formulas that break the sort order during recalculation.
- No circular reference warnings will be reported for cells other than the result cell.

Binary searches are typically tens or hundreds of times faster than linear searches.

5. Mixed Data

If SEARCH::MixedData is specified:

- The range may contain both numbers and text.
- If v is text, all values are converted to text before comparison.
- If v is numeric, text values representing numbers are converted to numbers.
- If sorting flags are used, the range must be sorted using mixed numeric/text ordering.

6. Performance Tip

If a workbook contains many hLookup() calls, you may use the numeric sum of flags instead of individual names for faster evaluation.

7. Error Handling

If no match is found, hLookup() returns #N/A!.

8. 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

9. Insert hLookup() Automatically

You can insert hLookup() with two clicks using **Insert** → **HLOOKUP()**. The dialog box determines optimal parameters, presets options, and builds the formula automatically.

10. Source Range List

The 16-element source range list is global and stored in the settings file. New ranges are added in a circular manner using:

“Copy as Location”

The “Enter” toolbar button

hstack(value-or-reference1, value-or-reference2 [,value-or-reference...])

Merges the specified data items (values, cell or range references) horizontally and returns the combined array. Rows without data are left empty.

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 function 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 'width' 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:

- VT_EMPTY (empty cells)
- VT_UI1 (error codes)
- VT_R8 (numbers)
- VT_BSTR (text strings)
- 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.
```

sparkLine(data-series, chart-specification-string)

Displays a mini-chart in cell (or merged cells). Please online see the full specification.

```
=SPARKLINE(D17,{"charttype","column";"ymin", 0; "ymax",2000;"firstcolor","#334960"})
```

transpose(array)

Transpose a given array.

```
=transpose({1, 2, 3; 4, 5, 6}) returns {1, 4; 2, 5; 3, 6}
```

udf(python_module, function, type, arg1, arg2, ...)

Executes a function from the specified Python module.

The "type" parameter specifies the type of the returned value:

0 - number

1 - matrix / array of numbers

- 2 - text string (up to 1024 characters)
- 3 - string / block of CSV data (of any size) to be pasted without parsing numbers in cells
- 4 - string / block of CSV data (of any size) to be pasted with parsing numbers in cells
- 5 - an image (binary data returned as standard Python "memoryview" binary data)

Additionally, you can add to the "type" parameter the following flags :

- 32 - all ranges/arrays passed from GS-Calc to Python will be saved and passed to Python as blocks/strings of CSV data
- 64 - same as 32, except that CSV data is created and passed instead of an array, only if there are any non-numeric/text cells within that GS-Calc range/array.

For details, please see the "Python integration" help topic.

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], [startIndex], [occurrence])

The **vLookup()** function searches for a value **v** in the leftmost column of an array. If a match is found, it returns the value from the same row and the **n-th** column of that array.

1. Parameters

- **v** — value to search for in the leftmost column.
- **array** — a 2-D array or range; the first column is searched.
- **n** — column index (within the array) from which the result is returned.
- **type** — search mode (exact, approximate, pattern, regex, sorted, etc.).
- **startIndex** — starting position for the search. Examples:

- 1 = first cell in the searched range
- 2 = second cell
- -1 = last cell
- -2 = second-to-last cell

Any non-zero integer can be used to specify where the search should start or end.

- **occurrence** — which occurrence of the matching value to use. Examples:

- 1 = first match (top-down)
- 2 = second match (top-down)
- -1 = last match (bottom-up)
- -2 = second-to-last match (bottom-up)

Any non-zero integer can be used to specify which occurrence (counted from **startIndex**) should be returned.

If **type** is omitted, it defaults to **1**.

2. Search Types and SEARCH:: Flags

2.1 Basic Types

- **0** — linear search for an exact match; **v** may be a pattern using **?** (any single character) and ***** (any string, including empty). To search for literal **?** or *****, prefix them with **~**.
- **1** — if no exact match is found, returns the largest value that is not greater than **v**; no pattern matching; the searched range must be sorted ascending.
- **-1** — if no exact match is found, returns the smallest value that is not smaller than **v**; no pattern matching; the searched range must be sorted descending.

2.2 SEARCH:: Flags

- **SEARCH::MatchNotGreater (2)** — largest value $\leq v$.
- **SEARCH::MatchNotSmaller (4)** — smallest value $\geq v$.
- **SEARCH::SortAscending (8)** — binary search on ascending sorted range.
- **SEARCH::SortDescending (16)** — binary search on descending sorted range.
- **SEARCH::CaseSensitive (128)** — case-sensitive comparison.
- **SEARCH::FirstMatch (256)** — return the first match.
- **SEARCH::LastMatch (512)** — return the last match.
- **SEARCH::AutoSort (1024)** — automatically sort internally during the first update and then use binary search. In this case, **startIndex** refers to the internally sorted range.
- **SEARCH::MixedData (2048)** — searched range contains both text and numbers.
- **SEARCH::SortIndex (4096)** — only with AutoSort; returns index relative to the internally sorted range.
- **SEARCH::RegEx (8192)** — **v** is a regular expression.

- **SEARCH::IgnorePunctuation (16384)** — ignore certain punctuation marks (word sort order).
- **SEARCH::NeutralSortOrder (32768)** — language-independent comparison.
- **SEARCH::Pattern (65536)** — **v** is a wildcard pattern; cannot be used with fast binary search.

2.3 Equivalences

- **1** = SortAscending + MatchNotGreater
- **-1** = SortDescending + MatchNotSmaller
- **0** = plain linear exact match

2.4 Restrictions

- SEARCH::Pattern and SEARCH::RegEx cannot be used with:
 - SEARCH::MatchNotGreater
 - SEARCH::MatchNotSmaller
 - SEARCH::StringSort
 - SEARCH::CaseSensitive
 - SEARCH::SortAscending
 - SEARCH::SortDescending
- SEARCH::MatchNotGreater and SEARCH::MatchNotSmaller cannot be used with SEARCH::FirstMatch or SEARCH::LastMatch.
- If neither SEARCH::FirstMatch nor SEARCH::LastMatch is specified:
 - linear search returns the first match
 - binary search may return any of the existing matches

2.5 Important Notes

If SEARCH::SortAscending or SEARCH::SortDescending is specified, the searched range must not contain formulas that break the sort order during recalculation. In such cases, circular references are not reported for cells other than the result cell.

Binary searches enabled by SortAscending or SortDescending are typically tens or hundreds of times faster than plain linear searches.

2.6 AutoSort

If SEARCH::AutoSort is specified, GS-Calc creates and maintains sort indices for unsorted ranges. These internal indices allow quick binary searches without requiring manual sorting. All indices are created during the first update and then updated as needed.

The speed gain depends on how many vLookUp() formulas are present, how often ranges are reused, and how large the searched ranges are. For very large worksheets, performance can improve by hundreds of times.

SEARCH::AutoSort must be used together with SEARCH::SortAscending or SEARCH::SortDescending. If the data is already partially sorted, use the sort order flag that best matches that partial sorting.

2.7 MixedData

If SEARCH::MixedData is specified, the searched range is assumed to contain both numeric and text cells:

- If **v** is text, numeric values are internally converted to text before comparison.
- If **v** is numeric, text strings representing numbers are converted to numbers.

If sorting flags are used with MixedData, the range must be sorted using a mixed text/numeric method. With AutoSort, GS-Calc handles this internally, but it may significantly slow down sorting. Otherwise, use the options in **Tools** → **Sort Cell Range**.

2.8 Numeric Codes

For workbooks containing extremely large numbers of vLookup() functions, you can use the numeric sum of flags instead of individual names to specify the **type** argument.

If no match is found, vLookup() returns the **#N/A!** error value.

3. Examples

=vLookup(2, {1, 2, 3; "a", "b", "c"}, 2, 0) → **"b"**

=vLookup(2.5, {1, 2, 3; "a", "b", "c"}, 2, -1) → **"c"**

=vLookup(2.5, {1, 2, 3; "a", "b", "c"}, 2, 1) → **"b"**

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

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

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

4. Inserting vLookup() via the Menu

You can insert the **vLookup()** function with two clicks using:

Insert → **VLOOKUP()**

The "Insert VLOOKUP()" dialog determines optimal parameters, presets all options, and automatically creates the formula.

The source range 16-element list is stored globally in the settings file. Subsequent source ranges are added in a circular manner using:

Copy as Location (menu command)

Enter (toolbar button)

vstack(value-or-reference1, value-or-reference2 [,value-or-reference...])

Merges the specified data items (values, cell or range references) vertically and returns the combined array. Columns without data are left empty.

[GS-Calc Help](#) > *Calculation Functions* > *Optimization/Solver Functions*

Function Index

[LProg](#) [LProgBin](#) [LProgInt](#) [QProg](#) [minMC](#) [minSimplex](#) [root](#)

Optimization/Solver Functions

LProg(A, s, b, c, vector, [epsilon], [BigM])

1. Purpose

Finds one or more solutions for a linear programming problem with **m** constraints of the form:

(1) $\sum_{j=1..n} (x[j] \cdot a[i,j]) \leq b[i]$

(2) $\sum_{j=1..n} (x[j] \cdot a[i,j]) = b[i]$

(3) $\sum_{j=1..n} (x[j] \cdot a[i,j]) \geq b[i]$

and a maximized objective function:

$$\sum_{j=1..n} (x[j] \cdot c[j])$$

2. Arguments

- **A** — m×n matrix of coefficients a[i,j]
- **b** — m-element column vector
- **c** — n-element column vector
- **s** — m-element column vector specifying constraint types:
 - s[i] = 1 → ≤
 - s[i] = 0 → =
 - s[i] = -1 → ≥
- **vector** — return mode:

- $0 \rightarrow$ number of optimum vectors found
- $i > 0 \rightarrow$ i-th optimum vector
- **epsilon** — precision threshold; coefficients with absolute value $< \text{epsilon}$ are treated as 0 (default: $1e-8$)
- **BigM** — large artificial-variable coefficient used in the Big-M method; if omitted, an appropriate Big-M value is chosen automatically

3. Behavior

- All returned $x[i]$ values are non-negative.
- If the model allows negative variables, each such variable must be represented by a pair of non-negative variables.
- To minimize the objective function, negate it and use the default maximization.
- If the objective function is unconstrained, LProg returns **#N/A!**.
- If the constraints are inconsistent and no solution exists, LProg returns **#NULL!**.

4. Example

Initial constraints:

$$2 \cdot x_1 + x_2 + 2 \cdot x_3 \leq 20$$

$$3 \cdot x_1 - x_2 + x_3 = 10$$

Objective function:

$$3 \cdot x_1 + x_2 - x_3 \rightarrow \max$$

Formula:

`=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], [BigM])

1. Purpose

Finds one or more solutions for a linear programming problem with constraints of the same form as LProg, where some or all variables $x[i]$ are required to be binary (0 or 1).

2. Arguments

- **A, b, c, s, epsilon, BigM** — as in LProg.
- **bin** — n-element column vector defining variable types:
 - $\text{bin}[i] = 1 \rightarrow x[i] \in \{0, 1\}$
 - $\text{bin}[i] = 0 \rightarrow x[i] \geq 0$
- **vector** — return mode:

- 0 → number of optimum vectors found
- $i > 0$ → i -th optimum vector

3. Behavior

- All returned $x[i]$ values are non-negative; binary variables are restricted to 0 or 1.
- To minimize the objective function, negate it and use the maximization procedure.
- If the objective function is unconstrained, LProgBin returns **#N/A!**.
- If the constraints are inconsistent and no solution exists, LProgBin returns **#NULL!**.

4. Example

Initial constraints:

$$2.5 \cdot x_1 + 1.5 \cdot x_2 + x_3 \leq 15$$

$$x_1 + 1.5 \cdot x_2 + 3 \cdot x_3 = 10$$

Objective function:

$$x_1 + x_2 + 2 \cdot x_3 \rightarrow \max$$

Formula:

=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], [BigM])

1. Purpose

Finds one or more solutions for a linear programming problem with constraints of the same form as LProg, where some or all variables $x[i]$ are required to be integers.

2. Arguments

- **A, b, c, s, epsilon, BigM** — as in LProg.
- **int** — n -element column vector defining variable types:
 - $\text{int}[i] = 1$ → $x[i]$ must be integer
 - $\text{int}[i] = 0$ → $x[i] \geq 0$
- **vector** — return mode:
 - 0 → number of optimum vectors found
 - $i > 0$ → i -th optimum vector

3. Behavior

- All returned $x[i]$ values are non-negative; selected variables must be integers.
- To minimize the objective function, negate it and use the maximization procedure.
- If the objective function is unconstrained, LProgInt returns **#N/A!**.
- If the constraints are inconsistent and no solution exists, LProgInt returns **#NULL!**.

4. Example

Initial constraints:

$$2.5 \cdot x_1 + 1.5 \cdot x_2 + x_3 \leq 15$$

$$x_1 + 1.5 \cdot x_2 + 3 \cdot x_3 = 10$$

Objective function:

$$x_1 + x_2 + 2 \cdot x_3 \rightarrow \max$$

Formula:

=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}

QProg(A, s, b, p, C, vector, [epsilon])

1. Purpose

Finds one or more solutions for a quadratic programming problem with **m** constraints:

$$(1) \sum_{j=1..n} (x[j] \cdot a[i,j]) \leq b[i]$$

$$(2) \sum_{j=1..n} (x[j] \cdot a[i,j]) = b[i]$$

$$(3) \sum_{j=1..n} (x[j] \cdot a[i,j]) \geq b[i]$$

and a maximized objective function:

$$p^T \cdot x - x^T \cdot C \cdot x$$

where **p** is an n-element column vector and **C** is a positive-definite n×n matrix representing a quadratic form.

2. Arguments

- **A** — m×n matrix of coefficients $a[i,j]$
- **b** — m-element column vector
- **p** — n-element column vector
- **C** — n×n positive-definite matrix
- **s** — m-element column vector specifying constraint types:

- $s[i] = 1 \rightarrow \leq$
- $s[i] = 0 \rightarrow =$
- $s[i] = -1 \rightarrow \geq$
- **vector** — return mode:
 - 0 → number of optimum vectors found
 - $i > 0 \rightarrow i$ -th optimum vector
- **epsilon** — precision threshold (default: $1e-8$)

3. Behavior

- All $x[i]$ are non-negative.
- If negative variables are needed, each must be represented by a pair of non-negative variables.
- To minimize the objective function, change its sign and use the maximization procedure.
- If the objective function is unconstrained, QProg returns **#N/A!**.
- If the constraints are inconsistent and no solution exists, QProg returns **#NULL!**.

4. Example

Constraints:

$$x_1 + 2 \cdot x_2 \leq 10$$

$$x_1 + x_2 \leq 9$$

$$x_1 \geq 0, x_2 \geq 0$$

Objective function:

$$f(x_1, x_2) = 10 \cdot x_1 + 25 \cdot x_2 - 10 \cdot x_1^2 - x_2^2 - 4 \cdot x_1 \cdot x_2 \rightarrow \max$$

Data:

$$A = \{1, 2; 1, 1\}$$

$$s = \{1; 1\}$$

$$b = \{10; 9\}$$

$$p = \{10; 25\}$$

$$C = \{10, 2; 2, 1\}$$

Formula:

$$= \text{QProg}(\{1, 2; 1, 1\}, \{1; 1\}, \{10; 9\}, \{10; 25\}, \{10, 2; 2, 1\}, 1,)$$

$$\text{returns } \{0; 5\}$$

minMC(f(x), x, v, from, to, points)

1. Purpose

Performs n-dimensional minimization of a given function using the Monte-Carlo method. The function generates a number of random points in n dimensions and returns the point for which f(x) is smallest. This method can be used to obtain an initial vector for **minSimplex()**.

2. Arguments

- **f(x)** — reference to a cell containing a numeric formula with n variables.
- **x** — reference to a one-column vector of n cells (numbers or numeric formulas) representing the variables in f(x).
- **v** — one-column vector indicating how to use each variable:
 - $v[i] = 0 \rightarrow$ i-th variable has a fixed value and is not modified
 - $v[i] = 1 \rightarrow$ i-th variable is modified
- **from, to** — search limits for all x[i].
- **points** — number of sample n-dimensional points to generate.

3. Notes and Errors

- Since points are generated randomly for each calculation, results change after each update.
- If f(x) is not a reference to a cell containing a valid numeric formula, or x is not a reference to n non-empty numeric cells, minMC returns **#REF!**.

4. Example (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}
```

5. Example (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)

1. Purpose

Performs n-dimensional minimization of a given function using the downhill simplex method.

2. Arguments

- **f(x)** — reference to a cell containing a numeric formula with n variables.
- **x** — reference to a one-column vector of n cells (numbers or numeric formulas) representing the variables in f(x). Initial values of x can be obtained with **minMC()**.
- **v** — one-column vector indicating how to use each variable:
 - $v[i] = 0$ → i-th variable has a fixed value and is not modified
 - $v[i] = 1$ → i-th variable will be modified
- **start** — initial simplex size; if 0 or omitted, default is $1e-5$.
- **epsilon** — precision used to evaluate changes in the function value; if 0 or omitted, default is $1e-25$.
- **max_steps** — maximum number of steps; default is 2000.
- **result** — return mode:
 - 1 → x vector for the found minimum
 - 2 → minimum f(x) value
 - 3 → number of steps performed

3. Errors

- If f(x) is not a reference to a valid numeric formula or x is not a reference to n non-empty numeric cells, minSimplex returns **#REF!**.
- If the procedure is not convergent after max_steps steps, the function returns **#NUM!**.

4. Example (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

5. Example (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

root(f(x), x, x0, x1, [epsilon])

1. Purpose

Finds a root of a one-dimensional equation $f(x) = 0$ using a modified secant method. The method is convergent if $f(x)$ is continuous on $[x0, x1]$ and $f(x0) \cdot f(x1) < 0$.

2. Arguments

- **f(x)** — reference to a cell containing a numeric formula with one variable.
- **x** — single cell reference representing the variable.
- **x0, x1** — initial search limits such that $f(x0) \cdot f(x1) < 0$.
- **epsilon** — precision used to evaluate changes in the function value; if 0 or omitted, default is $1e-15$.

3. Errors

- If $f(x)$ is not a reference to a valid numeric formula or x is not a reference to a numeric cell, root returns **#REF!**.

4. Example

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

[GS-Calc Help](#) > Calculation Functions > Financial Functions

Function Index

cumIPMT cumPRINC db ddb dollarDe dollarFr effect fv fvSchedule ipmt
irr mirr nominal nper npv pmt ppmt pv rate sln syd xirr xnpv

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) → -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) → -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) → 185912.959716189

=db(1000000, 100000, 6, 7, 7) → 15867.888384391

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) → 1.31506849315068

=ddb(2400, 300, 10, 2, 1.5) → 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) → 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) → 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) → 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 type specifies the payment type: 0 – at the end of each period, 1 – at the beginning of each period. If pmt is omitted, pv must be specified. If pv is omitted, it is assumed to be 0 and pmt must be specified. The default value of type is 0.

=fV(0.5%, 10, -200, -500, 1) → 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}) → 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 is assumed to be 0.

=ipmt(0.1/12, 1, 36, 8000) → -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 be 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%) → 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%) → 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) → 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 is assumed to be 0. The default value of type is 0.

=nper(12%/12, -100, -1000, 10000, 1) → 59.6738656742946

=nper(1%, -100, 1000) → 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) → 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 is assumed to be 0. The default value of type is 0.

=pmt(8%/12, 10, 10000) → -1037.03208935916
=pmt(8%/12, 10, 10000, 0, 1) → -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 is assumed to be 0. The default value of type is 0.

=ppmt(10%/12, 1, 24, 2000) → -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 is assumed to be 0. The default value of type is 0.

=pv(0.08/12, 12*20, 500, , 0) → -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. If fv is omitted, it is assumed to be 0. The default value of type is 0.

=rate(48, -200, 8000) → 0.0077014724882

sln(cost, salvage, life)

Returns the straight-line depreciation of an asset for one period. The cost argument is initial value 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) → 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) → 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 order 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) → 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 order 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"}) → 2086.64760205153

[GS-Calc Help](#) > *Calculation Functions* > *Output Functions*

Function Index

[imagelf](#) [messagelf](#) [playSoundlf](#) [setFormatlf](#)

Output Functions

imagelf(if_condition, path_true, path_false, type)

Displays an image in a cell (overflowing cell boundaries if necessary).

if_condition:

If non-zero, **path_true** is displayed; otherwise **path_false**.

path_true / path_false:

Image file paths or names.

- If **type = 1**: paths refer to external disk files (e.g., **e:\image_file1.png**).
- If **type = 0**: names refer to images inserted into the workbook using **Insert > Image**.

The list of inserted images can be managed via the **Set Cell Background Image** toolbar button or the **Cell Style Properties** dialog.

type: Determines how paths are interpreted.

Images are drawn immediately after formula evaluation is complete.

=imagelf(P251 > 0, "e:\image_file1.png", "e:\image_file2.png", 1)

=imagelf(S10:S120 = 0, "folder1\music_file1.jpg", "image_file2.gif", 0)

messagelf(message, type, if_condition, reference-as-text)

Displays a message box with **OK** and **Cancel**.

message: Text to display.

type:

- 1 — information icon
- 2 — warning icon
- 3 — question icon

if_condition:

Message appears only if the condition evaluates to non-zero.

reference (as text):

Optional full worksheet path. If provided and you click **OK**, GS-Calc jumps to the referenced cell/range.

Messages appear immediately after formula evaluation is complete. You can use

if_condition = 1 to signal the end of updating.

```
=messageIf("need to inspect this cell", 1, isError(folder2\sheet2!P251),  
"folder2\sheet2!P251")
```

```
=messageIf("need to inspect this cells", 1, isError(S120), "S121:S200")
```

playSoundIf(path, type, if_condition, loops, volume)

Plays MP3 and WAV files.

path:

File path.

- If **type = 1**: external disk file (e.g., **e:\music_file1.mp3**).
- If **type = 0**: full worksheet path of a file inserted via **Insert > File**.

type: Determines how the path is interpreted.

if_condition: File plays only if non-zero.

loops:

Number of times to play. **-1** = endless loop.

volume:

Value from 0 to 1000.

Files play immediately after formula evaluation is complete. Use **if_condition = 1** to signal the end of updating.

If multiple files are queued, they play sequentially.

To stop playback, press **ESC** or use **Tools > Cancel Update**.

```
=playSoundIf("e:\music_file1.mp3", 1, true, 1, 40)
```

=playSoundIf("folder1\music_file1.mp3", 0, true, 100, 80)

setFormatIf(reference, action, if_condition, format_true, format_false)

This function formats a cell or range of cells. The numeric **action** parameter specifies which formatting attributes to set or clear (restore defaults).

It is often convenient to keep formatting formulas in a separate worksheet so a single control cell can affect formatting across multiple sheets.

The function works on both existing cells and empty regions. There are no limits on the number or complexity of conditional formatting operations — millions of cells may be formatted using nested IFs or other formulas.

Formatting Layers

If **reference** specifies entire columns or rows, formatting applies to column or row styles. Formatting precedence:

table style < column styles < row styles < cell styles

Cell styles override all other styles.

format_true / format_false Values

Values may be **-1** (restore default) or one of the following:

- 1 — numeric style (index or name)
- 2 — custom style pattern (e.g., **\$#,##0.00**)
- 3 — font size (4–256)
- 4 — font name
- 6 — bold (0/1)
- 7 — italic (0/1)
- 8 — underline (0/1)
- 9 — strikethrough (0/1)
- 11 — font color (RGB value, “#RRGGBB”, or color name)
- 14 — horizontal alignment (0 left, 1 center, 2 right)
- 15 — vertical alignment (0 top, 1 center, 2 bottom)
- 16 — wrap text (0/1)
- 17 — horizontal indent (0–64 px)
- 18 — vertical indent (0–64 px)
- 19 — text rotation (–180 to 180 degrees)
- 20 — cell borders (“position width style color”)
- 21 — borders around selection (“width style color”)
- 22 — background color (RGB value, “#RRGGBB”, or color name)
- 25 — print / don't print (0/1)
- 26 — hyperlink (0/1)
- 27 — filter cell (0/1)
- 28 — image list cell (0/1)

- 29 — area in-cell chart (0/1)
- 30 — column in-cell chart (0/1)
- 31 — bar in-cell chart (0/1)
- 32 — apply all attributes at once using a named custom style

Custom Cell Styles

action = 32 applies all formatting attributes using a named custom style. Custom styles are created in **Format > Custom Cell Styles** and saved in the workbook.

They may also be saved to or loaded from the global GS-Calc cache using:

- **Save As Default App Styles**
- **Load Default App Styles**

Border Formatting (action = 20)

position may include:

- top
- left
- bottom
- right
- diag-left
- diag-right

width may be:

- thin, medium, thick
- numeric values with px, pt, in, mm, cm, pc

style may be:

- solid
- dotted
- dash
- long-dash
- dot-dash
- dot-dot-dash
- wave
- double

color may be an RGB value, “#RRGGBB”, or one of:

- black
- maroon
- green
- olive
- navy
- purple
- teal
- gray

- silver
- red
- lime
- yellow
- blue
- fuchsia
- aqua
- white

Return Value

The function returns the evaluated value of **if_condition**.

=setFormatIf(c120, 11, c120 > 0, "green", "red")

=setFormatIf(c:e, 11, c120 > 0, "green", "red")

=setFormatIf(10:20, 11, c120 > 0, "green", "red")

=setFormatIf(c120, 20, c120 > 0, "all 2px solid green", "diag-left diag-right red")

=setFormatIf(d100:d999, 1, c1="use format", "currency", 0)

=setFormatIf(d99, 20, a1, "bottom 2px dotted green", -1)

=setFormatIf(d99, 32, a10 > 10, "my-style-1", "my-style-2")

[GS-Calc Help](#) > *Calculation Functions* > *Engineering Functions*

Function Index

[besselJ](#) [besselY](#) [bin2dec](#) [bin2hex](#) [bin2oct](#) [complex](#) [dec2bin](#) [dec2hex](#)
[dec2oct](#) [delta](#) [erf](#) [ERF.PRECISE](#) [erfc](#) [ERFC.PRECISE](#) [getStep](#) [hex2bin](#)
[hex2dec](#) [hex2oct](#) [imAbs](#) [imaginary](#) [imArgument](#) [imConjugate](#) [imCos](#) [imDiv](#)
[imExp](#) [imLn](#) [imLog10](#) [imLog2](#) [imPower](#) [imProduct](#) [imReal](#) [imSin](#) [imSqrt](#)
[imSub](#) [imSum](#) [oct2bin](#) [oct2dec](#) [oct2hex](#)

Engineering Functions

besselJ(x, n)

Returns the Bessel function of the n-th order for a given value **x**.

=besselJ(1.9, 2) → 0.32992572769239

besselY(x, n)

Returns the Bessel/Weber function of the n-th order for a given value **x**.

=besselY(2.5, 1) → 0.14591813796679

bin2dec(n)

Converts **n** representing a binary number to a decimal number. The argument may contain up to 10 characters. The most significant bit determines the sign. Negative numbers use two's complement.

=bin2dec("1100100") → 100

=bin2dec("111111111") → -1

bin2hex(n, [l])

Converts a binary number **n** to a hexadecimal string of length **l**. may contain up to 10 characters. The most significant bit determines the sign. Negative numbers use two's complement. If **l** is omitted, it defaults to 10.

=bin2hex(11111011, 4) → "00FB"

=bin2hex("111111111") → "FFFFFFFF"

bin2oct(n, [l])

Converts a binary number **n** to an octal string of length **l**. may contain up to 10 characters. The most significant bit determines the sign. Negative numbers use two's complement. If **l** is omitted, it defaults to 10.

=bin2oct(1001, 3) → "011"

=bin2oct(111111111) → "777777777"

complex(x, y, [i])

Converts real and imaginary coefficients into a complex number string. The optional **i** argument specifies the imaginary suffix (default: "i").

`=complex(1, 2) → "1 + 2i"`
`=complex(3.5, 4.6, "j") → "3.5 + 4.6j"`

dec2bin(n, [l])

Converts a decimal number **n** (range: -512 to 511) to a binary string of length **l**. Negative numbers use two's complement. If **l** is omitted, it defaults to 10.

`=dec2bin(9, 4) → "1001"`
`=dec2bin(-1) → "1111111111"`

dec2hex(n, [l])

Converts a decimal number **n** to a hexadecimal string of length **l**. Negative numbers use two's complement. If **l** is omitted, it defaults to 10.

`=dec2hex(100, 4) → "064"`
`=dec2hex(-1) → "FFFFFFFFF"`

dec2oct(n, [l])

Converts a decimal number **n** (range: -536870912 to 536870911) to an octal string of length **l**. Negative numbers use two's complement. If **l** is omitted, it defaults to 10.

`=dec2oct(58, 3) → "1001"`
`=dec2oct(-1) → "7777777777"`

delta(x, [y])

Returns **1** if **x = y**, otherwise **0**. If **y** is omitted, it defaults to 0.

`=delta(12, 11) → 0`
`=delta(0) → 1`

erf(x1, [x2])

Returns the value of the error function integrated between the specified limits. If **x2** is omitted, integration is performed between 0 and **x1**.

`=erf(0.745) → 0.70792891807065`

ERF.PRECISE(x1, x2)

Returns the error function evaluated over the interval from **x1** to **x2**.

The returned value is:

$\text{ERF}(x2) - \text{ERF}(x1)$

Both **x1** and **x2** must be numeric. If **x2** is omitted, the function returns $\text{ERF}(x1)$.

Examples:

$\text{ERF.PRECISE}(0, 1)$ returns $\text{ERF}(1) - \text{ERF}(0)$.

$\text{ERF.PRECISE}(1, 2)$ returns $\text{ERF}(2) - \text{ERF}(1)$.

$\text{ERF.PRECISE}(1)$ returns $\text{ERF}(1)$.

erfc(x)

Returns the complementary error function integrated between **x** and infinity.

$=\text{erfc}(1) \rightarrow 0.15722921001143$

ERFC.PRECISE(x1, x2)

Returns the complementary error function evaluated over the interval from **x1** to **x2**.

The returned value is:

$\text{ERFC}(x1) - \text{ERFC}(x2)$

Both **x1** and **x2** must be numeric. If **x2** is omitted, the function returns $\text{ERFC}(x1)$.

Examples:

$\text{ERFC.PRECISE}(0, 1)$ returns $\text{ERFC}(0) - \text{ERFC}(1)$.

$\text{ERFC.PRECISE}(1, 2)$ returns $\text{ERFC}(1) - \text{ERFC}(2)$.

$\text{ERFC.PRECISE}(1)$ returns $\text{ERFC}(1)$.

getStep(x, [y])

Returns **1** if $x > y$, otherwise **0**. Default value of **y** is 0.

$=\text{getStep}(1.4) \rightarrow 1$

hex2bin(n, [l])

Converts a hexadecimal number **n** to a binary string of length **l**. may contain up to 10 characters. Negative numbers use two's complement. If **l** is omitted, it defaults to 10.

`=hex2bin("F", 8) → "00001111"`

`=hex2bin("FFFFFFFF") → "1111111111"`

hex2dec(n)

Converts a hexadecimal number **n** to a decimal number. may contain up to 10 characters. Negative numbers use two's complement.

`=hex2dec("A5") → 165`

`=hex2dec("FFFFFFFF") → -1`

hex2oct(n, [l])

Converts a hexadecimal number **n** to an octal string of length **l**. may contain up to 10 characters. Negative numbers use two's complement. If **l** is omitted, it defaults to 10.

`=hex2oct("F", 3) → "017"`

`=hex2oct("FFFFFFFF") → "7777777777"`

imAbs(z)

Returns the absolute value of a complex number **z**.

`=imAbs("5 + 12i") → 13`

imaginary(z)

Returns the imaginary coefficient of a complex number **z**.

`=imaginary("5 + 12i") → 12`

imArgument(z)

Returns the argument (angle) **t** such that:

$z = |z|(\cos(t) + \sin(t)i)$.

`=imArgument("3 + 4i") → 0.92727521800161`

imConjugate(z)

Returns the complex conjugate of **z**.

=imConjugate("3 + 4i") → "3 - 4i"

imCos(z)

Returns the cosine of a complex number **z**.

=imCos("1 + 1i") → "0.83373002513115 + 0.98889770576287i"

imDiv(z1, z2)

Divides **z1** by **z2** and returns the result.

=imDiv("1 + 2i", "2 + 1i") → "0.8 + 0.6i"

imExp(z)

Returns e^z for a complex number **z**.

=imExp("1 + 1i") → "1.43869393991589 + 2.28735528717884i"

imLn(z)

Returns the natural logarithm of a complex number **z**.

=imLn("3 + 4i") → "1.6094379124341 + 0.92729521800161i"

imLog10(z)

Returns the base-10 logarithm of a complex number **z**.

=imLog10("3+4i") → "0.69897000433601 + 0.40271919627337i"

imLog2(z)

Returns the base-2 logarithm of a complex number **z**.

=imLog2("3 + 4i") → "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 square 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 '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 "FFFFFFFF"

Function Index

[mtxBkSubst](#) [mtxChl](#) [mtxDiag](#) [mtxFwSubst](#) [mtxGauss](#) [mtxLSC](#) [mtxLSW](#) [mtxMlt](#)
[mtxRand](#) [mtxRand2](#) [mtxSeries](#) [mtxSVD](#) [mtxT](#)

Matrix and Equation System Functions

mtxBkSubst(U, B)

Performs backsubstitution for a given upper triangular $n \times n$ **U** matrix and an $n \times I$ **B** matrix containing **I** vectors of right-hand values.

The **mtxBkSubst** function returns **I** column vectors that are—typically—solutions of the equation $L \cdot U \cdot x = B$.

Example:

```
=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 $n \times n$ **A** matrix ($x^T \cdot A \cdot x > 0$ for each $x \neq 0$). The **mtxChl** function can be used to:

1. solve a linear equation set $A \cdot x = B$, where **A** is an $n \times n$ matrix, **x** is an **n**-element column vector and **B** is an $n \times I$ matrix,
2. find the upper triangular **U** matrix such that $U^T \cdot U = A$,
3. find the inverse and determinant of **A**.

The **A** and **B** arguments specify the matrices used in the $A \cdot x = B$ equation. The **B** argument is used only when **result** = **1**. If **B** has **I** columns ($I > 1$), the procedure will use each column as different right-hand values and it will return **I** solution (column) vectors.

The **iterations** 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 function returns **I** solution vectors of the $A \cdot x = B$ equation system.

If **result** = **2**, the function returns an upper triangular $n \times n$ **U** matrix such that $U^T \cdot U = A$.

If **result = 3**, the function returns a transposed upper triangular $n \times n$ U^T matrix such that $U^T \cdot U = A$.

If **result = 4**, the function returns the inverse of **A**.

If **result = 5**, the function returns the determinant of **A**.

Examples:

```
=mtxChl({1, 0.5, 0.3333333; 0.5, 0.3333333, 0.25; 0.3333333, 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; 0.3333333, 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 an $n \times n$ 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.

Examples:

```
=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 $n \times n$ **L** matrix and an $n \times I$ **B** matrix containing **I** vectors of right-hand values.

The **mtxFwSubst** function returns **I** (column) vectors.

Example:

```
=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 $A \cdot x = B$, where **A** is an $n \times n$ matrix, **x** is an **n**-row column vector and **B** is an $n \times I$ 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 $\mathbf{A} \cdot \mathbf{x} = \mathbf{B}$ equation. The **B** argument is used only when **result** = 1. If **B** has **I** columns ($I > 1$), the procedure will use each column as different right-hand values and it will return **I** 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** = 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 $\mathbf{A} \cdot \mathbf{x} = \mathbf{B}$ and returns **I** solution vectors, where **I** is the number of columns in **B**.

If **result** = 2, the function returns an $n \times n$ lower triangular matrix **L** such that $\mathbf{L} \cdot \mathbf{U} = \mathbf{A}'$ where **A'** is a row-wise permutation of **A**.

If **result** = 3, the function returns an $n \times n$ upper triangular matrix **U** such that $\mathbf{L} \cdot \mathbf{U} = \mathbf{A}'$ where **A'** is a row-wise permutation of **A**.

If **result** = 4, the function returns an $n \times n$ permutation matrix **P** such that $\mathbf{P} \cdot \mathbf{A} = \mathbf{A}'$ 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**.

Initial equation system:

$$2 \cdot x_1 - x_2 + 3 \cdot x_3 = 6$$

$$1 \cdot x_1 + 3 \cdot x_2 + x_3 = 8$$

$$4 \cdot x_1 + x_2 + x_3 = 2$$

Examples:

=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 $\mathbf{A} \cdot \mathbf{x} \approx \mathbf{b}$ with constraints in the form of $\mathbf{E} \cdot \mathbf{x} = \mathbf{d}$, where **E** is an $I \times n$ matrix and $I < n$.

The **A** argument is an $m \times n$ matrix and the **b** argument is a column vector with **m** rows. The **E** argument is an $I \times n$ matrix and the **d** argument is a column vector with **I** 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** = 1, the function returns the **n**-row column vector **x**.

If **result** = 2, the function returns the rest value **r** such that $(\mathbf{A} \cdot \mathbf{x} - \mathbf{b})^T \cdot (\mathbf{A} \cdot \mathbf{x} - \mathbf{b}) = \mathbf{r}$.

If the SVD decomposition is not convergent, the **#NUM!** error value is returned.

Examples:

=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 $\mathbf{A} \cdot \mathbf{x} \approx \mathbf{b}$ with weights specified by the **G** matrix so that the minimized function has the form:

$$\mathbf{r} = (\mathbf{A} \cdot \mathbf{x} - \mathbf{b})^T \cdot \mathbf{G} \cdot (\mathbf{A} \cdot \mathbf{x} - \mathbf{b}) \rightarrow \min$$

The **A** argument is an **m** × **n** matrix and the **b** argument is a column vector with **m** rows.

The **G** argument is an **n** × **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** = 1, the function returns the **n**-row column vector **x**.

If **result** = 2, the function returns the rest value **r** such that $(\mathbf{A} \cdot \mathbf{x} - \mathbf{b})^T \cdot \mathbf{G} \cdot (\mathbf{A} \cdot \mathbf{x} - \mathbf{b}) = \mathbf{r}$.

If the SVD decomposition is not convergent, the **#NUM!** error value is returned.

Examples:

=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.

Example:

=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 **2.3×10¹⁸** 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 start 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.

Example:

```
=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 start 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.

Example:

```
=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

Examples:

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.

Examples:

```
=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 $\mathbf{A} \cdot \mathbf{x} \approx \mathbf{B}$ where $\mathbf{B}$ is an $\mathbf{m} \times \mathbf{l}$ matrix containing $\mathbf{l}$ right-hand values vectors,
2. find the number of non-zero singular values,
3. find $\mathbf{l}$ rest values $\mathbf{r}[\mathbf{i}]$ such that $(\mathbf{A} \cdot \mathbf{x} - \mathbf{B}_i)^T \cdot (\mathbf{A} \cdot \mathbf{x} - \mathbf{B}_i) = \mathbf{r}[\mathbf{i}]$ where $\mathbf{B}_i$ is the $\mathbf{i}$ -th column of $\mathbf{B}$,
4. find the (pseudo-)inverse $\mathbf{A}^+$ of $\mathbf{A}$,
5. find the $\mathbf{U}[\mathbf{m},\mathbf{n}]$ matrix, $\mathbf{S}[\mathbf{m},\mathbf{n}]$ matrix and $\mathbf{V}[\mathbf{n},\mathbf{n}]$ matrix such that $\mathbf{U} \cdot \mathbf{S} \cdot \mathbf{V}^T = \mathbf{A}$.

The $\mathbf{A}$ and $\mathbf{B}$ arguments are matrices used in the equation system $\mathbf{A} \cdot \mathbf{x} \approx \mathbf{B}$. The $\mathbf{B}$ argument is ignored for **result** values 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 $\mathbf{l}$ $\mathbf{n}$ -row column vectors that are solutions of $\mathbf{A} \cdot \mathbf{x} \approx \mathbf{B}$.

If **result** = **2**, the function returns a column vector of $\mathbf{l}$ rest values.

If **result** = **3**, the function returns the number of non-zero singular values.

If **result** = **4**, the function returns the inverse $\mathbf{A}^+$ of $\mathbf{A}$.

If **result** = **5**, **6** or **7**, the function returns respectively the $\mathbf{U}$, $\mathbf{S}$ and $\mathbf{V}$ matrices described above.

If the SVD decomposition is not convergent, the **#NUM!** error value is returned.

Examples:

=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)

Transposes a given matrix and returns the result.

Example:

=mtxT({1, 2, 3; 4, 5, 6}) returns {1, 4; 2, 5; 3, 6}

Using Regular Expressions in Formulas

1. Overview

Regular expressions can be used in:

- Find() and Replace()
- Lookup and matching formulas: Match(), vLookup(), hLookup()

To enable regular expressions, specify the **SEARCH::RegEx** flag (value 8192) as described in the **Formula Composer** window.

In Find(), you may also use **SEARCH::RegExStr**. If used, the function returns the matching substring instead of the index.

Examples:

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

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

2. Returning Capturing Groups

You can specify which capturing group should be returned:

- 32 — first group
- 64 — second group
- 128 — third group

Useful for extracting file components:

Extension:

=find("(\\.[^.]*)\$", "c:\file.txt", 1, 32+SEARCH::RegExStr)

Folder:

=find("(^.*[\\W])?(\\.*.*?)(\\.[^.]*)\$", "c:\folder1\file.txt", 1, 32+SEARCH::RegExStr)

File name without extension:

=find("(^.*[\\W])?(\\.*.*?)(\\.[^.]*)\$", "c:\file.txt", 1, 64+SEARCH::RegExStr)

3. Regular Expression Examples

abc	Cells containing "abc"
.bc	Any character + "bc"
\Aabc	Starts with "abc"
abc\z	Ends with "abc"
\Aabc.*123\z	Starts with "abc", ends with "123"
abc\d\z	Ends with "abc" + digit
^a\d+	Line starting with "a" + digits
^a\d*	Line starting with "a" + optional digits

a\d*\$	Line ending with “a” + optional digits
[ab]+c	“abc”, “aabc”, “abbabc”... but not “c” alone
[ab]*c	Same as above, including “c”
^[^ab]+c	Ends with “c”, contains no “a” or “b”
\w\d{2,3}	Letter + 2–3 digits
^ab\d{2,3}c	Starts with “ab”, 2–3 digits, “c”
abc xyz	Contains “abc” or “xyz”

See [PCRE regular expression syntax summary](#).

4. Replace() with Regular Expressions

If **SEARCH::RegEx** is set, the replace string may contain:

1. Capturing group references: \1, \2, ...
2. \l, \L, \u, \U — case conversion
3. \r, \n — line feed / newline
4. \s — space

5. Examples

Example 1

Find:	ab+
Content:	abcdefaabb
Replace:	x
Result:	xcdefax

Example 2

Find:	(.)a+\d{1,3}
Content:	abc aa0102
Replace:	\1
Result:	abc 2

Example 3

Find:	(ab)
Content:	abcdef ghijk abb123
Replace:	\u\1\\1xyz\s
Result:	ABabxyz cdef ghijk ABabxyz b123

Example 4

Find:	\R
Content:	abc def ghi
Replace:	\s
Result:	abc def ghi

Find & Replace — Full-Text Searches and Replacing

1. Overview

To perform full-text searching, use **Edit > Find (F3)**. Enter the search text in the **Find** field of the **Find & Replace** toolbar. Click **Options** to choose whether the search text is treated as a regular expression or as plain text.

Selecting a range limits searching and replacing to that range. Clicking any cell cancels the active range and searching resumes across the entire worksheet. Changing search/replace options or text strings starts a new search.

2. What Is Searched

- Text and numeric cells — their unformatted contents.
- Formulas — the formula text itself.

3. Search Modes

3.1 Regular Expressions

The **Find** string is interpreted as a regular expression. Examples:

abc	Cells containing “abc”
.bc	Any character followed by “bc”

\Aabc	Starts with “abc”
abc\z	Ends with “abc”
\Aabc.*123\z	Starts with “abc”, ends with “123”
abc\d\z	Ends with “abc” + digit
^a\d+	Line starting with “a” + digits
^a\d*	Line starting with “a” + optional digits
a\d*\$	Line ending with “a” + optional digits
[ab]+c	“abc”, “aabc”, “abbabc”... but not “c” alone
[ab]*c	Same as above, including “c”
[^ab]+c	Ends with “c”, contains no “a” or “b”
\w\d{2,3}	Letter + 2–3 digits
^ab\d{2,3}c	Starts with “ab”, 2–3 digits, “c”
abc xyz	Contains “abc” or “xyz”
(?=.*abc)(?=.*xyz)	Contains both “abc” and “xyz”
\A\z	Empty cells

Additional options:

- **Allow empty matches**
- **Match case**

Empty matches occur for patterns like **a?** or **\A\z**, allowing searches for empty cells within data blocks.

See [PCRE regular expression syntax summary](#).

Replace String Features

- \1, \2... — capturing group references
- \l, \L, \u, \U — case conversion
- \r, \n — line feed / newline (Ctrl+Enter inserts **\r\n**)
- \s — space

Notes:

- **To find/replace Unicode or non-ASCII characters, use Unicode regex classes such as **\p{L}** instead of **\w**, and **\P{L}** instead of **\W**.**
- **Matching empty cells requires enabling Options > Allow Empty Matches.**

The “RegEx” menu in GS-Base inserts regex tags easily.

The screenshot shows the 'RegEx' menu in GS-Base. The menu is open, displaying a list of tags and their descriptions. The background shows a spreadsheet interface with columns M, N, O, P, and Q.

Tag	Description
\	Quote a single control character
\Q...\E	Quote selection
\A	Match the beginning of a field
\Z	Match the ending of a field
\A...\Z	Match the entire (also empty) field
.	Any ASCII character
\w	Any word character
\W	Non-word character
\d	Decimal digit
\D	Any character except decimal digits
\s	Whitespace character
\S	Character that is not whitespace
\xhh	Character with hex code hh
\x{hh...}	Character with hex code hh...
\b	Word boundary
\B	Not a word boundary
\n	New line
\R	New line + carriage return
Properties and Unicode characters	
(...)	Group characters
...?	0 or 1 character or group
...*	0 or more characters or groups
...+	1 or more characters or group
...{n}	n characters or groups
...{n,m}	From n to m characters or groups
...{n,}	At least n characters or groups
expr expr expr...	Logical OR of expressions
[...]	Character must belong to the list
[^...]	Character must not belong to the list
[x-y]	Character in the ASCII or hex range
(?# ...)	Comments
\p	Character with the specified property
\P	Character without the specified property
\p{L}	Letter
\p{Lm}	Modifying Letter
\p{LI}	Lower case Letter
\p{Lu}	Upper case Letter
\p{N}	Digit
\p{Nd}	Decimal digit
\p{NI}	Numeral letter
\p{M}	Modifying mark: e.g. accent
\p{Ms}	Spacing mark
\p{Me}	Enclosing mark
\p{Mn}	Non-spacing mark
\p{S}	Symbol
\p{Sc}	Currency Symbol
\p{Sk}	Modifying symbol
\p{Sm}	Mathematical symbol
\p{P}	Any punctuation
\p{Ps}	Opening bracket
\p{Pe}	Closing bracket
\p{Pd}	Dash
\p{Pi}	Initial quote
\p{Pf}	Ending quote
\p{Pc}	Connector: e.g. underscore
\p{Z}	Separator

4. Examples

Find:	cat
Replace:	dog
Result:	Replaces “cat” with “dog”

Find:	\\
Replace:	/
Result:	Replaces “\” with “/”

Find:	\A\z
Replace:	NULL
Result:	Fills empty fields with “NULL”

Find:	\ANULL\z
Replace:	
Result:	Deletes fields containing only “NULL”

Find:	\b(\w+)(?:\W+\1\b)+
Replace:	\1
Result:	Removes duplicate words

Find:	(\b\w)(\w*(\W+ \z))
Replace:	\u\1\2
Result:	Capitalizes first letters

Find:	(\b\w)(\w*(\W+ \z))
Replace:	\1
Result:	Creates abbreviations

Find:	(\b\w(\d*))(\w*(\W+ \z))
Replace:	\1
Result:	Abbreviations; numbers preserved

Find:	\b((\d{1,3}\.){3})\d{1,3}\b
Replace:	\1*
Result:	Masks IP addresses

Find:	(\S*)(\s*)(\S*)(\s*)(\S*)
--------------	---------------------------

Replace:	\g5\g4\g3\g2\g1
Result:	Reverses first three words

Find:	\R
Replace:	\s
Result:	Replaces line breaks with spaces

Find:	ab+
Database field content:	abcdefaabb
Replace:	x
Result:	xcdefax

Find:	(.)a+\d{1,3}
Database field content:	abc aa0102
Replace:	\1
Result:	abc 2

Find:	(ab)
Database field content:	abcdef ghijk abb123
Replace:	\u\1\\1xyz\s
Result:	ABabxyz cdef ghijk ABabxyz b123

Find:	\R
Database field content:	abc def ghi
Replace:	\s
Result:	abc def ghi

3.2 Plain Text — Partial Matching

Searches for substrings. Supports wildcards **?** (any character) and ***** (any string). Literal “?” or “*” must be prefixed with **~**.

3.3 Plain Text — Full Content Matching

Matches the entire cell contents. Same wildcard rules as above.

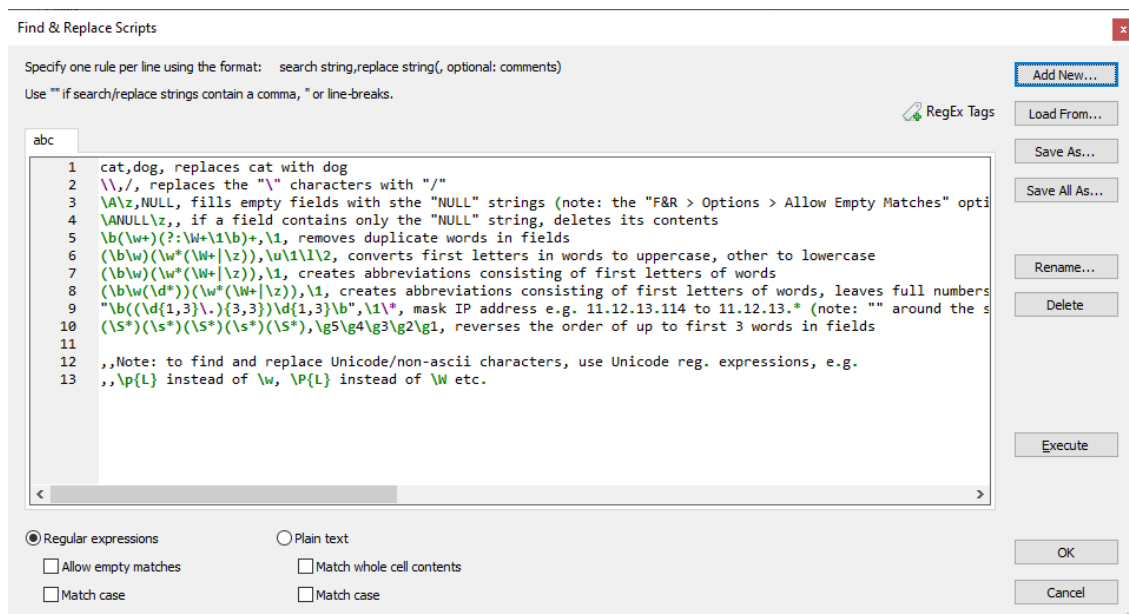
3.4 Scripts

The **Scripts** button performs quick mass text replacements in a table.

Note:

The “search” and “replace” strings must follow CSV quoting rules:

- Text containing commas, line breaks, or quotes must be enclosed in "".
- Inner quotes must be doubled.



pcresyntax man page

Return to the [PCRE index page](#).

This page is part of the PCRE HTML documentation. It was generated automatically from the original man page. If there is any nonsense in it, please consult the man page, in case the conversion went wrong.

- [PCRE REGULAR EXPRESSION SYNTAX SUMMARY](#)
- [QUOTING](#)
- [CHARACTERS](#)
- [CHARACTER TYPES](#)
- [GENERAL CATEGORY PROPERTIES FOR \p and \P](#)
- [PCRE SPECIAL CATEGORY PROPERTIES FOR \p and \P](#)
- [SCRIPT NAMES FOR \p AND \P](#)
- [CHARACTER CLASSES](#)
- [QUANTIFIERS](#)

- [ANCHORS AND SIMPLE ASSERTIONS](#)
- [MATCH POINT RESET](#)
- [ALTERNATION](#)
- [CAPTURING](#)
- [ATOMIC GROUPS](#)
- [COMMENT](#)
- [OPTION SETTING](#)
- [LOOKAHEAD AND LOOKBEHIND ASSERTIONS](#)
- [BACKREFERENCES](#)
- [SUBROUTINE REFERENCES \(POSSIBLY RECURSIVE\)](#)
- [CONDITIONAL PATTERNS](#)
- [BACKTRACKING CONTROL](#)
- [NEWLINE CONVENTIONS](#)
- [WHAT \R MATCHES](#)
- [CALLOUTS](#)
- [SEE ALSO](#)
- [AUTHOR](#)
- [REVISION](#)

[PCRE REGULAR EXPRESSION SYNTAX SUMMARY](#)

The full syntax and semantics of the regular expressions that are supported by PCRE are described in the [pcrepattern](#) documentation. This document contains a quick-reference summary of the syntax.

[QUOTING](#)

<code>\x</code>	where x is non-alphanumeric is a literal x
<code>\Q...\E</code>	treat enclosed characters as literal

[CHARACTERS](#)

<code>\a</code>	alarm, that is, the BEL character (hex 07)
<code>\cx</code>	"control-x", where x is any ASCII character
<code>\e</code>	escape (hex 1B)
<code>\f</code>	form feed (hex 0C)
<code>\n</code>	newline (hex 0A)
<code>\r</code>	carriage return (hex 0D)
<code>\t</code>	tab (hex 09)
<code>\ddd</code>	character with octal code ddd, or backreference
<code>\xhh</code>	character with hex code hh
<code>\x{hhh..}</code>	character with hex code hhh..

[CHARACTER TYPES](#)

<code>.</code>	any character except newline; in dotall mode, any character whatsoever
<code>\C</code>	one data unit, even in UTF mode (best avoided)
<code>\d</code>	a decimal digit
<code>\D</code>	a character that is not a decimal digit
<code>\h</code>	a horizontal white space character
<code>\H</code>	a character that is not a horizontal white space character
<code>\N</code>	a character that is not a newline

<code>\p{xx}</code>	a character with the xx property
<code>\P{xx}</code>	a character without the xx property
<code>\R</code>	a newline sequence
<code>\s</code>	a white space character
<code>\S</code>	a character that is not a white space character
<code>\v</code>	a vertical white space character
<code>\V</code>	a character that is not a vertical white space character
<code>\w</code>	a "word" character
<code>\W</code>	a "non-word" character
<code>\X</code>	an extended Unicode sequence

In PCRE, by default, `\d`, `\D`, `\s`, `\S`, `\w`, and `\W` recognize only ASCII characters, even in a UTF mode. However, this can be changed by setting the `PCRE_UCP` option.

[GENERAL CATEGORY PROPERTIES FOR `\p` and `\P`](#)

C	Other
Cc	Control
Cf	Format
Cn	Unassigned
Co	Private use
Cs	Surrogate
L	Letter
Ll	Lower case letter
Lm	Modifier letter
Lo	Other letter
Lt	Title case letter
Lu	Upper case letter
L&	Ll, Lu, or Lt
M	Mark
Mc	Spacing mark
Me	Enclosing mark
Mn	Non-spacing mark
N	Number
Nd	Decimal number
Nl	Letter number
No	Other number
P	Punctuation
Pc	Connector punctuation
Pd	Dash punctuation
Pe	Close punctuation
Pf	Final punctuation
Pi	Initial punctuation
Po	Other punctuation
Ps	Open punctuation
S	Symbol
Sc	Currency symbol
Sk	Modifier symbol
Sm	Mathematical symbol
So	Other symbol
Z	Separator
Zl	Line separator
Zp	Paragraph separator
Zs	Space separator

[PCRE SPECIAL CATEGORY PROPERTIES FOR `\p` and `\P`](#)

Xan	Alphanumeric: union of properties L and N
Xps	POSIX space: property Z or tab, NL, VT, FF, CR
Xsp	Perl space: property Z or tab, NL, FF, CR
Xwd	Perl word: property Xan or underscore

SCRIPT NAMES FOR \p AND \P

Arabic, Armenian, Avestan, Balinese, Bamum, Batak, Bengali, Bopomofo, Brahmi, Braille, Buginese, Buhid, Canadian_Aboriginal, Carian, Chakma, Cham, Cherokee, Common, Coptic, Cuneiform, Cypriot, Cyrillic, Deseret, Devanagari, Egyptian_Hieroglyphs, Ethiopic, Georgian, Glagolitic, Gothic, Greek, Gujarati, Gurmukhi, Han, Hangul, Hanunoo, Hebrew, Hiragana, Imperial_Aramaic, Inherited, Inscriptional_Pahlavi, Inscriptional_Parthian, Javanese, Kaithi, Kannada, Katakana, Kayah_Li, Kharoshthi, Khmer, Lao, Latin, Lepcha, Limbu, Linear_B, Lisu, Lycian, Lydian, Malayalam, Mandaic, Meetei_Mayek, Meroitic_Cursive, Meroitic_Hieroglyphs, Miao, Mongolian, Myanmar, New_Tai_Lue, Nko, Ogham, Old_Italic, Old_Persian, Old_South_Arabian, Old_Turkic, Ol_Chiki, Oriya, Osmanya, Phags_Pa, Phoenician, Rejang, Runic, Samaritan, Saurashtra, Sharada, Shavian, Sinhala, Sora_Sompeng, Sundanese, Syloti_Nagri, Syriac, Tagalog, Tagbanwa, Tai_Le, Tai_Tham, Tai_Viet, Takri, Tamil, Telugu, Thaana, Thai, Tibetan, Tifinagh, Ugaritic, Vai, Yi.

CHARACTER CLASSES

[...]	positive character class
[^...]	negative character class
[x-y]	range (can be used for hex characters)
[[:xxx:]]	positive POSIX named set
[[:^xxx:]]	negative POSIX named set
alnum	alphanumeric
alpha	alphabetic
ascii	0-127
blank	space or tab
cntrl	control character
digit	decimal digit
graph	printing, excluding space
lower	lower case letter
print	printing, including space
punct	printing, excluding alphanumeric
space	white space
upper	upper case letter
word	same as \w
xdigit	hexadecimal digit

In PCRE, POSIX character set names recognize only ASCII characters by default, but some of them use Unicode properties if PCRE_UCP is set. You can use \Q...\E inside a character class.

QUANTIFIERS

?	0 or 1, greedy
?+	0 or 1, possessive
??	0 or 1, lazy
*	0 or more, greedy
*+	0 or more, possessive
*?	0 or more, lazy
+	1 or more, greedy
++	1 or more, possessive
+	1 or more, lazy
{n}	exactly n

<code>{n,m}</code>	at least n, no more than m, greedy
<code>{n,m}+</code>	at least n, no more than m, possessive
<code>{n,m}?</code>	at least n, no more than m, lazy
<code>{n,}</code>	n or more, greedy
<code>{n,}+</code>	n or more, possessive
<code>{n,}?</code>	n or more, lazy

ANCHORS AND SIMPLE ASSERTIONS

<code>\b</code>	word boundary
<code>\B</code>	not a word boundary
<code>^</code>	start of subject also after internal newline in multiline mode
<code>\A</code>	start of subject
<code>\$</code>	end of subject also before newline at end of subject also before internal newline in multiline mode
<code>\Z</code>	end of subject also before newline at end of subject
<code>\z</code>	end of subject
<code>\G</code>	first matching position in subject

MATCH POINT RESET

<code>\K</code>	reset start of match
-----------------	----------------------

ALTERNATION

`expr|expr|expr...`

CAPTURING

<code>(...)</code>	capturing group
<code>(?<name>...)</code>	named capturing group (Perl)
<code>(?'name'...)</code>	named capturing group (Perl)
<code>(?P<name>...)</code>	named capturing group (Python)
<code>(?:...)</code>	non-capturing group
<code>(? ...)</code>	non-capturing group; reset group numbers for capturing groups in each alternative

ATOMIC GROUPS

<code>(?>...)</code>	atomic, non-capturing group
-------------------------	-----------------------------

COMMENT

<code>(?#.)</code>	comment (not nestable)
-------------------------	------------------------

OPTION SETTING

<code>(?i)</code>	caseless
<code>(?J)</code>	allow duplicate names
<code>(?m)</code>	multiline
<code>(?s)</code>	single line (dotall)

(?U)	default ungreedy (lazy)
(?x)	extended (ignore white space)
(?-...)	unset option(s)

The following are recognized only at the start of a pattern or after one of the newline-setting options with similar syntax:

(*NO_START_OPT)	no start-match optimization (PCRE_NO_START_OPTIMIZE)
(*UTF8)	set UTF-8 mode: 8-bit library (PCRE_UTF8)
(*UTF16)	set UTF-16 mode: 16-bit library (PCRE_UTF16)
(*UCP)	set PCRE_UCP (use Unicode properties for \d etc)

LOOKAHEAD AND LOOKBEHIND ASSERTIONS

(?=...)	positive look ahead
(?!...)	negative look ahead
(?<=...)	positive look behind
(?<!=...)	negative look behind

Each top-level branch of a look behind must be of a fixed length.

BACKREFERENCES

\n	reference by number (can be ambiguous)
\gn	reference by number
\g{n}	reference by number
\g{-n}	relative reference by number
\k<name>	reference by name (Perl)
\k'name'	reference by name (Perl)
\g{name}	reference by name (Perl)
\k{name}	reference by name (.NET)
(?P=name)	reference by name (Python)

SUBROUTINE REFERENCES (POSSIBLY RECURSIVE)

(?R)	recurse whole pattern
(?n)	call subpattern by absolute number
(?+n)	call subpattern by relative number
(?-n)	call subpattern by relative number
(?&name)	call subpattern by name (Perl)
(?P>name)	call subpattern by name (Python)
\g<name>	call subpattern by name (Oniguruma)
\g'name'	call subpattern by name (Oniguruma)
\g<n>	call subpattern by absolute number (Oniguruma)
\g'n'	call subpattern by absolute number (Oniguruma)
\g<+n>	call subpattern by relative number (PCRE extension)
\g'+n'	call subpattern by relative number (PCRE extension)
\g<-n>	call subpattern by relative number (PCRE extension)
\g'-n'	call subpattern by relative number (PCRE extension)

CONDITIONAL PATTERNS

(?(condition)yes-pattern)	
(?(condition)yes-pattern no-pattern)	
(?(n)...	absolute reference condition
(?(+n)...	relative reference condition
(?(-n)...	relative reference condition
(?(<name>)...	named reference condition (Perl)
(?('name')...	named reference condition (Perl)

<code>(?(name)...</code>	named reference condition (PCRE)
<code>(?(R)...</code>	overall recursion condition
<code>(?(Rn)...</code>	specific group recursion condition
<code>(?(R&name)...</code>	specific recursion condition
<code>(?(DEFINE)...</code>	define subpattern for reference
<code>(?(assert)...</code>	assertion condition

[BACKTRACKING CONTROL](#)

The following act immediately they are reached:

<code>(*ACCEPT)</code>	force successful match
<code>(*FAIL)</code>	force backtrack; synonym <code>(*F)</code>
<code>(*MARK:NAME)</code>	set name to be passed back; synonym <code>(*:NAME)</code>

The following act only when a subsequent match failure causes a backtrack to reach them. They all force a match failure, but they differ in what happens afterwards. Those that advance the start-of-match point do so only if the pattern is not anchored.

<code>(*COMMIT)</code>	overall failure, no advance of starting point
<code>(*PRUNE)</code>	advance to next starting character
<code>(*PRUNE:NAME)</code>	equivalent to <code>(*MARK:NAME)(*PRUNE)</code>
<code>(*SKIP)</code>	advance to current matching position
<code>(*SKIP:NAME)</code>	advance to position corresponding to an earlier <code>(*MARK:NAME)</code> ; if not found, the <code>(*SKIP)</code> is ignored
<code>(*THEN)</code>	local failure, backtrack to next alternation
<code>(*THEN:NAME)</code>	equivalent to <code>(*MARK:NAME)(*THEN)</code>

[NEWLINE CONVENTIONS](#)

These are recognized only at the very start of the pattern or after a `(*BSR_...)`, `(*UTF8)`, `(*UTF16)` or `(*UCP)` option.

<code>(*CR)</code>	carriage return only
<code>(*LF)</code>	linefeed only
<code>(*CRLF)</code>	carriage return followed by linefeed
<code>(*ANYCRLF)</code>	all three of the above
<code>(*ANY)</code>	any Unicode newline sequence

[WHAT \R MATCHES](#)

These are recognized only at the very start of the pattern or after a `(*...)` option that sets the newline convention or a UTF or UCP mode.

<code>(*BSR_ANYCRLF)</code>	CR, LF, or CRLF
<code>(*BSR_UNICODE)</code>	any Unicode newline sequence

[CALLOUTS](#)

<code>(?C)</code>	callout
<code>(?Cn)</code>	callout with data n

[SEE ALSO](#)

pcrpattern(3), pcreapi(3), pcrecallout(3), pcrematching(3), pcre(3).

[AUTHOR](#)

Philip Hazel
University Computing Service
Cambridge CB2 3QH, England.

[REVISION](#)

Last updated: 10 January 2012
Copyright © 1997-2012 University of Cambridge.

Return to the [PCRE index page](#).

Filtering tables

The FILTER() function enables you to filter data with the number of filters up to the maximum 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 or columns.

Filter types include: Regex (regular expressions), patterns (* ? ~), similar to (fuzzy searches), all 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 (because the first output column may additionally contain hyperlinks).

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

If you specify the hyperlink option (see below) and apply the **Format > Hyperlink** style to the first column of the returned array, this first column will contain hyperlinks to the original rows in search-range.

Thus the filtered rows or records can be easily edited. After clicking a given link (or pressing SPACE) and completing editing of the source data, click the back bookmark toolbar button (or press Alt+F11) to return to browsing results in the same place.

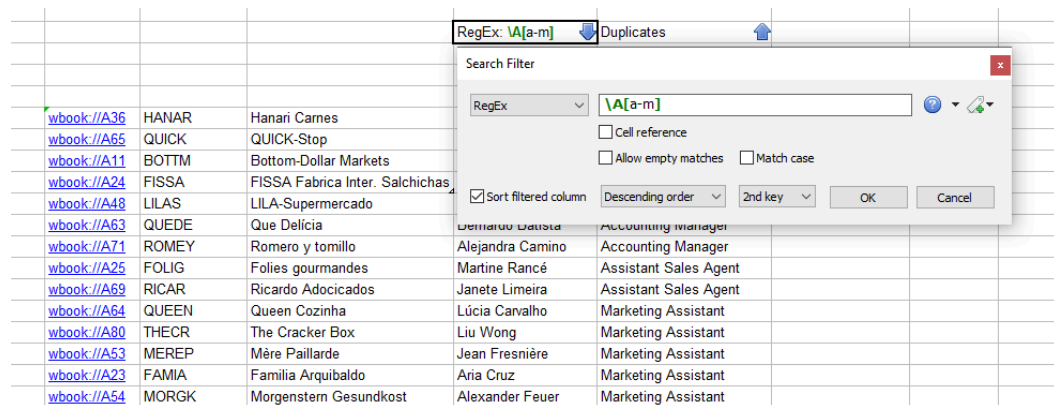
filters

The filters argument represents a range or an array with filtering expressions for the subsequent columns in the search-range. The number of cells in filters must not exceed

the number of columns in search-range. The n-th cell position in filters is the filter expression for the n-th column in search-range. If there is no filter for a given column, the corresponding filter cell may be left empty.

Entering filters in worksheet cells is easy: you can use the **Format > Search Filter** format style for the desired cells. This displays formatted filters with syntax coloring for Regex expressions and graphic sort order indicators.

When editing such cells, GS-Calc displays a cell-aligned dialog box with all filter specification and sort options:



Example

The filter options are as follows:

- Cell reference - the filter text is a 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 punctuation and non-word characters
- Sorting and sort key index - for single-column sorting it is always 1st key; for multi-column sorting you must specify 2nd, 3rd, ... 6th key indices. If indices are repeated, FILTER() returns the #VALUE! error code.

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

[You can also specify filters directly as text cells](#), for example if you want to generate them using formulas.

hyperlink-path

The hyperlink-path argument represents a worksheet or workbook path that will be used along with cell references if the include hyperlinks option is specified.

Examples:

sheet1

folder1\sheet1

c:\documents\ [sample.gsc]sheet1

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

options

The options argument is a number and can be 0 or a combination (sum) of the following values:

- 1 - include hyperlinks to the original search-range rows in the first column of the FILTER() results
- 2 - perform searching for duplicates after all other filters; by default duplicates are searched first
- 4 - by default FILTER() does not create calculation chains with formulas placed in search-range for speed reasons; if some column in search-range contains formulas and you want FILTER() to execute after they are updated, add 4 to options. This will result in significantly slower filtering.

The default value is 0.

Note: If you are using FILTER() for very large data sets and expect to change options frequently, use a cell reference for options instead of a hard-coded value. Re-editing the FILTER() formula cell may activate Undo for most of the target range and cause delays before filtering.

empty-string

The optional empty-string argument represents a number or a string that FILTER() should return if no matching rows are found. If omitted, FILTER() returns the #N/A! error code.

Insert > Filter()

You can insert the Filter() function with two clicks using the **Insert > Filter()** command. The Insert FILTER() dialog box determines 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 source styles and adds hyperlink and filter expression formatting.

B192	=Filter('Pivot tables\orders'!A2:G44, C191:I191, "Pivot tables\orders", 1.)									
	A	B	C	D	E	F	G	H	I	J
189			ID	ProductID	ProductName	UnitPrice	Qty	Total	Date	
190										
191										
192		wbook://Pivot tables	ALFKI	3	Aniseed Syrup	\$10.00	2	\$20.00	3/1/2011	
193		wbook://Pivot tables	BOLID	5	Chef Anton's Gumbo Mix	\$21.35	1	\$21.35	2/28/2011	
194		wbook://Pivot tables	CENTC	11	Queso Cabrales	\$21.00	3	\$63.00	3/4/2011	
195		wbook://Pivot tables	EASTC					\$44.00	3/5/2011	
196		wbook://Pivot tables	BLAUS					\$105.00	2/27/2011	
197		wbook://Pivot tables	ERNSH					\$81.00	3/4/2011	
198		wbook://Pivot tables	WELLI					\$23.25	3/1/2011	
199		wbook://Pivot tables	VAFFE					\$27.60	2/28/2011	
200		wbook://Pivot tables	ERNSH					\$81.00	3/4/2011	
201		wbook://Pivot tables	WELLI					\$23.25	1/2/2011	
202		wbook://Pivot tables	VAFFE					\$27.60	2/28/2011	
203		wbook://Pivot tables	CENTC					\$63.00	3/4/2011	
204		wbook://Pivot tables	EASTC					\$44.00	1/5/2011	
205		wbook://Pivot tables	BLAUS					\$105.00	2/23/2011	
206		wbook://Pivot tables	CENTC					\$63.00	3/8/2011	
207		wbook://Pivot tables	EASTC					\$44.00	3/31/2011	
208		wbook://Pivot tables	ALFKI					\$20.00	3/1/2011	
209		wbook://Pivot tables	BOLID					\$21.35	2/28/2011	
210		wbook://Pivot tables	ALFKI					\$20.00	3/1/2011	
211		wbook://Pivot tables	ALFKI					\$20.00	3/1/2011	
212		wbook://Pivot tables	ERNSH					\$81.00	3/3/2011	
213		wbook://Pivot tables	WELLI					\$23.25	3/1/2011	
214		wbook://Pivot tables	WELLI					\$23.25	3/1/2011	
215		wbook://Pivot tables	ERNSH					\$81.00	3/2/2011	
216		wbook://Pivot tables	WELLI					\$23.25	3/1/2011	
217		wbook://Pivot tables	EASTC					\$44.00	3/5/2011	
218		wbook://Pivot tables	BLAUS					\$105.00	2/27/2011	
219		wbook://Pivot tables	CENTC	11	Queso Cabrales	\$21.00	3	\$63.00	3/4/2011	
220		wbook://Pivot tables	EASTC	4	Chef Anton's Gumbo Mix	\$22.00	2	\$44.00	3/10/2011	

Insert FILTER()

Source cell range:

'Pivot tables\orders'!A1:G44

☒ First row contains column names
 ☒ Copy styles

Insert FILTER() in:

B192

Filter expressions in:

C191:I191

Hyperlink path:

"Pivot tables\orders"

☒ Return hyperlinks to source rows in the first column
☐ Search for duplicates after all other filters are applied
☐ Source range contains formulas

Empty string:

=Filter('Pivot tables\orders'!A2:G44, C191:I191, "Pivot tables\orders", 1.)

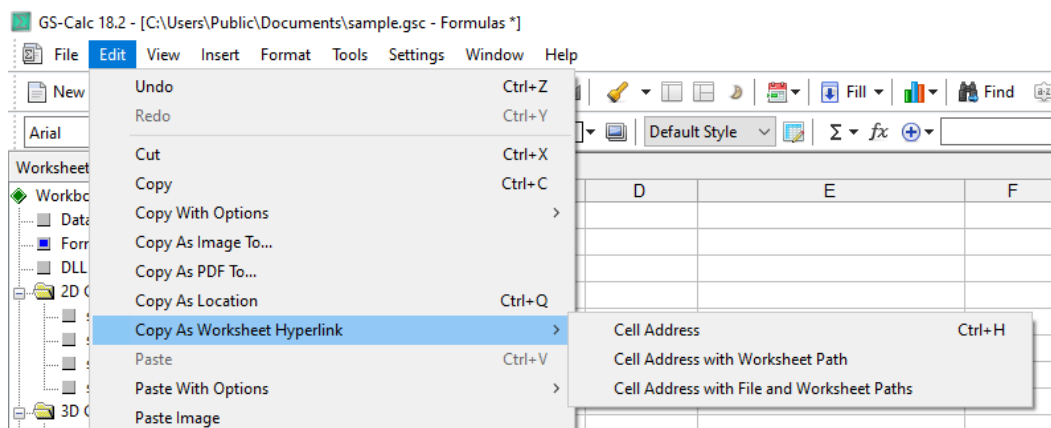
☐ Insert only values returned by formula

OK

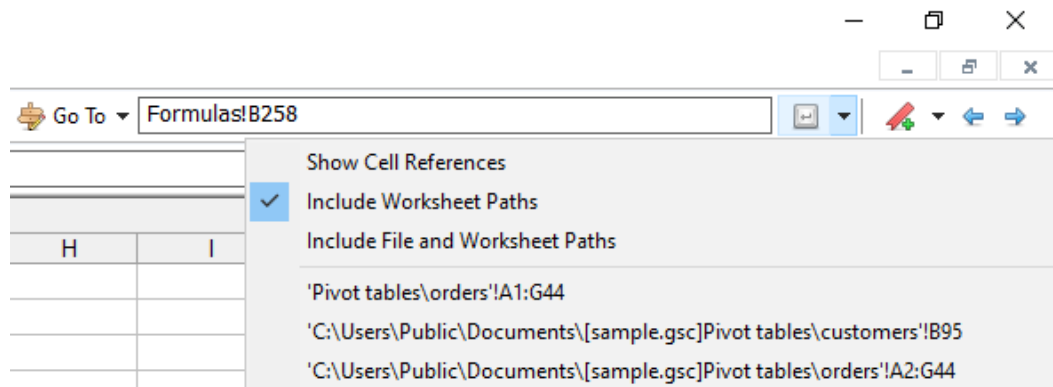
Cancel

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, for example by formulas, you need to use them as plain text cells in the following form:

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

Elements in brackets are optional.

- **filter-type**

- a - regular expression
- b - starts with / 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 or 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 (includes cells with spaces only)
- (optional) **sort-key** is a sort key index. For single-column sorting it is always 1. For multi-key sorting these must be numbers from 1 to 6, and each number may appear only once in the filters array. The number must be followed by + or - to indicate ascending or descending sort order.
- (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 (ignore punctuation and non-word characters when sorting or searching duplicates)

Thus n can be any number from 1 to 7.

- The : or = character indicates the start of the filter text. If it is "=", the text is assumed to be a cell reference (for j: between filters, two references are required).
- (optional) **filter-text** is the actual filtering expression. It is ignored for filter types n and o. For 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. The filters parameter can also be entered directly as an array within the FILTER() formula, for example:

=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)

Using Python Functions as UDF()

1. Overview

The built-in **UDF()** function allows GS-Calc formulas to call Python modules and functions. This greatly expands functionality through Python's scientific, financial, and graphics libraries.

You can pass numbers, text strings, ranges (as numeric arrays/matrices), or automatically saved CSV blocks to Python.

Python functions may return numbers, text, arrays, CSV blocks (parsed or unparsed), or images (binary data).

2. UDF() Syntax

UDF(module, function, type, parameter1, parameter2, ...)

- **module** — full path to a Python module, e.g. "c:\my_modules\some_functions.py"
- **function** — function name inside the module
- **type** — expected return type (0–5):
 - 0 — number
 - 1 — numeric array/matrix
 - 2 — text string (≤1024 chars)

- 3 — CSV block (no numeric parsing)
- 4 — CSV block (with numeric parsing)
- 5 — image (Python **memoryview** binary data)

Additional flags for **type**:

- **32** — pass ranges as CSV blocks
- **64** — pass ranges as CSV only if they contain non-numeric/text cells

If neither flag is used, ranges are passed as numeric arrays (text ignored).

3. Example Python Module

e:\example.py:

```
def function1():
    return 10

def function2(a, b):
    return a + b

def function3(text):
    return 'A new text with ' + text + ' inserted in the middle'

def function4():
    csvdata = 'abc,def,ghi\n123,456,789'
    return csvdata

def function5(csvdata):
    return csvdata

def function6(csvFilePath):
    f = open(csvFilePath, 'rb')
    file_content = f.read()
    f.close()
    return file_content

def function7(imageFilePath):
    f = open(imageFilePath, 'rb')
    file_content = f.read()
```

```
f.close()  
return bytearray(file_content)
```

4. Usage Examples in GS-Calc

- **UDF("e:\example.py", "function1", 0)**

Returns 10.

- **UDF("e:\example.py", "function2", 0, 10, 20)**

Returns the sum of two numbers.

- **UDF("e:\example.py", "function3", 2, "a word")**

Returns a modified text string.

- **UDF("e:\example.py", "function4", 4)**

Returns a CSV block that fills a 2×3 range.

- **UDF("e:\example.py", "function5", 4 + 32, b10:d10000)**

Receives a CSV block and returns it for parsing.

- **UDF("e:\example.py", "function6", 4, "e:\some_csv_file.csv")**

Loads a CSV file and returns its contents.

- **UDF("e:\example.py", "function7", 5, "e:\some_image.jpg")**

Returns an image; GS-Calc displays it at original size.

Related Topics

[Installing Python and integrating it with GS-Calc](#)

Installing Python and Integrating It with GS-Calc

1. Installing Python

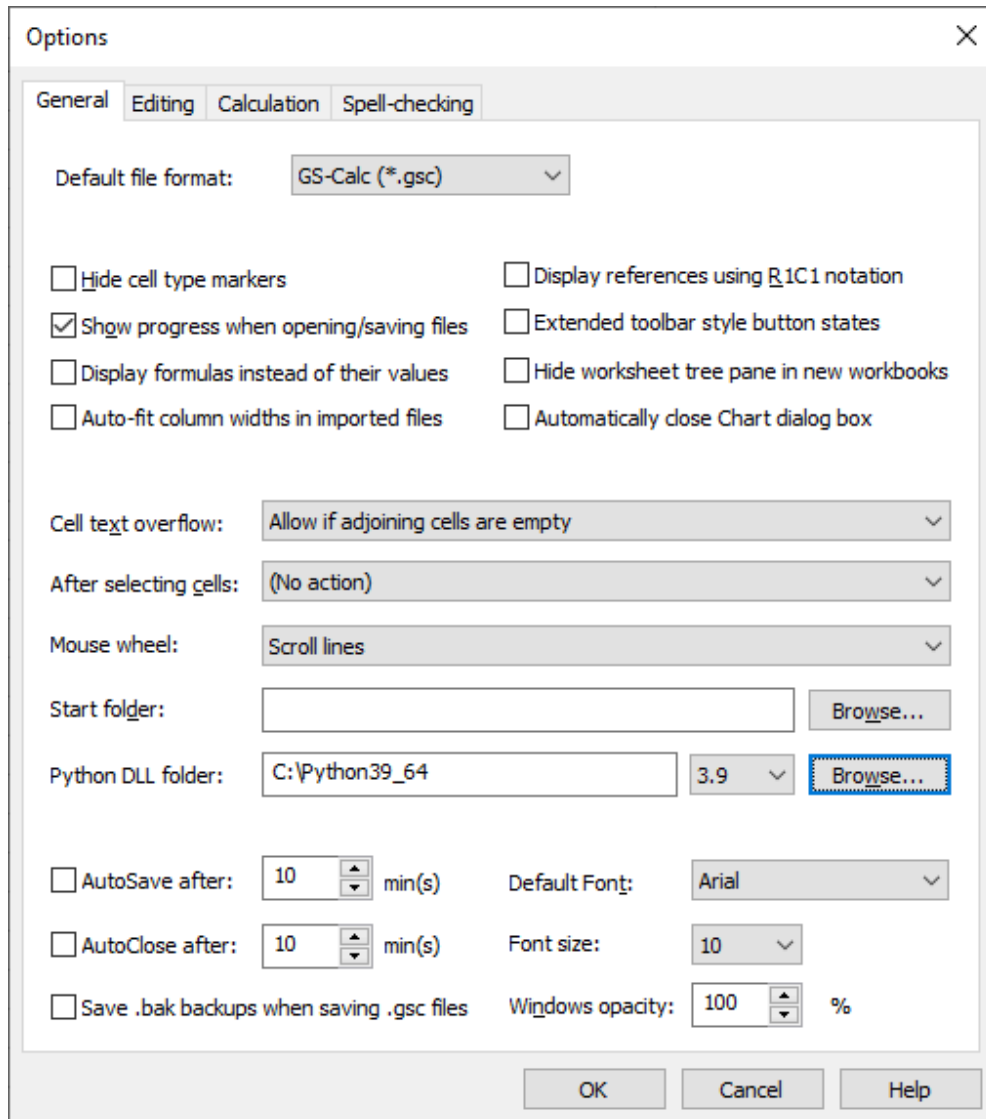
You can download Python from the Microsoft Store or from the official Python website.

2. Selecting the Python Version

Specify the Python version to use in GS-Calc via **Settings > Options > General**.

3. Specifying the Python DLL Location

If Windows system variables do not already point to the Python installation folder, you must specify the folder containing the Python core DLL (e.g., **python39.dll**) in **Settings > Options > General**.



Inspecting Cells and Finding Certain Cell Types: Inspect Formulas Pane

1. Overview

The **View > Inspect Formulas** pane allows you to quickly view a list of formulas in the current workbook, along with their evaluated values and any comments you have added.

To add new cells to the list, click **Add**.

2. Empty Cells

Cells that are currently empty appear in red in the list.

Comparing Workbooks, Worksheets and Cell Ranges

1. Overview

Use the **File > Compare Workbooks** command to find differences between two data sets. These data sets may be:

- entire workbooks in any supported file format,
- single worksheets,
- cell ranges.

You can freely choose where the first data set comes from and where the second one is located. The comparison always starts from the top-left cell of each data set and proceeds left to right. When comparing cell ranges to other ranges, worksheets or entire workbooks, the ranges may be located anywhere within their worksheets.

If the dimensions of the compared data sets differ, non-empty cells from the larger data set are treated as differences unless you select the option to compare only existing, non-empty cells from both data sets.

When comparing entire workbooks, sheets are compared one by one, ignoring sheet names. Cells from missing sheets are treated as differences, except when the option to compare only existing cells is selected. The comparison concerns only cell contents: text, numbers, formulas and their values. Formatting and other metadata are ignored.

2. Compare Workbooks Dialog

The **Compare Workbooks** dialog box:

Compare Data

Compare:

☒ Workbook: E:\sample.gsc Browse...

☐ Worksheet: \Pivot tables\customers

☐ Cell range: D1:E100

To:

☒ Workbook: C:\Users\Public\Documents\sample.gsc Browse...

☐ Worksheet: \Pivot tables\customers

☐ Cell range: D101:E200

☐ Compare only not empty cells from both data sources

☒ Save comparison results in active workbook

New sheet name: compare 12/4/2024 12:27:39 PM

In folder: \

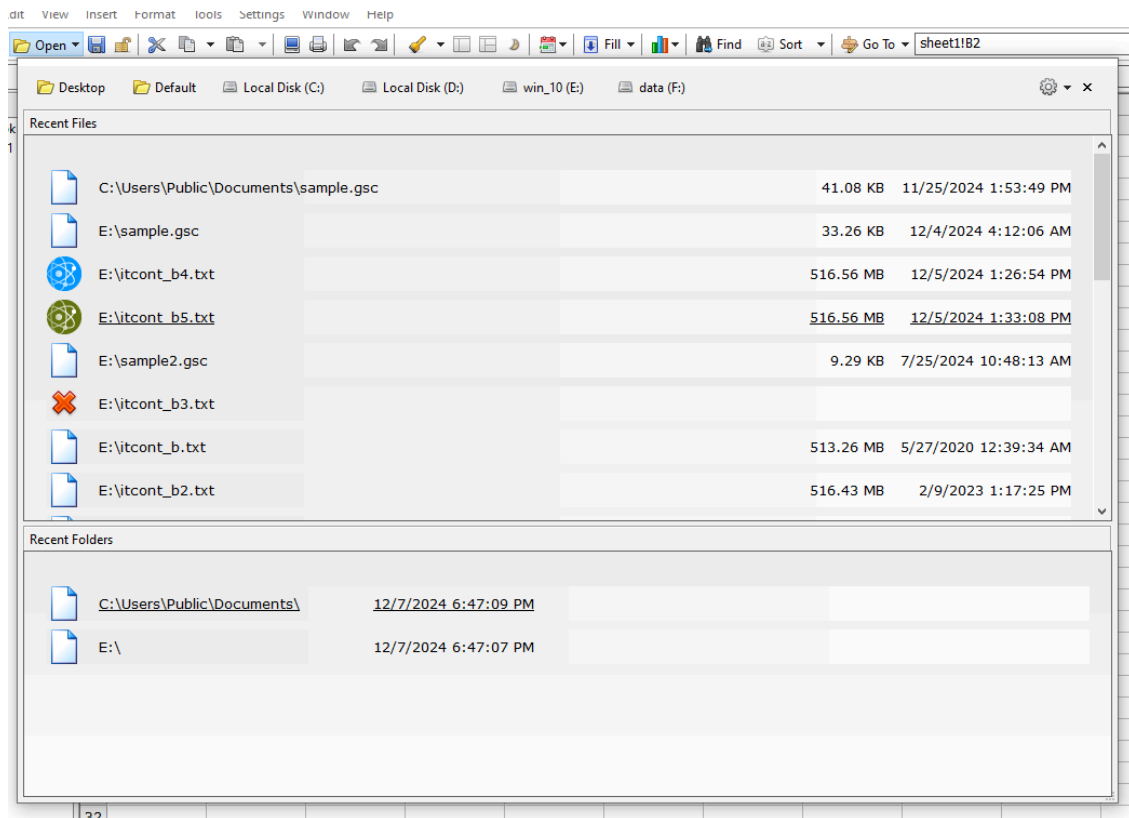
OK Cancel

You can choose the source data options independently for each workbook or data set. At least one of the three options must be selected, and you can mix them freely. For example:

- If the **Workbook** option is not selected, the **Sheet** and **Cell range** options refer to the workbook that was active when the dialog was opened. If only the **Workbook** option is selected, the entire workbook will be compared.
- If the **Sheet** option is not selected, the specified cell range will refer to each sheet in the workbook. If only the **Sheet** option is selected, only that sheet will be compared.
- If the **Cell range** option is selected, only cells from that range will be compared. If the **Sheet** option is not selected at the same time, the same range will be compared across all sheets.

3. Selecting Workbooks

Clicking the **Browse** button opens the standard **Open** dialog with recently used files and folders. The selected workbook is opened in the same way as with the regular **Open File** command, including all applicable file format options.



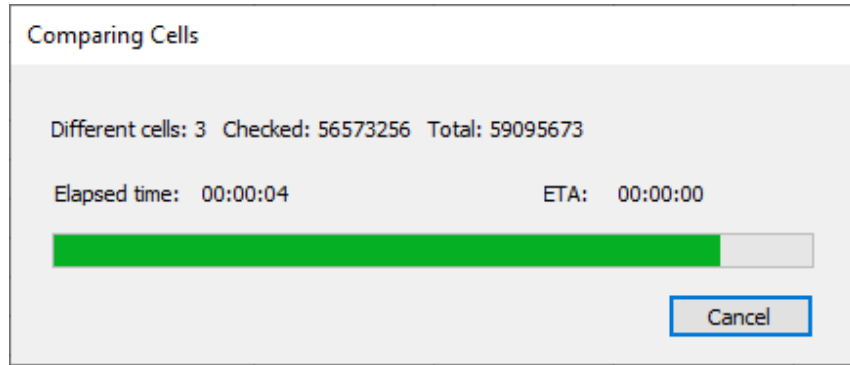
4. Saving Comparison Results

If the **Save comparison results** check box is clear, GS-Calc will only display the number of differences found.

If the **Save comparison results** check box is selected, Gs-Calc will save all found differences as a new time-stamped sheet in the workbook that was active when you clicked **Compare Workbooks**. You can also specify the folder within that workbook where the results should be saved.

5. Progress Dialog

The progress dialog box shows the current status of the comparison:



6. Comparison Report

The comparison report contains two groups of four columns each and may include up to 32 million rows of found differences. For example:

The first column in each group contains hyperlinks to the differing cells in the first and second data sets. Clicking these links allows you to quickly edit, correct or update the cells. To return to the same position in the report workbook, click the rightmost “back” bookmark button on the toolbar. Note that jumping back requires the report workbook to be saved and named.

Formatting: Using Data Styles and Numeric Formats

1. Overview

GS-Base supports a variety of built-in numeric styles, each with its own configurable options. These styles control how numbers, dates, times, percentages, fractions, and scientific values are displayed.

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

Default

- No specific formatting.
- Numbers with more than 15 digits are displayed in exponential scientific format.

General

- decimal places (0 to 14)
- leading zeros (0 to 14)
- thousand separator
- 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
- fixed or system-dependent year/month/day order

Time

- time/period pattern

Date/Time

- date pattern

- time pattern
- 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

2. User-Defined Numeric Formats

You can define your own numeric formats by entering a custom style pattern in the **Style** dialog. Custom formats can be saved as new user styles for reuse.

To understand user-defined formats, it helps to see how standard formats are constructed. For example, the accounting format is:

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

A fraction format with a fixed denominator of 4 can be defined as:

`?\ ?/4`

3. Conditional Formatting with Colors

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

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

This displays:

- positive numbers in green
- negative numbers in red
- zero values in blue
- text labels in gray

Another example:

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

This displays:

- numbers < 10 in red
- numbers ≤ 50 in yellow
- numbers < 400 in green
- numbers ≥ 400 in magenta

Color values may be expressed as RGB values. For example:

[red][<10]#0.00 is equivalent to [#FF0000][<10]#0.00

You may also use one of the following predefined 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]

Formatting: Using Cell Style Palettes

1. Overview

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 currently in use across all worksheets in a workbook (excluding charts and inserted objects). Style palettes update dynamically as formatting changes are applied.

2. Editing Style Palette Entries

GS-Calc allows editing of all properties of existing style palette entries, including the **default cell style**.

Changing the default style affects all cells that do not have user-defined formatting. For example, modifying the default background color will change the background of all unformatted cells.

3. Managing Style Palettes

Style palette management tools are available in:

- **Format** → **Style Palette** (resizable window)
- **Format** → **Cell Format** dialog (when **Format** → **Advanced Cell Format Dialog Options** → **Show Style Lists** is enabled)

Formatting: Using Custom Styles

1. Overview

A cell style is a set of formatting properties (font, data style, borders, background) used to display a single cell.

Custom styles are user-defined collections of cell styles stored inside a workbook. Unlike [style palettes](#), custom styles do not change automatically — they remain fixed until manually edited, added, or deleted.

2. Managing Custom Styles

GS-Calc provides multiple tools for creating, editing, and managing custom styles. These options are available in:

- **Format** → **Custom Styles** (resizable window)
- **Format** → **Cell Format** dialog box (when **Format** → **Advanced Cell Format Dialog Options** → **Show Style Lists** is enabled)

Custom styles can include any combination of formatting properties:

- fonts
- data styles
- borders
- backgrounds
- colors

1. Changing the Default Cell Style

To modify the default cell style:

1. Open **Format** → **Cell Format**.
2. In the **Style Palette** list, right-click the first entry ("Default Style").
3. Select **Edit Style Properties**.
4. Adjust any formatting property:
 - font
 - data style
 - borders
 - background
 - colors

All cells that do not have user-defined formatting will automatically adopt the updated default style.

Conditional Cell Formatting

1. Function Overview

setFormatIf(reference, action, if_condition, format_true, format_false)

This function formats a cell or range of cells. The numeric **action** parameter specifies which formatting attributes to set or clear (restore defaults).

It is often convenient to keep formatting formulas in a separate worksheet so a single control cell can affect formatting across multiple sheets.

The function works on both existing cells and empty regions. There are no limits on the number or complexity of conditional formatting operations — millions of cells may be formatted using nested IFs or other formulas.

2. Formatting Layers

If **reference** specifies entire columns or rows, formatting applies to column or row styles.

Formatting precedence:

table style < column styles < row styles < cell styles

Cell styles override all other styles.

3. `format_true` / `format_false` Values

Values may be **-1** (restore default) or one of the following:

- 1 — numeric style (index or name)
- 2 — custom style pattern (e.g., `$#,##0.00`)
- 3 — font size (4–256)
- 4 — font name
- 6 — bold (0/1)
- 7 — italic (0/1)
- 8 — underline (0/1)
- 9 — strikethrough (0/1)
- 11 — font color (RGB value, “#RRGGBB”, or color name)
- 14 — horizontal alignment (0 left, 1 center, 2 right)
- 15 — vertical alignment (0 top, 1 center, 2 bottom)
- 16 — wrap text (0/1)
- 17 — horizontal indent (0–64 px)
- 18 — vertical indent (0–64 px)
- 19 — text rotation (–180 to 180 degrees)
- 20 — cell borders (“position width style color”)
- 21 — borders around selection (“width style color”)
- 22 — background color (RGB value, “#RRGGBB”, or color name)
- 25 — print / don't print (0/1)
- 26 — hyperlink (0/1)
- 27 — filter cell (0/1)
- 28 — image list cell (0/1)
- 29 — area in-cell chart (0/1)
- 30 — column in-cell chart (0/1)
- 31 — bar in-cell chart (0/1)
- 32 — apply all attributes at once using a named custom style

4. Custom Cell Styles

action = 32 applies all formatting attributes using a named custom style. Custom styles are created in **Format > Custom Cell Styles** and saved in the workbook.

They may also be saved to or loaded from the global GS-Calc cache using:

- **Save As Default App Styles**
- **Load Default App Styles**

5. Border Formatting (**action = 20**)

position may include:

- top
- left
- bottom

- right
- diag-left
- diag-right

width may be:

- thin, medium, thick
- numeric values with px, pt, in, mm, cm, pc

style may be:

- solid
- dotted
- dash
- long-dash
- dot-dash
- dot-dot-dash
- wave
- double

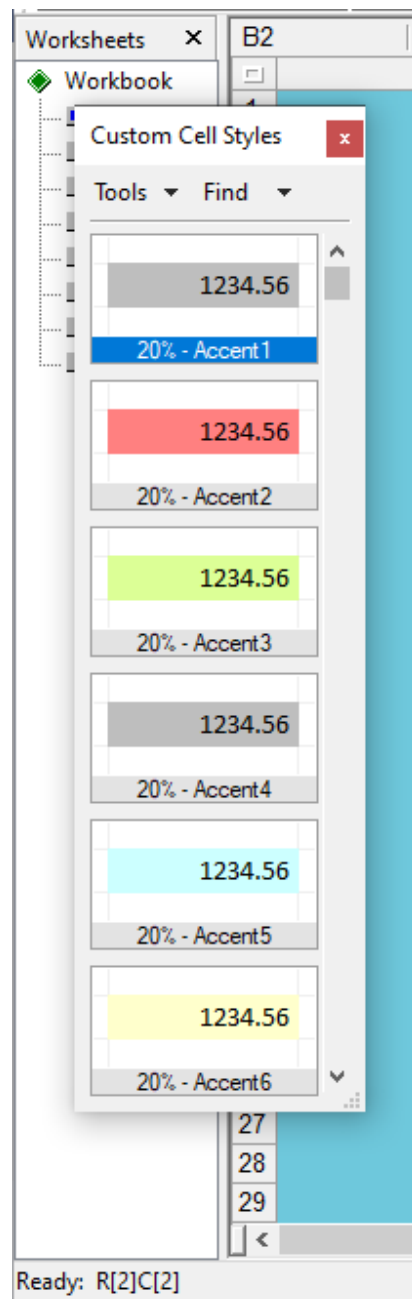
color may be an RGB value, “#RRGGBB”, or one of:

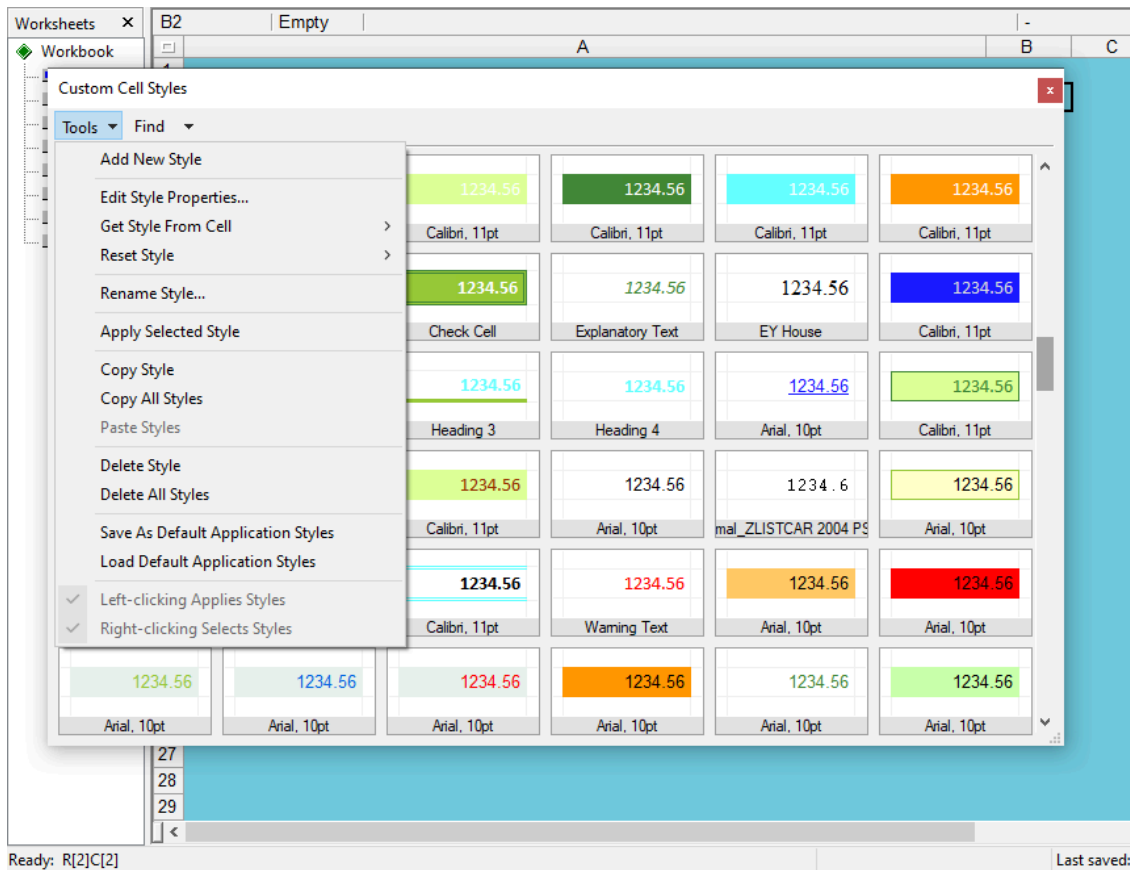
- black
- maroon
- green
- olive
- navy
- purple
- teal
- gray
- silver
- red
- lime
- yellow
- blue
- fuchsia
- aqua
- white

6. Return Value

The function returns the evaluated value of **if_condition**.

7. Custom Cell Styles Dialog





8. Examples

```
setFormatIf(c120, 11, c120 > 0, "green", "red")
```

```
setFormatIf(c:e, 11, c120 > 0, "green", "red")
```

```
setFormatIf(10:20, 11, c120 > 0, "green", "red")
```

```
setFormatIf(c120, 20, c120 > 0, "all 2px solid green", "diag-left diag-right red")
```

```
setFormatIf(d100:d999, 1, c1="use format", "currency", 0)
```

```
setFormatIf(d99, 20, a1, "bottom 2px dotted green", -1)
```

```
setFormatIf(d99, 32, a10 > 10, "my-style-1", "my-style-2")
```

Formatting: Hyperlinks

1. Overview

The **Format > Hyperlink** style allows you to treat cell contents as hyperlinks activated with a single click or by pressing **SPACE**. To edit a hyperlink, double-click it or press **ENTER/F2** (unless

ENTER has been redefined in global or AutoScroll options).

2. Supported Link Types

A hyperlink may point to:

- a location in the current workbook,
- another workbook file,
- a web address,
- an e-mail address,
- a file on a local disk,
- or any other supported protocol.

You can explicitly specify the link type by using one of the following prefixes:

- wbook:// (e.g., **wbook://folder2/sheet2!b100**)
- http://
- https://
- file://
- mailto:
- ftp://
- gopher://
- nntp://
- news://
- wais://
- telnet://
- prospero://
- res://

3. Navigation and Backtracking

Note: Clicking a workbook hyperlink jumps to the specified location. If the link points within the same workbook and you want to return to the previous location, click the **Back** bookmark button or press **Alt+F11**.

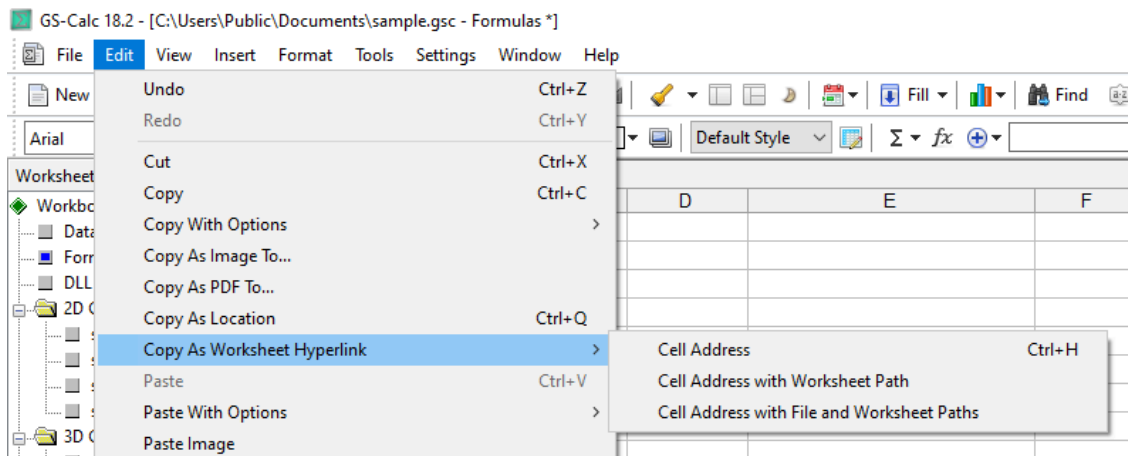
This jump-back option remains available until you use any bookmark command for another purpose.

4. Creating Hyperlinks

The easiest way to create a hyperlink to the current cell is to use one of the **Edit > Copy as Hyperlink** commands.

To paste such a hyperlink — already formatted — into the same or another workbook, simply use the standard **Paste** command.

The **Ctrl+H** shortcut is automatically assigned to the most recently used hyperlink command variant.



Images in Cells

1. Overview

The **Format** → **Images** style allows you to enter a sequence of image names in a cell and display them inline.

Each workbook includes a predefined set of image names such as buttons, flags, and symbols. For example, entering:

star, star, star, star, star-half, star-empty, star-empty

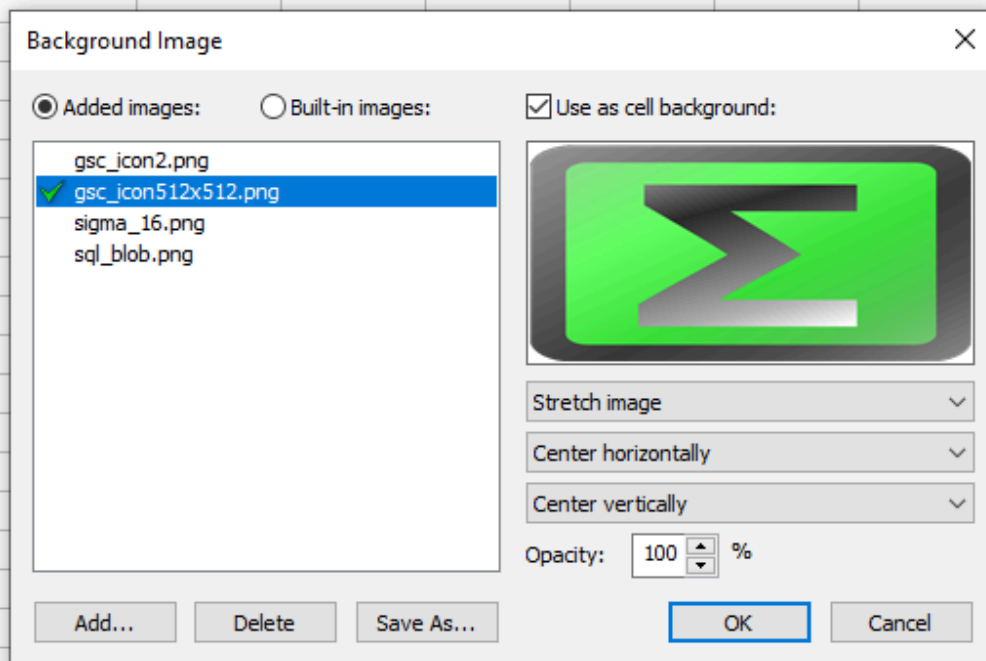
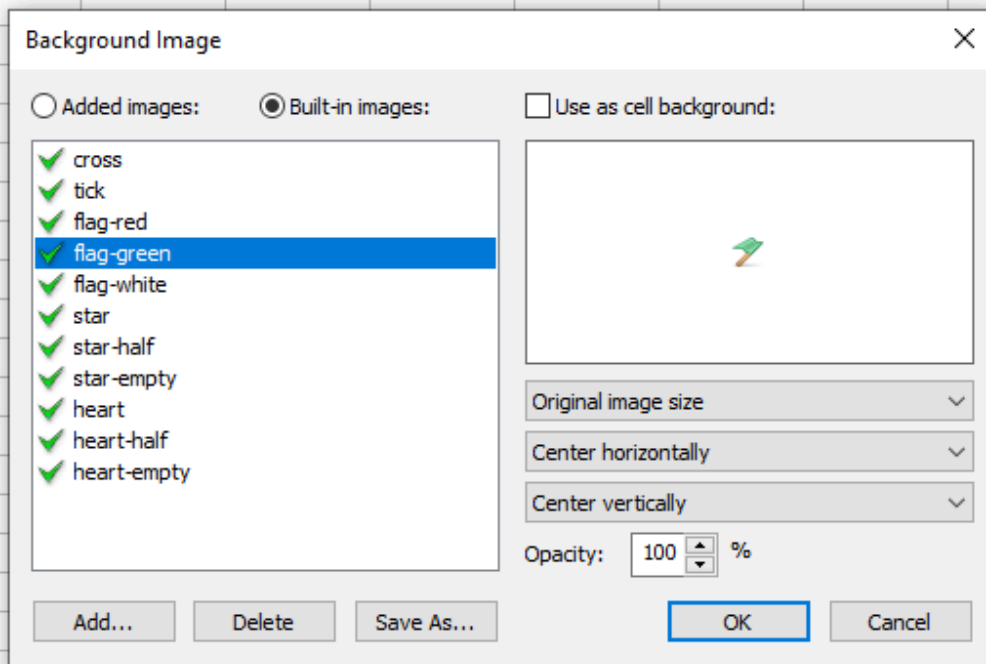
will display a 7-star rating with 4.5 stars:

Text	heart, heart, heart, heart, heart-half, heart-empty, heart-empty		
C	D	E	F
	★★★★★☆☆		
	♥♥♥♥♥♥♥		

2. Adding Custom Images

To add your own images to a workbook, use the **Images List** toolbar button or the **Background Image** command in the **Cell Format** dialog.

The **Workbook Images** dialog allows you to view and manage all added images:



1. Overview

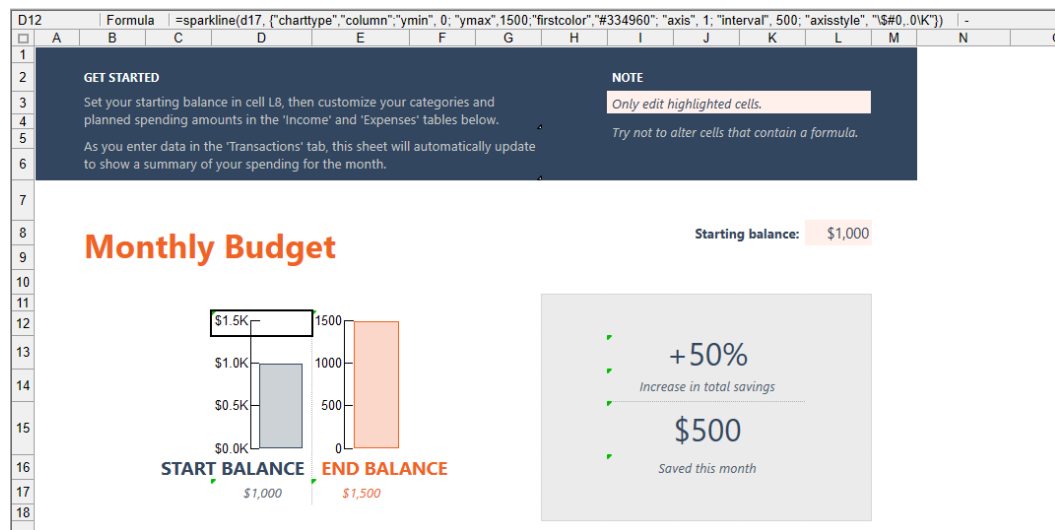
You can display mini-charts directly inside cells using two methods:

1. SPARKLINE() Function

The **SPARKLINE()** function creates a mini-chart using two parameters:

- the data point values
- an array of chart setup parameters

This is a standard Google Sheets function; its full description is available online. GS-Calc additionally supports the **interval** and **axisstyle** parameters.

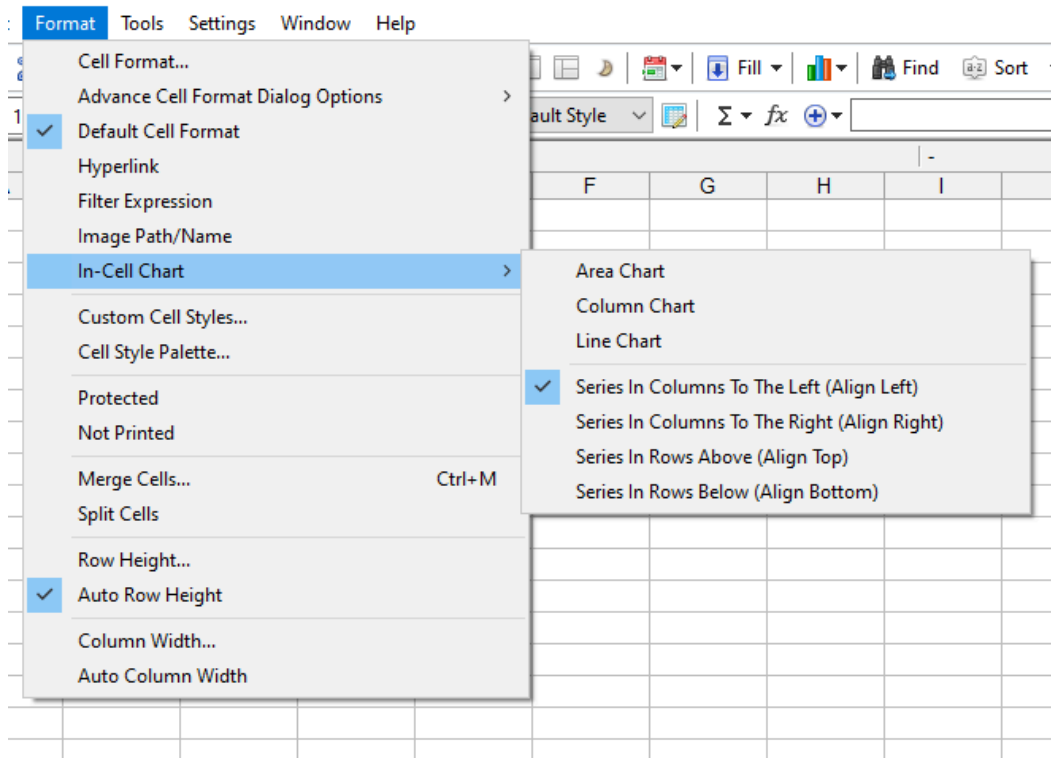


2. Format → In-Cell Chart

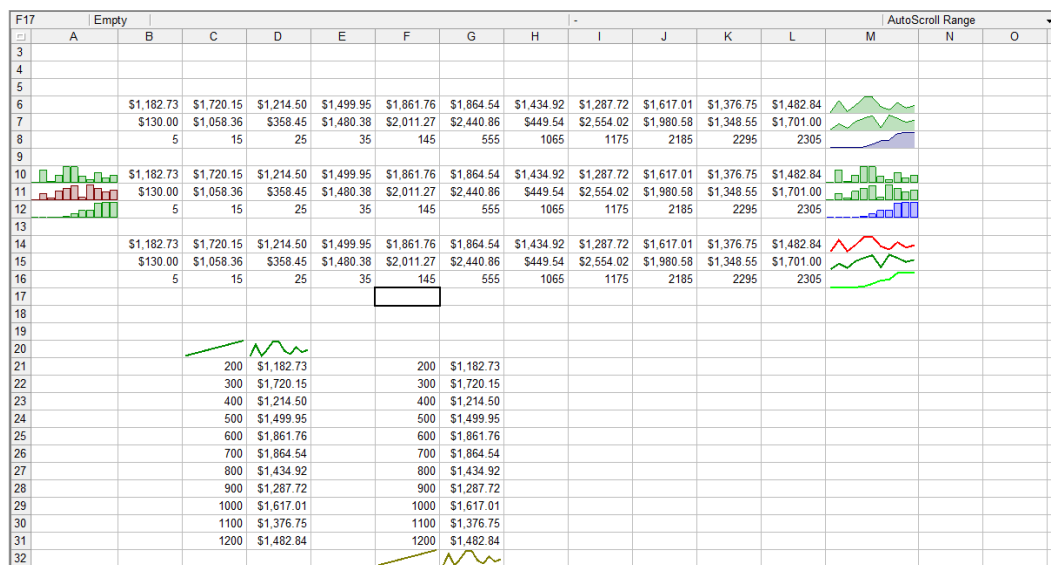
The **Format → In-Cell Chart** command allows you to create any number of cell mini-charts with a single click. All parameters and display options are configured automatically.

By default, this in-cell format is applied to cells at the end of a data series in a row. However, charts can also be displayed at the beginning of a row series, or above/below a column series — the chart position is determined by the cell's text alignment.

Format → In-Cell Chart menu:



Sample in-cell charts:



Charts and Images: Creating Charts

1. Overview

You can create all popular chart types, both 2D and 3D. This includes column/bar charts, area charts, line charts, pie charts, scatter charts, contour charts, and color maps.

A single data series can contain up to **32 million data points**. Each chart may include up to **16 axes**, each with independent scaling and parameters, and you may add custom gridlines.

Contour charts and color maps support both automatic parameters and user-defined isolines and isobands. All 2D charts may be mixed on a single XY plane except color maps, which normally fill the entire XY plane.

2. Specifying Chart Data Series

1. Select a continuous range of cells containing data series in rows or columns and use **Insert Chart > Series In Rows/Columns**. GS-Calc automatically creates the corresponding data series.



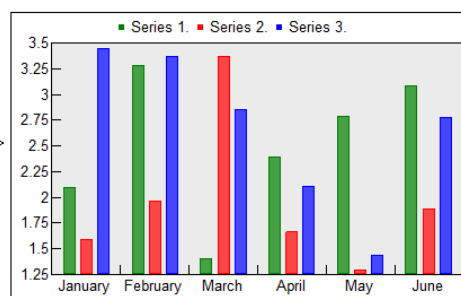
2. Use **Insert Chart** to create an empty chart and add series manually. Click **New Series** in the **Chart** dialog box.

Example for a **2D/3D chart**:



The “Series 1” tab should look like:

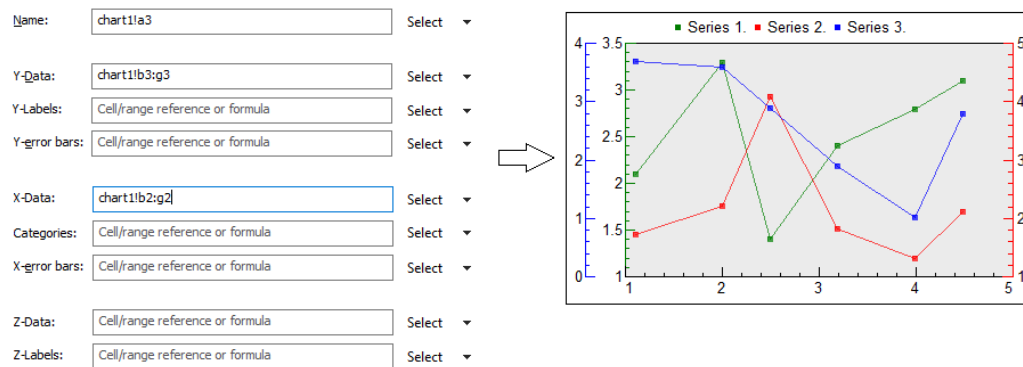
Name:	chart1!a3	Select ▼
Y-Data:	chart1!b3:g3	Select ▼
Y-Labels:	Cell/range reference or formula	Select ▼
Y-error bars:	Cell/range reference or formula	Select ▼
X-Data:	Cell/range reference or formula	Select ▼
Categories:	chart1!b2:g2	Select ▼
X-error bars:	Cell/range reference or formula	Select ▼
Z-Data:	Cell/range reference or formula	Select ▼
Z-Labels:	Cell/range reference or formula	Select ▼



Example for a **scatter (XY) chart**:

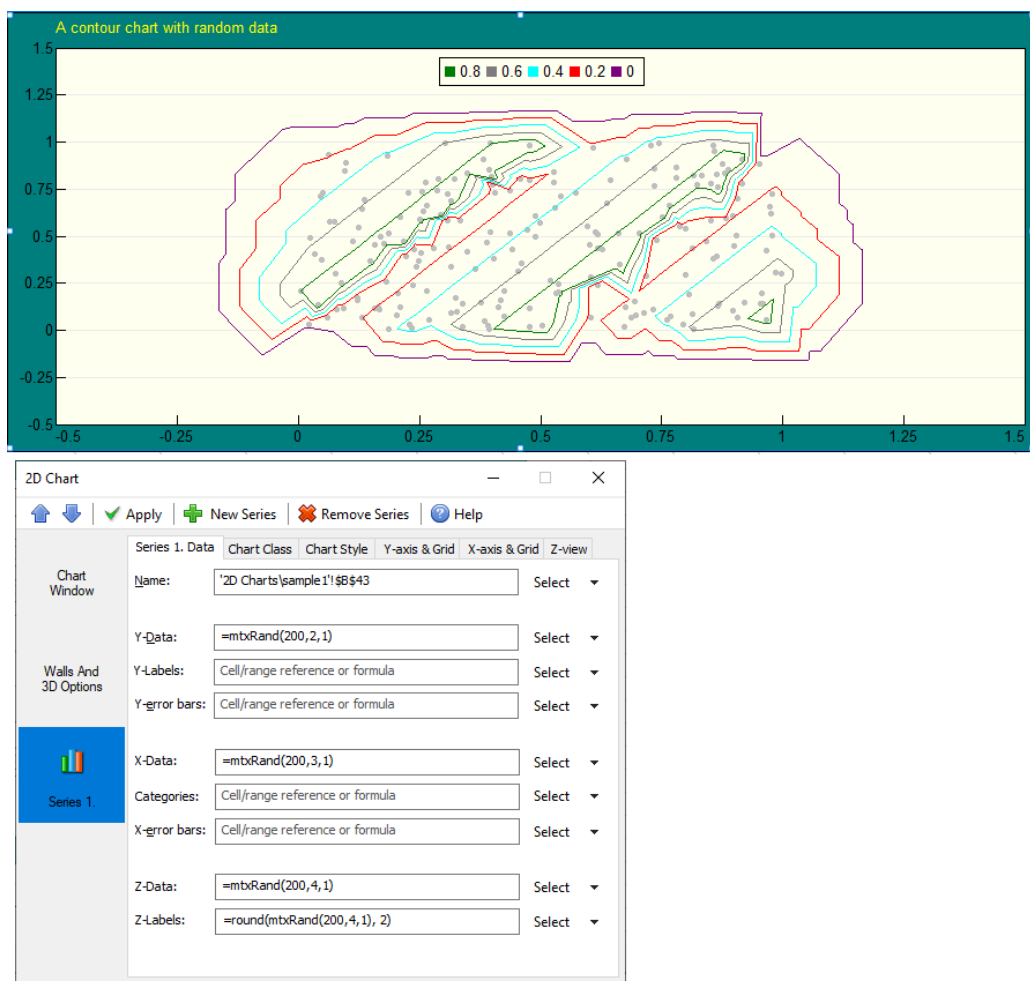


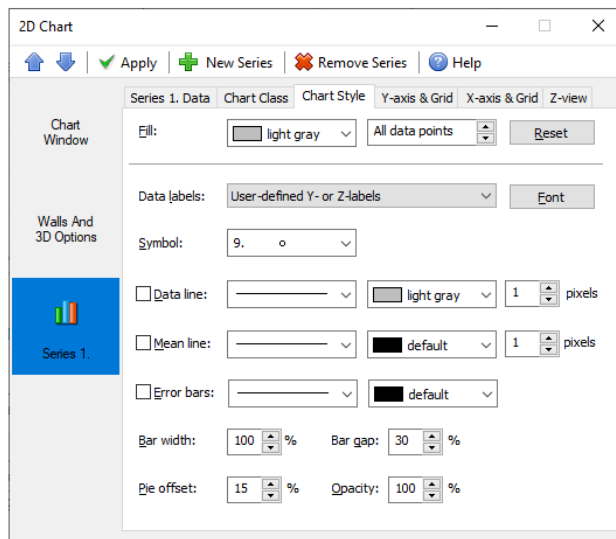
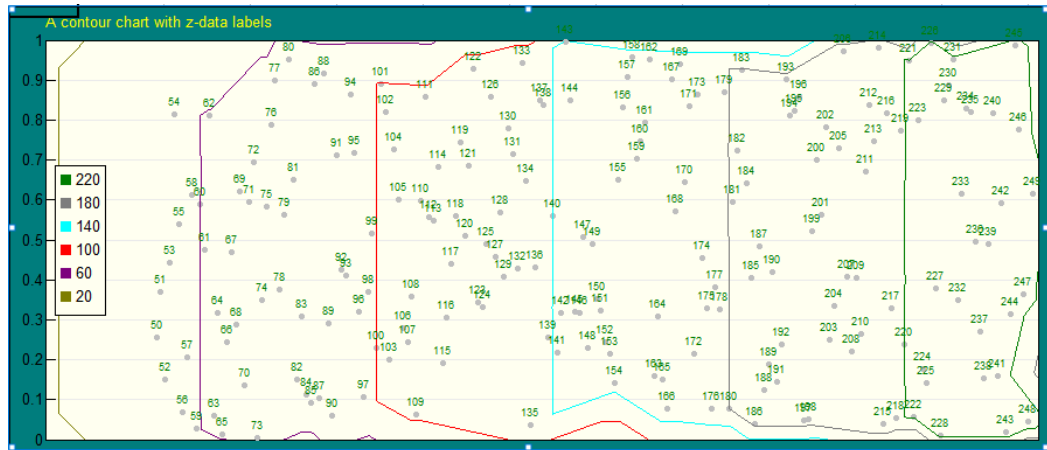
The “Series 1” tab should look like:



If the **X-Data/Domain** field is left empty, GS-Calc creates a domain series of integers from 1 to the number of Y-Data points.

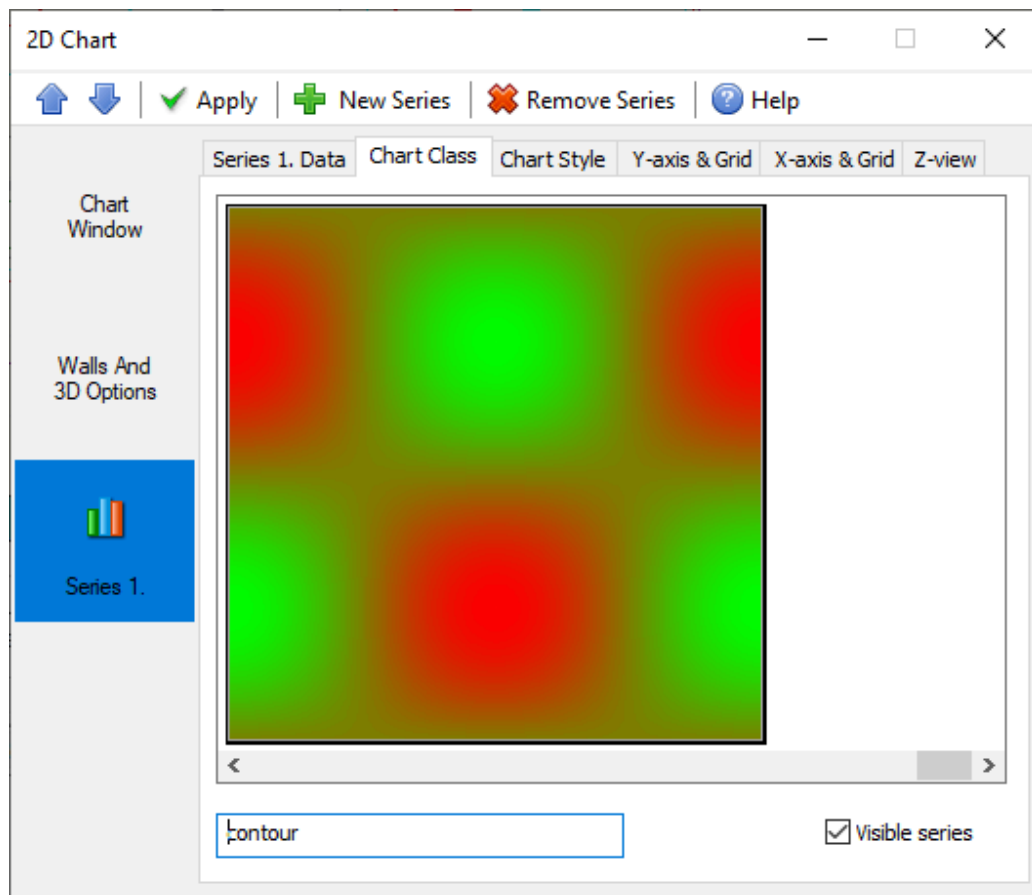
Contour charts and color maps additionally use a third **Z-Data** series assigning values to each XY point. GS-Calc draws isolines around XY points whose Z-values exceed given isoline thresholds.



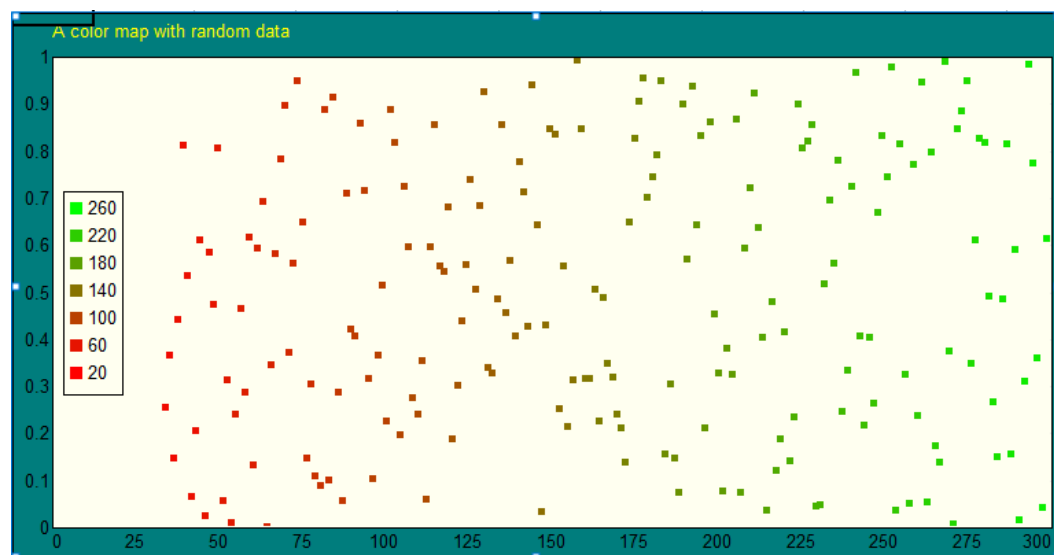


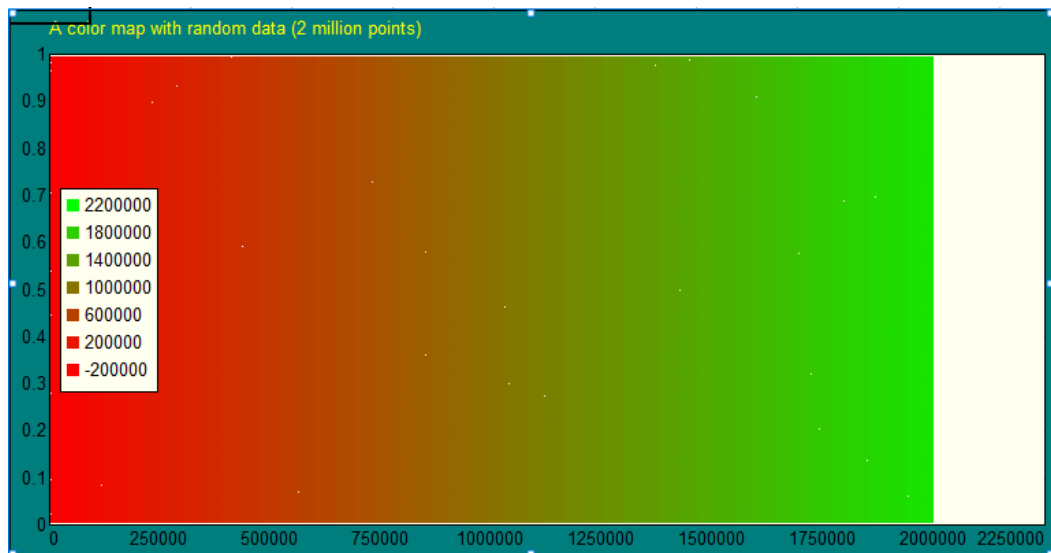
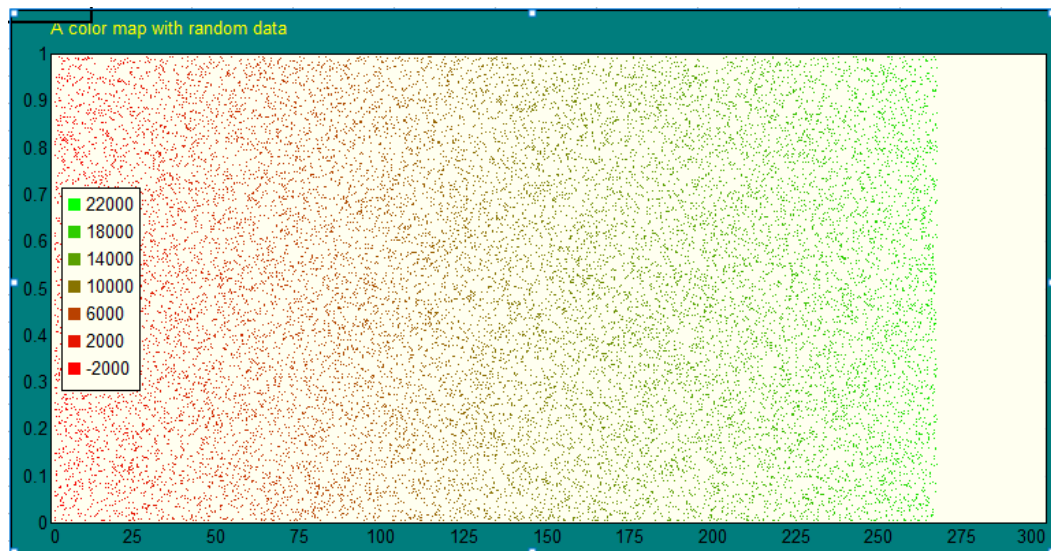
Color maps are similar to contour charts but optimized for very large datasets where isolines would be indistinguishable. Each XY point is colored according to its Z-value using either the default gradient or user-defined colors.

Example: $Z = \sin(x) * \cos(y)$ for x, y in $(0, 2\pi)$.



Color maps are designed to fill the entire XY plane quickly. Depending on dataset size, point size decreases from rectangles down to single pixels.





3. For 2D charts, you may add labels to all or selected data points. Enable labels on the **Chart > Style** tab.

The first two label types (“values” and “percentage”) are generated automatically. The **y-labels** and **z-labels** require specifying source ranges or formulas.

These ranges must contain the same number of cells as the main data series, but may contain empty cells to label only selected points.

If both y-labels and z-labels are specified, **z-labels take precedence**. To display both, create z-labels containing combined text.

4. All chart parameters are specified in the **Chart** dialog box.

Click **Apply** to apply changes.

Double-click a chart to edit it.

XY Chart

Apply
 New Series
 Remove Series
 Help

Chart Window
 Walls And 3D Options
 Series 1.
 Series 2.

Series 1. Data | Chart Class | Chart Style | Y-axis & Grid | X-axis & Grid | Z-view

Axis title:
 Shared axis Id:

Orientation:

☐ Interval: Divisor:

☒ Minimum axis scaling value:
☐ Visible axis

☒ Maximum axis scaling value:
☐ Right side

☒ Axis line:

☒ Major grid:

☒ Minor grid:

Other gridlines:

XY Chart

Apply
 New Series
 Remove Series
 Help

Chart Window
 Walls And 3D Options
 Series 1.
 Series 2.

Series 1. Data | Chart Class | Chart Style | Y-axis & Grid | X-axis & Grid | Z-view

☐ Minimum: ☐ Maximum:

☐ Interval:

☒ Isolines: ☐ Automatic

☐ Isobands: ☒ Automatic

3. Data Types Required in the Chart Dialog

Chart data type	Must be specified as
-----------------	----------------------

Series name	A single cell address or plain text string, e.g.: (1) Worksheet1!\$E\$2 (2) '2D Charts\sample2'!\$B\$24 (3) sales in February
Chart data series, domain, series and domain error bars, data categories	A cell range, list of numbers, list of text strings, or array formula, e.g.: (1) Worksheet1!\$H\$1:\$H\$100 (2) 23 2.3 45 12.1 44 (3) ab "c d e" "a1" "fg h" "in ""quote"" (4) ={1, 2, 3.1, 4.2, 5.7} (5) =row(b1:b100)*rand() (6) =mtxSeries(30,1,1) + 4*mtxRand(30,1,1) Note: When entering a list of text strings, use quotation marks for strings containing spaces or quotes. Inner quotes must be doubled.
Chart titles, axis titles	A plain text string or cell address, e.g.: (1) sample chart title (2) Worksheet1!E2

4. Notes

- Changes in the **Chart** dialog apply only after clicking **Apply**. Exceptions: **3D rotation** and **lamp parameters** update instantly.
- The **Chart** dialog always requires A1 notation for cell references.
- **Shared axes:**

Each series has its own visible axes by default. To ensure identical axis limits across series:

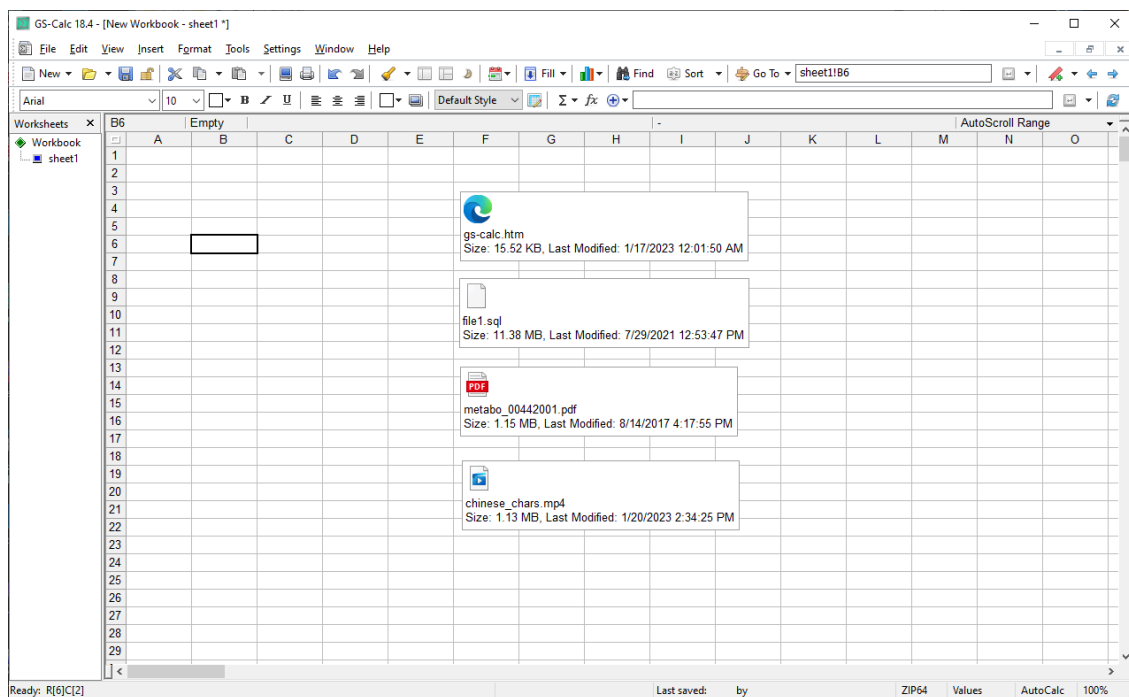
1. Specify identical fixed axis limits for each series (and hide all but one axis), or
 2. Specify a shared axis by entering the axis ID of another series in **Chart > Axis > Shared axis id**. Shared axes are displayed only once per group.
- If only minimum or only maximum axis limits are specified, automatic interval calculation may cause displayed limits to differ by one interval. To enforce exact limits, specify: minimum, maximum, and interval.
 - New charts show only one set of horizontal gridlines (for the first series) and axes for the first two series. Additional gridlines and axes may be enabled manually.
 - If **Categories** (2D/3D charts) or **Domain** (XY charts) is left blank, GS-Calc uses integers starting from 1.

1. Overview

To insert images or files, use the **Insert > Image / File** command.

Images are displayed in their natural size and proportions. If you resize the frame containing an image, double-click the image to restore the frame to full size.

Other inserted files (non-executables only) appear as frames containing an icon and file details. Right-click such an object to view more information or to save/export the original file.



Password Protection: Protecting and Encrypting Workbooks

To encrypt and password-protect a file, use **File > Encrypt**, specify a password of at least six characters, and choose an encryption algorithm.

When a workbook is encrypted, the “shield” toolbar button turns green.

Note: GS-Calc uses standard ODF encryption and encrypts all workbook content (tables, styles, charts, images, settings, etc.). However, names of inserted images remain unencrypted inside *.ods files.

Password Protection: Protecting Workbook Structure

To disable or enable adding, deleting, or moving worksheets, use **Tools > Protect Structure**. The password must contain at least six characters.

Note: If the workbook is not encrypted, anyone with access to the file can modify this setting using other software capable of reading/writing *.ods files.

Password Protection: Cell Protection

To enable or disable cell protection for a worksheet, use **Tools > Cell Protection**. A password of at least six characters is required.

Once protection is active, use **Format > Protected** to toggle protection for individual cells. Protected cells cannot be deleted.

Each time you protect or unprotect cells, you must enter the defined password.

Note: If the workbook is not encrypted, someone with full access to the file can change these settings using other *.ods-compatible software.

Opening and Saving Files: Content and Statistics

To view file statistics and change options concerning 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 must perform an initial update or recalculation after opening the file to recreate formula values.

This option is available only for the GS-Calc *.gsc file format.

Default: On

Save selected formula values

If the **Save values of formulas** check box is on, this option allows explicitly specifying cells or ranges whose formula values should 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 cell or range reference does not contain a worksheet path, it is assumed to refer to the currently active worksheet.

This option is available only for the GS-Calc *.gsc file format.

Default: Off

Save thumbnail

Specifies whether a thumbnail image should be added to the saved workbook.

Thumbnail images can be saved in *.gsc and *.ods files. They represent the portion of the worksheet visible when the worksheet was saved.

For encrypted files, the thumbnail is replaced by a static 128x128 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:

Thumbnails/thumbnail.png

Note: The ODF specification requires thumbnails in *.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.

The range may be:

- a simple cell range,
- a cell range with a worksheet reference,
- or only a worksheet reference/path without a specific cell range.

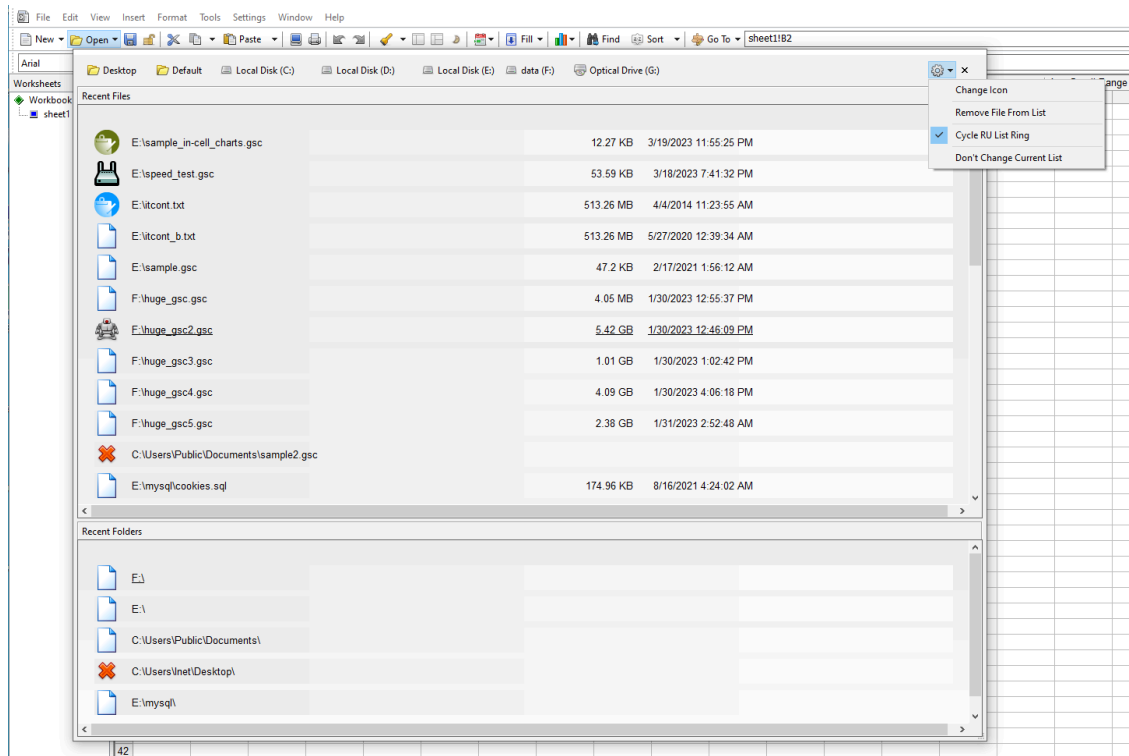
If not specified, the thumbnail represents the portion of the worksheet visible when the worksheet was saved.

Default: Off

1. Overview

Click the **Open** toolbar button to display the lists of recently used files and folders. The lists appear in a split window. The file list shows paths, sizes, modification dates, and states. The folder list shows paths and the last access date.

You can assign one of many built-in icons to each file or folder. Each list can contain up to 50 items, added cyclically. Both lists can be locked to prevent new additions or reordering.



2. Opening Files

To open a file or a file from a listed folder, click its link or use the cursor keys and press Space.

3. Default Folder

The **Default** button opens the folder specified in **Settings > Options > General > Start Folder**.

4. Changing Icons

To change a file's icon, right-click its current icon.

5. Removing Items

To delete a file from the RU list, right-click it and choose **Remove**.

6. Preserving List Order

To preserve the current order of the RU file list, right-click the list and uncheck **Cycle RU List Ring**. Otherwise, the most recently opened database moves to the top.

7. Restoring the Old Short “Open” Button

To restore the old icon-only **Open** button, edit **settings.xml** and change:

```
<OldRecentFileList>0</OldRecentFileList>
```

to:

```
<OldRecentFileList>1</OldRecentFileList>
```

*Opening and Saving GS-Base Databases (Saved as *.xls Workbooks)*

1. Saving a Workbook in *.xlsx / *.xls / *.xml Format

Use **File** → **Save As** and choose **Excel (*.xlsx, *.xml)** from the **File of type** list.

Note (1):

As of version 20.0, table references defined inside worksheets are not yet supported. Formulas such as:

```
=Meals[[#This Row],[Budget]] - Meals[[#This Row],[Actual]]
```

are loaded and saved with their values, but they are not recalculated.

Note (2):

Although old *.xls files are still supported, if they contain formulas, GS-Calc loads and saves only formula values.

1.1 Save Options (Save Excel XLSX File Dialog)

- **Split 1M+ row tables into Excel worksheets (sheet, sheet(1), ..., sheet(n))**
If selected, sheets exceeding Excel's row limits (1M for *.xlsx, 65K for *.xls) are split into multiple worksheets.
- **Split 1M+ row tables into Excel files (file.xlsx, file(1).xlsx, ..., file(n).xlsx)**
Sheets exceeding Excel row limits are saved into multiple Excel files. Previously saved files with higher indices are deleted. All files share identical sheet structures. Sheets with fewer rows are saved as empty sheets in subsequent files.

When opening file.xlsx again in GS-Calc, all partial workbooks are automatically merged.

- **Truncate 1M+ row tables**

Rows above Excel's limits are not saved.

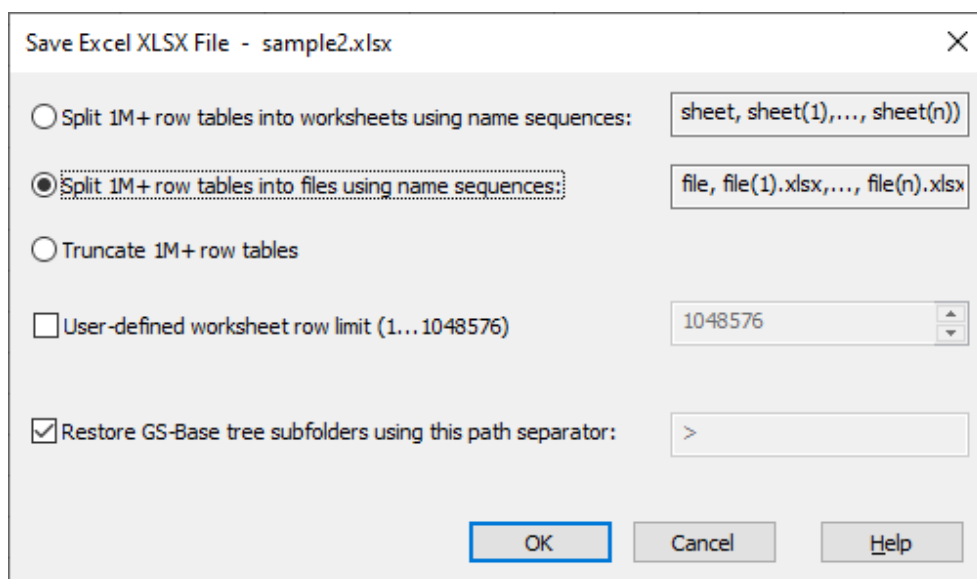
- **User-defined worksheet limit**

Allows specifying a custom threshold (1 to Excel's row limit) for splitting sheets when saving *.xlsx files.

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

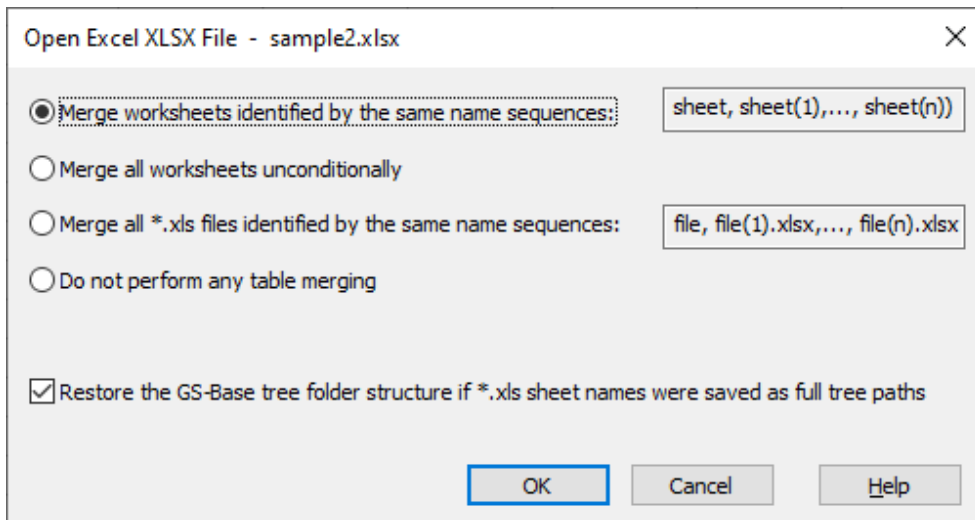
If selected, Excel sheet names are generated using full GS-Calc sheet tree paths, allowing GS-Calc to reconstruct the original folder structure when reopening the workbook back in GS-Calc.

Since “\” cannot be used in Excel sheet names, it is replaced with “>”.



2. Opening a GS-Calc File Saved in *.xlsx / *.xls Format

Use **File** → **Open** and choose **Excel (*.xlsx, *.xls)** from the **File of type** list.



3. Related Topics

[GS-Calc *.gsc and ODF *.ods files](#)

[Excel \(*.xlsx, *.xml\) files](#)

[PDF files](#)

[Text files \(*.csv, *.txt, *.tab ...\)](#)

[Text *.zip archives](#)

[GS-Base \(*.gsb\) files](#)

[dBase/Clipper/FoxPro files](#)

[SQLite \(*.db\) files](#)

[MySQL \(*.sql\) files](#)

Opening and Saving Files: Text *.zip Archives

1. Overview

GS-Calc can load and save multiple text files inside a single ***.zip** archive. This is supported through the “Files of Type” filter:

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

To save a text archive for the first time, choose this filter in **File** → **Save As**, enter a file name, and append the **.zip** extension.

2. Advantages of Text *.zip Archives

- All worksheets can be saved at once as text files inside a single archive.
- No need to repeatedly adjust text file options (separators, encoding, parsing rules). When opening a text archive, the **Text File Options** dialog appears only once unless you choose to display it for individual files.
- Each saved archive contains a small ***.xml** file storing text import/export parameters, allowing seamless editing and saving without reopening option dialogs.
- All nested worksheet subfolders are preserved when saving a *.gsc file into a text archive.
- You can automate workflows by generating archives containing multiple ***.xml** option files for selected text files.

3. Opening a Text *.zip Archive

Use **File** → **Open** and select:

Text | Text Archive

Text archives must contain files with extensions:

- *.txt
- *.csv
- *.tsv
- *.tab

Files may be stored in subfolders inside the archive; these subfolders are recreated in the imported workbook.

3.1 Text File Options

Text file options (separators, encoding, parsing rules, formatting) may be specified:

- by a single **[zip-archive-name].xml** file containing options for all text files
- by one or more **[text-file-name].xml** files containing options for specific files
- in the **Open Text File** dialog box (shown only for files without corresponding *.xml settings)

Checking “apply to all (...)” in the dialog applies the chosen options to all remaining files.

3.2 XML File Structure

```
<?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" -> serial date number

keep-styles="false"           // retain formatting after parsing

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

iso-code-page="Default Ansi code page"/>
                               // e.g. "ISO 8859-15 Latin 9"

```

4. Notes on Text-Only Import

If a text file is opened entirely as text (no number or formula parsing), all worksheet columns created from that file are automatically formatted using the **Text** style.

To remove the Text style from a column:

- Click the column header
- Press **Del** → “Formatting”, or
- Select **Default Style** on the formatting toolbar

5. Saving a Text *.zip Archive

Use **File** → **Save As**, choose:

Text | Text Archive

Enter a file name and append **.zip**.

The **Save Text File** dialog appears to specify text file options. Subsequent **File** → **Save** operations reuse these settings automatically.

Text archives are saved with a single **[zip-archive-name].xml** file containing options for all included text files.

6. Auto-Resizing Columns

6.1 Automatic

Enable:

Auto-fit column widths

in either:

- **Open Text File** dialog
- **Settings** → **Options** → **General**

(See [Settings](#))

6.2 Manual

- **All columns:** Click a row heading to select the row, then use **Format** → **Auto-fit column widths**.
- **Selected columns:** Select cells in the desired columns, then use **Format** → **Auto-fit column widths**.

7. Related Topics

[GS-Calc *.gsc and ODF *.ods files](#)

[Excel \(*.xlsx, *.xml\) files](#)

[PDF files](#)

[Text files \(*.csv, *.txt, *.tab ...\)](#)

[Text *.zip archives](#)

[GS-Base \(*.gsb\) files](#)

[dBase/Clipper/FoxPro files](#)

[SQLite \(*.db\) files](#)

[MySQL \(*.sql\) files](#)

Saving Files: PDF Files

1. Saving Data as PDF

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

- **File > Save Worksheet As PDF** and **File > Save All As PDF** — the first saves the current worksheet, the second saves the entire workbook. Page layout and content options can be adjusted via **File > Page Settings**.

- **Edit > Copy As PDF To...** — saves the currently selected range, chart, or image.
- **File > Preview** — in the preview window, click **Save Page** to save the displayed page, or **Save All** to save all generated pages.

Note: Preview pages depend on **File > Print > General** settings. For example, choosing **File > Print > General > Printed Data > Selection** limits the preview to selected cells or the selected object.

2. Font Mapping

Note: When saving a PDF, GS-Calc maps all workbook fonts to the standard set of 14 PostScript Type 1 fonts. These fonts are guaranteed to exist in all PDF viewers and operating systems.

3. Advanced PDF Save Options

Use **File > Advanced PDF Save Options** to specify:

- the language/code page used when saving text,
- a custom character encoding list to ensure correct glyph display.

4. Example Encoding Lists

iso-8859-1

```
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

```

Windows-1250

```

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

```

Opening and Saving Files: Text Files

1. Opening a Text File

Use **File** → **Open** and choose **Text** from the **File of type** list. Then specify additional text file options as described below.

Cell separator A single character used to separate cells in one line.

Fixed cell widths A comma-separated list of fixed cell width values, e.g.:

10,30,15,40

If fewer widths are specified than needed, 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 newline characters must be enclosed in quoting symbols. Inner quoting symbols must be doubled.

If you need to open a text file that does not conform to this standard, consider disabling quoting or specifying a unique quoting symbol that does not occur in the file.

Escaping symbol

A symbol used to mark the occurrence of a single byte value specified as a two-digit hexadecimal number preceded by the **x** letter.

Examples:

\x66

\xA0\x45

Escaping symbols not representing such sequences must be doubled.

Escaping can be selected when opening a text file. When saving, GS-Calc always uses quoting and UTF-8 encoding. Use escaping only if you are certain the file uses this encoding method.

Note:

If a text file is to be opened entirely as textual data (no number or formula parsing), clear all parsing options below. All worksheet columns created from that file will automatically use the **Text** style.

To remove the Text style and enable number/formula parsing, select the column and either:

- press **Del** → “Formatting”, or
- choose any numeric style on the formatting toolbar

Formula conversion	Determines whether loaded text strings should be evaluated and converted to formulas. The default No Conversion treats all cells as text labels.
Convert text to numbers	Converts plain numeric text strings to numbers. Only unformatted numbers can be converted. The decimal point is determined by Settings → Locales and system regional settings.
Parse formatted numbers	Converts formatted numeric strings (e.g., currency, parentheses) to numbers. Decimal point rules follow Settings → Locales and system regional settings.
Convert date/time strings	Converts text strings representing dates and/or times into date serial numbers.
Retain styles	Specifies whether cells containing numbers (after parsing) should retain their original data styles.
Encoding	Specifies the text encoding: UTF-8, ANSI 8-bit, ISO/OEM.
Maximum columns in sheet	By default, GS-Calc warns if a text file contains more than 16K columns. If enabled, additional sheets are created automatically to load columns beyond 16K. You may specify any splitting limit from 1 to 16384 columns. Text files may contain up to 1 million columns, split into 16384-column sheets.

If multiple sheets are created, saving them back as a text file prompts for a new ZIP64 archive name. To reopen such archives, choose: **Text | Text Archive (*.txt, *.csv, *.tsv, *.zip)**.

Maximum rows in sheet Same as above, but for rows. You may split text files into multiple sheets with any number of rows from 1 to 32 million.

Open Text Archive File: users102501.csv - E:\citadel_test.zip

Sample file data:

CustomerID	CompanyName	ContactName	ContactTitle	Address	City	Region	PostalCode	Cour
ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57	Berlin		12209	Ger
ANATR	Ana Trujillo Emparedados y helados	Ana Trujillo	Owner	Avda. de la Constitución 2222				
ANTON	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312	México D.F.		051	

☒ Cell separator: comma ☐ Fixed field widths:

☒ Quoting symbol: " ☐ Escape symbol: \

Enter widths in characters of the subsequent columns separated by commas. For example, 10,8,15

Formula conversion: Don't parse formulas Text encoding: UTF-8 Windows Code Page:

☒ Convert text to numbers ☐ Convert date/time strings to date/time serial numbers

☐ Parse formatted numbers ☐ Save styles for converted number and date/time strings

☐ Maximum columns in one sheet: 16384

☐ Maximum rows in one sheet: 33554432

☐ Auto-fit column widths ☒ Apply these settings to all files in this ZIP archive

Load predefined settings ▼ OK Cancel Help

2. Auto-Resizing Columns

To automatically resize columns when opening text files, enable:

Settings → Options → General → Auto-fit column widths in imported files

(See [Settings](#))

3. Saving a Text File

Use **File** → **Save** and choose **Text** from the **File of type** list. Specify the same text file options as when opening.

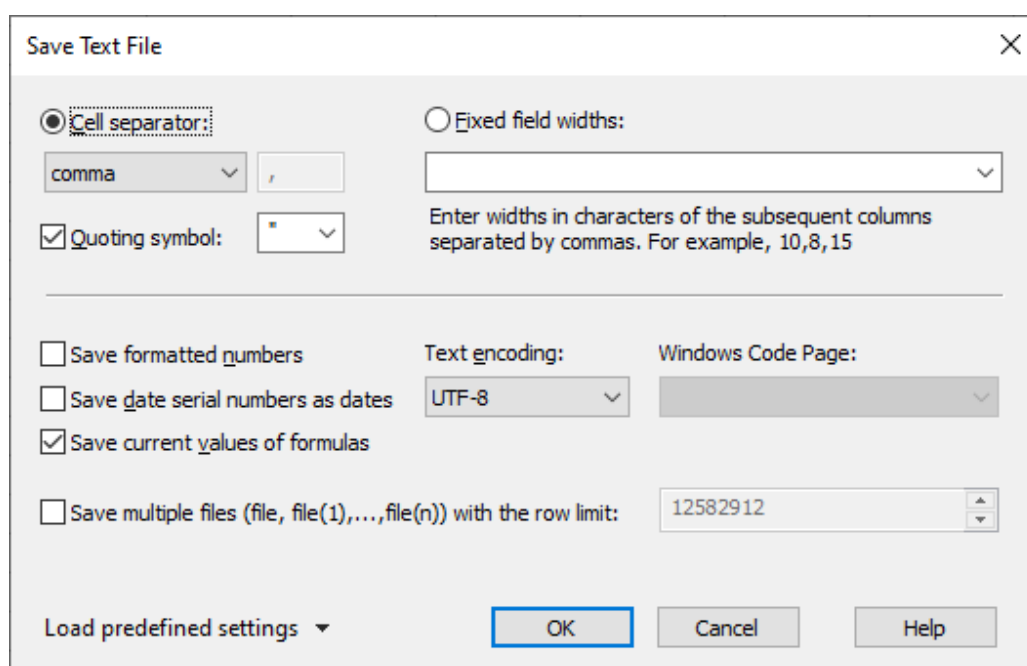
By default, choosing:

Text | Text Archive (*.txt, *.csv, *.tsv, *.zip)

saves only the current sheet as a standalone text file.

To save all sheets at once, specify a file name with the ***.zip** extension. This instructs GS-Base to save all text files into a single ZIP64 archive, along with a short XML file containing text parameters for future automatic loading.

When saving ZIP text archives, row-splitting options are disabled.



The "Save Text File" dialog box contains the following elements:

- Cell separator:** A radio button that is selected, with a dropdown menu showing "comma" and a small text input field containing a comma character.
- Fixed field widths:** An unselected radio button with an empty dropdown menu.
- Quoting symbol:** A checked checkbox with a dropdown menu showing a double quote character.
- Text encoding:** A dropdown menu showing "UTF-8".
- Windows Code Page:** A dropdown menu that is currently empty.
- Save formatted numbers:** An unchecked checkbox.
- Save date serial numbers as dates:** An unchecked checkbox.
- Save current values of formulas:** A checked checkbox.
- Save multiple files (file, file(1),...,file(n)) with the row limit:** An unchecked checkbox with a text input field showing "12582912" and a small up/down arrow.
- Enter widths in characters of the subsequent columns separated by commas. For example, 10,8,15:** A text input field.
- Buttons:** "OK", "Cancel", and "Help".
- Load predefined settings:** A dropdown menu.

4. Related Topics

[GS-Calc *.gsc and ODF *.ods files](#)

[Excel \(*.xlsx, *.xml\) files](#)

[PDF files](#)

[Text files \(*.csv, *.txt, *.tab ...\)](#)

[Text *.zip archives](#)

[GS-Base \(*.gsb\) files](#)

[dBase/Clipper/FoxPro files](#)

[SQLite \(*.db\) files](#)

[MySQL \(*.sql\) files](#)

Saving Cell Ranges As HTML/CSS Code

1. Overview

To copy (and save) a selected range of cells as HTML code, use the **Edit > Copy As HTML** command. You can paste that copied clipboard data directly into the <body> section of any HTML page as a single continuous block. When displayed in web browsers, the copied HTML/CSS data looks the same as in the source worksheet window, including styles and formatting, except for text rotation. Also, the **accounting** numeric style requires that the related cells in GS-Calc use a fixed-width (monospace) font, such as Courier New.

2. Examples

- Worksheet window screenshot:

N27

Empty

</

- The same worksheet pasted into a web browser as HTML/CSS data:

APPRAISAL REPORT				0	Commercial P&L	
Development Project Appraisal Model					Proportion	Values
ABC						
		£	£	0		£
REVENUE						
Commercial NCV	2,550,000 sq ft NIA		5,423,588,944	0		5,423,588,944
Residential NCV	3,400,000 sq ft NIA		5,391,011,488	0		
Total NCV	5,950,000 sq ft NIA		10,814,600,432			
Commercial Rent Received			280,193,634	0		280,193,634
Total Revenue		£	11,094,794,066	0		5,703,782,578
COSTS						
Land	10.0000 acre		1,135,328,314	0		
SDLT	4.70% on land		53,360,431	0		
Acquisition Fees	1.50% on land		17,029,925	0		
			1,205,718,670	50.15%		604,675,364
Construction - Commercial	3,000,000 sq ft GRS		1,685,771,834	0		1,685,771,834
Construction - Residential	4,000,000 sq ft GRS		1,831,723,207	0		
Construction - Overall			-0	42.86%		-0
	7,000,000 sq ft GRS		3,517,495,041	0		
plus:	0 pkg spaces					
Professional Fees	35.12% on const'n		1,235,352,931	-0		592,047,224
Construction Cost Contingency	10.00% on const'n		351,749,504	0		168,577,183
			5,104,597,476			
Infrastructure			75,592,215	0		
Pre-Construction			1,015,951	-0	42.86%	435,408
Void - Commercial			102,403,810	0		102,403,810
Letting (Commercial)			37,526,879	0		37,526,879
Selling (Commercial)	1.00% on NCV		54,235,889	0		54,235,889
Selling (Residential)	3.00% on NCV		161,730,345	0		
Marketing	1.72% on NCV		185,732,076	-0	50.15%	93,145,784
Other Costs			60,573,703	0	42.86%	25,960,159
Development Cost Contingency	1.25% on dev costs		7,559,221	-0		3,931,224
			610,777,874			
Overall Contingency	0.64% on C&D		36,771,432	-0		17,783,178
Total Costs Before Interest			7,033,457,667			3,386,493,936

Opening and Saving Files: xBase Files

1. Opening an xBase File

Use **File** → **Open** and choose one of the following formats from the **File of type** list:

- dBase III
- dBase IV
- FoxPro 2.x
- Clipper

Then specify additional xBase file options:

Encoding Specifies the text encoding for the 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 the red font color from the corresponding worksheet row.

2. Auto-Resizing Columns

To automatically resize columns when opening xBase files, enable:

Settings → Options → General → Auto-fit column widths in imported files

(See [Settings](#))

3. Saving an xBase File

Use **File → Save** and choose one of the following formats from the **File of type** list:

- dBase III
- dBase IV
- FoxPro 2.x
- Clipper

Then specify additional xBase file options:

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

To mark 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 **File → Reload** to refresh the view.

4. Related Topics

[GS-Calc *.gsc and ODF *.ods files](#)

[Excel \(*.xlsx, *.xml\) files](#)

[PDF files](#)

[Text files \(*.csv, *.txt, *.tab ...\)](#)

[Text *.zip archives](#)

[GS-Base \(*.gsb\) files](#)

[dBase/Clipper/FoxPro files](#)

[SQLite \(*.db\) files](#)

[MySQL \(*.sql\) files](#)

1. MySQL *.sql File Format Support

You can load, create, edit, and save databases in the ***.sql** file format using all types of SQL fields. Similar to the SQLite format, each table begins with a first row containing field descriptions: **field_name**, **field definition**.

	A	B	C	D	E	F	G	H
1	id, INT(11) NOT NULL AUTO_INCREMENT	md5, VARCHAR(32) NOT NULL	descr, MEDIUMTEXT NOT NULL	toc, MEDIUMTEXT NOT NULL	TimeLastModified, TIMESTAMP NOT NULL DEFAULT current_time			
2	81	ae10fb5c80b603c436a0d28fabbfb4f	Presents the basic theory of real a		2010-06-30 05:04:29			
3	82	ecee5cab217bd4b28573a3ecaf255207	The present Handbook has been d		2011-01-22 15:59:54			
4	83	b828328031b4bee4d3971026b5789e0c	Book DescriptionThis handbook is		2010-06-30 05:04:29			

2. Auto-Fit Column Widths

To automatically resize columns in an opened MySQL file, enable:

Settings > Options > General > Auto-fit column widths in imported files

(See [Settings](#))

3. Related Topics

[GS-Calc *.gsc and ODF *.ods files](#)

[Excel \(*.xlsx, *.xml\) files](#)

[PDF files](#)

[Text files \(*.csv, *.txt, *.tab ...\)](#)

[Text *.zip archives](#)

[GS-Base \(*.gsb\) files](#)

[dBase/Clipper/FoxPro files](#)

[SQLite \(*.db\) files](#)

[MySQL \(*.sql\) files](#)

*Opening and Saving Files: SQLite *.db Files*

1. SQLite *.db File Format Support

GS-Calc can load, create, edit, and save databases in the **SQLite *.db** file format. All SQLite field types are supported.

Each table stores field descriptions in the first row, using the format:

- **field_name**
- **field_definition**

	A	B	C	D	E	F	G
1	CustomerID, TEXT	CompanyName, TEXT	ContactName, TEXT	ContactTitle, TEXT	Address, TEXT	ID_1, INT	ID_2, DOUBLE
2	ALFKI	Alfreds Futterkiste	Maria Anders	Sales Representative	Obere Str. 57	1	1.5
3	ANATR	Ana Trujillo Emparedados	Ana Trujillo	Owner	Avda. de la Constitución 2222	2	2.5
4	ANTON	Antonio Moreno Taquería	Antonio Moreno	Owner	Mataderos 2312	3	3.5
5	AROUT	Around the Horn	Thomas Hardy	Sales Representative	120 Hanover Sq.	4	4.5
6	BERGS	Berglunds snabbköp	Christina Berglund	Order Administrator	Berguvsvägen 8	5	5.5

2. Auto-Resizing Columns

To automatically resize columns when opening SQLite files, enable:

Settings → Options → General → Auto-fit column widths in imported files

See also: [Settings](#).

3. Related Topics

[GS-Calc *.gsc and ODF *.ods files](#)

[Excel \(*.xlsx, *.xml\) files](#)

[PDF files](#)

[Text files \(*.csv, *.txt, *.tab ...\)](#)

[Text *.zip archives](#)

[GS-Base \(*.gsb\) files](#)

[dBase/Clipper/FoxPro files](#)

[SQLite \(*.db\) files](#)

[MySQL \(*.sql\) files](#)

Printing and Print Previewing

1. Overview

By default, choosing **Print** prints the entire current worksheet using the content options defined in **Print Worksheet > Content**.

2. Printing a Range of Cells

Select the desired range and choose **Selection** in **Print Worksheet > General**.

3. Printing Selected Pages

Specify the list of pages in **Print Worksheet > General**.

The list may contain single pages or ranges separated by commas, e.g.:

1,2,4-8,10

4. Printing an Inserted Chart or Image

Select the object before choosing **Print**. Charts are automatically stretched or shrunk to fit the printed page.

5. Adding Headers and Footers

In **Print > Layout**, enter header/footer text using these special codes:

&p	Current page number
&p+number	Page number plus “number”
&p-number	Page number minus “number”
&f	Full file path
&a	Worksheet name
&d	Current date
&t	Current time
&&	Literal ampersand

6. Changing the View Scale in Print Preview

Click **Zoom In**, then right-click and drag to select the rectangle to zoom into.

Settings

1. Overview

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

GS-Calc does not store settings in the Windows registry except optional COM/shell entries required by other applications. These entries may be added or removed using **Settings > Register-Unregister**. All program settings are stored in **settings.xml**.

2. Where settings.xml is loaded from

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

3. Where settings.xml is saved

1. The original file, if it exists
2. The local application data folder (if installed in the default “Programs Folder”)
3. The installation folder (if not installed in the default “Programs Folder”)

You can save or load different profiles using **Settings > Save – Load Profile**.

4. Undo/Redo Level

Number of actions that can be undone via **Edit/Undo-Redo**. Valid range: <2, 200> Default: 20

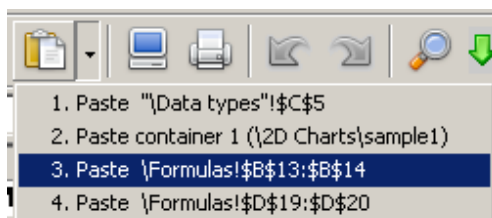
Notes:

- Very high undo levels + large copy/paste operations may significantly increase memory usage.
- Drag-drop copying counts as two actions; minimum undo level is 2.

5. Copy/Paste Level

Number of cyclic copy “slots”. Valid range: <1, 16> Default: 1

To paste older items, click the arrow next to **Paste**:



6. Formula Notation for New ODF (*.ods) Files

Select which formula namespace should be used when saving newly created ODS files:

- Excel 2010/2013 (default)
- OpenFormula
- Google Docs

Existing ODS files retain their notation unless changed via **Save As**.

Using the correct namespace is important — unsupported namespaces may cause formulas to be removed and replaced with values when opened in other applications.

7. Auto-Format Cells

If **on**, GS-Calc identifies the style of newly entered data and adjusts formatting accordingly.
Default: On

8. Use Date Picker

If **on**, the system Date Picker control appears when editing date cells. See: [editing and filling cells](#).
Default: On

9. Parse Dates in Formatted Cells Only

If **on**, date strings convert only in cells formatted as Date, Time, or Date/Time. Default: Off

10. Enable Syntax Coloring

If **on**, formulas edited in cells use syntax coloring (via the Scintilla project). Default: Off

11. Update Copied Absolute Cell References

Controls whether absolute references like **\$A\$1** update when copying or inserting/deleting rows/columns. Default: Off

12. Update Copied Relative Cell References

Controls whether relative references like **A1** update when copying or inserting/deleting rows/columns. Default: On

13. Hide Cell Markers

Hides markers for formulas, drop-down lists, comments, circular references (>1), and truncated text. Default: On

14. Show Progress When Saving/Loading Files

If **on**, a progress bar is displayed. Default: On

15. Display Formulas Instead of Values

Displays formulas instead of their results. Default: Off

16. Display References Using R1C1 Notation

Changes the cell reference notation. See: [Cell references](#). Must use the current notation when entering formulas. Internally, A1 notation is always used. Default: Off (A1)

17. Extended Toolbar Button Styles

Adds a third “undeterminate” state for formatting buttons when a selection contains mixed formatting. Default: Off

Note:

May significantly slow down selecting very large ranges.

18. Auto-Fit Column Widths in Imported Files

If **on**, after opening a **text** or **xBase** file, all non-empty columns are automatically resized to fit their contents.

The resizing respects the limits defined in **settings.xml**:

- **<AutoFitMin>** — minimum width (default: 20 px)
- **<AutoFitMax>** — maximum width (default: 640 px)

Both values may be modified within the range <2; 2000>.

Note:

May slow down opening very large text or xBase files.

Default: Off

19. Automatically Close Chart Dialog Box

If **on**, the Chart dialog closes when switching worksheets. Leaving it open is useful when the chart is in one sheet and its data in another. Default: Off

20. Cell Text Overflow

Controls whether and how text longer than the cell width is truncated. Default: Allow if adjoining cells are empty

21. After Selecting Cells

Controls whether GS-Base scrolls back to the selection origin. If off, click the address in the toolbar to return. Default: No action

22. Mouse Wheel

Controls how GS-Base reacts to a single wheel movement:

- scroll lines
- scroll pages

Default: Scroll lines

23. After Pressing Enter

Controls scrolling after finishing cell editing. Default: No action

24. Start Folder

Specifies the folder shown first in the **Open File** dialog after launching GS-Calc.

25. AutoSave After

Period of inactivity after which GS-Calc automatically saves all open workbooks. Default: Never

26. AutoClose After

Period of inactivity after which GS-Calc automatically closes all open workbooks. Modified workbooks are saved before closing. Default: Never

27. Windows Opacity

Dialog box transparency. Valid range: <10, 100> Default: 100%

28. Default Font

Specifies the default font name. Default: Arial

29. Recursion Level

Number of circular reference iterations. Level ≥ 2 displays circular markers. Use "Inspect Cells" → "Add" to list circular formulas. Valid range: <1, 255> Default: 1

30. Perform Background Calculations

If **on**, formulas update in background processes. Updated results appear only after completion. No partial updates are possible.

Editing cell data restarts the update cycle; formatting does not.

The status bar shows progress and number of active processes.

Background updating requires more memory. Default: On

31. Keep Workbooks Referenced in Calculations Open

If **on**, referenced workbooks remain open until the referencing workbook closes. If **off**, referenced workbooks open and close repeatedly during recalculation. Default: On

32. Number of CPUs

Number of CPUs/threads used for updating formulas, auto-fitting widths, and some editing actions.

Using more threads than available cores does not improve performance.

This value applies to workbooks whose individual “Number of CPUs” setting is “Default”. Default: 2

33. Threads Priority

Priority of update threads (Windows priority classes). High priorities may hamper other system processes.

Applies to workbooks whose individual “Threads priority” is “Default”. Default: 2

34. Automatic Update Mode

Recalculates the entire workbook after each cell edit.

Applies to workbooks whose individual “Update Mode” is “Default”. Default: On

35. Favor Compound Formulas

Optimizes calculations assuming formulas use both built-in functions and references.

Applies to workbooks whose individual “Update Optimization Options” are “Default”. Default: On

36. Assume Short Calculation Chains

Optimizes calculations assuming worksheets contain short linked cell groups.

Applies to workbooks whose individual “Update Optimization Options” are “Default”. Default: On

37. Bottom-Up Calculation

Recalculates cells from right to left and bottom to top (reverse of default order).

Applies to workbooks whose individual “Update Optimization Options” are “Default”. Default: Off

GS-Calc 11 introduced the following changes concerning the behavior of a few formulas, to ensure better ODF file compatibility. If you're opening a file created by version 10.x or 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 included.
 - **GS-Calc 11 and newer:**
Text representations of numbers in references and arrays are not included.
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 included.
 - **GS-Calc 11 and newer:**
Text representations of numbers in references and arrays are not included.
You can also use the AVERAGEA(), MINA(), MAXA() and COUNTA() functions, respectively, to include text representations of numbers.
- **INDEX(range, row, column)**
 - **GS-Calc 10.x and older:**
Specifying 0 as either the row or the column argument causes the function to return the #NUM! error.
 - **GS-Calc 11 and newer:**
Specifying 0 as either the row or the column argument has the same meaning as leaving the parameter blank: it returns either the full column or the full row of the range, respectively.
- **MATCH(), VLOOKUP(), HLOOKUP()**
 - **GS-Calc 10.x and older:**
Specifying 1 or -1 as the search mode argument requires the searched range to be in ascending order.
 - **GS-Calc 11 and newer:**
Specifying -1 as the search mode argument requires the searched range to be in descending order.
- **NoPattern and StringSort flags in MATCH(), VLOOKUP(), HLOOKUP(), PIVOTDATA()**
 - **GS-Calc 11.x:**
These 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 applied (that is, punctuation marks are not ignored).

Enabling Scripting

To use Python scripting, you must have the Python package installed. Enabling Python scripting requires the same configuration steps as using Python UDF functions and formulas. Please refer to [Installing Python and integrating it with GS-Calc](#) for detailed instructions.

Creating Scripts

You can create scripts as **global scripts**, which are saved in the program settings and available to all workbooks, or as **local scripts** stored within a specific GS-Calc workbook, which are only available when that workbook is open.

To create these scripts, use the **File > Application Scripts** or **File > Database Scripts** commands.

The "Scripts" dialog box allows you to organize your scripts into subfolders, as well as copy, import, or export them.

[View a sample "Scripts" dialog screen.](#)

Notes:

- If a global script is executed while no workbook is open, it should begin with one of the application [Open\(...\)](#) or [New\(..\)](#) functions.
- If a workbook is already loaded, an executed script (whether global or local) automatically refers to that active workbook. Using the [Open\(...\)](#) functions with the same workbook path will have no effect.
- Backslashes used in script text, primarily within file path parameters, must always be doubled. For example: `f:\my_folder\file01.gsc` .

GS-Calc Functions Available in Scripts

- [XBaseParams](#)
 - [TextParams](#)
 - [MergeParams](#)
 - [FormatParams](#)
 - [Worksheet](#)
 - [Workbook](#)
 - [Application](#)
-

XBaseParams

setFormat(str format) -> bool / getFormat() -> str

A string specifying the xBase file format:

- dbaseIII
- dbaseIV
- clipper
- foxpro

Example:

```
xBaseParams = gscalc.CreateXBaseParams();  
xBaseParams.setFormat("dbaseIV");
```

setEncoding(str encoding) -> bool / getEncoding() -> str

A string specifying the character encoding in the xBase file:

- windows
- dos

Example:

```
xBaseParams = gscalc.CreateXBaseParams();  
xBaseParams.setEncoding("dos");
```

GetFieldCount()

Returns the number of defined database fields.

Example:

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

SetField(int index, str name, str type, int length, int decimals) -> bool

AddField(str name, str type, int length, int decimals) -> bool

InsertField(int index, str name, str type, int length, int decimals) -> bool

Updates the name and type of an existing database field.

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:

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

GetFieldName(int index) -> str

Returns the name of a given database field.

index

A one-based index of the field.

Example:

```
xBaseParams = gscalC.CreateXBaseParams();  
xBaseParams.AddField("name", "C", 40, 0);  
name = GetFieldName(1); # returns "name"
```

GetFieldType(int index) -> str

Returns the type of a given database field.

index

A one-based index of the field.

Example:

```
xBaseParams = gscalC.CreateXBaseParams();  
xBaseParams.AddField("name", "C", 40, 0);  
name = GetFieldType(1); # returns "C"
```

GetFieldLength(int index) -> int

Returns the length of a given database field.

index

A one-based index of the field.

Example:

```
xBaseParams = gscalC.CreateXBaseParams();  
xBaseParams.AddField("name", "C", 40, 0);  
name = GetFieldLength(1); # returns 40
```

GetFieldDecimals(int index) -> int

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

index

A one-based index of the field.

Example:

```
xBaseParams = gscalC.CreateXBaseParams();  
xBaseParams.AddField("name", "C", 40, 0);  
name = GetFieldLength(1); # returns 40
```

DeleteField(int index) -> bool

Deletes a given database field.

index

A one-based index of the field.

Example:

```
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);  
name = DeleteField(3);
```

TextParams

setSeparator(str separator) -> None / getSeparator() -> str

Specifies a character separating columns in a text file.

Example:

```
textParams1 = gscalC.CreateTextParams();  
textParams1.setSeparator("\t"); # tab-separated values  
textParams2 = gscalC.CreateTextParams();
```

```
textParams2.setSeparator(","); # comma-separated values
```

setEncoding(str encoding) -> bool / getEncoding() -> str

A string specifying the character encoding in the text file:

- utf8
- utf16
- windows
- dos

Example:

```
textParams = gscalc.CreateTextParams();  
textParams.setEncoding("utf8");
```

setQuotingSymbol(str symbol) -> None / getQuotingSymbol() -> str

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

Example:

```
textParams = gscalc.CreateTextParams();  
textParams.setQuotingSymbol("\"");
```

setLoadNumbers(bool value) -> None / getLoadNumbers() -> bool

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:

```
textParams = gscalc.CreateTextParams();  
textParams.setLoadNumbers(True);
```

setLoadNumbers(bool value) -> None / getLoadFmtNumbers() -> bool

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:

```
textParams = gscalc.CreateTextParams();  
textParams.setLoadFmtNumbers(True);
```

setLoadDates(bool value) -> None / getLoadDates() -> bool

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:

```
textParams = gscalc.CreateTextParams();  
textParams.setLoadDates(False);
```

setLoadDateStyles(bool value) -> None / getLoadDateStyles() -> bool

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:

```
textParams = gscalc.CreateTextParams();  
textParams.setLoadDateStyles(False);
```

setParsingMode(int parsing) -> bool / getParsingMode() -> int

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:


```
textParams = gscalc.CreateTextParams();  
textParams.setParsingMode(3);
```

setColumnWidths(str columnWidths) -> bool / getColumnWidths() -> str

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:

```
textParams = gscalc.CreateTextParams();  
textParams.setColumnWidths("10, 30, 15, 20");
```

setSaveFmtNumbers(bool value) -> None / getSaveFmtNumbers() -> bool

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:

```
textParams = gscalc.CreateTextParams();  
textParams.setSaveFmtNumbers(False);
```

setSaveFmtNumbers(bool value) -> None / getSaveFmtDates() -> bool

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

Example:

```
textParams = gscalc.CreateTextParams();  
textParams.setSaveFmtDates(True);
```

setSaveFormulaValues(bool value) -> None / getSaveFormulaValues() -> bool

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

Example:

```
textParams = gscalc.CreateTextParams();  
textParams.setSaveFormulaValues(False);
```

setAutoFitColumns(bool value) -> None / getAutoFitColumns() -> bool

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:

```
textParams = gscalc.CreateTextParams();  
textParams.setAutoFitColumns(False);
```

MergeParams

A set of parameters used by rows-merging functions.

- **setPath(str path) -> None / getPath() -> str**

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.

- **setTable(str table) -> None / getTable() -> str**

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.

- **setMasterField(int field) -> None / getMasterField() -> int**

- **setSlaveField(int field) -> None / getSlaveField() -> int**

- **setMergeType(int type) -> None / getMergeType() -> int**

- **setFieldNames(bool val) -> None / getFieldNames() -> bool**

Specifies whether merged tables contain field/column names in the 1st row (which are then excluded from merging). 1/0, True/False

Default: 0

- **setMatchFieldNames(bool val) -> None / getMatchFieldNames() -> bool**

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

- **setIgnoreEmpty(bool val) -> None / getIgnoreEmpty() -> bool**

- **setEnableUndo(bool val) -> None / getEnableUndo() -> bool**

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:

```
mergeParams = gscal.CreateMergeParams();
mergeParams.setPath("e:\\data_files\\*.gsc");
mergeParams.setTable("customers");
mergeParams.setTable("pivot table reports\\data");
mergeParams.setFieldNames(1);
mergeParams.setMatchFieldNames(0);
mergeParams.setEnableUndo(0);
```

FormatParams

setDataStyleName(str name) -> bool / getDataStyleName() -> str

A string specifying the predefined data style name:

- Default
- General
- Currency
- Accounting
- Date
- Time

- Date/Time
- Fraction
- Percentage
- Scientific
- Custom

Example:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
formatParams = wsheet.CreateFormatParams();
formatParams.setDataStyleName("Currency");
```

SetGeneralNumberFormat(str decimals, BYTE zeroes, bool brackets, bool inRed, bool separators, int scaling) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
formatParams = wsheet.CreateFormatParams();
formatParams.SetGeneralNumberFormat(4, 1, False, False,
False, 0);
```

```
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

SetCurrencyFormat(str decimals, str position, str symbol, bool brackets, bool inRed, int scaling) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.SetCurrencyFormat(0, "$1.1", "$", False, False,  
0);  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

SetAccountingFormat(str decimals, str symbol, int scaling) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.SetAccountingFormat(0, "$", 0);  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

SetDateFormat(str pattern, bool systemOrder) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.SetDateFormat("m/d", True);  
wsheet.setSelectedRange("d4");
```

```
wsheet.SetCellFormat(formatParams);
```

SetTimeFormat(str pattern) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.SetTimeFormat("hh:mm AM/PM");  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

SetDateTimeFormat(str datePattern, str timePattern, bool systemOrder, bool timeFirst) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.SetDateTimeFormat("m/d", "hh:mm AM/PM", True,  
False);  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

SetPercentFormat(str decimals, int scaling) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
formatParams = wsheet.CreateFormatParams();
formatParams.SetPercentFormat(0, 0);
wsheet.setSelectedRange("d4");
wsheet.SetCellFormat(formatParams);
```

SetFractionFormat(long denominator, bool digits) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
formatParams = wsheet.CreateFormatParams();
formatParams.SetFractionFormat(2, False);
wsheet.setSelectedRange("d4");
wsheet.SetCellFormat(formatParams);
```

SetScientificFormat(str decimals, const str exponent) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
formatParams = wsheet.CreateFormatParams();
formatParams.SetScientificFormat("auto", 12);
wsheet.setSelectedRange("d4");
wsheet.SetCellFormat(formatParams);
```

setFontName(str name) -> bool / getFontName() -> str

A string specifying the font name. To specify the default font name, use an empty string.

Example:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
formatParams = wsheet.CreateFormatParams();
formatParams.setFontName("Tahoma");
wsheet.setSelectedRange("d4");
wsheet.SetCellFormat(formatParams);
```

setFontSize(int size) -> bool / getFontSize() -> int

The size of the font in points. To specify the default size, use zero.

Example:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
formatParams = wsheet.CreateFormatParams();
formatParams.setFontSize(8);
wsheet.setSelectedRange("d4");
wsheet.SetCellFormat(formatParams);
```

setLanguage(str language) -> bool / getLanguage() -> str

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:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.setLanguage("en-GB");  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

setBoldFont(bool bold) -> bool / getBoldFont() -> bool

Toggles the "bold" font attribute on/off.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.setBoldFont(True);  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

setItalicFont(bool italic) -> bool / getItalicFont() -> bool

Toggles the "italic" font attribute on/off.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.setItalicFont(True);  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

setUnderlineFont(bool uline) -> bool / getUnderlineFont() -> bool

Toggles the "underline" font attribute on/off.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.setUnderlineFont(True);  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

setStrikeoutFont(bool sline) -> bool / getStrikeoutFont() -> bool

Toggles the "strikeout" font attribute on/off.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.setStrikeoutFont(True);  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

setFontColor(str color) -> bool / getFontColor() -> str

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:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.setFontColor("#FF0000");  
formatParams.setFontColor("green");  
formatParams.setFontColor("auto");  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

setHorzAlignment(str horzAlign) -> bool / getHorzAlignment() -> str

Specifies the horizontal text alignment in cells. Accepted values are the following strings:

- left
- center
- right
- default

Example:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
formatParams = wsheet.CreateFormatParams();
formatParams.setHorzAlignment("center");
wsheet.setSelectedRange("d4");
wsheet.SetCellFormat(formatParams);
```

setVertAlignment(str vertAlign) -> bool / getVertAlignment() -> str

Specifies the vertical text alignment in cells. Accepted values are the following strings:

- top
- center
- bottom
- base-line
- default

Example:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
formatParams = wsheet.CreateFormatParams();
formatParams.setVertAlignment("center");
wsheet.setSelectedRange("d4");
wsheet.SetCellFormat(formatParams);
```

setHorzIndent(int indent) -> bool / getHorzIndent() -> int

Specifies the horizontal text indent in cells. Accepted values are between 0 and 255. The default value is 4.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.setHorzIndent(4);  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

setVertIndent(int indent) -> bool / getVertIndent() -> int

Specifies the vertical text indent in cells. Accepted values are between 0 and 255. The default value is 2.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.setVertIndent(1);  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

setWrapText(bool value) -> bool / getWrapText() -> bool

A logical value that toggles wrapping text in cells on/off. Setting this property to True automatically sets the shrinkText property to False.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.setWrapText(True);  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

setShrinkText(bool value) -> bool / getShrinkText() -> bool

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:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.setShrinkText(True);  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

setTextRotation(int rotation) -> bool / getTextRotation() -> int

Specifies the rotation of the text in cells. Acceptable values are from -90 to 90 degrees.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.setTextRotation(25);  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

setHideFormula(bool value) -> bool / getHideFormula() -> bool

A logical value that toggles hiding formulas on/off.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.setHideFormula(True);  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

setProtectedCell(bool value) -> bool / getProtectedCell() -> bool

A logical value that toggles cell protection on/off.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();
```

```
formatParams.setProtectedCell(True);  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

setPrintedCell(bool value) -> bool / getPrintedCell() -> bool

A logical value that toggles printing cell contents on/off.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.setPrintedCell(True);  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

setBorderStyle(str style) -> bool / getBorderStyle() -> str

Specifies the cell border style. The following style names are accepted:

- none
- solid
- dot
- dash
- dash-dot
- dash-dot-dot

Specifying "none" removes all borders around the cell.

Example:

```
wbook = gscalc.ActiveWorkbook(); wsheet = wbook.GetActiveWorksheet(); formatParams =  
wsheet.CreateFormatParams(); formatParams.setBorderColor("green");  
formatParams.setBorderPosition(15); formatParams.setBorderStyle("solid");  
formatParams.setBorderWidth(1); formatParams.setBorderDoubleLine(True);  
wsheet.setSelectedRange("d4"); wsheet.SetCellFormat(formatParams);
```

setBorderWidth(int width) -> bool / getBorderWidth() -> int

Specifies the cell border width. Accepted values are from 1 to 16 logical pixels.

Example:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
formatParams = wsheet.CreateFormatParams();
formatParams.setBorderColor("green");
formatParams.setBorderPosition(15);
formatParams.setBorderStyle("solid");
formatParams.setBorderWidth(1);
formatParams.setBorderDoubleLine(True);
wsheet.setSelectedRange("d4");
wsheet.SetCellFormat(formatParams);
```

setBorderDoubleLine(bool doubleLine) -> bool / getBorderDoubleLine() -> bool

A logical value that toggles displaying double borders.

Example:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
formatParams = wsheet.CreateFormatParams();
formatParams.setBorderColor("green");
formatParams.setBorderPosition(15);
formatParams.setBorderStyle("solid");
formatParams.setBorderWidth(1);
formatParams.setBorderDoubleLine(True);
wsheet.setSelectedRange("d4");
wsheet.SetCellFormat(formatParams);
```

setBorderColor(str color) -> bool / getBorderColor() -> str

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:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
formatParams = wsheet.CreateFormatParams();
formatParams.setBorderColor("green");
formatParams.setBorderPosition(15);
formatParams.setBorderStyle("solid");
```



```
formatParams.setBorderWidth(1);
formatParams.setBorderDoubleLine(True);
wsheet.setSelectedRange("d4");
wsheet.SetCellFormat(formatParams);
```

setBorderPosition(int pos) -> bool / getBorderPosition() -> int

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:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
formatParams = wsheet.CreateFormatParams();
formatParams.setBorderColor("green");
formatParams.setBorderPosition(15);
formatParams.setBorderStyle("solid");
formatParams.setBorderWidth(1);
formatParams.setBorderDoubleLine(True);
wsheet.setSelectedRange("d4");
wsheet.SetCellFormat(formatParams);
```

setBkgColor(str color) -> bool / getBkgColor() -> str

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:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
formatParams = wsheet.CreateFormatParams();
formatParams.setBkgColor("#FF0000");
formatParams.setBkgColor("green");
formatParams.setBkgColor("auto");
wsheet.setSelectedRange("d4");
wsheet.SetCellFormat(formatParams);
```

setBkgImageName(str name) -> bool / getBkgImageName() -> str

Specifies the cell background image. The name must represent an image available via the workbook list of images (see: "Cell Format > Background > Images").

Example:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
formatParams = wsheet.CreateFormatParams();
formatParams.setBkgImageName("blue-sphere.png");
formatParams.setBkgImageOpacity(100);
formatParams.setBkgImageHorzPos("left");
formatParams.setBkgImageVertPos("center");
wsheet.setSelectedRange("d4");
wsheet.SetCellFormat(formatParams);
```

setBkgImageRepeat(str repeat) -> bool / getBkgImageRepeat() -> str

Specifies how the cell background image should be displayed within the cell. The following text string 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:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
formatParams = wsheet.CreateFormatParams();
formatParams.setBkgImageName("blue-sphere.png");
formatParams.setBkgImageRepeat("repeat");
formatParams.setBkgImageOpacity(100);
formatParams.setBkgImageHorzPos("left");
```

```
formatParams.setBkgImageVertPos("center");  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

setBkgImageHorzPos(str horzPos) -> bool / getBkgImageHorzPos() -> str

Specifies the horizontal position of the cell background image. The following text string values are accepted:

- left
- center
- right

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.setBkgImageName("blue-sphere.png");  
formatParams.setBkgImageHorzPos("right");  
formatParams.setBkgImageOpacity(100);  
formatParams.setBkgImageHorzPos("left");  
formatParams.setBkgImageVertPos("center");  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

setBkgImageVertPos(str vertPos) -> bool / getBkgImageVertPos() -> str

Specifies the vertical position of the cell background image. The following text string values are accepted:

- top
- center
- bottom

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.setBkgImageName("blue-sphere.png");  
formatParams.setBkgImageVertPos("top");  
formatParams.setBkgImageOpacity(100);  
formatParams.setBkgImageHorzPos("left");
```

```
formatParams.setBkgImageVertPos("center");  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

setBkgImageOpacity(int opacity) -> bool / getBkgImageOpacity() -> int

Specifies the cell background image opacity. This is a numeric value between 0 (a fully transparent image) to 100 (a fully opaque image).

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.setBkgImageName("blue-sphere.png");  
formatParams.setBkgImageVertPos("top");  
formatParams.setBkgImageOpacity(70);  
formatParams.setBkgImageHorzPos("right");  
formatParams.setBkgImageVertPos("center");  
wsheet.setSelectedRange("d4");  
wsheet.SetCellFormat(formatParams);
```

Reset() -> None

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:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.setBoldFont(True);  
formatParams.setItalicFont(True);  
wsheet.setSelectedRange("b10:b20");  
wsheet.SetCellFormat(formatParams);
```

is the equivalent of clicking the "Bold" toolbar button, then the "Italic" format button. If the following code reuses the same FormatParams object without resetting:

```
formatParams.setDataStyleName("Currency");  
wsheet.setSelectedRange("b10:b20");
```

```
wsheet.SetCellFormat(formatParams);
```

it is the equivalent of clicking the "Bold" toolbar button, then the "Italic" format button, then choosing the "Currency" style.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.Reset();
```

Worksheet

setSelectedRange(str range) -> None / getSelectedRange() -> str

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:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
wsheet.setSelectedRange("c20");  
wsheet.UpdateWindow();  
gscalc.Sleep(2000);  
wsheet.setSelectedRange("c20:d22");  
wsheet.UpdateWindow();  
gscalc.Sleep(2000);  
wsheet.setSelectedRange("2:2");  
wsheet.UpdateWindow();  
gscalc.Sleep(2000);  
wsheet.setSelectedRange("H:I");
```

setTopLeftCell(str range) -> None / getTopLeftCell() -> str

Specifies the top-left cell of the worksheet window.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
wsheet.setTopLeftCell("c20");
```

Scroll(str type, bool scrollSelection) -> None

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 execution 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:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
for i in range(0, 50) :  
    wsheet.Scroll("line-down", True);  
    gscalc.Sleep(200);
```

Address(int column, bool relativeColumn, int row, bool relativeRow) -> str

Creates and returns 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:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
cell = wsheet.Address(3, True, 20, False); # returns "C$20"  
cell = wsheet.Address(3, False, 0, False); # returns "$C"
```

RC(str range) -> int left, int top, int right, int bottom

Returns four integers representing the top-left and bottom-right coordinates of the specified cell or range. The numbers are returned in the following order: the left column, the top row, the right column and the bottom row.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
cell = wsheet.RC("b10") # returns 2, 10, 2, 10  
cell = wsheet.RC("b10:c20"); # returns 2, 10, 3, 20
```

Column(str cell) -> int

Returns the column number of the specified cell address.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
column = wsheet.Column("C20"); # returns 3
```

Row(str cell) -> int

Returns the column number of the specified cell address.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveSheet();  
column = wsheet.Column("C20"); # returns 20
```

GetFirstColumn() -> int

Returns the number of the first non empty column.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveSheet();  
column = wsheet.GetFirstColumn();
```

GetFirstRow() -> int

Returns the number of the first non empty row.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveSheet();  
row = wsheet.GetFirstRow();
```

GetLastColumn() -> int

Returns the number of the last non empty column.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveSheet();  
column = wsheet.GetLastColumn();
```

GetLastRow() -> int

Returns the number of the last non empty row.

Example:


```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
row = wsheet.GetLastRow();
```

GetMaxColumn() -> int

Returns the highest usable column number.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
column = wsheet.GetMaxColumn();
```

GetMaxRow() -> int

Returns the highest usable row number.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
row = wsheet.GetMaxRow();
```

CreateFormattingParams() -> FormatParams

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:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
formatParams.setBoldFont(True);  
formatParams.setItalicFont(True);  
wsheet.setSelectedRange("b1:b20");  
wsheet.SetCellFormat(formatParams);
```

GetCellFormat(FormatParams params) -> None

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:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
wsheet.setSelectedRange("b10:b20");  
wsheet.GetCellFormat(formatParams);
```

GetColumnFormat(FormatParams params) -> None

Retrieves a column format relative to the selected cell or range.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
wsheet.setSelectedRange("b10:b20");  
wsheet.GetColumnFormat("b10:b20", formatParams);
```

GetRowFormat(FormatParams params) -> None

Retrieves a row format relative to the selected cell or range.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
formatParams = wsheet.CreateFormatParams();  
wsheet.setSelectedRange("b10:b20");  
wsheet.GetRowFormat(formatParams);
```

SetCellFormat(FormatParams params) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
formatParams = wsheet.CreateFormatParams();
formatParams.setBoldFont(True);
formatParams.setItalicFont(True);
wsheet.setSelectedRange("b10:b20");
wsheet.SetCellFormat(formatParams);
```

SetCellCustomStyle(str name) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
wsheet.setSelectedRange("b10:b20");
wsheet.SetCellCustomStyle("cell style 1.");
```

DeleteCells(bool dialogBox, str contentType) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
wsheet.setSelectedRange("c10:c20");
wsheet.DeleteCells(False, "data");
wsheet.setSelectedRange("d10");
wsheet.DeleteCells(False, "formatting");
wsheet.setSelectedRange("b4");
wsheet.DeleteCells(True, "formatting, lists, comments");
```

CopyCells(int copyType) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
wsheet.setSelectedRange("c10:c20");
wsheet.CopyCells(0);
wsheet.setSelectedRange("d10");
wsheet.PasteCells(True);
wsheet.setSelectedRange("b4");
wsheet.CopyCells(0);
wsheet.setSelectedRange("b5");
wsheet.PasteCells(True);
wbook.WaitForUpdate(); # to allow the background updating to
complete before further actions
# ...
```

PasteCells(bool textAndFormatting) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
wsheet.setSelectedRange("c10:c20");
wsheet.CopyCells(0);
wsheet.setSelectedRange("d10");
wsheet.PasteCells(True);
wsheet.setSelectedRange("b4");
wsheet.CopyCells(0);
wsheet.setSelectedRange("b5:b10");
wsheet.PasteCells(True);
```

GetColumnWidth() -> str

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:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
width = wsheet.GetColumnWidth();
```

SetColumnWidth(str width) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
wsheet.setSelectedRange("b5:e5");  
wsheet.SetColumnWidth("auto");  
wsheet.setSelectedRange("f5");  
wsheet.SetColumnWidth("30");
```

GetRowHeight() -> str

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:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
height = wsheet.GetRowHeight();
```

SetRowHeight(str height) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
wsheet.setSelectedRange("b5:e5");  
wsheet.SetRowHeight("auto");  
wsheet.setSelectedRange("f5");  
wsheet.SetRowHeight("30");
```

InsertColumns(int type) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveSheet();
wsheet.setSelectedRange("b5:e5");
wsheet.InsertColumns(1);
wsheet.setSelectedRange("f5");
wsheet.InsertColumns(3);
wbook.WaitForUpdate(); # to allow the background updating to
complete before further actions
# ...
```

DeleteColumns() -> bool

Deletes columns based on the currently selected range.

Example:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveSheet();
wsheet.setSelectedRange("b5:e5");
wsheet.DeleteColumns();
wsheet.setSelectedRange("f5");
wsheet.DeleteColumns();
```

InsertRows(int type) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
wsheet.setSelectedRange("b5:b10");
wsheet.InsertRows(1);
wsheet.setSelectedRange("b11");
wsheet.InsertRows(3);
wbook.WaitForUpdate(); # to allow the background updating to
complete before further actions
# ...
```

DeleteRows() -> bool

Deletes rows based on the currently selected range.

Example:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
wsheet.setSelectedRange("b5:b10");
wsheet.DeleteRows(1);
wsheet.setSelectedRange("b11");
wsheet.DeleteRows(3);
```

InsertSeries() -> bool

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:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
wsheet.setSelectedRange("b5:b100");
```



```
wsheet.InsertSeries();  
wbook.WaitForUpdate(); # to allow the background updating to  
complete before further actions  
# ...
```

IsText(str cell-or-range) -> bool / IsTextRC(int row, int column) -> bool

Returns **True** if the specified cell contains a text or 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:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
check1 = wsheet.IsText("b5:b100");  
check2 = wsheet.IsText("b101");
```

IsNumber(str cell-or-range) -> bool / IsNumberRC(int row, int column) -> bool

Returns **True** if 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:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
check1 = wsheet.IsNumber("b5:b100");  
check2 = wsheet.IsNumber("b101");
```

IsFormula(str cell-or-range) -> bool / IsFormulaRC(int row, int column) -> bool

Returns **True** if 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:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
check1 = wsheet.IsFormula("b5:b100");  
check2 = wsheet.IsFormula("b101");
```

IsFormulaNumericRC(int row, int column) -> bool

IsFormulaTextualRC(int row, int column) -> bool

IsError(str cell-or-range) -> int / IsErrorRC(int row, int column) -> int

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:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
check1 = wsheet.IsError("b5:b100");  
check2 = wsheet.IsError("b101");
```

InsertData(str cell-or-range, str data, bool parse) -> bool

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 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 shown in the 2nd example below.

Example 1.:

```
wbook = gscalc.ActiveWorkbook();
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.:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
gscalc.MinimizeAppWindow();
updateMode = wbook.getUpdateMode();
wbook.setUpdateMode("manual");

for i in range(1, 200000) :
    ref = "b" + i;
    wsheet.InsertData(ref, "some text data", False);
wbook.setUpdateMode(updateMode);
gscalc.RestoreAppWindow();
```

InsertNumber(str cell, float value) -> bool / InsertNumberRC(int row, int column, float value) -> bool

InsertText(str cell, str text) -> bool / InsertTextRC(int row, int column, str text) -> bool

InsertFormula(str cell, str formula) -> bool / InsertFormulaRC(int row, int column, str formula) -> bool

InsertComments(str cell, str comments) -> bool / InsertCommentsRC(int row, int column, str comments) -> bool

Adds comments to specified cell. The **comments** argument can be any text string containing up to 1024 characters.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveSheet();  
wsheet.InsertComments("b5", "comments in b5");  
wsheet.InsertComments("b6", "comments in b6");
```

InsertMatrix(str topLeftCell, matrix) -> bool

GetData(str cell, bool formatted) -> str

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

If formatted=True, the returned string will include the corresponding formatting.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveSheet();  
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);  
sum = wsheet.GetData("b5:b10"); # sum = 12.8
```

GetNumber(str cell) -> float / GetNumberRC(int row, int column) -> float

GetText(str cell) -> str / GetTextRC(int row, int column) -> str

GetFormula(str cell) -> str / GetFormulaRC(int row, int column) -> str

Returns formula from the specified cell that contains some formula.

If the specified cell doesn't contain a formula, an empty string is returned.

Example:

```

wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveSheet();
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);
formula = wsheet.GetFormula("b11"); # formula =
"sum(b5:b10)"

```

GetComments(str cell) -> str / GetCommentsRC(int row, int column) -> str

Returns comments for the specified cell. If the cell doesn't have comments, an empty string is returned.

Example:

```

wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveSheet();
wsheet.InsertComments("b5", "comments in b5");
comments = wsheet.GetComments("b5"); # comments = "comments
in b5"

```

GetMatrix64F(str range) -> matrix (float64)

GetMatrix32F(str range) -> matrix (float32)

GetMatrix64I(str range) -> matrix (int64)

GetMatrix64U(str range) -> matrix (uint64)

GetMatrix32I(str range) -> matrix (int32)

GetMatrix32U(str range) -> matrix (uint32)

GetMatrix8I(str range) -> matrix (int8)

GetMatrix8U(str range) -> matrix (uint8)

GetSparseMatrix(str range, int type) -> array (uint32), array (uint32), array (float64)

InsertSparseMatrix(str topLeftCell, rows (array), columns (array), values (array)) -> bool

TerminateUpdateThreads() -> None

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:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
wsheet.setTopLeftCell("cv100");
wsheet.UpdateWindow();
gscalc.Sleep(3000);
wsheet.setSelectedRange("ab10");
wsheet.UpdateWindow();
```

SaveSelectionAsImage(str filePath) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();
wsheet = wbook.GetActiveWorksheet();
wsheet.setSelectedRange("b5:b10");
wsheet.SaveSelectionAsImage("d:\\image1.png");
```

Print(bool dialogBox, str range, str pageList) -> bool

Prints the active worksheet. The "rangeType" string can be one of the following:

- 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. If the "pages" type is specified, the pageList should be a string consisting of comma-separated page numbers.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.GetActiveWorksheet();  
wsheet.Print(False, "pages", "2,3,6-7");
```

Workbook

ActiveWorksheet() -> Sheet

Returns the active worksheet object. All methods called on this object always refers to the currently selected worksheet.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wbook.SelectTreeItem("sheet1");  
wsheet = wbook.GetActiveWorksheet();  
wsheet.InsertData("b5", "data entered in sheet1!b5", True);  
wbook.SelectTreeItem("sheet2");  
wsheet.InsertData("b5", "data entered in sheet2!b5", True);
```

SetActiveWorksheet(str path) -> bool

SetActiveFolder(str path) -> bool

GetActiveFolderPath() -> str

GetActiveWorksheetPath() -> str

GetWorksheetCount() -> int

Returns the number of worksheets in the workbook.

Example:

```
wbook = gscalc.ActiveWorkbook();  
counter = wbook.GetWorksheetCount();
```

GetFolderCount() -> int

Returns the number of folders in the workbook.

Example:

```
wbook = gscalc.ActiveWorkbook();  
counter = wbook.GetFolderCount();
```

IsFolderEmpty(str folder) -> int

Returns **True** if the specified folder doesn't contain any tree items (worksheets or folders) and **False** otherwise.

Example:

```
wbook = gscalc.ActiveWorkbook();  
empty = wbook.IsFolderEmpty("folder1");
```

GetFirstTreeItem(str folder) -> str

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 path.

Example:

```
wbook = gscalc.ActiveWorkbook();  
item1 = wbook.GetFirstTreeItem("");  
if wbook.IsFolder(item1) :  
    item2 = wbook.GetFirstTreeItem(item1);  
    # ...
```

GetPrevTreeItem(str path) -> str

Returns the previous sibling tree item (a worksheet or a folder) for the specified worksheet tree item.

Example:

```
wbook = gscalc.ActiveWorkbook();
item1 = wbook.GetFirstTreeItem("");
item2 = wbook.GetNextTreeItem(item1);
item3 = wbook.GetPrevTreeItem(item2);
# item1 == item3 if there are at least two sibling items in
the main folder of the worksheet tree.
item1 = wbook.GetFirstTreeItem("folder1\\subfolder2");
item2 = wbook.GetNextTreeItem(item1);
item3 = wbook.GetPrevTreeItem(item2);
# item1 == item3 if there are at least two sibling items in
the "subfolder2" folder.
```

GetNextTreeItem(str path) -> str

Returns the next sibling tree item (a worksheet or a folder) for the specified worksheet tree item.

Example:

```
wbook = gscalc.ActiveWorkbook();
item1 = wbook.GetFirstTreeItem("");
item2 = wbook.GetNextTreeItem(item1);
item3 = wbook.GetPrevTreeItem(item2);
# item1 == item3 if there are at least two sibling items in
the main folder of the worksheet tree.
item1 = wbook.GetFirstTreeItem("folder1\\subfolder2");
item2 = wbook.GetNextTreeItem(item1);
item3 = wbook.GetPrevTreeItem(item2);
# item1 == item3 if there are at least two sibling items in
the "subfolder2" folder.
```

GetParentFolder(str path) -> str

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:

```
wbook = gscalc.ActiveWorkbook();
item = wbook.GetFirstTreeItem("folder1\\subfolder2");
```

```
parent = wbook.GetParentFolder(item);  
# parent == "folder1\subfolder2"
```

IsWorksheet(str path) -> int

Returns the logical **True** value if the specified worksheet tree item is a worksheet. Otherwise **False** is returned.

Example:

```
wbook = gscalc.ActiveWorkbook();  
check1 = wbook.IsWorksheet("folder1\sheet10");  
check2 = wbook.IsWorksheet("abc");
```

IsFolder(str path) -> int

Returns the logical **True** value if the specified worksheet tree item is a folder. Otherwise **False** is returned.

Example:

```
wbook = gscalc.ActiveWorkbook();  
check1 = wbook.IsFolder("folder1\sheet10");  
check2 = wbook.IsFolder("abc");
```

ExpandFolder(str folder) -> bool

Expand the specified folder in the worksheet tree window.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wbook.ExpandFolder("folder1");
```

CollapseFolder(str folder) -> bool

Collapse the specified folder in the worksheet tree window.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wbook.CollapseFolder("folder1");
```

SelectTreeItem(str path) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();  
wbook.SelectTreeItem("sheet1");  
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(str name) -> bool

Inserts a new folder into the current worksheet tree folder. The **name** argument must not include tree path elements.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wbook.SelectTreeItem("folder2");  
wbook.InsertFolder("nested-folder3");
```

InsertWorksheet(str name) -> bool

Inserts a new worksheet into the current worksheet tree folder. The **name** argument must not include tree path elements.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wbook.SelectTreeItem("");  
wbook.InsertFolder("sheet1");
```

DeleteTreeItem() -> bool

Deletes the currently selected worksheet tree item (which can be a single worksheet or a folder containing other folders and worksheets).

Example:

```
wbook = gscalc.ActiveWorkbook();
wbook.SelectTreeItem("folder4");
wbook.DeleteTreeItem();
wbook.SelectTreeItem("folder5\\sheet1");
wbook.DeleteTreeItem();
```

MoveTreeItem(str sourcePath, str targetPath) -> bool

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 inserted 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:

```
wbook = gscalc.ActiveWorkbook();
wbook.MoveTreeItem("folder1\\sheet11", "folder2");
wbook.MoveTreeItem("folder1\\sheet10", "");
```

CopyTreeItem(str sourcePath, str targetPath) -> bool

RenameTreeItem(str path) -> bool

Changes the name of the currently selected worksheet tree item (except the top/root item).

Example:

```
wbook = gscalc.ActiveWorkbook();
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:

```
wbook = gscalc.ActiveWorkbook();  
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** argument and can be "name" or "range".

Example:

```
wbook = gscalc.ActiveWorkbook();  
name = wbook.GetNamedRange(2, "name");  
cell = wbook.GetNamedRange(2, "range");
```

SetNamedRange(index, rangeName, range)

Set the existing named range specified by **index**.

Example:

```
wbook = gscalc.ActiveWorkbook();  
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:

```
wbook = gscalc.ActiveWorkbook();  
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:

```
wbook = gscalc.ActiveWorkbook();  
while wbook.GetNamedRangeCount() >= 1 :  
    wbook.RemoveNamedRange(1);
```

RemoveAllNamedRanges()

Removes all defined named ranges.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wbook.RemoveAllNamedRanges();
```

Save() -> bool

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:

```
wbook = gscalc.ActiveWorkbook();  
wbook.Save();
```

SaveAs(str filePath) -> bool

Saves the current workbook to a new file. If **path** is an empty string, **Save As** dialog box is displayed.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wbook.SaveAs("d:\\wbook1.gsc");
```

SaveAsPDFFile(str filePath, bool saveAllWorksheets) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();  
wbook.SaveAsPDFFile("d:\\copy_of_wbook1.pdf");
```

SaveAsExcelFile(str filePath) -> bool

Saves the current workbook as a new Excel XML file.

If **path** is an empty string, the **Save As** dialog box is displayed.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wbook.SaveAsExcelFile("d:\\wbook1.xml");
```

SaveAsTextFile(str filePath, bool dialogBox, TextParams params) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();  
textParams = gscalc.CreateTextParams();  
textParams.setSeparator("\t"); # tab-separated values  
textParams.setQuotingSymbol("\"");
```

```
textParams.setEncoding("utf8");  
wbook.SaveAsTextFile("d:\\wbook1.txt", False, textParams);  
wbook.SaveAsTextFile("d:\\wbook2.txt", True, textParams);
```

SaveAsXBaseFile(str filePath, bool dialogBox, XBaseParams params) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();  
xBaseParams = gscalc.CreateXBaseParams();  
xBaseParams.setFormat("dbaseIV");  
xBaseParams.AddField("item", "C", 40, 0);  
xBaseParams.AddField("price", "N", 5, 2);  
xBaseParams.setEncoding("windows");  
wbook.SaveAsXBaseFile("d:\\wbook1.dbf", False, xBaseParams);  
wbook.SaveAsXBaseFile("d:\\wbook2.dbf", True, xBaseParams);
```

Reload() -> bool

Reloads the workbook. All changes are discarded.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wbook.Reload();
```

Close() -> bool

Closes the workbook. If the **modified** property is **True**, users are prompted to save it.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wbook.Close();  
counter = wbook.GetWorksheetCount();
```


GetFileInfo(str filePath, int type) -> str

Returns information about the specified file without opening it. The **type** argument determines what is returned:

- 1 - file size, in bytes
- 2 - last modification date

Example:

```
# file size in bytes
size = gscalc.GetFileInfo("c:\\file01.gsc", 1);
# last modification date in the format YYYY-MM-DD (YYYY-MM-DDTHH:MM:SS)
modificationData = gscalc.GetFileInfo("c:\\file01.gsc", 2);
```

ReleaseFile() -> str

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:

```
wbook = gscalc.ActiveWorkbook();
path = wbook.ReleaseFile();
```

AttachFile(str filePath) -> int

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:

```
wbook = gscalc.ActiveWorkbook();
wbook.AttachFile("c:\\text_file01.csv");
```

MergeRows(MergeParams params) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();
merge = gscalc.CreateMergeParams();
# merge records from all *.gsc files with names starting
with "gsc_file"
merge.setPath("c:\\gsc_file*.gsc");
merge.setTable("");
wbook.MergeRows(merge);
```

MergeRowsFromODSFile(MergeParams params) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();
merge = gscalc.CreateMergeParams();
# merge records from all *.ods files with names starting
with "ods_file"
merge.setPath("c:\\ods_file*.ods");
merge.setTable("");
wbook.MergeRowsFromODSFile(merge);
```

MergeRowsFromTextFile(MergeParams params, TextParams params) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();
textParams = gscalc.CreateTextParams();
textParams.setSeparator(",");
mergeParams = gscalc.CreateMergeParams();
# merge records from all *.csv files with names starting
with "csv_file"
```

```
mergeParams.setPath("c:\\csv_file*.csv");
mergeParams.setTable("");
wbook.MergeRowsFromTextFile(mergeParams, textParams);
```

MergeRowsFromExcelFile(MergeParams params) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();
merge = gscalc.CreateMergeParams();
# merge records from all *.xlsx files with names starting
with "xlsx_file"
merge.setPath("c:\\xlsx_file*.xlsx");
merge.setTable("");
wbook.MergeRowsFromExcelFile(merge);
```

MergeRowsFromXBaseFile(MergeParams params, XBaseParams params) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();
merge = gscalc.CreateMergeParams();
# merge records from all *.dbf files with names starting
with "dbf_file"
merge.setPath("c:\\dbf_file*.dbf");
merge.setTable("");
xbaseParams = gscalc.CreateXBaseParams();
xbaseParams.setEncoding("windows");
wbook.MergeRowsFromXBaseFile(merge, xbaseParams);
```

MergeRowsFromMySQLFile(MergeParams params) -> bool

MergeRowsFromSQLiteFile(MergeParams params) -> bool

MergeTable(str filePath, str tablePath) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();
# merge tables from all *.gsc files with names starting with
"gsc_file"
wbook.MergeTable("c:\\gsc_file*.gsc", "customers");
```

MergeTableFromODSFile(str filePath, str table) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();
# merge tables from all *.ods files with names starting with
"ods_file"
wbook.MergeTableFromODSFile("c:\\ods_file*.ods",
"customers");
```

MergeTableFromTextFile(str filePath, str table, TextParams params) -> bool

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:

```
textParams = gscalc.CreateTextParams();
textParams.setSeparator(",");
wbook = gscalc.ActiveWorkbook();
# merge tables from all *.csv files with names starting with
"csv_file"
wbook.MergeTableFromTextFile("c:\\csv_file*.csv", "",
textParams);
```

MergeTableFromExcelFile(str filePath, str table) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();
# merge default tables from all *.xlsx files with names
starting with "xlsx_file"
wbook.MergeTableFromExcelFile("c:\\xlsx_file*.xlsx", "");
```

MergeTableFromXBaseFile(str filePath, str table, XBaseParams params) -> bool

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:

```

xBaseParams = gscalc.CreateXBaseParams();
xBaseParams.setEncoding("windows");
wbook = gscalc.ActiveWorkbook();
# merge tables from all *.dbf files with names starting with
"dbf_file"
wbook.MergeTableFromXBaseFile("c:\\dbf_file*.dbf", "",
xBaseParams);

```

MergeTableFromMySQLFile(str filePath, str table) -> bool

MergeTableFromSQLiteFile(str filePath, str table) -> bool

SetFilePassword(bool enable, str cryptMethod, str oldPassword, str newPassword) -> bool

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 must contain a new password consisting of at least 6 characters.

Example:

```

wbook = gscalc.ActiveWorkbook();
wbook.SetFilePassword(True, "blowfish", "Df8ER1", "97uYwp");
wbook = gscalc.NewWorkbook();
wbook.SetPassword(True, "blowfish", "", "97uYwp");

```

SetCellPassword(bool enable, str oldPassword, str newPassword) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();  
wbook.SetCellPassword(True, "Df8ER1", "97uYwp");
```

SetTreePassword(bool enable, str oldPassword, str newPassword) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();  
wbook.SetTreePassword(True, "Df8ER1", "97uYwp");
```

UpdateAllWorksheets() -> bool

Updates all formulas in all worksheets in this workbook. (Same as the **Update All - F9** command.)

Example:

```
wbook = gscalc.ActiveWorkbook();  
wbook.UpdateAllWorksheets();  
wbook.WaitForUpdate(); # to allow the background updating to  
complete before further actions
```

UpdateActiveWorksheet() -> bool

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:

```
wbook = gscalc.ActiveWorkbook();  
wbook.UpdateActiveWorksheet();  
wbook.WaitForUpdate(); # to allow the background updating to  
complete before further actions
```

UpdateActiveWorksheetRegion(str range) -> bool

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:

```
wbook = gscalc.ActiveWorkbook();  
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() -> None

If the background updating mode is 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 actions.

Example:


```
wbook = gscalc.ActiveWorkbook();  
wbook.UpdateAllWorksheets();  
wbook.WaitForUpdate();
```

MaximizeWindow() -> None

Maximizes the workbook window.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wbook.MaximizeWindow();
```

RestoreWindow() -> None

Restores the previous size of the workbook window after it's maximized.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wbook.RestoreWindow();
```

setModified(bool value) -> bool / getModified() -> bool

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:

```
wbook = gscalc.ActiveWorkbook();  
wbook.setModified(False);  
wbook.Close()
```

setUpdateMode(str mode) -> bool / getUpdateMode() -> str

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:

```
wbook = gscalc.ActiveWorkbook();
wbook.setUpdateMode("manual");
```

setUpdateThreads(str threads) -> bool / getUpdateThreads() -> str

Specifies the number of threads/CPU's used when updating workbook formulas:

- default - application defaults
- any number from 1 to 16

For more details, please see the **Settings** help topic.

Example:

```
wbook = gscalc.ActiveWorkbook();
wbook.setUpdateThreads(4);
# ...
wbook.setUpdateThreads("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:

```
wbook = gscalc.ActiveWorkbook();
xBaseParams = gscalc.CreateXBaseParams();
xBaseParams.setFormat("dBaseIV");
```

```
xBaseParams.AddField("item", "C", 40, 0);
xBaseParams.AddField("price", "N", 5, 2);
xBaseParams.setEncoding("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:

```
wbook = gscalc.ActiveWorkbook();
textParams = gscalc.CreateTextParams();
textParams.setSeparator("\t"); # tab-separated values
textParams.setQuotingSymbol("\"");
textParams.setEncoding("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 (which are then 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:

```
mergeParams = gscalc.CreateMergeParams();
mergeParams.setPath("e:\\data_files\\*.gsc");
mergeParams.setTable("customers");
mergeParams.setTable("pivot table reports\\data");
mergeParams.setFieldNames(1);
mergeParams.setMatchFieldNames(0);
mergeParams.setEnableUndo(0);
```

ActiveWorkbook(str path, str password) -> Workbook

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:

```
wbook = gscalc.ActiveWorkbook();
wbook2 = gscalc.NewWorkbook();
wbook3 = gscalc.OpenWorkbook("d:\\sales2013\\june.gsc", "");
```

NewWorkbook() -> Workbook

Creates a new workbook and returns the corresponding **Workbook** object.

Example:

```
wbook = gscalc.ActiveWorkbook();
wbook2 = gscalc.NewWorkbook();
wbook3 = gscalc.OpenWorkbook("d:\\sales2013\\june.gsc", "");
```

OpenWorkbook(str path) -> Workbook

Opens an existing workbook in the *.gsc or *.ods file format and returns the corresponding **Workbook** object.

If **path** is an empty string, the **Open File** dialog box is displayed.

Example:

```
wbook = gscalc.ActiveWorkbook();  
wbook2 = gscalc.NewWorkbook();  
wbook3 = gscalc.OpenWorkbook("d:\\sales2013\\june.gsc", "");  
wbook4 = gscalc.OpenWorkbook("d:\\sales2013\\june.ods",  
"FE20kP");
```

OpenTextFile(str path, bool dialogBox, TextParams params) -> Workbook

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:

```
textParams = gscalc.CreateTextParams();  
textParams.setSeparator("\t"); # tab-separated values  
textParams.setQuotingSymbol("\"");  
textParams.setEncoding("utf8");  
wbook1 = gscalc.OpenTextFile("d:\\wbook1.txt", False,  
textParams);  
wbook2 = gscalc.OpenTextFile("d:\\wbook2.txt", True,  
textParams);
```

OpenExcelFile(str path) -> Workbook

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:

```
wbook = gscalc.OpenExcelFile("d:\\sales2020\\june.xlsx");
```

OpenXBaseFile(str path, bool dialogBox, XBaseParams params) -> Workbook

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:

```
xBaseParams = gscalc.CreateXBaseParams();
xBaseParams.setFormat("dbaseIV");
wbook1 = gscalc.OpenXBaseFile("d:\\wbook1.dbf", False,
xBaseParams);
xBaseParams.setFormat("clipper");
wbook2 = gscalc.OpenXBaseFile("d:\\wbook2.dbf", True,
xBaseParams);
```

OpenMySQLFile(str path) -> Workbook

OpenSQLiteFile(str path) -> Workbook

LoadProfile(str path) -> bool

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(str path) -> bool

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() -> int

Returns the number of open workbooks.

Example:

```
counter = gscalc.GetWorkbookCount();
```

CloseAllWorkbooks() -> bool

Closes all open workbooks.

Example:

```
wbook = gscalc.ThisWorkbook();  
gscalc.CloseAllWorkbooks();  
counter = wbook.GetWorksheetCount();
```

TileWorkbooks() -> None

Tiles open workbook windows vertically or horizontally.

Example:

```
gscalc.TileWorkbooks(True);  
gscalc.TileWorkbooks(False);
```

MaximizeAppWindow() -> None

Maximizes the application window.

Example:

```
gscalc.MaximizeAppWindow();
```

MinimizeAppWindow() -> None

Minimizes the application window.

Example:

```
gscalc.MinimizeAppWindow();
```

RestoreAppWindow() -> None

Restores the previous size and position of the application window after it's minimized or maximized.

Example:

```
gscalc.RestoreAppWindow();
```

GetFilePath(bool saveAs, str title, str folder, str name, str ext) -> bool, str

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:

```
openFilePath = gscalc.GetFilePath(False, "Open XLSX File",  
"d:\\", "d:\\newfile.xlsx", ".xlsx");
```


GetFolder(str title, str folder) -> bool, str

Displays the (file) "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:

```
folderPath = gscalculator.GetFolder("Select Folder", "d:\\");
```

MessageBox(str message, str buttons, int activeButton, str icon) -> bool, str

Displays a message box using the following arguments:

- **message**
The message text to display.
- **buttons**
A text string specifying which buttons will appear in the message box. This must be one of the following strings:
 - ok
 - ok-cancel
 - retry-cancel
 - yes-no
 - yes-no-cancel
 - abort-retry-ignore
- **button**
An index 1-3 of the default message box button.
- **icon**
A text string specifying which icon appear in the message box. This must be one of the following strings:
 - exclamation
 - warning
 - information
 - question

- error

The method returns a text string representing the name of the clicked/pressed button:

- abort
- cancel
- continue
- ignore
- no
- ok
- retry
- tryagain
- yes

Example:

```

        accepted, ret = gscalc.MessageBox("Open this file:
calc1.gsc", "yes-no", 1, "question")
        if ret == "yes" :
            wbook =
gscalc.OpenWorkbook("d:\\sales2013\\calc1.gsc", "");
            # ...

```

InputBox(str title, bool password, str initialValue) -> bool, str

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 user clicks the **Cancel** button, the 1st returned value is False.

Example:

```

        accepted, 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);
```

SetStatusBarText

Specifies the text displayed in the 1st pane of status bar.

Example:

```
gscalc.SetStatusBarText("Executing script...");
```

Related Topics

[Samples](#)

Scripting: Samples

1. Enabling Scripting

To use Python scripting, you must have the Python package installed. Enabling Python scripting requires the same configuration steps as using Python UDF functions and formulas. Please refer to [Installing Python and integrating it with GS-Calc](#) for detailed instructions.

2. Creating Scripts

You can create scripts as **global scripts**, which are saved in the program settings and available to all workbooks, or as **local scripts** stored within a specific GS-Calc workbook, which are only available when that workbook is open.

To create these scripts, use the **File > Application Scripts** or **File > Database Scripts** commands.

The "Scripts" dialog box allows you to organize your scripts into subfolders, as well as copy, import, or export them.

[View a sample "Scripts" dialog screen.](#)

Notes:

- If a global script is executed while no workbook is open, it should begin with one of the application [Open\(...\)](#) or [New\(..\)](#) functions.
- If a workbook is already loaded, an executed script (whether global or local) automatically refers to that active workbook. Using the [Open\(...\)](#) functions with the same workbook path will have no effect.
- Backslashes used in script text, primarily within file path parameters, must always be doubled. For example: f:\\my_folder\\file01.gsc.

3. Sample Index

- [Merging rows from multiple text files from a folder to the current worksheet](#)
 - [Merging rows from multiple Excel XLSX files from a folder to the current worksheet](#)
 - [Importing tables from multiple text files from a folder to the current workbook](#)
 - [Importing tables from multiple Excel XLSX files from a folder to the current workbook](#)
 - [Releasing open workbook files \(gsc/xlsx/txt/csv...\) to allow concurrent updates by other programs](#)
 - [Attaching a released workbook file to check for changes and enable saving](#)
 - [Activating the n-th worksheet in the current workbook \(ignoring subfolders\)](#)
 - [Iterating through the entire tree of subfolders in the current workbook](#)
 - [Editing worksheets](#)
 - [Adding worksheets and folders](#)
 - [Opening a text file and saving it as a new *.gsc file](#)
-

Merging rows from multiple text files from a given folder to the current worksheet

```
import sys
from ctypes import *
CDLL("gsc2python39.dll")    # Python 3.9.13
import gscalc

wbook = gscalc.ActiveWorkbook();

merge = gscalc.CreateMergeParams();
```

```

merge.setPath("e:\\sample??.txt")
# or e.g. merge.setPath("e:\\my_folder\\some_text_file*.txt")
# or e.g. merge.setPath("e:\\my_folder\\some_text_file_01.txt")
# or e.g. merge.setPath("e:\\my_folder\\*.csv")
# or e.g. merge.setPath("e:\\my_folder\\")
# or e.g. merge.setPath(gscalc.GetFolder("Find folder", "e:\\"))
# or e.g. merge.setPath(gscalc.GetFilePath(false, "Merge from text
file", "e:\\", "d:\\my_folder", ".csv"))
merge.setTable("")
merge.setFieldNames(False);

text = gscalc.CreateTextParams();
text.setSeparator(",")
text.setEncoding("utf8");

wbook.MergeRowsFromTextFile(merge, text);

```

Merging rows from multiple Excel XLSX files from a given folder to the current worksheet

```

import sys
from ctypes import *
CDLL("gsc2python39.dll") # Python 3.9.13
import gscalc

wbook = gscalc.ActiveWorkbook();

var merge = gscalc.CreateMergeParams();
merge.setPath("e:\\my_folder\\some_text_file??.xlsx")

# no table name - merge rows from the default worksheet in each XLSX
file
merge.setTable("")
merge.setFieldNames(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

```

wbook = gscalc.ActiveWorkbook();
wsheet = wbook.ActiveWorksheet();

```

```

text = gscalc.CreateTextParams();
text.setSeparator("|")
text.setEncoding("utf8");

wbook.MergeTableFromTextFile("e:\\sample??.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

```

wbook = gscalc.ActiveWorkbook();

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

You can continue editing the workbook after releasing it because it is already loaded into memory. However, you must use the `AttachFile()` method to reconnect the file before saving; otherwise, the "Save File As" dialog will be displayed.

```
wbook = gscalc.ActiveWorkbook()  
wsheet = wbook.ActiveWorksheet()  
  
filePath = wbook.ReleaseFile();  
gscalc.MessageBox("Closed file: " + filePath, "ok", 1, "information");
```

Attaching a released workbook file (gsc/xlsx/txt/csv...) to check for changes and enable saving that file

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.ActiveWorksheet();  
  
if wbook.AttachFile("") == 1 :  
    wbook.Reload()  
    # 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.  
  
# After checking and reloading, you can release it again  
wbook.ReleaseFile();
```

Activating the n-th worksheet in the current workbook (ignoring subfolders)

```
wbook = gscalc.ActiveWorkbook();  
wsheet = wbook.ActiveWorksheet();  
  
# Get the first "child" element in the specified folder - in this case:  
# the main folder  
# If path.length = 0 then there are no more worksheets or subfolders  
path = wbook.GetFirstTreeItem("");  
  
counter = 0;
```

```

# Iterate through direct "child" elements of the specified folder
# (ignoring nested subfolders)
# to find the 3rd worksheet

while path != "" :
    if (wbook.IsFolder(path)) :
        gscalc.MessageBox("folder - " + path, "ok", 1, "information");
    else : counter = counter + 1
    if counter == 3:
        break;
    path = wbook.GetNextTreeItem(path);

if path != "" :
    # activate the 3rd worksheet
    wbook.SetActiveWorksheet(path);

```

Iterating through the entire tree of subfolders of the current workbook (max. 10 nesting levels)

```

wbook = gscalc.ActiveWorkbook();
wsheet = wbook.ActiveWorksheet();

folders = list()

treeItem = wbook.GetFirstTreeItem("");

while True :
    if wbook.IsFolder(treeItem) :
        gscalc.MessageBox("Folder: " + treeItem, "ok", 1,
"information");
        if not wbook.IsFolderEmpty(treeItem) :
            folders.append(treeItem)
            treeItem = wbook.GetFirstTreeItem(treeItem)
            continue
    elif wbook.IsWorksheet(treeItem) :
        gscalc.MessageBox("Worksheet: " + treeItem, "ok", 1,
"information");

    treeItem = wbook.GetNextTreeItem(treeItem);
    if treeItem == "" and len(folders) > 0 :
        treeItem = wbook.GetNextTreeItem(folders.pop());
    if treeItem == "":
        break;

```

Editing worksheets


```

wbook = gscalc.ActiveWorkbook();
wsheet = wbook.ActiveWorksheet();

wbook.SelectTreeItem("formulas");

# display amount in "b9" as thousands of dollars

formatParams = wsheet.CreateFormattingParams();
formatParams.SetCurrencyFormat("0", "$ 1.1", "$", False, False, 1);

wsheet.setSelectedRange("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.InsertFormula("b10", "=fv(0.65%, 36, -500, -5500, 0)");
wbook.WaitForUpdate();

# repeat the formatting
wsheet.setSelectedRange("b10")
wsheet.SetCellFormat(formatParams);

# Save as a new encrypted file. Specifying no path displays the "Save"
dialog box.
wbook.SetFilePassword(True, "blowfish", "", "dk3zPi");
wbook.SaveAs("");

```

Adding worksheets and folders

```

wbook = gscalc.OpenWorkbook("e:\\sample3.gsc", "dk3zPi");
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();
```

Opening a text file and saving it to new *.gsc file

```
textParams = gscalc.CreateTextParams();
textParams.setSeparator("|")
textParams.setQuotingSymbol("\"")
textParams.setEncoding("utf8");
wbook = gscalc.OpenTextFile("e:\\sample1.txt", False, textParams);
wbook.SaveAs("d:\\some_test_file.gsc");
```

Reading / inserting a sparse matrix

```
from ctypes import *
CDLL("gsc2python39.dll")
import gscalc2

wbook = gscalc2.ActiveWorkbook();
wsheet = wbook.ActiveWorksheet()

m1, m2, m3 = wsheet.GetSparseMatrix("c4:g1000", 0)

wsheet.InsertSparseMatrix("k4", m1, m2, m3)
```

Custom 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 even be used as a replacement for the default built-in functions. They can accept and return all the argument types and can also "return" images displayed directly in worksheets, or messages displayed after each update. Importantly, using these functions GS-Calc retains full multicore support along with 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 with the **Settings > Imported Function Libraries** command, as shown in the screenshot below. The internal function names must match the original C/C++ names. The


```

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 function is called. For example, if a worksheet cell executes 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 written 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 second 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 arrays, for best performance when reading data from or writing data to arrays, subsequent `read_array()` and `write_array()` calls must access the array elements in left-to-right, top-to-bottom order.

On output, index 15 (`DLL_ARGC_RET`) is used to specify the returned type (in `types[15]`), the possible error code (in `errors[15]`) and the 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: `strings[15]` already contains a valid buffer (up to 1024 characters + NULL) allocated by GS-Calc, 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 **messagelf**, **filter**, **image** and **fractal** functions:

[gsclib.h](#)
[dllmain.cpp](#)
[gsclib.cpp](#)

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(69), helpstring("method MergeRowsFromMySQLFile")]
HRESULT MergeRowsFromMySQLFile([in] IMergeParams* mergeParams);
        [id(70), helpstring("method MergeRowsFromSQLiteFile")]
HRESULT MergeRowsFromSQLiteFile([in] IMergeParams* mergeParams);

        [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);
        [id(67), helpstring("method MergeTableFromMySQLFile")]
HRESULT MergeTableFromMySQLFile([in] BSTR path, [in] BSTR table);
        [id(68), helpstring("method MergeTableFromSQLiteFile")]
HRESULT MergeTableFromSQLiteFile([in] BSTR path, [in] BSTR table);

};

```



```

        [
            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(31), helpstring("method OpenMySQLFile")] HRESULT
        OpenMySQLFile([in] BSTR path, [out, retval] IWorkbook** ppWorkbook);
        [id(32), helpstring("method OpenSQLiteFile")] HRESULT
        OpenSQLiteFile([in] BSTR path, [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);
};

```

Related Topics

[Samples](#)

COM Programming: Samples

Sample Index

[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 Microsoft 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 for brevity. These function wrappers use exceptions as a method of handling errors. If you do not want to use exceptions, please refer to those header files for instructions on using 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 other available objects/interfaces remain generally the same, except that you can access object properties directly without the `get_...` and `put_...` functions. In both cases, the methods and functions are identical and use the same parameters described in the [Methods, properties and sample usage](#) scripting help topic.

Before these interfaces are accessible to other programs in the Windows system, they must be registered by GS-Calc using the **Register GS-Calc COM Interfaces** command (found in the GS-Calc "Settings" menu).

Creating and saving a new workbook

```
#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, (double)i);
}

//alternatively, the InsertData() function can be used, however
//it performs additional checks, 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:c10000)"));

//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 until 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 <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 C5 (Row 5, Col 3)
wsheet->InsertFormulaRC(5, 3, _bstr_t("=sum(c1:c4)"));

//if the update mode it set to "manual", update formulas explicitly;
//wbook->UpdateActiveWorksheet();

//if updating is performed in the background, wait until it's complete
wbook->WaitForUpdate();

//from the C5 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 C1 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 explicitly;
//wbook->UpdateActiveWorksheet();

//if updating is performed in the background, wait until 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 C1

```

```

wsheet->InsertData(wsheet->AddressRC(1, 3), _bstr_t("#N/A!"), true);

//if the update mode it set to "manual", update formulas explicitly;
//wbook->UpdateActiveWorksheet();

//if updating is performed in the background, wait until 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 <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 until it's complete
    wbook->WaitForUpdate();

    //save changes
    wbook->Save();
    wbook->Close();
}
catch(_com_error error)
{
    // ...
}

```

Formatting


```

#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;
//border 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"));
//border 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 <comdef.h>
#include <assert.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->GetNextTreeItem(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 <comdef.h>

```

```

#import "E:\gscalculator\gscalculator.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()/InsertWorksheet() functions require names without
    paths as parameters
    //and folders/sheets are inserted in the active folder
    wbook->SetActiveFolder((BSTR)NULL);
    wbook->InsertFolder(_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()/InsertWorksheet() 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() function applies to 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"));

//pause execution to allow for manual result verification
::Sleep(10000);

//close without saving
wbook->Close();
}
catch(_com_error error)
{
    // ...
}

```

Password protection and encryption

```

#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 <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 <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 <comdef.h>

#import "E:\gscalculator\gscalculator.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)
{
    // ...
}

```

Related Topics

[Interfaces](#)

Support

If you have any questions, comments or suggestions, please visit the [support forum](#) or email Citadel5 directly at info@citadel5.com.

Support

Copyright © 2026 Citadel5, J.Piechura

Some icons by [Yusuke Kamiyamane](#).

```

#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 : 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 : 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
    expanded 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 resolution 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 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]/*...*/)
        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;
}
}

```