Manual for Package PgfplotsTable

Component of pgfplots, Version 1.7

http://sourceforge.net/projects/pgfplots

Dr. Christian Feuersänger
cfeuersaenger@users.sourceforge.net

October 26, 2012

Abstract

This package reads tab-separated numerical tables from input and generates code for pretty-printed LaTeX-tabulars. It rounds to the desired precision and prints it in different number formatting styles.

Contents

1 Introduction

PgfplotsTable is a lightweight sub-package of pgfplots which employs its table input methods and the number formatting techniques to convert tab-separated tables into tabulars.

Its input is a text file containing space-separated rows, possibly starting with column names. Its output is a LaTeX tabular[1] which contains selected columns of the text table, rounded to the desired precision, printed in the desired number format (fixed point, integer, scientific etc.). The output is LaTeX code, and that code is finally typeset by LaTeX.

In other words, PgfplotsTable is nothing but a more-or-less smart code generator which spits out something like `\begin{tabular}....\end{tabular}`. Use it if you'd like to customize row- or column–dependent styles or if you have numerical data for which you want to have automatically formatted content.

It is used with

```
\usepackage{pgfplotstable}
%  recommended:
% \usepackage{booktabs}
% \usepackage{array}
% \usepackage{colortbl}
```

and requires pgfplots and pgf $\geq$ 2.00 installed.

`\pgfplotstableset{`⟨*key-value-options*⟩`}`

The user interface of this package is based on key-value-options. They determine what to display, how to format and what to compute.

Key-value pairs can be set in two ways:

1. As default settings for the complete document (or maybe a part of the document), using `\pgfplotstableset{`⟨*options*⟩`}`. For example, the document's preamble may contain

   ```
   \pgfplotstableset{fixed zerofill,precision=3}
   ```

   to configure a precision of 3 digits after the period, including zeros to get exactly 3 digits for all fixed point numbers.

2. As option which affects just a single table. This is provided as optional argument to the respective table typesetting command, for example `\pgfplotstabletypeset[`⟨*options*⟩`]{`⟨*file*⟩`}`.

Both ways are shown in the examples below.

Knowledge of `pgfkeys` is useful for a deeper insight into this package, as `/.style`, `/.append style` etc. are specific to `pgfkeys`. Please refer to the pgf manual, [2, Section pgfkeys] or the shorter introduction [3] to learn more about `pgfkeys`. Otherwise, simply skip over to the examples provided in this document.

You will find key prefixes `/pgfplots/table/` and `/pgf/number format/`. These prefixes can be skipped if they are used in PgfplotsTable; they belong to the "default key path" of `pgfkeys`.

2 Loading and Displaying data

2.1 Text Table Input Format

PgfplotsTable works with plain text file tables in which entries ("cells") are separated by a separation character. The initial separation character is "white space" which means "at least one space or tab" (see option `col sep` below). Those tables can have a header line which contains column names and most other columns typically contain numerical data.
The following listing shows `pgfplotstable.example1.dat` and is used often throughout this documentation.

[1] Please see the remarks in Section 5.4 for plain TeX and ConTeXt.

```
# Convergence results
# fictional source, generated 2008
level    dof       error1             error2   info     grad(log(dof),log(error2)) quot(error1)
1        4         2.50000000e-01     7.57858283e-01   48       0                    0
2        16        6.25000000e-02     5.00000000e-01   25       -3.00000000e-01   4
3        64        1.56250000e-02     2.87174589e-01   41       -3.99999999e-01   4
4        256       3.90625000e-03     1.43587294e-01   8        -5.00000003e-01   4
5        1024      9.76562500e-04     4.41941738e-02   22       -8.49999999e-01   4
6        4096      2.44140625e-04     1.69802322e-02   46       -6.90000001e-01   4
7        16384     6.10351562e-05     8.20091159e-03   40       -5.24999999e-01   4
8        65536     1.52587891e-05     3.90625000e-03   48       -5.35000000e-01   3.99999999e+00
9        262144    3.81469727e-06     1.95312500e-03   33       -5.00000000e-01   4.00000001e+00
10       1048576   9.53674316e-07     9.76562500e-04   2        -5.00000000e-01   4.00000001e+00
```

Lines starting with '`%`' or '`#`' are considered to be comment lines and are ignored.

There is future support for a second header line which must start with '`$flags `' (the space is mandatory, even if the column separator is *not* space!). Currently, such a line is ignored. It may be used to provide number formatting options like precision and number format.

`\pgfplotstabletypeset`[⟨*optional arguments*⟩]{⟨*file name* or *\macro* or *inline table*⟩}

Loads (or acquires) a table and typesets it using the current configuration of number formats and table options.

In case the first argument is a file name, the table will be loaded from disk. If it is an already loaded table (see `\pgfplotstableread` or `\pgfplotstablenew`), it will be used. Otherwise, if it is inline table data, this data will be parsed just as if it was found in a file (see `\pgfplotstableread`).

a	b
5,000	$1.23 \cdot 10^{5}$
6,000	$1.63 \cdot 10^{5}$
7,000	$2.10 \cdot 10^{5}$
9,000	$1.00 \cdot 10^{6}$

```
\pgfplotstabletypeset[sci zerofill]{
    a b
    5000 1.234e5
    6000 1.631e5
    7000 2.1013e5
    9000 1000000
}
```

level	dof	error1	error2	info	grad(log(dof),log(error2))	quot(error1)
1	4	0.25	0.76	48	0	0
2	16	$6.25 \cdot 10^{-2}$	0.5	25	−0.3	4
3	64	$1.56 \cdot 10^{-2}$	0.29	41	−0.4	4
4	256	$3.91 \cdot 10^{-3}$	0.14	8	−0.5	4
5	1,024	$9.77 \cdot 10^{-4}$	$4.42 \cdot 10^{-2}$	22	−0.85	4
6	4,096	$2.44 \cdot 10^{-4}$	$1.7 \cdot 10^{-2}$	46	−0.69	4
7	16,384	$6.1 \cdot 10^{-5}$	$8.2 \cdot 10^{-3}$	40	−0.52	4
8	65,536	$1.53 \cdot 10^{-5}$	$3.91 \cdot 10^{-3}$	48	−0.54	4
9	$2.62 \cdot 10^{5}$	$3.81 \cdot 10^{-6}$	$1.95 \cdot 10^{-3}$	33	−0.5	4
10	$1.05 \cdot 10^{6}$	$9.54 \cdot 10^{-7}$	$9.77 \cdot 10^{-4}$	2	−0.5	4

```
\pgfplotstabletypeset{pgfplotstable.example1.dat}
```

The configuration can be customized with ⟨*optional arguments*⟩. Configuration can be done for the complete table or for particular columns (or rows).

level	Dof	e_1	e_2	info	∇e_2	$\frac{e_1^{(n)}}{e_1^{(n-1)}}$
1	4	2.5^{-1}	$7.58{\cdot}10^{-1}$	+48.0	–	–
2	16	6.3^{-2}	$5.00{\cdot}10^{-1}$	+25.0	−0.3	4
3	64	1.6^{-2}	$2.87{\cdot}10^{-1}$	+41.0	−0.4	4
4	256	3.9^{-3}	$1.44{\cdot}10^{-1}$	+8.0	−0.5	4
5	1 024	9.8^{-4}	$4.42{\cdot}10^{-2}$	+22.0	−0.85	4
6	4 096	2.4^{-4}	$1.70{\cdot}10^{-2}$	+46.0	−0.69	4
7	16 384	6.1^{-5}	$8.20{\cdot}10^{-3}$	+40.0	−0.52	4
8	65 536	1.5^{-5}	$3.91{\cdot}10^{-3}$	+48.0	−0.54	4
9	262 144	3.8^{-6}	$1.95{\cdot}10^{-3}$	+33.0	−0.5	4
10	1 048 576	9.5^{-7}	$9.77{\cdot}10^{-4}$	+2.0	−0.5	4

```
\pgfplotstableset{%  global config, for example in the preamble
    %  these columns/<colname>/.style={<options>} things define a style
    %  which applies to <colname> only.
    columns/dof/.style={int detect,column type=r,column name=\textsc{Dof}},
    columns/error1/.style={
        sci,sci zerofill,sci sep align,precision=1,sci superscript,
        column name=$e_1$,
    },
    columns/error2/.style={
        sci,sci zerofill,sci sep align,precision=2,sci 10e,
        column name=$e_2$,
    },
    columns/{grad(log(dof),log(error2))}/.style={
        string replace={0}{}, %  erase '0'
        column name={$\nabla e_2$},
        dec sep align,
    },
    columns/{quot(error1)}/.style={
        string replace={0}{}, %  erase '0'
        column name={$\frac{e_1^{(n)}}{e_1^{(n-1)}}$}
    },
    empty cells with={--}, %  replace empty cells with '--'
    every head row/.style={before row=\toprule,after row=\midrule},
    every last row/.style={after row=\bottomrule}
}
\pgfplotstabletypeset[ %  local config, applies only for this table
    1000 sep={\,},
    columns/info/.style={
        fixed,fixed zerofill,precision=1,showpos,
        column type=r,
    }
]
{pgfplotstable.example1.dat}
```

All of these options are explained in all detail in the following sections.

You may also use an input format similar to the tabular environment:

level	dof	error
1	4	0.25
2	16	$6.25 \cdot 10^{-2}$
3	64	$1.56 \cdot 10^{-2}$
4	256	$3.91 \cdot 10^{-3}$
5	1,024	$9.77 \cdot 10^{-4}$
6	4,096	$2.44 \cdot 10^{-4}$
7	16,384	$6.10 \cdot 10^{-5}$
8	65,536	$1.53 \cdot 10^{-5}$
9	$2.62 \cdot 10^{5}$	$3.81 \cdot 10^{-6}$
10	$1.05 \cdot 10^{6}$	$9.54 \cdot 10^{-7}$

```
\pgfplotstabletypeset
    [col sep=&,row sep=\\,sci zerofill]
{
    level &  dof &    error \\
    1 &      4 &      2.50000000e-01 \\
    2 &      16 &     6.25000000e-02 \\
    3 &      64 &     1.56250000e-02 \\
    4 &      256 &    3.90625000e-03 \\
    5 &      1024 &   9.76562500e-04 \\
    6 &      4096 &   2.44140625e-04 \\
    7 &      16384 &  6.10351562e-05 \\
    8 &      65536 &  1.52587891e-05 \\
    9 &      262144 & 3.81469727e-06 \\
    10 &     1048576 &9.53674316e-07 \\
}
```

Technical note: every opened file will be protocolled into your log file.

`\pgfplotstabletypesetfile`[⟨*optional arguments*⟩]{⟨*file name*⟩}

Loads the table ⟨*file name*⟩ and typesets it. As of PgfplotsTable 1.2, this command is an alias to `\pgfplotstabletypeset`, that means the first argument can be either a file name or an already loaded table.

`\pgfplotstableread`{⟨*file name*⟩}{⟨*\macro*⟩}
`\pgfplotstableread`{⟨*inline table*⟩}{⟨*\macro*⟩}

Loads a table into the TEX-macro ⟨*\macro*⟩. This macro will store the table as internal structure and can be used multiple times.

dof	error1	dof	error2
4	0.25	4	0.76
16	$6.25 \cdot 10^{-2}$	16	0.5
64	$1.56 \cdot 10^{-2}$	64	0.29
256	$3.91 \cdot 10^{-3}$	256	0.14
1,024	$9.77 \cdot 10^{-4}$	1,024	$4.42 \cdot 10^{-2}$
4,096	$2.44 \cdot 10^{-4}$	4,096	$1.7 \cdot 10^{-2}$
16,384	$6.1 \cdot 10^{-5}$	16,384	$8.2 \cdot 10^{-3}$
65,536	$1.53 \cdot 10^{-5}$	65,536	$3.91 \cdot 10^{-3}$
$2.62 \cdot 10^{5}$	$3.81 \cdot 10^{-6}$	$2.62 \cdot 10^{5}$	$1.95 \cdot 10^{-3}$
$1.05 \cdot 10^{6}$	$9.54 \cdot 10^{-7}$	$1.05 \cdot 10^{6}$	$9.77 \cdot 10^{-4}$

```
\pgfplotstableread{pgfplotstable.example1.dat}\loadedtable
\pgfplotstabletypeset[columns={dof,error1}]\loadedtable
\hspace{2cm}
\pgfplotstabletypeset[columns={dof,error2}]\loadedtable
```

The first argument can be a ⟨*file name*⟩ as in the example here. It is also possible to provide the table data directly:

```
%  Alternative: inline table data:
\pgfplotstableread{
level   dof     error1  error2  info    grad(log(dof),log(error2))      quot(error1)
1       4       2.50000000e-01  7.57858283e-01  48      0                   0
2       16      6.25000000e-02  5.00000000e-01  25      -3.00000000e-01 4
3       64      1.56250000e-02  2.87174589e-01  41      -3.99999999e-01 4
4       256     3.90625000e-03  1.43587294e-01  8       -5.00000003e-01 4
5       1024    9.76562500e-04  4.41941738e-02  22      -8.49999999e-01 4
6       4096    2.44140625e-04  1.69802322e-02  46      -6.90000001e-01 4
7       16384   6.10351562e-05  8.20091159e-03  40      -5.24999999e-01 4
8       65536   1.52587891e-05  3.90625000e-03  48      -5.35000000e-01 3.99999999e+00
9       262144  3.81469727e-06  1.95312500e-03  33      -5.00000000e-01 4.00000001e+00
10      1048576 9.53674316e-07  9.76562500e-04  2       -5.00000000e-01 4.00000001e+00
}\loadedtable
%  can be used as above:
\pgfplotstabletypeset[columns={dof,error1}]\loadedtable
\hspace{2cm}
\pgfplotstabletypeset[columns={dof,error2}]\loadedtable
```

It is checked automatically whether the first argument contains inline data or a file name.

This check whether the first argument is inline data or a file name works as follows: if `format=auto`, the first argument is considered to be a file name unless it contains the `row sep` character (see `row sep`). If `format=inline`, it is always considered to be inline data. If `format=file`, it is a file name.

Special cases and more details:

- The inline data format is "fragile". If you experience problems, terminate your tables with '\\' combined with `row sep=\\` (the docs for `row sep` contain alternative ways and more explanation).
- There are variants of this command which do not really build up a struct, but which report every line to a "listener". There is also a struct which avoids protection by TEX scopes. In case you need such things, consider reading the source code comments.
- Technical note: every opened file will be protocolled into your log file.
- Note: avoid using '`\table`' as name, it conflicts with `\begin{table}` of LATEX.

`/pgfplots/table/col sep=space|tab|comma|semicolon|colon|braces|&|ampersand` (initially `space`)

Specifies the column separation character for table reading. The initial choice, `space`, means "at least one white space". White spaces are tab stops or spaces (newlines characters always delimit lines).

For example, the file `pgfplotstable.example1.csv` uses commas as separation characters.

```
# Convergence results
# fictional source  generated 2008
level,dof,error1,error2,info,{grad(log(dof),log(error2))},quot(error1)
1,9,2.50000000e-01,7.57858283e-01,48,0,0
2,25,6.25000000e-02,5.00000000e-01,25,-1.35691545e+00,4
3,81,1.56250000e-02,2.87174589e-01,41,-1.17924958e+00,4
4,289,3.90625000e-03,1.43587294e-01,8,-1.08987331e+00,4
5,1089,9.76562500e-04,4.41941738e-02,22,-1.04500712e+00,4
6,4225,2.44140625e-04,1.69802322e-02,46,-1.02252239e+00,4
7,16641,6.10351562e-05,8.20091159e-03,40,-1.01126607e+00,4
8,66049,1.52587891e-05,3.90625000e-03,48,-1.00563427e+00,3.99999999e+00
9,263169,3.81469727e-06,1.95312500e-03,33,-1.00281745e+00,4.00000001e+00
10,1050625,9.53674316e-07,9.76562500e-04,2,-1.00140880e+00,4.00000001e+00
```

Thus, we need to specify `col sep=comma` when we read it.

level	dof	error1	error2	info	grad(log(dof),log(error2))	quot(error1)
1	4	0.25	0.76	48	0	0
2	16	$6.25 \cdot 10^{-2}$	0.5	25	−0.3	4
3	64	$1.56 \cdot 10^{-2}$	0.29	41	−0.4	4
4	256	$3.91 \cdot 10^{-3}$	0.14	8	−0.5	4
5	1,024	$9.77 \cdot 10^{-4}$	$4.42 \cdot 10^{-2}$	22	−0.85	4
6	4,096	$2.44 \cdot 10^{-4}$	$1.7 \cdot 10^{-2}$	46	−0.69	4
7	16,384	$6.1 \cdot 10^{-5}$	$8.2 \cdot 10^{-3}$	40	−0.52	4
8	65,536	$1.53 \cdot 10^{-5}$	$3.91 \cdot 10^{-3}$	48	−0.54	4
9	$2.62 \cdot 10^{5}$	$3.81 \cdot 10^{-6}$	$1.95 \cdot 10^{-3}$	33	−0.5	4
10	$1.05 \cdot 10^{6}$	$9.54 \cdot 10^{-7}$	$9.77 \cdot 10^{-4}$	2	−0.5	4

```
\pgfplotstabletypeset[col sep=comma]{pgfplotstable.example1.csv}
```

You may call `\pgfplotstableset{col sep=comma}` once in your preamble if all your tables use commas as column separator.

Please note that if cell entries (for example column names) contain the separation character, you need to enclose the column entry in *braces*: `{grad(log(dof),log(error2))}`. If you want to use unmatched braces, you need to write a backslash before the brace. For example the name '`column{withbrace`' needs to be written as '`column\{withbrace`'.

For `col sep`≠`space`, spaces will be considered to be part of the argument (there is no trimming). However, (as usual in TEX), multiple successive spaces and tabs are replace by a single white space. Of course, if `col sep=tab`, tabs are the column separators and will be treated specially.

Furthermore, if you need empty cells in case `col sep=space`, you have to provide `{}` to delimit such a cell since `col sep=space` uses *at least* one white space (consuming all following ones).

The value `col sep=braces` is special since it actually uses two separation characters. Every single cell entry is delimited by an opening and a closing brace, ⟨*entry*⟩, for this choice. Furthermore, any white space (spaces and tabs) between cell entries are *skipped* in case `braces` until the next ⟨*entry*⟩ is found.

A further specialty of `col sep=braces` is that it has support for *multi-line* cells: everything within balanced braces is considered to be part of a cell. This includes newlines[2].

The `col sep=&` case (probably together with `row sep=\\`) allows to read tables as you'd usually type them in LATEX. This will automatically enable `trim cells`.

`/pgfplots/table/trim cells=true|false` (initially `false`)

If enabled, leading and trailing white space will be removed while tables are read.

This might be necessary if you have `col sep`≠`space` but your cells contain spaces. It will be activated automatically for `col sep=&`.

`/pgfplots/table/header=true|false|has colnames` (initially `true`)

Configures if column names shall be identified automatically during input operations.

[2]This treatment of newlines within balanced braces actually applies to every other column separator as well (it is a TEX readline feature). In other words: you *can* have multi-line cells for every column separator if you enclose them in balanced curly braces. However, `col sep=braces` has the special treatment that end-of-line counts as white space character; for every other `col sep` value, this white space is suppressed to remove spurious spaces.

The first non-comment line *can* be a header which contains column names. The `header` key configures how to detect if that is really the case.

The choice `true` enables auto–detection of column names: If the first non-comment line contains at least one non-numerical entry (for example '`a name`'), each entry in this line is supposed to be a column name. If the first non-comment line contains only numerical data, it is used as data row. In this case, column indices will be assigned as column "names".

The choice `false` is identical to this last case, i.e. even if the first line contains strings, they won't be recognised as column names.

Finally, the choice `has colnames` is the opposite of `false`: it assumes that the first non–comment line *contains* column names. In other words: even if only numbers are contained in the first line, they are considered to be column *names*.

Note that this key only configures headers in *input* tables. To configure *output* headers, you may want to look at `every head row`.

/pgfplots/table/format=auto|inline|file (initially auto)

Configures the format expected as first argument for `\pgfplotstableread{`⟨*input*⟩`}`.

The choice `inline` expects the table data directly as argument where rows are separated by `row sep`. Inline data is "fragile", because TEX may consume end–of–line characters (or `col sep` characters). See `row sep` for details.

The choice `file` expects a file name.

The choice `auto` searches for a `row sep` in the first argument supplied to `\pgfplotstableread`. If a `row sep` has been found, it is inline data, otherwise it is a file name.

/pgfplots/table/row sep=newline|\\ (initially newline)

Configures the character to separate rows of the inline table data format (see `format=inline`).

The choice `newline` uses the end of line as it appears in the table data (i.e. the input file or any inline table data).

The choice `\\` uses '\\' to indicate the end of a row.

Note that `newline` for inline table data is "fragile": you can't provide such data inside of TEX macros (this does not apply to input files). Whenever you experience problems, proceed as follows:

1. First possibility: call `\pgfplotstableread{`⟨*data*⟩`}\yourmacro` *outside* of any macro declaration.
2. Use `row sep=\\`.

The same applies if you experience problems with inline data and special `col sep` choices (like `col sep=tab`).

The reasons for such problems is that TEX scans the macro bodies and replaces newlines by white space. It does other substitutions of this sort as well, and these substitutions can't be undone (maybe not even found).

/pgfplots/table/ignore chars={⟨*comma-separated-list*⟩} (initially empty)

Allows to define an "ignore list" for single characters. Any characters found in an input file which occur also in ⟨*comma-separated-list*⟩ will silently by thrown away. The processing is exactly the same as if you did not write them at all in the first place.

For example, suppose we are given `pgfplotstable.example5.dat` with

```
first,second
(1)(0),2 1#2)
(3)(0),4 1#3)
(5)(0),6 1#3)
```

then, we can ignore several of the characters by writing

first	second
10	212
30	413
50	613

```
\pgfplotstabletypeset
    [col sep=comma,ignore chars={(,),\ ,\#}]
    {pgfplotstable.example5.dat}
```

The ⟨*comma-separated-list*⟩ should contain exactly one character in each list element, and the single characters should be separated by commas. Some special characters like commas, white space, hashes, percents or backslashes need to be escaped by prefixing them with a backslash.

Besides normal characters, it is also supported to eliminate any binary code from your input files. For example, suppose you have binary characters of code `0x01` (hex notation) in your files. Then, use

```
\pgfplotstableset{ignore chars={\^^01}}
```

to eliminate them silently. The `^^`⟨*digit*⟩⟨*digit*⟩ notation is a TEX feature to provide characters in hexadecimal encoding where ⟨*digit*⟩ is one of `0123456789abcdef`. I don't know if the backslash in `\^^01` is always necessary, try it out. There is also a character based syntax, in which `\^^M` is ⟨*newline*⟩ and `\^^I` is ⟨*tab*⟩. Refer to [1] for more details.

Note that after stripping all these characters, the input table must be valid – it should still contain column separators and balanced columns.

This setting applies to `\addplot table` and `\addplot file` for PGFPLOTS as well.

Note that `ignore chars` is "fragile" when it is applied to `format=inline` or `format=auto`. Consider `format=file` if you experience problems[3].

/pgfplots/table/**white space chars**={⟨*comma-separated-list*⟩} (initially `empty`)

Allows to define a list of single characters which are actually treated like white space (in addition to tabs and spaces). It might be useful in order to get more than one column separator character.

The `white space chars` list is used in exactly the same way as `ignore chars`, and the same remarks as above apply as well.

/pgfplots/table/**comment chars**={⟨*comma-separated-list*⟩} (initially `empty`)

Allows to add one or more *additional* comment characters. Each of these characters has a similar effect as the `#` character, i.e. all following characters of that particular input line are skipped.

0	1
1	0
2	−10
3	0

```
\pgfplotstabletypeset[comment chars=!]{
! Some comments
1 0
2 -10
! another comment line
3 0
}
```

The example above uses '`!`' as additional comment character (which allows to parse Touchstone files).

/pgfplots/table/**skip first n**={⟨*integer*⟩} (initially `0`)

Allows to skip the first ⟨*integer*⟩ lines of an input file. The lines will not be processed.

A	B	C
1	2	3
4	5	6

```
\pgfplotstabletypeset[skip first n=4]{% <- this '%' is important. Otherwise, the
                                      % newline here would delimit an (empty) row.
    XYZ Format,
    Version 1.234
    Date 2010-09-01
    @author Mustermann
    A B C
    1 2 3
    4 5 6
}
```

2.2 Selecting Columns and their Appearance Styles

/pgfplots/table/**columns**={⟨*comma-separated-list*⟩}

Selects particular columns of the table. If this option is empty (has not been provided), all available columns will be selected.

Inside of ⟨*comma-separated-list*⟩, column names as they appear in the table's header are expected. If there is no header, simply use column indices. If there are column names, the special syntax `[index]`⟨*integer*⟩ can be used to select columns by index. The first column has index 0.

[3] See also `row sep` for more information about dealing with fragile inline table formats.

dof	level	info
4	1	48
16	2	25
64	3	41
256	4	8
1,024	5	22
4,096	6	46
16,384	7	40
65,536	8	48
$2.62 \cdot 10^5$	9	33
$1.05 \cdot 10^6$	10	2

```
\pgfplotstabletypeset[columns={dof,level,[index]4}]{pgfplotstable.example1.dat}
```

The special pgfkeys feature `\pgfplotstableset{columns/.add={}{,a further col}}` allows to *append* a value, in this case '`,a further col`' to the actual value. See `/.add` for details.

`/pgfplots/table/alias/⟨col name⟩/.initial={⟨real col name⟩}`

Assigns the new name ⟨*col name*⟩ for the column denoted by ⟨*real col name*⟩. Afterwards, accessing ⟨*col name*⟩ will use the data associated with column ⟨*real col name*⟩.

a	newname
1	2
3	4
5	6

```
%  in preamble:
\pgfplotstableset{
    alias/newname/.initial=b,
}

%  in document:
\pgfplotstabletypeset[
    columns={a,newname},%  access to 'newname' is the same as to 'b'
]{
    a b
    1 2
    3 4
    5 6
}%
```

You can use `columns/`⟨*col name*⟩`/.style` to assign styles for the alias, not for the original column name.

If there exists both an alias and a column of the same name, the column name will be preferred. Furthermore, if there exists a `create on use` statement with the same name, this one will also be preferred.

In case ⟨*col name*⟩ contains characters which are required for key settings, you need to use braces around it: "`alias/{name=wi/th,special}/.initial={othername}`".

This key is used whenever columns are queried, it applies also to the `\addplot table` statement of PGFPLOTS.

`/pgfplots/table/columns/⟨column name⟩/.style={⟨key-value-list⟩}`

Sets all options in ⟨*key-value-list*⟩ exclusively for ⟨*column name*⟩.

level	Dof	L_2	A	info	grad(log(dof),log(error2))	quot(error1)
1	4	2.500_{-1}	7.58_{-1}	48	0	0
2	16	6.250_{-2}	5.00_{-1}	25	−0.3	4
3	64	1.563_{-2}	2.87_{-1}	41	−0.4	4
4	256	3.906_{-3}	1.44_{-1}	8	−0.5	4
5	1,024	9.766_{-4}	4.42_{-2}	22	−0.85	4
6	4,096	2.441_{-4}	1.70_{-2}	46	−0.69	4
7	16,384	6.104_{-5}	8.20_{-3}	40	−0.52	4
8	65,536	1.526_{-5}	3.91_{-3}	48	−0.54	4
9	262,144	3.815_{-6}	1.95_{-3}	33	−0.5	4
10	1,048,576	9.537_{-7}	9.77_{-4}	2	−0.5	4

```
\pgfplotstabletypeset[
    columns/error1/.style={
        column name=$L_2$,
        sci,sci zerofill,sci subscript,
        precision=3},
    columns/error2/.style={
        column name=$A$,
        sci,sci zerofill,sci subscript,
        precision=2},
    columns/dof/.style={
        int detect,
        column name=\textsc{Dof}
    }
]
    {pgfplotstable.example1.dat}
```

If your column name contains commas ‘,’, slashes ‘/’ or equal signs ‘=’, you need to enclose the column name in braces.

Dof	L_2	slopes L_2
4	2.500_{-1}	0.0
16	6.250_{-2}	−0.3
64	1.563_{-2}	−0.4
256	3.906_{-3}	−0.5
1,024	9.766_{-4}	−0.8
4,096	2.441_{-4}	−0.7
16,384	6.104_{-5}	−0.5
65,536	1.526_{-5}	−0.5
262,144	3.815_{-6}	−0.5
1,048,576	9.537_{-7}	−0.5

```
\pgfplotstabletypeset[
    columns={dof,error1,{grad(log(dof),log(error2))}},
    columns/error1/.style={
        column name=$L_2$,
        sci,sci zerofill,sci subscript,
        precision=3},
    columns/dof/.style={
        int detect,
        column name=\textsc{Dof}},
    columns/{grad(log(dof),log(error2))}/.style={
        column name=slopes $L_2$,
        fixed,fixed zerofill,
        precision=1}
]
    {pgfplotstable.example1.dat}
```

If your tables don’t have column names, you can simply use integer indices instead of ⟨*column name*⟩ to refer to columns. If you have column names, you can’t set column styles using indices.

/pgfplots/table/display columns/⟨*index*⟩/.style={⟨*key-value-list*⟩}

Applies all options in ⟨*key-value-list*⟩ exclusively to the column which will appear at position ⟨*index*⟩ in the output table.

In contrast to the `table/columns/`⟨*name*⟩ styles, this option refers to the output table instead of the input table. Since the output table has no unique column name, you can only access columns by index.

Indexing starts with ⟨*index*⟩ = 0.

Display column styles override input column styles.

/pgfplots/table/every col no ⟨*index*⟩ (style, no value)

A style which is identical with `display columns/`⟨*index*⟩: it applies exclusively to the column at position ⟨*index*⟩ in the output table.

See `display columns/`⟨*index*⟩ for details.

/pgfplots/table/column type={⟨*tabular column type*⟩} (initially `c`)

Contains the column type for `tabular`.

If all column types are empty, the complete argument is skipped (assuming that no `tabular` environment is generated).

Use `\pgfplotstableset{column type/.add={`⟨*before*⟩`}{`⟨*after*⟩`}}` to *modify* a value instead of overwriting it. The `/.add` key handler works for other options as well.

dof	error1	info
4	0.25	48
16	$6.25 \cdot 10^{-2}$	25
64	$1.56 \cdot 10^{-2}$	41
256	$3.91 \cdot 10^{-3}$	8
1,024	$9.77 \cdot 10^{-4}$	22
4,096	$2.44 \cdot 10^{-4}$	46
16,384	$6.1 \cdot 10^{-5}$	40
65,536	$1.53 \cdot 10^{-5}$	48
$2.62 \cdot 10^{5}$	$3.81 \cdot 10^{-6}$	33
$1.05 \cdot 10^{6}$	$9.54 \cdot 10^{-7}$	2

```
\pgfplotstabletypeset[
    columns={dof,error1,info},
    column type/.add={|}{}%  results in '|c'
]
    {pgfplotstable.example1.dat}
```

/pgfplots/table/column name={⟨*TEX display column name*⟩}

Sets the column name in the current context.

It is advisable to provide this option inside of a column-specific style, i.e. using `columns/{`⟨*lowlevel colname*⟩`}/.style={column name={`⟨*TEX display column name*⟩`}}` .

/pgfplots/table/assign column name/.code={⟨...⟩}

Allows to *modify* the value of `column name`.

Argument `#1` is the current column name, that means after evaluation of `column name`. After `assign column` name, a new (possibly modified) value for `column name` should be set.

That means you can use `column name` to assign the name as such and `assign column name` to generate final TEX code (for example to insert `\multicolumn{1}{c}{#1}`).

Default is empty which means no change.

/pgfplots/table/multicolumn names={⟨*tabular column type*⟩} (style, initially `c`)

A style which typesets each column name in the current context using a `\multicolumn{1}{`⟨*tabular column type*⟩`}{`⟨*the column name*⟩`}` statement.

Here,⟨*the column name*⟩ is set with `column name` as usual.

/pgfplots/table/dec sep align={⟨*header column type*⟩} (style, initially `c`)

A style which aligns numerical columns at the decimal separator.

The first argument determines the alignment of the header column.

Please note that you need `\usepackage{array}` for this style.

dof	error1	error2	info	grad(log(dof),log(error2))
4	0.25	7.58_{-1}	48	0
16	$6.25 \cdot 10^{-2}$	5.00_{-1}	25	−0.3
64	$1.56 \cdot 10^{-2}$	2.87_{-1}	41	−0.4
256	$3.91 \cdot 10^{-3}$	1.44_{-1}	8	−0.5
1,024	$9.77 \cdot 10^{-4}$	4.42_{-2}	22	−0.85
4,096	$2.44 \cdot 10^{-4}$	1.70_{-2}	46	−0.69
16,384	$6.1 \cdot 10^{-5}$	8.20_{-3}	40	−0.52
65,536	$1.53 \cdot 10^{-5}$	3.91_{-3}	48	−0.54
$2.62 \cdot 10^{5}$	$3.81 \cdot 10^{-6}$	1.95_{-3}	33	−0.5
$1.05 \cdot 10^{6}$	$9.54 \cdot 10^{-7}$	9.77_{-4}	2	−0.5

```
%  requires \usepackage{array}
\pgfplotstabletypeset[
    columns={dof,error1,error2,info,{grad(log(dof),log(error2))}},
    columns/error1/.style={dec sep align},
    columns/error2/.style={sci,sci subscript,sci zerofill,dec sep align},
    columns/info/.style={fixed,dec sep align},
    columns/{grad(log(dof),log(error2))}/.style={fixed,dec sep align}
]
    {pgfplotstable.example1.dat}
```

Or with comma as decimal separator:

dof	error1	error2	info	grad(log(dof),log(error2))
4	0,25	$7{,}58_{-1}$	48	0
16	$6{,}25 \cdot 10^{-2}$	$5{,}00_{-1}$	25	−0,3
64	$1{,}56 \cdot 10^{-2}$	$2{,}87_{-1}$	41	−0,4
256	$3{,}91 \cdot 10^{-3}$	$1{,}44_{-1}$	8	−0,5
1.024	$9{,}77 \cdot 10^{-4}$	$4{,}42_{-2}$	22	−0,85
4.096	$2{,}44 \cdot 10^{-4}$	$1{,}70_{-2}$	46	−0,69
16.384	$6{,}1 \cdot 10^{-5}$	$8{,}20_{-3}$	40	−0,52
65.536	$1{,}53 \cdot 10^{-5}$	$3{,}91_{-3}$	48	−0,54
$2{,}62 \cdot 10^{5}$	$3{,}81 \cdot 10^{-6}$	$1{,}95_{-3}$	33	−0,5
$1{,}05 \cdot 10^{6}$	$9{,}54 \cdot 10^{-7}$	$9{,}77_{-4}$	2	−0,5

```
%  requires \usepackage{array}
\pgfplotstabletypeset[
    use comma,
    columns={dof,error1,error2,info,{grad(log(dof),log(error2))}},
    columns/error1/.style={dec sep align},
    columns/error2/.style={sci,sci subscript,sci zerofill,dec sep align},
    columns/info/.style={fixed,dec sep align},
    columns/{grad(log(dof),log(error2))}/.style={fixed,dec sep align}
]
    {pgfplotstable.example1.dat}
```

It may be advisable to use `fixed zerofill` and/or `sci zerofill` to force at least one digit after the decimal separator to improve placement of exponents:

dof	error1	error2	info	grad(log(dof),log(error2))
4	0,25	$7{,}58_{-1}$	48	0,00
16	$6{,}25 \cdot 10^{-2}$	$5{,}00_{-1}$	25	−0,30
64	$1{,}56 \cdot 10^{-2}$	$2{,}87_{-1}$	41	−0,40
256	$3{,}91 \cdot 10^{-3}$	$1{,}44_{-1}$	8	−0,50
1.024	$9{,}77 \cdot 10^{-4}$	$4{,}42_{-2}$	22	−0,85
4.096	$2{,}44 \cdot 10^{-4}$	$1{,}70_{-2}$	46	−0,69
16.384	$6{,}10 \cdot 10^{-5}$	$8{,}20_{-3}$	40	−0,52
65.536	$1{,}53 \cdot 10^{-5}$	$3{,}91_{-3}$	48	−0,54
$2{,}62 \cdot 10^{5}$	$3{,}81 \cdot 10^{-6}$	$1{,}95_{-3}$	33	−0,50
$1{,}05 \cdot 10^{6}$	$9{,}54 \cdot 10^{-7}$	$9{,}77_{-4}$	2	−0,50

```
%  requires \usepackage{array}
\pgfplotstabletypeset[
    use comma,
    columns={dof,error1,error2,info,{grad(log(dof),log(error2))}},
    columns/error1/.style={dec sep align,sci zerofill},
    columns/error2/.style={sci,sci subscript,sci zerofill,dec sep align},
    columns/info/.style={fixed,dec sep align},
    columns/{grad(log(dof),log(error2))}/.style={fixed,dec sep align,fixed zerofill}
]
    {pgfplotstable.example1.dat}
```

The style `dec sep align` actually introduces two new `tabular` columns[4], namely `r@{}l`. It introduces multicolumns for column names accordingly and handles numbers which do not have a decimal separator.

Note that for fixed point numbers, it might be an alternative to use `fixed zerofill` combined with `column type=r` to get a similar effect.

Please note that this style overwrites `column type`, `assign cell content` and some number formatting settings.

`/pgfplots/table/sci sep align={`⟨*header column type*⟩`}` (style, initially `c`)

A style which aligns numerical columns at the exponent in scientific representation.

The first argument determines the alignment of the header column.

[4] Unfortunately, `dec sep align` is currently not very flexible when it comes to column type modifications. In particular, it is not possible to use colored columns or cells in conjunction with `dec sep align`. The `\rowcolor` command works properly; the color hangover introduced by `colortbl` is adjusted automatically.

It works similiarly to dec sep align, namely by introducing two artificial columns r@{}l for alignment.

Please note that you need \usepackage{array} for this style.

Please note that this style overwrites column type, assign cell content and some number formatting settings.

/pgfplots/table/dcolumn={⟨*tabular column type*⟩}{⟨*type for column name*⟩} (style, initially {D{.}{.}{2}}{c})

A style which can be used together with the dcolumn package of David Carlisle. It also enables alignment at the decimal separator. However, the decimal separator needs to be exactly one character which is incompatible with '{,}' (the default setting for use comma).

/pgfplots/table/sort={⟨*true,false*⟩} (initially false)

If set to true, \pgfplotstabletypeset will sort the table before applying its operation.

See the description of \pgfplotstablesort for how to configure sort key and sort cmp.

dof	error1	error2
$1.05 \cdot 10^{6}$	9.54_{-7}	9.77_{-4}
$2.62 \cdot 10^{5}$	3.81_{-6}	1.95_{-3}
65,536	1.53_{-5}	3.91_{-3}
16,384	6.10_{-5}	8.20_{-3}
4,096	2.44_{-4}	1.70_{-2}
1,024	9.77_{-4}	4.42_{-2}
256	3.91_{-3}	1.44_{-1}
64	1.56_{-2}	2.87_{-1}
16	6.25_{-2}	5.00_{-1}
4	2.50_{-1}	7.58_{-1}

```
\pgfplotstabletypeset[
    sort,sort key=error2,
    columns={dof,error1,error2},
    columns/error1/.style={sci,sci subscript,sci zerofill,dec sep align},
    columns/error2/.style={sci,sci subscript,sci zerofill,dec sep align},
]
    {pgfplotstable.example1.dat}
```

The sort mechanism is applied before the actual typesetting routine starts, i.e. it has the same effect as if you'd call \pgfplotstablesort manually before typesetting the table (however, the sort key has the advantage of respecting the include outfiles caching mechanism). Any create on use specifications are resolved before calling the sort key.

/pgfplots/table/every first column (style, no value)

A style which is installed for every first column only.

level	dof	error1	error2	info	grad(log(dof),log(error2))	quot(error1)
1	4	0.25	0.76	48	0	0
2	16	$6.25 \cdot 10^{-2}$	0.5	25	−0.3	4
3	64	$1.56 \cdot 10^{-2}$	0.29	41	−0.4	4
4	256	$3.91 \cdot 10^{-3}$	0.14	8	−0.5	4
5	1,024	$9.77 \cdot 10^{-4}$	$4.42 \cdot 10^{-2}$	22	−0.85	4
6	4,096	$2.44 \cdot 10^{-4}$	$1.7 \cdot 10^{-2}$	46	−0.69	4
7	16,384	$6.1 \cdot 10^{-5}$	$8.2 \cdot 10^{-3}$	40	−0.52	4
8	65,536	$1.53 \cdot 10^{-5}$	$3.91 \cdot 10^{-3}$	48	−0.54	4
9	$2.62 \cdot 10^{5}$	$3.81 \cdot 10^{-6}$	$1.95 \cdot 10^{-3}$	33	−0.5	4
10	$1.05 \cdot 10^{6}$	$9.54 \cdot 10^{-7}$	$9.77 \cdot 10^{-4}$	2	−0.5	4

```
\pgfplotstabletypeset[
 every head row/.style={before row=\hline,after row=\hline\hline},
 every last row/.style={after row=\hline},
 every first column/.style={
  column type/.add={|}{}
 },
 every last column/.style={
  column type/.add={}{|}
 }]
    {pgfplotstable.example1.dat}
```

/pgfplots/table/every last column (style, no value)

A style which is installed for every last column only.

/pgfplots/table/every even column (style, no value)

A style which is installed for every column with even column index (starting with 0).

```
\pgfplotstableset{
    columns={dof,error1,{grad(log(dof),log(error2))},info},
    columns/error1/.style={
        column name=$L_2$,
        sci,sci zerofill,sci subscript,
        precision=3},
    columns/dof/.style={
        int detect,
        column name=\textsc{Dof}},
    columns/{grad(log(dof),log(error2))}/.style={
        column name=slopes $L_2$,
        fixed,fixed zerofill,
        precision=1}}
```

Dof	L_2	slopes L_2	info
4	2.500_{-1}	0.0	48
16	6.250_{-2}	−0.3	25
64	1.563_{-2}	−0.4	41
256	3.906_{-3}	−0.5	8
1,024	9.766_{-4}	−0.8	22
4,096	2.441_{-4}	−0.7	46
16,384	6.104_{-5}	−0.5	40
65,536	1.526_{-5}	−0.5	48
262,144	3.815_{-6}	−0.5	33
1,048,576	9.537_{-7}	−0.5	2

```
%  requires \usepackage{colortbl}
\pgfplotstabletypeset[
 every even column/.style={
  column type/.add={>{\columncolor[gray]{.8}}}{}
}]
    {pgfplotstable.example1.dat}
```

/pgfplots/table/every odd column (style, no value)

A style which is installed for every column with odd column index (starting with 0).

\pgfplotstablecol

During the evaluation of row or column options, this command expands to the current column's index.

\pgfplotstablecolname

During the evaluation of column options, this command expands to the current column's name. It is valid while \pgfplotstabletypeset processes the column styles (including the preprocessing step explained in Section 3.3), prepares the output cell content and checks row predicates.

\pgfplotstablerow

During the evaluation of row or column options, this command expands to the current row's index.

Note that it will have the special value −1 for the header row.

\pgfplotstablecols

During the evaluation of row or column options, this command expands to the total number of columns in the output table.

\pgfplotstablerows

During evaluation of *columns*, this command expands to the total number of *input* rows. You can use it inside of `row predicate`.

During evaluation of *rows*, this command expands to the total number of *output* rows.

\pgfplotstablename

During \pgfplotstabletypeset, this macro contains the table's macro name as top-level expansion. If you are unfamiliar with "top-level-expansions" and '\expandafter', you will probably never need this macro.

Advances users may benefit from expressions like

\expandafter\pgfplotstabletypeset\pgfplotstablename.

For tables which have been loaded from disk (and have no explicitly assigned macro name), this expands to a temporary macro.

2.3 Configuring Row Appearance: Styles

The following styles allow to configure the final table code *after any cell contents have been assigned.*

/pgfplots/table/before row={⟨*TEX code*⟩}

Contains TEX code which will be installed before the first cell in a row.

Keep in mind that PGFPLOTSTABLE does no magic – it is simply a code generator which produces `tabular` environments. Consequently, you can add any TEX code which you would normally write into your `tabular` environment here.

An example could be a multicolumn heading for which PGFPLOTSTABLE has no own solution:

	Singular		Plural	
	English	Gaeilge	English	Gaeilge
1st	at me	agam	at us	againn
2st	at you	agat	at you	agaibh
3st	at him	aige	at them	acu
	at her	aici		

```
%  \usepackage{booktabs}
\pgfplotstabletypeset[
    column type=l,
    every head row/.style={
        before row={%
            \toprule
            & \multicolumn{2}{c}{Singular} & \multicolumn{2}{c}{Plural}\\
        },
        after row=\midrule,
    },
    every last row/.style={
        after row=\bottomrule},
    columns/person/.style       ={column name=},
    columns/singGaeilge/.style  ={column name=Gaeilge},
    columns/pluralGaeilge/.style={column name=Gaeilge},
    columns/singEnglish/.style  ={column name=English},
    columns/pluralEnglish/.style={column name=English},
    col sep=&,row sep=\\,
    string type,
]{
person & singEnglish  & singGaeilge & pluralEnglish  & pluralGaeilge\\
1st    & at me        & agam        & at us          & againn\\
2st    & at you       & agat        & at you         & agaibh\\
3st    & at him       & aige        & at them        & acu\\
       & at her       & aici        &                &\\
}
```

The example declares a lot of options and is finally followed by a short inline table. The `every head row` style configures `\multicolumn` headings by means of verbatim `tabular` code, together with `booktabs`

rules. It might be helpful to consider the `debug` or `outfile` keys during experiments. The `columns/...` styles are necessary to change the column headings.

Sometimes, one wants to combine `\multicolumn` and `\rowcolor`. From what I know about LaTeX, this is a little bit complicated: it requires the use of `\columncolor` inside of the `\multicolumn`. As in the example above, it is necessary to modify the code generated by PgfplotsTable a little bit. Keep in mind that PgfplotsTable is just a code generator for `tabular` environments – modify whatever you want. The following example demonstrates the combination of `\multicolumn` and `\rowcolor`. It has been taken out of an – admittedly advanced – application:

Quantenzahlen			Term–
n	ℓ	λ	bezeichnung
1	0	0	$1s\sigma$
2	0	0	$2s\sigma$
2	1	0	$2p\sigma$
2	1	1	$2p\pi$
3	2	0	$3d\sigma$
3	2	2	$3d\delta$

```
\newcolumntype{C}{>{\centering\arraybackslash}p{6mm}}%  a centered fixed-width-column
\pgfplotstabletypeset[
    col sep=&,
    row sep=\\,
    every head row/.style={
    %  as in the previous example, this patches the first row:
        before row={
            \hline
            \rowcolor{lightgray}
            \multicolumn{3}{|>{\columncolor{lightgray}}c|}{Quantenzahlen} & Term--\\
            \rowcolor{lightgray}
        },
        after row=\hline,
    },
    every last row/.style={
        after row=\hline},
    %  define column-specific styles:
    columns/n/.style={column type=|C,column name=$n$},
    columns/l/.style={column type=|C,column name=$\ell$},
    columns/lambda/.style={column type=|C,column name=$\lambda$},
    columns/text/.style={column type=|c|,column name=bezeichnung,
        string type % <-it contains formatted data
    },
]
{
n & l & lambda & text\\
1 & 0 & 0 & $1 s\sigma$ \\
2 & 0 & 0 & $2 s\sigma$ \\
2 & 1 & 0 & $2 p\sigma$ \\
2 & 1 & 1 & $2 p \pi $\\
3 & 2 & 0 & $3 d\sigma$ \\
3 & 2 & 2 & $3 d\delta$ \\
}
```

Up to the number formatting (which actually invokes `\pgfmathprintnumber`), the code above is equivalent to the listing

```
\newcolumntype{C}{>{\centering\arraybackslash}p{6mm}}%  a centered fixed-width-column
\begin{tabular}{|C|C|C|c|}
\hline
\rowcolor{lightgray} \multicolumn{3}{|>{\columncolor{lightgray}}c|}{Quantenzahlen} & Term--\\
\rowcolor{lightgray} $n$ & $\ell$ & $\lambda$ & bezeichnung\\
\hline
$1$ & $0$ & $0$ & $1 s\sigma$ \\
$2$ & $0$ & $0$ & $2 s\sigma$ \\
$2$ & $1$ & $0$ & $2 p\sigma$ \\
$2$ & $1$ & $1$ & $2 p \pi$ \\
$3$ & $2$ & $0$ & $3 d\sigma$ \\
$3$ & $2$ & $2$ & $3 d\delta$ \\
\hline
\end{tabular}
```

Clearly, the overhead introduced by defining a lot of styles is only worth the effort if you require number printing, automated processing, or have a huge bulk of similar tables.

/pgfplots/table/after row={⟨*TEX code*⟩}

Contains TEX code which will be installed after the last cell in a row (i.e. after \\).

/pgfplots/table/every even row (style, no value)

A style which is installed for each row with even row index. The first row is supposed to be a "head" row and does not count. Indexing starts with 0.

```
\pgfplotstableset{
    columns={dof,error1,{grad(log(dof),log(error2))}},
    columns/error1/.style={
        column name=$L_2$,
        sci,sci zerofill,sci subscript,
        precision=3},
    columns/dof/.style={
        int detect,
        column name=\textsc{Dof}},
    columns/{grad(log(dof),log(error2))}/.style={
        column name=slopes $L_2$,
        fixed,fixed zerofill,
        precision=1}}
```

Dof	L_2	slopes L_2
4	2.500_{-1}	0.0
16	6.250_{-2}	−0.3
64	1.563_{-2}	−0.4
256	3.906_{-3}	−0.5
1,024	9.766_{-4}	−0.8
4,096	2.441_{-4}	−0.7
16,384	6.104_{-5}	−0.5
65,536	1.526_{-5}	−0.5
262,144	3.815_{-6}	−0.5
1,048,576	9.537_{-7}	−0.5

```
%  requires \usepackage{booktabs}
\pgfplotstabletypeset[
    every head row/.style={
        before row=\toprule,after row=\midrule},
    every last row/.style={
        after row=\bottomrule},
]
    {pgfplotstable.example1.dat}
```

Dof	L_2	slopes L_2
4	2.500_{-1}	0.0
16	6.250_{-2}	−0.3
64	1.563_{-2}	−0.4
256	3.906_{-3}	−0.5
1,024	9.766_{-4}	−0.8
4,096	2.441_{-4}	−0.7
16,384	6.104_{-5}	−0.5
65,536	1.526_{-5}	−0.5
262,144	3.815_{-6}	−0.5
1,048,576	9.537_{-7}	−0.5

```
%  requires \usepackage{booktabs,colortbl}
\pgfplotstabletypeset[
    every even row/.style={
        before row={\rowcolor[gray]{0.9}}},
    every head row/.style={
        before row=\toprule,after row=\midrule},
    every last row/.style={
        after row=\bottomrule},
]
    {pgfplotstable.example1.dat}
```

/pgfplots/table/every odd row (style, no value)

A style which is installed for each row with odd row index. The first row is supposed to be a "head" row and does not count. Indexing starts with 0.

/pgfplots/table/every head row (style, no value)

A style which is installed for each first row in the tabular. This can be used to adjust options for column names or to add extra lines/colours.

	Col A	B	C
The first column	E	F	

```
\pgfplotstabletypeset[
  %  suppress the header row 'col1  col2  col3':
  every head row/.style={output empty row},
  col sep=comma,
  columns/col1/.style={string type,column type=r},
  columns/col2/.style={string type,column type=l},
  columns/col3/.style={string type,column type=l},
  ]
{
    col1,col2,col3
    Col A,B,C
    The first column,E,F
}
```

/pgfplots/table/every first row (style, no value)

A style which is installed for each first *data* row, i.e. after the head row.

/pgfplots/table/every last row (style, no value)

A style which is installed for each last row.

/pgfplots/table/every row no ⟨*index*⟩ (style, no value)

A style which is installed for the row with index ⟨*index*⟩.

/pgfplots/table/every nth row={⟨*integer*⟩}{⟨*options*⟩}
/pgfplots/table/every nth row={⟨*integer*[⟨*shift*⟩]⟩}{⟨*options*⟩}

This allows to install ⟨*options*⟩ for every nth row with $n =$⟨*integer*⟩.

a	b
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8

```
\pgfplotstabletypeset[
    every nth row={3}{before row=\midrule},
    every head row/.style={
        before row=\toprule,after row=\midrule},
    every last row/.style={
        after row=\bottomrule},
]{
    a b
    0 0
    1 1
    2 2
    3 3
    4 4
    5 5
    6 6
    7 7
    8 8
}
```

Only data rows are considered for `every nth row`; it will never apply to column names and data rows are numbered like $i = 0, 1, 2, \ldots$ (the example above applies it to the rows with $a = 3, 6$). Since the case $i = 0$ can be handled by `every first row`, it is not considered for `every nth row`.

The second syntax allows to provide an additional ⟨*shift*⟩:

a	b
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10

```
\pgfplotstabletypeset[
    every nth row={3[+1]}{before row=\midrule},
]{
    a b
    0 0
    1 1
    2 2
    3 3
    4 4
    5 5
    6 6
    7 7
    8 8
    9 9
    10 10
}
```

Here, the style is applied to rows $i = 1, 4, 7, 10$ (mathematically, it is applied if (i mod n) $=\langle shift\rangle$). The $\langle shift\rangle$ can be negative.

You can define many `every nth row` styles, they are processed in the order of occurance (consider using `before row/.add={`⟨*before existing*⟩`}{`⟨*after existing*⟩`}` to modify an existing value).

Note that `every nth row/.style 2 args=...` is the same as `every nth row=...`.

`/pgfplots/table/output empty row` (style, no value)

A style which suppresses output for the current row.

This style is evaluated very late, after all column-specific content modifications have been applied. It is equivalent to

```
\pgfplotstableset{
    output empty row/.style={
        typeset cell/.style={@cell content={}}
    },
}
```

See `every head row` for an application.

2.4 Configuring Single–Cell Appearance: Styles

Besides the possibilities to change column styles and row styles, there are also a couple of styles to change single cells.

`/pgfplots/table/every row` ⟨*rowindex*⟩ `column` ⟨*colindex*⟩ (style, no value)

A style which applies to at most one cell: the one with row index ⟨*rowindex*⟩ and column index ⟨*colindex*⟩. Each of these indices starts with 0.

The style is evaluated in the same context as the `preproc cell content`, `assign cell content`, and `postproc cell content` keys and it is a legitimate possibility to modify any of these parameters. It is also possible to replace the initial cell value by assigning something to `@cell content`.

For example, consider this unmodified table:

colA	colB	colC
11	12	13
21	22	23

```
\pgfplotstabletypeset[
    col sep=&,row sep=\\]{
  colA & colB & colC \\
  11   & 12   & 13   \\
  21   & 22   & 23   \\
}
```

Now, we change the number format of one of its cells, and at the same time we change the formatting of another (single) cell:

colA	colB	colC
11	12	13
21	22	$2.3 \cdot 10^{1}$

```
\pgfplotstabletypeset[
    every row 1 column 2/.style={/pgf/number format/sci},
    every row 0 column 0/.style={postproc cell content/.style={@cell content=\textbf{##1}}},
    col sep=&,row sep=\\]{
  colA & colB & colC \\
  11   & 12   & 13   \\
  21   & 22   & 23   \\
}
```

Note that this style is (only) applied to input row/column values.

`/pgfplots/table/every row no` ⟨*rowindex*⟩ `column no` ⟨*colindex*⟩ (style, no value)

This is actually the same – `row no` and `row` are both supported, the same for `column` and `column no`.

`/pgfplots/table/every row` ⟨*rowindex*⟩ `column` ⟨*colname*⟩ (style, no value)

A similar style as above, but it allows column *names* rather than column indices. Column names need to be provided in the same way as for other column–specific styles (including the extra curly braces in case ⟨*colname*⟩ contains special characters).

Our example from above can thus become:

colA	colB	colC
88	12	13
21	44	23

```
\pgfplotstabletypeset[
    every row 1 column colB/.style={string replace*={2}{4}},
    every row 0 column colA/.style={preproc/expr={##1*8}},
    col sep=&,row sep=\\]{
  colA & colB & colC \\
  11   & 12   & 13   \\
  21   & 22   & 23   \\
}
```

The example employs the `string replace*` preprocessor style and the `preproc/expr` style. All preprocessor or postprocessor styles can be used.

Please refer to Section 3 for predefined choices.

2.5 Customizing and Getting the Tabular Code

The following keys allow changes of alignment (`begin table`) and `font` and they allow to write the generated code to `outfile`s (see also `write to macro`). Furthermore, the generated code can be fine–tuned to provide other sorts of table output, beyond LaTeX.

/pgfplots/table/every table={⟨*file name*⟩}

A style which is installed at the beginning of every `\pgfplotstabletypeset` command[5].

The table file name is given as first argument.

/pgfplots/table/font={⟨*font name*⟩} (initially `empty`)

Assigns a font used for the complete table.

/pgfplots/table/begin table={⟨*code*⟩} (initially `\begin{tabular}`)

Contains {⟨*code*⟩} which is generated as table start.

The following example uses a `longtable` instead of `tabular`:

```
\pgfplotstableset{
    begin table=\begin{longtable},
    end table=\end{longtable},
}
```

Note that `longtable` allows the use of *replicated headers* for multi-page tables by means of its `\endhead` macro:

```
%  replicate column names on top of every page of a multi-page table:
\pgfplotstableset{
    row sep=\\,
    begin table=\begin{longtable},
    end table=\end{longtable},
    every head row/.append style={after row=\endhead},
}
```

If the first page should have a different header, you can use `\endfirsthead` provided by the `longtable` package:

```
%  replicate column names on top of every page of a multi-page table,
%  but with different values for first page:
\pgfplotstableset{
    row sep=\\,
    begin table=\begin{longtable},
    end table=\end{longtable},
    every head row/.append style={after row={%
            \caption{The caption}%
            \endfirsthead
            \multicolumn{3}{c}{{\bfseries \tablename\ \thetable{} -- continued from previous page}} \\
            \endhead
        },
    },
}
```

[5] The `every table` style is installed *after* options provided to `\pgfplotstabletypeset`; it has higher precedence.

The preceding example uses the `longtable` macros `\caption`, `\endfirsthead`, `\thetable`, and `\endhead`. In addition, it requires to provide the number of columns (`{3}` in this case) *explicitly*.

It is also possible to *change* the value of `begin table`. For example,

```
\pgfplotstableset{
    begin table/.add={}{[t]},
}
```

prepends the empty string `{}` and appends the prefix `[t]`. Thus, '`\begin{tabular}`' becomes '`\begin{tabular}[t]`'.

`/pgfplots/table/end table={`⟨*code*⟩`}` (initially `\end{tabular}`)

Contains ⟨*code*⟩ which is generated as table end.

`/pgfplots/table/typeset cell/.code={`⟨...⟩`}`

A code key which assigns `/pgfplots/table/@cell content` to the final output of the current cell.

The first argument, `#1`, is the final cell's value. After this macro, the value of `@cell content` will be written to the output.

The default implementation is

```
\ifnum\pgfplotstablecol=\pgfplotstablecols
    \pgfkeyssetvalue{/pgfplots/table/cell content}{#1\\}%
\else
    \pgfkeyssetvalue{/pgfplots/table/cell content}{#1&}%
\fi
```

Attention: The value of `\pgfplotstablecol` starts with 1 in this context, i.e. it is in the range $1, \ldots, n$ where $n =$ `\pgfplotstablecols`. This simplifies checks whether we have the last column.

`/pgfplots/table/outfile={`⟨*file name*⟩`}` (initially `empty`)

Writes the generated tabular code into ⟨*file name*⟩. It can then be used with `\input{`⟨*file name*⟩`}`, PgfplotsTable is no longer required since it contains a completely normal `tabular`.

dof	error1
4	0.25
16	$6.25 \cdot 10^{-2}$
64	$1.56 \cdot 10^{-2}$
256	$3.91 \cdot 10^{-3}$
1,024	$9.77 \cdot 10^{-4}$
4,096	$2.44 \cdot 10^{-4}$
16,384	$6.1 \cdot 10^{-5}$
65,536	$1.53 \cdot 10^{-5}$
$2.62 \cdot 10^{5}$	$3.81 \cdot 10^{-6}$
$1.05 \cdot 10^{6}$	$9.54 \cdot 10^{-7}$

```
\pgfplotstabletypeset[
    columns={dof,error1},
    outfile=pgfplotstable.example1.out.tex]
    {pgfplotstable.example1.dat}
```

and `pgfplotstable.example1.out.tex` contains

```
\begin {tabular}{cc}%
dof&error1\\%
\pgfutilensuremath {4}&\pgfutilensuremath {0.25}\\%
\pgfutilensuremath {16}&\pgfutilensuremath {6.25\cdot 10^{-2}}\\%
\pgfutilensuremath {64}&\pgfutilensuremath {1.56\cdot 10^{-2}}\\%
\pgfutilensuremath {256}&\pgfutilensuremath {3.91\cdot 10^{-3}}\\%
\pgfutilensuremath {1{,}024}&\pgfutilensuremath {9.77\cdot 10^{-4}}\\%
\pgfutilensuremath {4{,}096}&\pgfutilensuremath {2.44\cdot 10^{-4}}\\%
\pgfutilensuremath {16{,}384}&\pgfutilensuremath {6.1\cdot 10^{-5}}\\%
\pgfutilensuremath {65{,}536}&\pgfutilensuremath {1.53\cdot 10^{-5}}\\%
\pgfutilensuremath {2.62\cdot 10^{5}}&\pgfutilensuremath {3.81\cdot 10^{-6}}\\%
\pgfutilensuremath {1.05\cdot 10^{6}}&\pgfutilensuremath {9.54\cdot 10^{-7}}\\%
\end {tabular}%
```

The command `\pgfutilensuremath` checks whether math mode is active and switches to math mode if necessary[6].

`/pgfplots/table/include outfiles={⟨boolean⟩}` (initially `false`)

If enabled, any already existing outfile will be `\input` instead of overwritten.

```
\pgfplotstableset{include outfiles} %  for example in the document's preamble
```

This allows to place any corrections manually into generated output files since PgfplotsTable won't overwrite the resulting tables automatically.

This will affect tables for which the `outfile` option is set. If you wish to apply it to every table, consider

```
\pgfplotstableset{every table/.append style={outfile={#1.out}}}
```

which will generate an `outfile` name for every table.

`/pgfplots/table/force remake={⟨boolean⟩}` (initially `false`)

If enabled, the effect of `include outfiles` is disabled. As all key settings only last until the next brace (or `\end⟨⟩`), this key can be used to regenerate some output files while others are still included.

`/pgfplots/table/write to macro={⟨\macroname⟩}`

If the value of `write to macro` is not empty, the completely generated (tabular) code will be written into the macro ⟨*macroname*⟩.

See the `typeset=false` key in case you need *only* the resulting macro.

`/pgfplots/table/skip coltypes=true|false` (initially `false`)

Allows to skip the ⟨*coltypes*⟩ in `\begin{tabular}{⟨coltypes⟩}`. This allows simplifications for other table types which don't have LaTeX's table format.

`/pgfplots/table/typeset=true|false` (initially `true`)

A boolean which disables the final typesetting stage. Use `typeset=false` in conjunction with `write to macro` if only the generated code is of interest and TeX should not attempt to produce any content in the output `pdf`.

`/pgfplots/table/debug={⟨boolean⟩}` (initially `false`)

If enabled, it will write every final tabular code to your log file.

`/pgfplots/table/TeX comment={⟨comment sign⟩}` (initially `%`)

The comment sign which is inserted into outfiles to suppress trailing white space.

As a last example, we use PgfplotsTable to write an `.html` file (including number formatting and rounding!):

```
<table>
<tr><td>level</td><td>dof</td><td>error1</td></tr>
<tr><td>1</td><td>4</td><td>0.25</td></tr>
<tr><td>2</td><td>16</td><td>6.25e-2</td></tr>
<tr><td>3</td><td>64</td><td>1.56e-2</td></tr>
<tr><td>4</td><td>256</td><td>3.91e-3</td></tr>
<tr><td>5</td><td>1024</td><td>9.77e-4</td></tr>
<tr><td>6</td><td>4096</td><td>2.44e-4</td></tr>
<tr><td>7</td><td>16384</td><td>6.1e-5</td></tr>
<tr><td>8</td><td>65536</td><td>1.53e-5</td></tr>
<tr><td>9</td><td>2.62e5</td><td>3.81e-6</td></tr>
<tr><td>10</td><td>1.05e6</td><td>9.54e-7</td></tr>
</table>
```

[6] Please note that `\pgfutilensuremath` needs to be replaced by `\ensuremath` if you want to use the output file independent of pgf. That can be done by `\let\pgfutilensuremath=\ensuremath` which enables the LaTeX-command `\ensuremath`.

```
\pgfplotstabletypeset[
    begin table={<table>}, end table={</table>},
    typeset cell/.style={
      /pgfplots/table/@cell content={<td>#1</td>}
    },
    before row=<tr>,after row=</tr>,
    skip coltypes, typeset=false,
    verbatim,%  configures number printer
    TeX comment=,
    columns={level,dof,error1},
    outfile=pgfplotstable.example1.out.html,
]{pgfplotstable.example1.dat}
\lstinputlisting
    [basicstyle=\ttfamily\footnotesize]
    {pgfplotstable.example1.out.html}
```

2.6 Defining Column Types for `tabular`

Besides input of text files, it is sometimes desirable to define column types for existing `tabular` environments.

`\newcolumntype{`⟨*letter*⟩`}[`⟨*number of arguments*⟩`]>{`⟨*before column*⟩`}`⟨*column type*⟩`<{`⟨*after column*⟩`}`

The command `\newcolumntype` is part of the `array` package and it defines a new column type ⟨*letter*⟩ for use in LaTeX tabular environments.

```
\usepackage{array}
```

-a+	b
-c+	d

```
\newcolumntype{d}{>{-}c<{+}}
\begin{tabular}{dl}
a & b \\
c & d \\
\end{tabular}
```

Now, the environment `pgfplotstablecoltype` can be used in ⟨*before column*⟩ and ⟨*after column*⟩ to define numerical columns:

9	2.50_{-1}
25	6.25_{-2}
81	1.56_{-2}
289	3.91_{-3}
1,089	9.77_{-4}
4,225	2.44_{-4}
16,641	6.10_{-5}
66,049	1.53_{-5}
263,169	3.81_{-6}
1,050,625	9.54_{-7}

```
%  requires \usepackage{array}
\newcolumntype{L}[1]
    {>{\begin{pgfplotstablecoltype}[#1]}r<{\end{pgfplotstablecoltype}}}

\begin{tabular}{L{int detect}L{sci,sci subscript,sci zerofill}}
9       & 2.50000000e-01\\
25      & 6.25000000e-02\\
81      & 1.56250000e-02\\
289     & 3.90625000e-03\\
1089    & 9.76562500e-04\\
4225    & 2.44140625e-04\\
16641   & 6.10351562e-05\\
66049   & 1.52587891e-05\\
263169 & 3.81469727e-06\\
1050625& 9.53674316e-07\\
\end{tabular}
```

The environment `pgfplotstablecoltype` accepts an optional argument which may contain any number formatting options. It is an error if numerical columns contain non-numerical data, so it may be necessary to use `\multicolumn` for column names.

Dof	Error
9	2.50_{-1}
25	6.25_{-2}
81	1.56_{-2}
289	3.91_{-3}
1,089	9.77_{-4}
4,225	2.44_{-4}
16,641	6.10_{-5}
66,049	1.53_{-5}
263,169	3.81_{-6}
1,050,625	9.54_{-7}

```
% requires \usepackage{array}
\newcolumntype{L}[1]
    {>{\begin{pgfplotstablecoltype}[#1]}r<{\end{pgfplotstablecoltype}}}

\begin{tabular}{L{int detect}L{sci,sci subscript,sci zerofill}}
\multicolumn{1}{r}{Dof} & \multicolumn{1}{r}{Error}\\
9      & 2.50000000e-01\\
25     & 6.25000000e-02\\
81     & 1.56250000e-02\\
289    & 3.90625000e-03\\
1089   & 9.76562500e-04\\
4225   & 2.44140625e-04\\
16641  & 6.10351562e-05\\
66049  & 1.52587891e-05\\
263169 & 3.81469727e-06\\
1050625& 9.53674316e-07\\
\end{tabular}
```

2.7 Number Formatting Options

The following extract of [2] explains how to configure number formats. The common option prefix `/pgf/number format` can be omitted; it will be recognized automatically.

All these number formatting options can also be applied to PGFPLOTS.

`\pgfmathprintnumber{`⟨*x*⟩`}`

Generates pretty-printed output for the (real) number ⟨*x*⟩. The input number ⟨*x*⟩ is parsed using `\pgfmathfloatparsenumber` which allows arbitrary precision.

Numbers are typeset in math mode using the current set of number printing options, see below. Optional arguments can also be provided using `\pgfmathprintnumber[`⟨*options*⟩`]{`⟨*x*⟩`}`.

`\pgfmathprintnumberto{`⟨*x*⟩`}{`⟨*macro*⟩`}`

Returns the resulting number into ⟨*macro*⟩ instead of typesetting it directly.

`/pgf/number format/fixed` (no value)

Configures `\pgfmathprintnumber` to round the number to a fixed number of digits after the period, discarding any trailing zeros.

4.57 0 0.1 24,415.98 123,456.12

```
\pgfkeys{/pgf/number format/.cd,fixed,precision=2}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-04}\hspace{1em}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{24415.98123}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

See Section 2.7.1 for how to change the appearance.

`/pgf/number format/fixed zerofill=`{⟨*boolean*⟩} (default `true`)

Enables or disables zero filling for any number drawn in fixed point format.

4.57 0.00 0.10 24,415.98 123,456.12

```
\pgfkeys{/pgf/number format/.cd,fixed,fixed zerofill,precision=2}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-04}\hspace{1em}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{24415.98123}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

This key affects numbers drawn with `fixed` or `std` styles (the latter only if no scientific format is chosen).

4.57 $5 \cdot 10^{-5}$ 1.00 $1.23 \cdot 10^{5}$

```
\pgfkeys{/pgf/number format/.cd,std,fixed zerofill,precision=2}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-05}\hspace{1em}
\pgfmathprintnumber{1}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

See Section 2.7.1 for how to change the appearance.

/pgf/number format/sci (no value)

Configures \pgfmathprintnumber to display numbers in scientific format, that means sign, mantissa and exponent (base 10). The mantissa is rounded to the desired precision (or sci precision, see below).

$4.57 \cdot 10^{0}$ $5 \cdot 10^{-4}$ $1 \cdot 10^{-1}$ $2.44 \cdot 10^{4}$ $1.23 \cdot 10^{5}$

```
\pgfkeys{/pgf/number format/.cd,sci,precision=2}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-04}\hspace{1em}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{24415.98123}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

See Section 2.7.1 for how to change the exponential display style.

/pgf/number format/sci zerofill={⟨*boolean*⟩} (default true)

Enables or disables zero filling for any number drawn in scientific format.

$4.57 \cdot 10^{0}$ $5.00 \cdot 10^{-4}$ $1.00 \cdot 10^{-1}$ $2.44 \cdot 10^{4}$ $1.23 \cdot 10^{5}$

```
\pgfkeys{/pgf/number format/.cd,sci,sci zerofill,precision=2}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-04}\hspace{1em}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{24415.98123}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

As with fixed zerofill, this option does only affect numbers drawn in sci format (or std if the scientific format is chosen).

See Section 2.7.1 for how to change the exponential display style.

/pgf/number format/zerofill={⟨*boolean*⟩} (style, default true)

Sets both fixed zerofill and sci zerofill at once.

/pgf/number format/std (no value)
/pgf/number format/std=⟨*lower e*⟩
/pgf/number format/std=⟨*lower e*⟩:⟨*upper e*⟩

Configures \pgfmathprintnumber to a standard algorithm. It chooses either fixed or sci, depending on the order of magnitude. Let $n = s \cdot m \cdot 10^{e}$ be the input number and p the current precision. If $-p/2 \le e \le 4$, the number is displayed using fixed format. Otherwise, it is displayed using sci format.

4.57 $5 \cdot 10^{-4}$ 0.1 24,415.98 $1.23 \cdot 10^{5}$

```
\pgfkeys{/pgf/number format/.cd,std,precision=2}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-04}\hspace{1em}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{24415.98123}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

The parameters can be customized using the optional integer argument(s): if ⟨*lower e*⟩ $\le e \le$ ⟨*upper e*⟩, the number is displayed in fixed format, otherwise in sci format. Note that ⟨*lower e*⟩ should be negative for useful results. The precision used for scientific format can be adjusted with sci precision if necessary.

/pgf/number format/relative*=⟨*exponent base 10*⟩

Configures \pgfmathprintnumber to format numbers relative to an order of magnitude, 10^r, where r is an integer number.

This key addresses different use-cases.

First use-case: provide a unified format for a *sequence* of numbers. Consider the following test:

0 1.2 6 20.6 87

```
\pgfkeys{/pgf/number format/relative*={1}}
\pgfmathprintnumber{6.42e-16}\hspace{1em}
\pgfmathprintnumber{1.2}\hspace{1em}
\pgfmathprintnumber{6}\hspace{1em}
\pgfmathprintnumber{20.6}\hspace{1em}
\pgfmathprintnumber{87}
```

With any other style, the 6.42e-16 would have been formatted as an isolated number. Here, it is rounded to 0 because when viewed relative to 10^1 (the exponent 1 is the argument for relative), it has no significant digits.

123 0 0

```
\pgfkeys{/pgf/number format/relative*={2}}
\pgfmathprintnumber{123.345}\hspace{1em}
\pgfmathprintnumber{0.0012}\hspace{1em}
\pgfmathprintnumber{0.0014}\hspace{1em}
```

The example above applies the initial precision=2 to 123.345 – relative to 100. Two significant digits of 123.345 relative to 100 are 123. Note that the "2 significant digits of 123.345" translates to "round 1.2345 to 2 digits", which would yield 1.2300. Similarly, the other two numbers are 0 compared to 100 using the given precision.

123.345 $1.2 \cdot 10^{-3}$ $1.4 \cdot 10^{-3}$

```
\pgfkeys{/pgf/number format/relative*={-3}}
\pgfmathprintnumber{123.345}\hspace{1em}
\pgfmathprintnumber{0.0012}\hspace{1em}
\pgfmathprintnumber{0.0014}\hspace{1em}
```

Second use-case: improve rounding in the presence of *inaccurate* numbers. Let us suppose that some limited-precision arithmetics resulted in the result 123456999 (like the fpu of PGF). You know that its precision is about five or six significant digits. And you want to provide a fixed point output. In this case, the trailing digits ....999 are a numerical artifact due to the limited precision. Use relative*=3,precision=0 to eliminate the artifacts:

$1.23457 \cdot 10^8$ $1.23457 \cdot 10^8$

```
\pgfkeys{/pgf/number format/.cd,relative*={3},precision=0}
\pgfmathprintnumber{123456999}\hspace{1em}
\pgfmathprintnumber{123456999.12}
```

Here, precision=0 means that we inspect 123456.999 and round that number to 0 digits. Finally, we move the period back to its initial position. Adding relative style=fixed results in fixed point output format:

123,457,000 123,457,000

```
\pgfkeys{/pgf/number format/.cd,relative*={3},precision=0,relative style=fixed}
\pgfmathprintnumber{123456999}\hspace{1em}
\pgfmathprintnumber{123456999.12}
```

Note that there is another alternative for this use-case which is discussed later: the fixed relative style.

123,457,000 123,457,000

```
\pgfkeys{/pgf/number format/.cd,fixed relative,precision=6}
\pgfmathprintnumber{123456999}\hspace{1em}
\pgfmathprintnumber{123456999.12}
```

You might wonder why there is an asterisk in the key's name. The short answer is: there is also a /pgf/number format/relative number printer which does unexpected things. The key relative* repairs this. Existing code will still use the old behavior.

Technically, the key works as follows: as already explained above, relative*=3 key applied to 123456999.12 moves the period by three positions and analyzes 123456.99912. Mathematically speaking, we are given a number $x = \pm m \cdot 10^e$ and we attempt to apply relative*=r. The method then rounds $x/10^r$ to precision digits. Afterwards, it multiplies the result by 10^r and typesets it.

/pgf/number format/every relative (style, no value)

A style which configures how the relative method finally displays its results.

The initial configuration is

```
\pgfkeys{/pgf/number format/every relative/.style=std}
```

Note that rounding is turned off when the resulting style is being evaluated (since relative already rounded the number).

Although supported, I discourage from using fixed zerofill or sci zerofill in this context – it may lead to a suggestion of higher precision than is actually used (because fixed zerofill might simply add .00 although there was a different information before relative rounded the result).

/pgf/number format/relative style={⟨*options*⟩}

The same as every relative/.append style={⟨*options*⟩}.

/pgf/number format/fixed relative (no value)

Configures \pgfmathprintnumber to format numbers in a similar way to the fixed style, but the precision is interpreted relatively to the number's exponent.

The motivation is to get the same rounding effect as for sci, but to display the number in the fixed style:

1,000 100 0.00001 0.0101 1.24 1,000 1,010

```
\pgfkeys{/pgf/number format/.cd,fixed relative,precision=3}
\pgfmathprintnumber{1000.0123}\hspace{1em}
\pgfmathprintnumber{100.0567}\hspace{1em}
\pgfmathprintnumber{0.000010003452}\hspace{1em}
\pgfmathprintnumber{0.010073452}\hspace{1em}
\pgfmathprintnumber{1.23567}\hspace{1em}
\pgfmathprintnumber{1003.75}\hspace{1em}
\pgfmathprintnumber{1006.75}\hspace{1em}
```

The effect of fixed relative is that the number is rounded to *exactly* the first ⟨*precision*⟩ non-zero digits, no matter how many leading zeros the number might have.

Use fixed relative if you want fixed and if you know that only the first n digits are correct. Use sci if you need a scientific display style and only the first n digits are correct.

Note that fixed relative ignores the fixed zerofill flag.

See also the relative* key. Note that the relative={⟨*exponent*⟩} key explicitly moves the period to some designated position before it attempts to round the number. Afterwards, it "rounds from the right", i.e. it rounds to that explicitly chosen digit position. In contrast to that, fixed relative "rounds from the left": it takes the *first* non-zero digit, temporarily places the period after this digit, and rounds that number. The rounding style fixed leaves the period where it is, and rounds everything behind that digit. The sci style is similar to fixed relative.

/pgf/number format/int detect (no value)

Configures \pgfmathprintnumber to detect integers automatically. If the input number is an integer, no period is displayed at all. If not, the scientific format is chosen.

15 20 $2.04 \cdot 10^{1}$ $1 \cdot 10^{-2}$ 0

```
\pgfkeys{/pgf/number format/.cd,int detect,precision=2}
\pgfmathprintnumber{15}\hspace{1em}
\pgfmathprintnumber{20}\hspace{1em}
\pgfmathprintnumber{20.4}\hspace{1em}
\pgfmathprintnumber{0.01}\hspace{1em}
\pgfmathprintnumber{0}
```

\pgfmathifisint{⟨*number constant*⟩}{⟨*true code*⟩}{⟨*false code*⟩}

A command which does the same check as `int detect`, but it invokes ⟨*true code*⟩ if the ⟨*number constant*⟩ actually is an integer and the ⟨*false code*⟩ if not.

As a side–effect, `\pgfretval` will contain the parsed number, either in integer format or as parsed floating point number.

The argument ⟨*number constant*⟩ will be parsed with `\pgfmathfloatparsenumber`.

15 is an int: 15. 15.5 is no int

```
15 \pgfmathifisint{15}{is an int: \pgfretval.}{is no int}\hspace{1em}
15.5 \pgfmathifisint{15.5}{is an int: \pgfretval.}{is no int}
```

/pgf/number format/int trunc (no value)

Truncates every number to integers (discards any digit after the period).

4 0 0 24,415 123,456

```
\pgfkeys{/pgf/number format/.cd,int trunc}
\pgfmathprintnumber{4.568}\hspace{1em}
\pgfmathprintnumber{5e-04}\hspace{1em}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{24415.98123}\hspace{1em}
\pgfmathprintnumber{123456.12345}
```

/pgf/number format/frac (no value)

Displays numbers as fractionals.

$\frac{1}{3}$ $\frac{1}{2}$ $\frac{16}{75}$ $\frac{3}{25}$ $\frac{2}{75}$ $-\frac{1}{75}$ $\frac{18}{25}$ $\frac{1}{15}$ $\frac{2}{15}$ $-\frac{1}{75}$ $3\frac{1}{3}$ $1\frac{16993}{72465}$ 1 -6

```
\pgfkeys{/pgf/number format/frac}
\pgfmathprintnumber{0.333333333333333}\hspace{1em}
\pgfmathprintnumber{0.5}\hspace{1em}
\pgfmathprintnumber{2.133333333333325e-01}\hspace{1em}
\pgfmathprintnumber{0.12}\hspace{1em}
\pgfmathprintnumber{2.666666666666646e-02}\hspace{1em}
\pgfmathprintnumber{-1.333333333333334e-02}\hspace{1em}
\pgfmathprintnumber{7.200000000000000e-01}\hspace{1em}
\pgfmathprintnumber{6.666666666666667e-02}\hspace{1em}
\pgfmathprintnumber{1.333333333333333e-01}\hspace{1em}
\pgfmathprintnumber{-1.333333333333333e-02}\hspace{1em}
\pgfmathprintnumber{3.3333333}\hspace{1em}
\pgfmathprintnumber{1.2345}\hspace{1em}
\pgfmathprintnumber{1}\hspace{1em}
\pgfmathprintnumber{-6}
```

/pgf/number format/frac TeX={⟨*macro*⟩} (initially `\frac`)

Allows to use a different implementation for `\frac` inside of the `frac` display type.

/pgf/number format/frac denom=⟨*int*⟩ (initially `empty`)

Allows to provide a custom denominator for `frac`.

$\frac{1}{10}$ $\frac{5}{10}$ $1\frac{2}{10}$ $-\frac{6}{10}$ $-1\frac{4}{10}$

```
\pgfkeys{/pgf/number format/.cd,frac, frac denom=10}
\pgfmathprintnumber{0.1}\hspace{1em}
\pgfmathprintnumber{0.5}\hspace{1em}
\pgfmathprintnumber{1.2}\hspace{1em}
\pgfmathprintnumber{-0.6}\hspace{1em}
\pgfmathprintnumber{-1.4}\hspace{1em}
```

/pgf/number format/frac whole=true|false (initially true)

Configures whether complete integer parts shall be placed in front of the fractional part. In this case, the fractional part will be less then 1. Use frac whole=false to avoid whole number parts.

$\frac{201}{10}$ $\frac{11}{2}$ $\frac{6}{5}$ $-\frac{28}{5}$ $-\frac{7}{5}$

```
\pgfkeys{/pgf/number format/.cd,frac, frac whole=false}
\pgfmathprintnumber{20.1}\hspace{1em}
\pgfmathprintnumber{5.5}\hspace{1em}
\pgfmathprintnumber{1.2}\hspace{1em}
\pgfmathprintnumber{-5.6}\hspace{1em}
\pgfmathprintnumber{-1.4}\hspace{1em}
```

/pgf/number format/frac shift={⟨*integer*⟩} (initially 4)

In case you experience stability problems, try experimenting with a different frac shift. Higher shift values k yield higher sensitivity to inaccurate data or inaccurate arithmetics.

Technically, the following happens. If $r < 1$ is the fractional part of the mantissa, then a scale $i = 1/r \cdot 10^k$ is computed where k is the shift; fractional parts of i are neglected. The value $1/r$ is computed internally, its error is amplified.

If you still experience stability problems, use \usepackage{fp} in your preamble. The frac style will then automatically employ the higher absolute precision of fp for the computation of $1/r$.

/pgf/number format/precision={⟨*number*⟩}

Sets the desired rounding precision for any display operation. For scientific format, this affects the mantissa.

/pgf/number format/sci precision=⟨*number or empty*⟩ (initially empty)

Sets the desired rounding precision only for sci styles.

Use sci precision={} to restore the initial configuration (which uses the argument provided to precision for all number styles).

2.7.1 Changing Number Format Display Styles

You can change the way how numbers are displayed. For example, if you use the 'fixed' style, the input number is rounded to the desired precision and the current fixed point display style is used to typeset the number. The same is applied to any other format: first, rounding routines are used to get the correct digits, afterwards a display style generates proper TEX-code.

/pgf/number format/set decimal separator={⟨*text*⟩}

Assigns ⟨*text*⟩ as decimal separator for any fixed point number (including the mantissa in sci format).

/pgf/number format/dec sep={⟨*text*⟩}

Just another name for set decimal separator.

/pgf/number format/set thousands separator={⟨*text*⟩}

Assigns ⟨*text*⟩ as thousands separator for any fixed point number (including the mantissa in sci format).

1234.56

```
\pgfkeys{/pgf/number format/.cd,
    fixed,
    fixed zerofill,
    precision=2,
    set thousands separator={}}
\pgfmathprintnumber{1234.56}
```

1234567890.00

```
\pgfkeys{/pgf/number format/.cd,
    fixed,
    fixed zerofill,
    precision=2,
    set thousands separator={}}
\pgfmathprintnumber{1234567890}
```

1.234.567.890.00

```
\pgfkeys{/pgf/number format/.cd,
    fixed,
    fixed zerofill,
    precision=2,
    set thousands separator={.}}
\pgfmathprintnumber{1234567890}
```

1, 234, 567, 890.00

```
\pgfkeys{/pgf/number format/.cd,
    fixed,
    fixed zerofill,
    precision=2,
    set thousands separator={,}}
\pgfmathprintnumber{1234567890}
```

1,234,567,890.00

```
\pgfkeys{/pgf/number format/.cd,
    fixed,
    fixed zerofill,
    precision=2,
    set thousands separator={{{{,}}}}}
\pgfmathprintnumber{1234567890}
```

The last example employs commas and disables the default comma-spacing.

/pgf/number format/1000 sep={⟨*text*⟩}

Just another name for set thousands separator.

/pgf/number format/1000 sep in fractionals={⟨*boolean*⟩} (initially `false`)

Configures whether the fractional part should also be grouped into groups of three digits.

The value `true` will activate the 1000 sep for both integer and fractional parts. The value `false` will activate 1000 sep only for the integer part.

1 234.123 456 7

```
\pgfkeys{/pgf/number format/.cd,
    fixed,
    precision=999,
    set thousands separator={\,},
    1000 sep in fractionals,
    }
\pgfmathprintnumber{1234.1234567}
```

1 234.123 456 700

```
\pgfkeys{/pgf/number format/.cd,
    fixed,fixed zerofill,
    precision=9,
    set thousands separator={\,},
    1000 sep in fractionals,
    }
\pgfmathprintnumber{1234.1234567}
```

/pgf/number format/min exponent for 1000 sep={⟨*number*⟩} (initially 0)

Defines the smalles exponent in scientific notation which is required to draw thousand separators. The exponent is the number of digits minus one, so $\langle number \rangle = 4$ will use thousand separators starting with $1e4 = 10000$.

5 000; 1 000 000

```
\pgfkeys{/pgf/number format/.cd,
    int detect,
    1000 sep={\,},
    min exponent for 1000 sep=0}
\pgfmathprintnumber{5000}; \pgfmathprintnumber{1000000}
```

1000; 5000

```
\pgfkeys{/pgf/number format/.cd,
    int detect,
    1000 sep={\,},
    min exponent for 1000 sep=4}
\pgfmathprintnumber{1000}; \pgfmathprintnumber{5000}
```

10 000; 1 000 000

```
\pgfkeys{/pgf/number format/.cd,
    int detect,
    1000 sep={\,},
    min exponent for 1000 sep=4}
\pgfmathprintnumber{10000}; \pgfmathprintnumber{1000000}
```

A value of `0` disables this feature (negative values are ignored).

`/pgf/number format/use period` (no value)

A predefined style which installs periods '`.`' as decimal separators and commas '`,`' as thousands separators. This style is the default.

12.35

```
\pgfkeys{/pgf/number format/.cd,fixed,precision=2,use period}
\pgfmathprintnumber{12.3456}
```

1,234.56

```
\pgfkeys{/pgf/number format/.cd,fixed,precision=2,use period}
\pgfmathprintnumber{1234.56}
```

`/pgf/number format/use comma` (no value)

A predefined style which installs commas '`,`' as decimal separators and periods '`.`' as thousands separators.

12,35

```
\pgfkeys{/pgf/number format/.cd,fixed,precision=2,use comma}
\pgfmathprintnumber{12.3456}
```

1.234,56

```
\pgfkeys{/pgf/number format/.cd,fixed,precision=2,use comma}
\pgfmathprintnumber{1234.56}
```

`/pgf/number format/skip 0.=`{⟨*boolean*⟩} (initially `false`)

Configures whether numbers like 0.1 shall be typeset as .1 or not.

.56

```
\pgfkeys{/pgf/number format/.cd,
    fixed,
    fixed zerofill,precision=2,
    skip 0.}
\pgfmathprintnumber{0.56}
```

0.56

```
\pgfkeys{/pgf/number format/.cd,
    fixed,
    fixed zerofill,precision=2,
    skip 0.=false}
\pgfmathprintnumber{0.56}
```

`/pgf/number format/showpos=`{⟨*boolean*⟩} (initially `false`)

Enables or disables display of plus signs for non-negative numbers.

+12.35

```
\pgfkeys{/pgf/number format/showpos}
\pgfmathprintnumber{12.345}
```

12.35

```
\pgfkeys{/pgf/number format/showpos=false}
\pgfmathprintnumber{12.345}
```

$+1.23 \cdot 10^{1}$

```
\pgfkeys{/pgf/number format/.cd,showpos,sci}
\pgfmathprintnumber{12.345}
```

`/pgf/number format/print sign=`{⟨*boolean*⟩}

A style which is simply an alias for `showpos=`⟨*boolean*⟩.

/pgf/number format/sci 10e (no value)

Uses $m \cdot 10^e$ for any number displayed in scientific format.

$1.23 \cdot 10^1$

```
\pgfkeys{/pgf/number format/.cd,sci,sci 10e}
\pgfmathprintnumber{12.345}
```

/pgf/number format/sci 10^e (no value)

The same as 'sci 10e'.

/pgf/number format/sci e (no value)

Uses the '1e+0' format which is generated by common scientific tools for any number displayed in scientific format.

1.23e+1

```
\pgfkeys{/pgf/number format/.cd,sci,sci e}
\pgfmathprintnumber{12.345}
```

/pgf/number format/sci E (no value)

The same with an uppercase 'E'.

1.23E+1

```
\pgfkeys{/pgf/number format/.cd,sci,sci E}
\pgfmathprintnumber{12.345}
```

/pgf/number format/sci subscript (no value)

Typesets the exponent as subscript for any number displayed in scientific format. This style requires very little space.

1.23_1

```
\pgfkeys{/pgf/number format/.cd,sci,sci subscript}
\pgfmathprintnumber{12.345}
```

/pgf/number format/sci superscript (no value)

Typesets the exponent as superscript for any number displayed in scientific format. This style requires very little space.

1.23^1

```
\pgfkeys{/pgf/number format/.cd,sci,sci superscript}
\pgfmathprintnumber{12.345}
```

/pgf/number format/sci generic={⟨*keys*⟩}

Allows to define a custom number style for the scientific format. Here, ⟨*keys*⟩ can be one of the following choices (omit the long key prefix):

/pgf/number format/sci generic/mantissa sep={⟨*text*⟩} (initially `empty`)

Provides the separator between the mantissa and the exponent. It might be `\cdot`, for example,

/pgf/number format/sci generic/exponent={⟨*text*⟩} (initially `empty`)

Provides text to format the exponent. The actual exponent is available as argument `#1` (see below).

1.23×10^1; 1.23×10^{-4}

```
\pgfkeys{
    /pgf/number format/.cd,
    sci,
    sci generic={mantissa sep=\times,exponent={10^{#1}}}}
\pgfmathprintnumber{12.345};
\pgfmathprintnumber{0.00012345}
```

The ⟨*keys*⟩ can depend on three parameters, namely on `#1` which is the exponent, `#2` containing the flags entity of the floating point number and `#3` is the (unprocessed and unformatted) mantissa.

Note that `sci generic` is *not* suitable to modify the appearance of fixed point numbers, nor can it be used to format the mantissa (which is typeset like fixed point numbers). Use `dec sep`, `1000 sep` and `print sign` to customize the mantissa.

/pgf/number format/@dec sep mark={⟨*text*⟩}

Will be placed right before the place where a decimal separator belongs to. However, ⟨*text*⟩ will be inserted even if there is no decimal separator. It is intended as place-holder for auxiliary routines to find alignment positions.

This key should never be used to change the decimal separator! Use `dec sep` instead.

/pgf/number format/@sci exponent mark={⟨*text*⟩}

Will be placed right before exponents in scientific notation. It is intended as place-holder for auxiliary routines to find alignment positions.

This key should never be used to change the exponent!

/pgf/number format/assume math mode={⟨*boolean*⟩} (default `true`)

Set this to `true` if you don't want any checks for math mode.

The initial setting installs a `\pgfutilensuremath` around each final number to change to math mode if necessary. Use `assume math mode=true` if you know that math mode is active and you don't want `\pgfutilensuremath`.

/pgf/number format/verbatim (style, no value)

A style which configures the number printer to produce verbatim text output, i.e. it doesn't contain TEX macros.

1.23e1; 1.23e-4; 3.27e6

```
\pgfkeys{
    /pgf/fpu,
    /pgf/number format/.cd,
    sci,
    verbatim}
\pgfmathprintnumber{12.345};
\pgfmathprintnumber{0.00012345};
\pgfmathparse{exp(15)}
\pgfmathprintnumber{\pgfmathresult}
```

The style resets `1000 sep`, `dec sep`, `print sign`, `skip 0.` and sets `assume math mode`. Furthermore, it installs a `sci generic` format for verbatim output of scientific numbers.

However, it will still respect `precision`, `fixed zerofill`, `sci zerofill` and the overall styles `fixed`, `sci`, `int detect` (and their variants). It might be useful if you intend to write output files.

3 From Input Data To Output Tables: Data Processing

The conversion from an unprocessed input table to a final typesetted `tabular` code uses four stages for every cell,

1. Loading the table,
2. Preprocessing,
3. Typesetting,
4. Postprocessing.

The main idea is to select one typesetting algorithm (for example "format my numbers with the configured number style"). This algorithm usually doesn't need to be changed. Fine-tuning can then be done using zero, one or more preprocessors and postprocessors. Preprocessing can mean to select only particular rows or to apply some sort of operation before the typesetting algorithm sees the content. Postprocessing means to apply fine-tuning to the resulting TEX output – for example to deal with empty cells or to insert unit suffixes or modify fonts for single cells.

3.1 Loading the table

This first step to typeset a table involves the obvious input operations. Furthermore, the "new column creation" operations explained in Section 4 are processed at this time. The table data is read (or acquired) as already explained earlier in this manual. Then, if columns are missing, column alias and `create on use` specifications will be processed as part of the loading procedure. See Section 4 for details about column creation.

3.2 Typesetting Cell Content

Typesetting cells means to take their value and "do something". In many cases, this involves number formatting routines. For example, the "raw" input data `12.56` might become `1.26 \cdot 10^1`. The result of this stage is no longer useful for content-based computations. The typesetting step follows the preprocessing step.

/pgfplots/table/assign cell content/.code={⟨...⟩}

Allows to redefine the algorithm which assigns cell contents. The argument `#1` is the (unformatted) contents of the input table.

The resulting output needs to be written to /pgfplots/table/@cell content.

	a	b
data	1	2
	3	4
	5	6
	7	8

```
\begin {tabular}{ccc}%
\toprule &a&b\\\midrule %
\multirow {4}{*}{data}&\pgfutilensuremath {1}&\pgfutilensuremath {2}\\%
&\pgfutilensuremath {3}&\pgfutilensuremath {4}\\%
&\pgfutilensuremath {5}&\pgfutilensuremath {6}\\%
&\pgfutilensuremath {7}&\pgfutilensuremath {8}\\\bottomrule %
\end {tabular}%
```

```
%  An example how to use
%  \usepackage{multirow} and
%  \usepackage{booktabs}:
\pgfplotstabletypeset[
    columns/Z/.style={
        column name={},
        assign cell content/.code={%  use \multirow for Z column:
            \ifnum\pgfplotstablerow=0
                \pgfkeyssetvalue{/pgfplots/table/@cell content}
                    {\multirow{4}{*}{##1}}%
            \else
                \pgfkeyssetvalue{/pgfplots/table/@cell content}{}%
            \fi
        },
    },
    %  use \booktabs as well (compare examples above):
    every head row/.style={before row=\toprule,after row=\midrule},
    every last row/.style={after row=\bottomrule},
    row sep=\\,col sep=&,
    outfile=pgfplotstable.multirow.out,%  write it to file
]{%  here: inline data in tabular format:
    Z    & a & b \\
    data & 1 & 2 \\
         & 3 & 4 \\
         & 5 & 6 \\
         & 7 & 8 \\
}
%  ... and show the generated file:
\lstinputlisting[basicstyle=\footnotesize\ttfamily]{pgfplotstable.multirow.out}
```

The example above uses `\usepackage{multirow}` to format column `Z`. More precisely, it uses `\multirow{4}{*}{data}` for row #0 of column `Z` and the empty string for any other row in column `Z`.

Please note that you may need special attention for `#1={⟨⟩}`, i.e. the empty string. This may happen if a column has less rows than the first column. PgfplotsTable will balance columns automatically in this case, inserting enough empty cells to match the number of rows of the first column.

Please note further that if any column has more entries than the first column, these entries will be skipped and a warning message will be issued into the log file.

This key is evaluated inside of a local TEX group, so any local macro assignments will be cleared afterwards.

/pgfplots/table/numeric type (style, no value)

A style which (re)defines assign cell content back to its original value which assumes numerical data.

It invokes \pgfmathprintnumberto and writes the result into @cell content.

/pgfplots/table/string type (style, no value)

A style which redefines `assign cell content` to simply return the "raw" input data, that means as text column. This assumes input tables with valid LaTeX content (verbatim printing is not supported).

`/pgfplots/table/verb string type` (style, no value)

A style which redefines `assign cell content` to return the "raw" as–is. Thus, it is quite similar to `string type` – but it will return control sequences and (many, not all) special characters without expanding them.

You may need to combine `verb string type` with `special chars`.

`/pgfplots/table/numeric as string type` (style, no value)

A style which redefines `assign cell content` such that it assumes numerical input data. It returns a string literal describing the input number either as integer or in scientific (exponential) notation. In contrast to `numeric type`, it does not apply number formatting.

`/pgfplots/table/date type={`⟨*date format*⟩`}`

A style which expects ISO dates of the form `YYYY-MM-DD` in each cell and produces pretty-printed strings on output. The output format is given as ⟨*date format*⟩. Inside of ⟨*date format*⟩, several macros which are explained below can be used.

date	account1	date	account1
2008-01-03	60	January 2008	60
2008-02-06	120	February 2008	120
2008-03-15	−10	March 2008	−10
2008-04-01	1,800	April 2008	1,800
2008-05-20	2,300	May 2008	2,300
2008-06-15	800	June 2008	800

```
%  Requires
%  \usepackage{pgfcalendar}
\pgfplotstableset{columns={date,account1}}

%  plotdata/accounts.dat contains:
%
%  date              account1  account2  account3
%  2008-01-03        60        1200      400
%  2008-02-06        120       1600      410
%  2008-03-15        -10       1600      410
%  2008-04-01        1800      500       410
%  2008-05-20        2300      500       410
%  2008-06-15        800       1920      410

%  Show the contents in 'string type':
\pgfplotstabletypeset[
    columns/date/.style={string type}
]{plotdata/accounts.dat}
\hspace{1cm}
%  Show the contents in 'date type':
\pgfplotstabletypeset[
    columns/date/.style={date type={\monthname\ \year}}
]{plotdata/accounts.dat}
```

This style **requires** to load the PGF **calendar package**:

```
\usepackage{pgfcalendar}
```

`\year`

Inside of ⟨*date format*⟩, this macro expands to the year as a number (like `2008`).

`\month`

Inside of ⟨*date format*⟩, this macro expands to the month as a number, starting with 1 (like `1`).

`\monthname`

Inside of ⟨*date format*⟩, this macro expands to the month's name as set in the current language (like `January`). See below for how to change the language.

`\monthshortname`

Inside of ⟨*date format*⟩, this macro expands to the month's short name as set in the current language (like `Jan`). See below for how to change the language.

`\day`

Inside of ⟨*date format*⟩, this macro expands to the day as number (like `31`).

`\weekday`

Inside of ⟨*date format*⟩, this macro expands to the weekday number (0 for Monday, 1 for Tuesday etc.).

`\weekdayname`

Inside of ⟨*date format*⟩, this macro expands to the weekday's name in the current language (like `Wednesday`). See below for how to change the language.

`\weekdayshortname`

Inside of ⟨*date format*⟩, this macro expands to the weekday's short name in the current language (like `Wed`). See below for how to change the language.

Changing the language for dates

The date feature is implemented using the PGF calendar module. This module employs the package `translator` (if it is loaded). I don't have more details yet, sorry. Please refer to [2] for more details.

3.3 Preprocessing Cell Content

The preprocessing step allows to change cell contents *before* any typesetting routine (like number formatting) has been applied. Thus, if tables contain numerical data, it is possible to apply math operations at this stage. Furthermore, cells can be erased depending on their numerical value. The preprocessing step follows after the data acquisition step ("loading step"). This means in particular that you can create (or copy) columns and apply operations on them.

`/pgfplots/table/preproc cell content/.code={⟨...⟩}`

Allows to *modify* the contents of cells *before* `assign cell content` is called.

The semantics is as follows: before the preprocessor, `@cell content` contains the raw input data (or, maybe, the result of another preprocessor call). After the preprocessor, `@cell content` is filled with a – possibly modified – value. The resulting value is then used as input to `assign cell content`.

In the default settings, `assign cell content` expects numerical input. So, the preprocessor is expected to produce numerical output.

It is possible to provide multiple preprocessor directives using `/.append code` or `/.append style` key handlers.

In case you don't want (or need) stackable preprocessors, you can also use '`#1`' to get the raw input datum as it is found in the file. Furthermore, the key `@unprocessed cell content` will also contain the raw input datum.

`/pgfplots/table/string replace={⟨`*cell match*`⟩}{⟨`*cell replacement*`⟩}`

Appends code to the current `preproc cell content` value which replaces any cell with exact match ⟨*cell match*⟩ by ⟨*cell replacement*⟩. No expansion is performed during this step; ⟨*cell match*⟩ must match literally.

level	dof	level	dof
1	4	1	4
2	16	2	16
3	64	3	64
4	256	4	−42
5	1,024	5	1,024
6	4,096	6	4,096
7	16,384	7	16,384
8	65,536	8	65,536
9	$2.62 \cdot 10^5$	9	$2.62 \cdot 10^5$
10	$1.05 \cdot 10^6$	10	$1.05 \cdot 10^6$

```
\pgfplotstabletypeset[columns={level,dof}]
    {pgfplotstable.example1.dat}


\pgfplotstabletypeset[
    columns={level,dof},
    columns/level/.style={string replace={A}{B}}, %  does nothing because there is no cell 'A'
    columns/dof/.style={string replace={256}{-42}}]  %  replace cell '256' with '-42'
    {pgfplotstable.example1.dat}
```

See the `string replace*` method for sub–string replacement.

/pgfplots/table/**string replace***={⟨*pattern*⟩}{⟨*replacement*⟩}

Appends code to the current `preproc cell content` value which replaces every occurence of ⟨*pattern*⟩ with ⟨*replacement*⟩. No expansion is performed during this step; ⟨*pattern*⟩ must match literally.

colA	colB	colC
11	16	13
61	66	63

```
\pgfplotstabletypeset[
    string replace*={2}{6},
    col sep=&,row sep=\\]{
  colA & colB & colC \\
  11   & 12   & 13   \\
  21   & 22   & 23   \\
}
```

/pgfplots/table/**clear infinite** (style, no value)

Appends code to the current `preproc cell content` value which replaces every infinite number with the empty string. This clears any cells with $\pm\infty$ and NaN.

/pgfplots/table/**preproc/expr**={⟨*math expression*⟩}

Appends code to the current `preproc cell content` value which evaluates ⟨*math expression*⟩ for every cell. Arithmetics are carried out in floating point.

Inside of ⟨*math expression*⟩, use one of the following expressions to get the current cell's value.

- The string '`##1`' expands to the cell's content as it has been found in the input file, ignoring preceeding preprocessors.

  This is usually enough.
- The command `\thisrow{`⟨*the currently processed column name*⟩`}` expands to the current cell's content. This will also include the results of preceeding preprocessors.

  Note that `\thisrow{}` in this context (inside of the preprocessor) is not as powerful as in the context of column creation routines: the argument must match exactly the name of the currently processed column name. You can also use the shorthand

  `\thisrow{\pgfplotstablecolname}`.
- The command `\pgfkeysvalueof{/pgfplots/table/@cell content}` is the same.

$2 \cdot \text{level} + 4$
6
8
10
12
14
16
18
20
22
24

```
\pgfplotstabletypeset[
    columns={level},
    columns/level/.style={
        column name={$2\cdot \text{level}+4$},
        preproc/expr={2*##1 + 4}
    }
]
    {pgfplotstable.example1.dat}
```

Empty cells won't be processed, assuming that a math expression with an "empty number" will fail.

Note that there is also an `create col/expr` which is more powerful than `preproc/expr`.

/pgfplots/table/multiply by={⟨*real number*⟩}

Appends code to the current `preproc cell content` value which multiplies every cell by ⟨*real number*⟩. Arithmetics are carried out in floating point.

/pgfplots/table/divide by={⟨*real number*⟩}

Appends code to the current `preproc cell content` value which divides every cell by ⟨*real number*⟩. Arithmetics are carried out in floating point.

/pgfplots/table/sqrt (style, no value)

Appends code to the current `preproc cell content` value which takes the square root $\sqrt{\cdot}$ of every non-empty cell. Arithmetics are carried out in floating point.

The following example copies the column `error1` and applies `sqrt` to the copy.

ϵ	$\sqrt{\epsilon}$
$2.500 \cdot 10^{-1}$	$5.000 \cdot 10^{-1}$
$6.250 \cdot 10^{-2}$	$2.500 \cdot 10^{-1}$
$1.563 \cdot 10^{-2}$	$1.250 \cdot 10^{-1}$
$3.906 \cdot 10^{-3}$	$6.250 \cdot 10^{-2}$
$9.766 \cdot 10^{-4}$	$3.125 \cdot 10^{-2}$
$2.441 \cdot 10^{-4}$	$1.562 \cdot 10^{-2}$
$6.104 \cdot 10^{-5}$	$7.813 \cdot 10^{-3}$
$1.526 \cdot 10^{-5}$	$3.906 \cdot 10^{-3}$
$3.815 \cdot 10^{-6}$	$1.953 \cdot 10^{-3}$
$9.537 \cdot 10^{-7}$	$9.766 \cdot 10^{-4}$

```
\pgfplotstableset{
    columns={error1,sqrterror1},
    create on use/sqrterror1/.style={create col/copy=error1},
    columns/error1/.style={column name=$\epsilon$},
    columns/sqrterror1/.style={sqrt,column name=$\sqrt \epsilon$},
    sci,sci 10e,precision=3,sci zerofill
}
\pgfplotstabletypeset{pgfplotstable.example1.dat}
```

Please take a look at Section 4 for details about `create on use`.

/pgfplots/table/multiply -1 (style, no value)

Appends code to current `preproc cell content` value which multiplies every cell by -1. This style does the same job as `multiply by=-1`, it is just faster because only the sign changes.

dof	error2	slopes2	dof	error2	slopes2
4	$7.58 \cdot 10^{-1}$	–	4	$7.58 \cdot 10^{-1}$	–
16	$5.00 \cdot 10^{-1}$	−0.3	16	$5.00 \cdot 10^{-1}$	0.3
64	$2.87 \cdot 10^{-1}$	−0.4	64	$2.87 \cdot 10^{-1}$	0.4
256	$1.44 \cdot 10^{-1}$	−0.5	256	$1.44 \cdot 10^{-1}$	0.5
1,024	$4.42 \cdot 10^{-2}$	−0.85	1,024	$4.42 \cdot 10^{-2}$	0.85
4,096	$1.70 \cdot 10^{-2}$	−0.69	4,096	$1.70 \cdot 10^{-2}$	0.69
16,384	$8.20 \cdot 10^{-3}$	−0.52	16,384	$8.20 \cdot 10^{-3}$	0.52
65,536	$3.91 \cdot 10^{-3}$	−0.54	65,536	$3.91 \cdot 10^{-3}$	0.54
$2.62 \cdot 10^{5}$	$1.95 \cdot 10^{-3}$	−0.5	$2.62 \cdot 10^{5}$	$1.95 \cdot 10^{-3}$	0.5
$1.05 \cdot 10^{6}$	$9.77 \cdot 10^{-4}$	−0.5	$1.05 \cdot 10^{6}$	$9.77 \cdot 10^{-4}$	0.5

```
\pgfplotstableset{
    columns={dof,error2,slopes2},
    columns/error2/.style={sci,sci zerofill},
    columns/slopes2/.style={dec sep align,empty cells with={\ensuremath{-}}},
    create on use/slopes2/.style=
        {create col/gradient loglog={dof}{error2}}}

\pgfplotstabletypeset{pgfplotstable.example1.dat}

\pgfplotstabletypeset[columns/slopes2/.append style={multiply -1}]
    {pgfplotstable.example1.dat}
```

/pgfplots/table/row predicate/.code={⟨...⟩}

A boolean predicate which allows to select particular rows of the input table, based on the current row's index. The argument #1 contains the current row's index (starting with 0, not counting comment lines or column names).

The return value is assigned to the TEX-if \ifpgfplotstableuserow. If the boolean is not changed, the return value is true.

level	dof	error1	error2	info	grad(log(dof),log(error2))	quot(error1)
1	4	0.25	0.76	48	0	0
2	16	$6.25 \cdot 10^{-2}$	0.5	25	−0.3	4
3	64	$1.56 \cdot 10^{-2}$	0.29	41	−0.4	4
4	256	$3.91 \cdot 10^{-3}$	0.14	8	−0.5	4
5	1,024	$9.77 \cdot 10^{-4}$	$4.42 \cdot 10^{-2}$	22	−0.85	4
9	$2.62 \cdot 10^{5}$	$3.81 \cdot 10^{-6}$	$1.95 \cdot 10^{-3}$	33	−0.5	4
10	$1.05 \cdot 10^{6}$	$9.54 \cdot 10^{-7}$	$9.77 \cdot 10^{-4}$	2	−0.5	4

```
%  requires \usepackage{booktabs}
\pgfplotstabletypeset[
    every head row/.style={
        before row=\toprule,after row=\midrule},
    every last row/.style={
        after row=\bottomrule},
    row predicate/.code={%
        \ifnum#1>4\relax
            \ifnum#1<8\relax
                \pgfplotstableuserowfalse
            \fi
        \fi}
]
    {pgfplotstable.example1.dat}
```

Please note that row predicate is applied *before* any other option which affects row (or column) appearance. It is evaluated before assign cell content. One of the consequences is that even/odd row styles refer to those rows for which the predicate returns true. In fact, you can use row predicate to truncate the complete table before it has actually been processed.

During row predicate, the macro \pgfplotstablerows contains the total number of *input* rows.

Furthermore, row predicate applies only to the typeset routines, not the read methods. If you want to plot only selected table entries with \addplot table, use the PGFPLOTS coordinate filter options.

/pgfplots/table/skip rows between index={⟨*begin*⟩}{⟨*end*⟩}

A style which appends a row predicate which discards selected rows. The selection is done by index where indexing starts with 0. Every row with index $\langle begin \rangle \leq i < \langle end \rangle$ will be skipped.

level	dof	error1	error2	info	grad(log(dof),log(error2))	quot(error1)
1	4	0.25	0.76	48	0	0
2	16	$6.25 \cdot 10^{-2}$	0.5	25	−0.3	4
5	1,024	$9.77 \cdot 10^{-4}$	$4.42 \cdot 10^{-2}$	22	−0.85	4
6	4,096	$2.44 \cdot 10^{-4}$	$1.7 \cdot 10^{-2}$	46	−0.69	4
7	16,384	$6.1 \cdot 10^{-5}$	$8.2 \cdot 10^{-3}$	40	−0.52	4
10	$1.05 \cdot 10^{6}$	$9.54 \cdot 10^{-7}$	$9.77 \cdot 10^{-4}$	2	−0.5	4

```
%  requires \usepackage{booktabs}
\pgfplotstabletypeset[
    every head row/.style={
        before row=\toprule,after row=\midrule},
    every last row/.style={
        after row=\bottomrule},
    skip rows between index={2}{4},
    skip rows between index={7}{9}
]
    {pgfplotstable.example1.dat}
```

/pgfplots/table/select equal part entry of={⟨*part no*⟩}{⟨*part count*⟩}

A style which overwrites row predicate with a subset selection predicate. The idea is to split the current column into ⟨*part count*⟩ equally sized parts and select only ⟨*part no*⟩.

This can be used to simulate multicolumn tables.

A	B
A1	B1
A2	B2
A3	B3
A4	B4
A5	B5
A6	B6
A7	B7
A8	B8
A9	B9
A10	B10
A11	B11
A12	B12
A13	B13
A14	B14
A15	B15
A16	B16
A17	B17
A18	B18
A19	B19
A20	B20
A21	B21

A	B	A	B
A1	B1	A12	B12
A2	B2	A13	B13
A3	B3	A14	B14
A4	B4	A15	B15
A5	B5	A16	B16
A6	B6	A17	B17
A7	B7	A18	B18
A8	B8	A19	B19
A9	B9	A20	B20
A10	B10	A21	B21
A11	B11		

```
%  requires \usepackage{booktabs}
\pgfplotstableset{
    every head row/.style={before row=\toprule,after row=\midrule},
    every last row/.style={after row=\bottomrule}}

\pgfplotstabletypeset[string type]{pgfplotstable.example2.dat}%
~
\pgfplotstabletypeset[
    columns={A,B,A,B},
    display columns/0/.style={select equal part entry of={0}{2},string type},%  first part of 'A'
    display columns/1/.style={select equal part entry of={0}{2},string type},%  first part of 'B'
    display columns/2/.style={select equal part entry of={1}{2},string type},%  second part of 'A'
    display columns/3/.style={select equal part entry of={1}{2},string type},%  second part of 'B'
]
    {pgfplotstable.example2.dat}
```

The example above shows the original file as-is on the left side. The right side shows columns A,B,A,B – but only half of the elements are shown, selected by indices #0 or #1 of #2. The parts are equally large, up to a remainder.

If the available number of rows is not divisible by ⟨*part count*⟩, the remaining entries are distributed equally among the first parts.

/pgfplots/table/unique={⟨*column name*⟩}

A style which appends a `row predicate` which suppresses successive occurances of the same elements in ⟨*column name*⟩. For example, if ⟨*column name*⟩ contains `1,1,3,5,5,6,5,0`, the application of `unique` results in `1,3,5,6,5,0` (the last `5` is kept – it is not directly preceded by another `5`).

The algorithm uses string token comparison to find multiple occurance[7].

The argument ⟨*column name*⟩ can be a column name, index, alias, or `create on use` specification (the latter one must not depend on other `create on use` statements). It is not necessary to provide a ⟨*column name*⟩ which is part of the output.

However, it *is* necessary that the `unique` predicate can be evaluated for all columns, starting with the first one. That means it is an error to provide `unique` somewhere deep in column–specific styles.

3.4 Postprocessing Cell Content

The postprocessing step is applied after the typesetting stage, that means it can't access the original input data. However, it can apply final formatting instructions which are not content-based.

/pgfplots/table/postproc cell content/.code={⟨...⟩}

Allows to *modify* assigned cell content *after* it has been assigned, possibly content-dependent. Ideas could be to draw negative numbers in red, typeset single entries in bold face or insert replacement text.

This key is evaluated *after* `assign cell content`. Its semantics is to modify an existing `@cell content` value.

There may be more than one `postproc cell content` command, if you use `/.append code` or `/.append style` to define them:

dof	info
4	**48€**
16	**25€**
64	**41€**
256	**8€**
1,024	**22€**
4,096	**46€**
16,384	**40€**
65,536	**48€**
$2.62 \cdot 10^5$	**33€**
$1.05 \cdot 10^6$	**2€**

```
%  requires \usepackage{eurosym}
\pgfplotstabletypeset[
    column type=r,
    columns={dof,info},
    columns/info/.style={
        %  stupid example for multiple postprocessors:
        postproc cell content/.append style={
            /pgfplots/table/@cell content/.add={$\bf}{$},
        },
        postproc cell content/.append style={
            /pgfplots/table/@cell content/.add={}{\EUR{}},
        }
    }]
    {pgfplotstable.example1.dat}
```

[7] To be more precise, the comparison is done using `\ifx`, i.e. cell contents won't be expanded. Only the tokens as they are seen in the input table will be used.

The code above modifies `@cell content` in two steps. The net effect is to prepend "`$\bf `" and to append "`$ \EUR`". It should be noted that `pgfkeys` handles `/.style` and `/.code` in (basically) the same way – both are simple code keys and can be used as such. You can combine both with `/.append style` and `/.append code`. Please refer to [2, section about pgfkeys] for details.

As in `assign cell content`, the code can evaluate helper macros like `\pgfplotstablerow` to change only particular entries. Furthermore, the postprocessor may depend on the unprocessed cell input (as it has been found in the input file or produced by the loading procedure) and/or the preprocessed cell value. These values are available as

- the key `@unprocessed cell content` which stores the raw input,
- the key `@cell content after rowcol styles` which stores the value of `@cell content` after evaluating cell–specific styles,
- the key `@preprocessed cell content` which stores the result of the preprocessor,
- the key `@cell content` which contains the result of the typesetting routine,
- the shorthand '`#1`' which is also the unprocessed input argument as it has been found in the input table.

Remember that you can access the key values using

```
\pgfkeysvalueof{/pgfplots/table/@preprocessed cell content}
```

at any time.

This allows complete context-based formatting options. Please remember that empty strings may appear due to column balancing – introduce special treatment if necessary.

There is one special case which occurs if `@cell content` itsself contains the cell separation character '`&`'. In this case, `postproc cell content` is invoked *separately* for each part before and after the ampersand and the ampersand is inserted afterwards. This allows compatibility with special styles which create artificial columns in the output (which is allowed, see `dec sep align`). To allow separate treatment of each part, you can use the macro `\pgfplotstablepartno`. It is defined only during the evaluation of `postproc cell content` and it evaluates to the current part index (starting with 0). If there is no ampersand in your text, the value will always be 0.

This key is evaluated inside of a local TEX group, so any local macro assignments will be cleared afterwards.

The following example can be used to insert a dash, −, in a slope column:

dof	error1	slopes1
4	$2.50 \cdot 10^{-1}$	–
16	$6.25 \cdot 10^{-2}$	−1
64	$1.56 \cdot 10^{-2}$	−1
256	$3.91 \cdot 10^{-3}$	−1
1,024	$9.77 \cdot 10^{-4}$	−1
4,096	$2.44 \cdot 10^{-4}$	−1
16,384	$6.10 \cdot 10^{-5}$	−1
65,536	$1.53 \cdot 10^{-5}$	−1
$2.62 \cdot 10^{5}$	$3.81 \cdot 10^{-6}$	−1
$1.05 \cdot 10^{6}$	$9.54 \cdot 10^{-7}$	−1

```
\pgfplotstableset{
    create on use/slopes1/.style=
        {create col/gradient loglog={dof}{error1}}}

\pgfplotstabletypeset[
    columns={dof,error1,slopes1},
    columns/error1/.style={sci,sci zerofill},
    columns/slopes1/.style={
        postproc cell content/.append code={%
            \ifnum\pgfplotstablerow=0
                \pgfkeyssetvalue{/pgfplots/table/@cell content}{\ensuremath{-}}%
            \fi
        }%
    }]
    {pgfplotstable.example1.dat}
```

Since this may be useful in a more general context, it is available as `empty cells with` style.

/pgfplots/table/**empty cells with**={⟨*replacement*⟩}

Appends code to `postproc cell content` which replaces any empty cell with ⟨*replacement*⟩.

If `dec sep align` is active, the replacement will be inserted only for the part before the decimal separator.

/pgfplots/table/**set content**={⟨*content*⟩}

A style which redefines `postproc cell content` to always return the value ⟨*content*⟩.

/pgfplots/table/**fonts by sign**={⟨*TEX code for positive*⟩}{⟨*TEX code for negative*⟩}

Appends code to `postproc cell content` which allows to set fonts for positive and negative numbers.

The arguments ⟨*TEX code for positive*⟩ and ⟨*TEX code for negative*⟩ are inserted right before the typesetted cell content. It is permissible to use both ways to change LaTeX fonts: the `\textbf{`⟨*argument*⟩`}` or the `{\bfseries {`⟨*argument*⟩`}}` way.

date	account1
January 2008	60
February 2008	120
March 2008	−10
April 2008	1,800
May 2008	2,300
June 2008	800

```
%  Requires
%  \usepackage{pgfcalendar}

%  plotdata/accounts.dat contains:
%
%  date              account1  account2  account3
%  2008-01-03        60        1200      400
%  2008-02-06        120       1600      410
%  2008-03-15        -10       1600      410
%  2008-04-01        1800      500       410
%  2008-05-20        2300      500       410
%  2008-06-15        800       1920      410

\pgfplotstabletypeset[
    columns={date,account1},
    column type=r,
    columns/date/.style={date type={\monthname\ \year}},
    columns/account1/.style={fonts by sign={}{\color{red}}}
]
    {plotdata/accounts.dat}
```

In fact, the arguments for this style don't need to be font changes. The style `fonts by sign` inserts several braces and the matching argument into `@cell content`. To be more precise, it results in

`{`⟨*TEX code for negative*⟩`{`⟨*cell value*⟩`}}` for negative numbers and

`{`⟨*TEX code for positive*⟩`{`⟨*cell value*⟩`}}` for all other numbers.

4 Generating Data in New Tables or Columns

It is possible to create new tables from scratch or to change tables after they have been loaded from disk.

4.1 Creating New Tables From Scratch

`\pgfplotstablenew[`⟨*options*⟩`]{`⟨*row count*⟩`}{`⟨*\table*⟩`}`
`\pgfplotstablenew*[`⟨*options*⟩`]{`⟨*row count*⟩`}{`⟨*\table*⟩`}`

Creates a new table from scratch.

The new table will contain all columns listed in the `columns` key. For `\pgfplotstablenew`, the `columns` key needs to be provided in `[`⟨*options*⟩`]`. For `\pgfplotstablenew*`, the current value of `columns` is used, no matter where and when it has been set.

Furthermore, there must be `create on use` statements (see the next subsection) for every column which shall be generated[8]. Columns are generated independently, in the order of appearance in `columns`. As soon as a column is complete, it can be accessed using any of the basic level access mechanisms. Thus, you can build columns which depend on each other.

The table will contain exactly ⟨*row count*⟩ rows. If ⟨*row count*⟩ is an `\pgfplotstablegetrowsof` statement, that statement will be executed and the resulting number of rows be used. Otherwise, ⟨*row count*⟩ will be evaluated as number.

new
4
5
6
7
8
9
10
—
—
—
—

```
%  this key setting could be provided in the document's preamble:
\pgfplotstableset{
    %  define how the 'new' column shall be filled:
    create on use/new/.style={create col/set list={4,5,6,7,...,10}}}
%  create a new table with 11 rows and column 'new':
\pgfplotstablenew[columns={new}]{11}\loadedtable
%  show it:
\pgfplotstabletypeset[empty cells with={---}]\loadedtable
```

new
$1.31 \cdot 10^{12}$
$2.09 \cdot 10^{13}$
$3.56 \cdot 10^{14}$
$6.4 \cdot 10^{15}$
$1.22 \cdot 10^{17}$
$2.43 \cdot 10^{18}$
$5.11 \cdot 10^{19}$
$1.12 \cdot 10^{21}$
$2.59 \cdot 10^{22}$
$6.2 \cdot 10^{23}$
$1.55 \cdot 10^{25}$

```
%  create a new table with 11 rows and column 'new':
\pgfplotstablenew[
    %  define how the 'new' column shall be filled:
    create on use/new/.style={create col/expr={factorial(15+\pgfplotstablerow)}},
    columns={new}]
    {11}
    \loadedtable
%  show it:
\pgfplotstabletypeset\loadedtable
```

`\pgfplotstablevertcat{`⟨*\table1*⟩`}{`⟨*\table2 or filename*⟩`}`

Appends the contents of ⟨*\table2*⟩ to ⟨*\table1*⟩ ("vertical concatenation"). To be more precise, only columns which exist already in ⟨*\table1*⟩ will be appended and every column which exists in ⟨*\table1*⟩ must exist in ⟨*\table2*⟩ (or there must be `alias` or `create on use` specifications to generate them).

If the second argument is a file name, that file will be loaded from disk.

If ⟨*\table1*⟩ does not exist, ⟨*\table2*⟩ will be copied to ⟨*\table1*⟩.

```
\pgfplotstablevertcat{\output}{datafile1} %  loads 'datafile1' -> '\output'
\pgfplotstablevertcat{\output}{datafile2} %  appends rows of datafile2
\pgfplotstablevertcat{\output}{datafile3} %  appends rows of datafile3
```

Remark: The output table ⟨*\table1*⟩ will be defined in the current TEX scope and it will be erased afterwards. The current TEX scope is delimited by an extra set of curly braces. However, every LATEX environment and, unfortunately, the Ti*k*Z `\foreach` statement as well, introduce TEX scopes.

PGFPLOTS has some some loop statements which do not introduce extra scopes. For example,

```
\pgfplotsforeachungrouped \i in {1,2,...,10} {%
    \pgfplotstablevertcat{\output}{datafile\i} %  appends 'datafile\i' -> '\output'
}%
```

These looping macros are explained in the manual of PGFPLOTS, reference section "Miscellaneous Commands"

[8] Currently, you need to provide at least one column: the implementation gets confused for completely empty tables. If you do not provide any column name, a dummy column will be created.

\pgfplotstableclear{⟨\table⟩}

Clears a table. Note that it is much more reliable to introduce extra curly braces '{ ... }' around table operations – these braces define the scope of a variable (including tables).

4.2 Creating New Columns From Existing Ones

\pgfplotstablecreatecol[⟨options⟩]{⟨new col name⟩}{⟨\table⟩}

Creates a new column named ⟨*new col name*⟩ and appends it to an already existing table ⟨\table⟩.

End users probably don't need to use \pgfplotstablecreatecol directly at all – there is the high–level framework create on use which invokes it internally and can be used with simple key–value assignments (see below). However, this documentation explains how to use values of existing columns to fill new cells.

This command offers a flexible framework to generate new columns. It has been designed to create new columns using the already existing values – for example using logical or numerical methods to combine existing values. It provides fast access to a row's value, the previous row's value and the next row's value.

The following documentation is for everyone who wants to *write* specialized columns. It is not particularly difficult; it is just technical and it requires some knowledge of pgfkeys. If you don't like it, you can resort to predefined column generation styles – and enable those styles in ⟨*options*⟩.

The column entries will be created using the command key create col/assign. It will be invoked for every row of the table. It is supposed to assign contents to create col/next content. During the evaluation, the macro \thisrow{⟨*col name*⟩} expands to the current row's value of the column identified by ⟨*col name*⟩. Furthermore, \nextrow{⟨*col name*⟩} expands to the *next* row's value of the designated column and \prevrow{⟨*col name*⟩} expands to the value of the *previous* row.

So, the idea is to simply redefine the command key create col/assign in such a way that it fills new cells as desired.

Two special assign routines are available for the first and last row: The contents for the *last* row is computed with create col/assign last. Its semantics is the same. The contents for the *first* row is computed with create col/assign first to simplify special cases here. These first and last commands are optional, their default is to invoke the normal assign routine.

The evaluation of the assign keys is done in local TEX groups (i.e. any local definitions will be cleared afterwards).

The following macros are useful during cell assignments:

1. \prevrow{⟨*col name*⟩} / \getprevrow{⟨*col name*⟩}{⟨\macro⟩}

   These two routines return the value stored in the *previous* row of the designated column ⟨*col name*⟩. The get routine stores it into ⟨\macro⟩.

   The argument ⟨*col name*⟩ has to denote either an existing column name or one for which an alias/⟨*col name*⟩ exists.

2. \thisrow{⟨*col name*⟩} / \getthisrow{⟨*col name*⟩}{⟨\macro⟩}

   These two routines return the *current* row's value stored in the designated column. The get routine stores it into ⟨\macro⟩.

   The argument ⟨*col name*⟩ has to denote either an existing column name or one for which an alias/⟨*col name*⟩ exists.

3. \nextrow{⟨*col name*⟩} / \getnextrow{⟨*col name*⟩}{⟨\macro⟩}

   These two routines return the *next* row's value.

   The argument ⟨*col name*⟩ has to denote either an existing column name or one for which an alias/⟨*col name*⟩ exists.

4. \pgfplotstablerow and \pgfplotstablerows which contain the current row's index and the total number of rows, respectively. See page 14 for details.

5. \pgfmathaccuma and \pgfmathaccumb can be used to transport intermediate results. Both maintain their value from one column assignment to the next. All other local variables will be deleted after leaving the assignment routines. The initial value is the empty string for both of them unless they are already initialized by column creation styles.

6. \pgfplotstablename a macro containing the name of the currently processed table (i.e. it contains the second argument of \pgfplotstablecreatecol).
7. commands which are valid throughout every part of this package, for example \pgfplotstablerow to get the current row index or \pgfplotstablerows to get the total number of rows.

The ⟨*col name*⟩ is expected to be a *physical* column name, no alias or column index is allowed (unless column indices and column names are the same).

The following example takes our well-known input table and creates a copy of the `level` column. Furthermore, it produces a lot of output to show the available macros. Finally, it uses `\pgfkeyslet` to assign the contents of the resulting `\entry` to `next content`.

level	new
1	thisrow=1; nextrow=2. (#0/10)
2	thisrow=2; nextrow=3. (#1/10)
3	thisrow=3; nextrow=4. (#2/10)
4	thisrow=4; nextrow=5. (#3/10)
5	thisrow=5; nextrow=6. (#4/10)
6	thisrow=6; nextrow=7. (#5/10)
7	thisrow=7; nextrow=8. (#6/10)
8	thisrow=8; nextrow=9. (#7/10)
9	thisrow=9; nextrow=10. (#8/10)
10	thisrow=10; nextrow=. (#9/10)

```
\pgfplotstableread{pgfplotstable.example1.dat}\loadedtable
\pgfplotstablecreatecol[
    create col/assign/.code={%
        \getthisrow{level}\entry
        \getnextrow{level}\nextentry
        \edef\entry{thisrow=\entry; nextrow=\nextentry.
            (\#\pgfplotstablerow/\pgfplotstablerows)}%
        \pgfkeyslet{/pgfplots/table/create col/next content}\entry
    }]
    {new}\loadedtable

\pgfplotstabletypeset[
    column type=l,
    columns={level,new},
    columns/new/.style={string type}
]\loadedtable
```

There is one more specialty: you can use `columns={`⟨*column list*⟩`}` to reduce the runtime complexity of this command. This works only if the `columns` key is provided directly into ⟨*options*⟩. In this case \thisrow and its variants are only defined for those columns listed in the `columns` value.

Limitations. Currently, you can only access three values of one column at a time: the current row, the previous row and the next row. Access to arbitrary indices is not (yet) supported.

Remark: If you'd like to create a table from scratch using this command (or the related `create on use` simplification), take a look at \pgfplotstablenew.

The default implementation of `assign` is to produce empty strings. The default implementation of `assign last` is to invoke `assign`, so in case you never really use the next row's value, you won't need to touch `assign last`. The same holds for `assign first`.

/pgfplots/table/create on use/⟨*col name*⟩/.style={⟨*create options*⟩}

Allows "lazy creation" of the column ⟨*col name*⟩. Whenever the column ⟨*col name*⟩ is queried by name, for example in an \pgfplotstabletypeset command, and such a column does not exist already, it is created on-the-fly.

error1	quot1
$2.50 \cdot 10^{-1}$	
$6.25 \cdot 10^{-2}$	4
$1.56 \cdot 10^{-2}$	4
$3.91 \cdot 10^{-3}$	4
$9.77 \cdot 10^{-4}$	4
$2.44 \cdot 10^{-4}$	4
$6.10 \cdot 10^{-5}$	4
$1.53 \cdot 10^{-5}$	4
$3.81 \cdot 10^{-6}$	4
$9.54 \cdot 10^{-7}$	4

```
% requires \usepackage{array}
\pgfplotstableset{% could be used in preamble
    create on use/quot1/.style=
        {create col/quotient={error1}}}

\pgfplotstabletypeset[
    columns={error1,quot1},
    columns/error1/.style={sci,sci zerofill},
    columns/quot1/.style={dec sep align}]
{pgfplotstable.example1.dat}
```

The example above queries `quot1` which does not yet exist in the input file. Therefore, it is checked whether a `create on use` style for `quot1` exists. This is the case, so it is used to create the missing column. The `create col/quotient` key is discussed below; it computes quotients of successive rows in column `error1`.

A `create on use` specification is translated into

`\pgfplotstablecreatecol`[⟨*create options*⟩]{⟨*col name*⟩}{⟨*the table*⟩},

or, equivalently, into

`\pgfplotstablecreatecol`[create on use/⟨*col name*⟩]{⟨*col name*⟩}{⟨*the table*⟩}.

This feature allows some laziness, because you can omit the lengthy table modifications. However, laziness may cost something: in the example above, the generated column will be *lost* after returning from `\pgfplotstabletypeset`.

The `create on use` has higher priority than `alias`.

In case ⟨*col name*⟩ contains characters which are required for key settings, you need to use braces around it: "`create on use/{name=wi/th,special}/.style={...}`".

More examples for `create on use` are shown below while discussing the available column creation styles.

Note that `create on use` is also available within PGFPLOTS, in `\addplot table` when used together with the `read completely` key.

4.3 Predefined Column Generation Methods

The following keys can be used in both `\pgfplotstablecreatecol` and the easier `create on use` frameworks.

4.3.1 Acquiring Data Somewhere

`/pgfplots/table/create col/set={`⟨*value*⟩`}`

A style for use in column creation context which creates a new column and writes ⟨*value*⟩ into each new cell. The value is written as string (verbatim).

level	my new col
1	–empty–
2	–empty–
3	–empty–
4	–empty–
5	–empty–
6	–empty–
7	–empty–
8	–empty–
9	–empty–
10	–empty–

```
\pgfplotstableset{
    create on use/my new col/.style={create col/set={--empty--}},
    columns/my new col/.style={string type}
}

\pgfplotstabletypeset[
    columns={level,my new col},
]{pgfplotstable.example1.dat}
```

`/pgfplots/table/create col/set list={`⟨*comma-separated-list*⟩`}`

A style for use in column creation context which creates a new column consisting of the entries in ⟨*comma-separated-list*⟩. The value is written as string (verbatim).

The ⟨*comma-separated-list*⟩ is processed via Ti*k*Z's `\foreach` command, that means you can use ... expressions to provide number (or character) ranges.

level	my new col
1	A
2	B
3	C
4	4
5	50
6	55
7	60
8	65
9	70
10	75

```
\pgfplotstableset{
    create on use/my new col/.style={
        create col/set list={A,B,C,4,50,55,...,100}},
    columns/my new col/.style={string type}
}

\pgfplotstabletypeset[
    columns={level,my new col},
]{pgfplotstable.example1.dat}
```

The new column will be padded or truncated to the required number of rows. If the list does not contain enough elements, empty cells will be produced.

`/pgfplots/table/create col/copy={`⟨*column name*⟩`}`

A style for use in column creation context which simply copies the existing column ⟨*column name*⟩.

level	Copy of level
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10

```
\pgfplotstableset{
    create on use/new/.style={create col/copy={level}}
}

\pgfplotstabletypeset[
    columns={level,new},
    columns/new/.style={column name=Copy of level}
]{pgfplotstable.example1.dat}
```

`/pgfplots/table/create col/copy column from table={`⟨*file name or \macro*⟩`}{`⟨*column name*⟩`}`

A style for use in column creation context which creates a new column consisting of the entries in ⟨*column name*⟩ of the provided table. The argument may be either a file name or an already loaded table (i.e. a ⟨*\macro*⟩ as returned by `\pgfplotstableread`).

You can use this style, possibly combined with `\pgfplotstablenew`, to merge one common sort of column from different tables into one large table.

The cell values are written as string (verbatim).

The new column will be padded or truncated to the required number of rows. If the list does not contain enough elements, empty cells will be produced.

4.3.2 Mathematical Operations

`/pgf/fpu=true|false` (initially `true`)

Before we start to describe the column generation methods, one word about the math library. The core is always the PGF math engine written by Mark Wibrow and Till Tantau. However, this engine has been written to produce graphics and is not suitable for scientific computing.

I added a high-precision floating point library to PGF which will be part of releases newer than PGF 2.00. It offers the full range of IEEE double precision computing in TEX. This FPU is also part of PGFPLOTSTABLE, and it is activated by default for `create col/expr` and all other predefined mathematical methods.

The FPU won't be active for newly defined numerical styles (although it is active for the predefined mathematical expression parsing styles like `create col/expr`). If you want to add own routines or styles, you will need to use

```
\pgfkeys{/pgf/fpu=true}
```

in order to activate the extended precision. The standard math parser is limited to fixed point numbers in the range of ±16384.00000.

/pgfplots/table/create col/expr={⟨*math expression*⟩}

A style for use in `\pgfplotstablecreatecol` which uses ⟨*math expression*⟩ to assign contents for the new column.

level	2·level
1	2
2	4
3	6
4	8
5	10
6	12
7	14
8	16
9	18
10	20

```
\pgfplotstableset{
    create on use/new/.style={
        create col/expr={\thisrow{level}*2}}
}

\pgfplotstabletypeset[
    columns={level,new},
    columns/new/.style={column name=$2\cdot $level}
]{pgfplotstable.example1.dat}
```

The macros `\thisrow{`⟨*col name*⟩`}` and `\nextrow{`⟨*col name*⟩`}` can be used to use values of the existing table.

Please see `\pgfplotstablecreatecol` for more information.

Accumulated columns: The `expr` style initializes `\pgfmathaccuma` to `0` before its first column. Whenever it computes a new column value, it redefines `\pgfmathaccuma` to be the result. That means you can use `\pgfmathaccuma` inside of ⟨*math expression*⟩ to accumulate columns. See `create col/expr accum` for more details.

About the precision and number range: Starting with version 1.2, `expr` uses a floating point unit. The FPU provides the full data range of scientific computing with a relative precision between 10^{-4} and 10^{-6}. The `/pgf/fpu` key provides some more details.

Accepted operations: The math parser of PGF, combined with the FPU, provides the following function and operators:

`+`, `-`, `*`, `/`, `abs`, `round`, `floor`, `mod`, `<`, `>`, `max`, `min`, `sin`, `cos`, `tan`, `deg` (conversion from radians to degrees), `rad` (conversion from degrees to radians), `atan`, `asin`, `acos`, `cot`, `sec`, `cosec`, `exp`, `ln`, `sqrt`, the constanst `pi` and `e`, `^` (power operation), `factorial`[9], `rand` (random between −1 and 1 following a uniform distribution), `rnd` (random between 0 and 1 following a uniform distribution), number format conversions `hex`, `Hex`, `oct`, `bin` and some more. The math parser has been written by Mark Wibrow and Till Tantau [2], the FPU routines have been developed as part of PGFPLOTS. The documentation for both parts can be found in [2]. **Attention:** Trigonometric functions work with degrees, not with radians!

/pgfplots/table/create col/expr accum={⟨*math expression*⟩}{⟨*accum initial*⟩}

A variant of `create col/expr` which also allows to define the initial value of `\pgfmathaccuma`. The case ⟨*accum initial*⟩=`0` is *equivalent* to `expr={`⟨*math expression*⟩`}`.

[9] Starting with PGF versions newer than 2.00, you can use the postfix operator `!` instead of `factorial`.

level	∑level	∏level
1	1	1
2	3	2
3	6	6
4	10	24
5	15	120
6	21	720
7	28	5,040
8	36	40,320
9	45	$3.63 \cdot 10^5$
10	55	$3.63 \cdot 10^6$

```
\pgfplotstableset{
    create on use/new/.style={
        create col/expr={\pgfmathaccuma + \thisrow{level}}},
    create on use/new2/.style={
        create col/expr accum={\pgfmathaccuma * \thisrow{level}}{1}% <- start with '1'
    }
}

\pgfplotstabletypeset[
    columns={level,new,new2},
    columns/new/.style={column name=$\sum$level},
    columns/new2/.style={column name=$\prod$level}
]{pgfplotstable.example1.dat}
```

The example creates two columns: the `new` column is just the sum of each value in the ⟨*level*⟩ column (it employs the default `\pgfmathaccuma=0`). The `new2` column initializes `\pgfmathaccuma=100` and then successively subtracts the value of ⟨*level*⟩.

/pgfplots/table/create col/quotient={⟨*column name*⟩}

A style for use in `\pgfplotstablecreatecol` which computes the quotient $c_i := m_{i-1}/m_i$ for every entry $i = 1, \dots, (n-1)$ in the column identified with ⟨*column name*⟩. The first value c_0 is kept empty.

error1	error2	quot1	quot2
$2.50 \cdot 10^{-1}$	$7.58 \cdot 10^{-1}$		
$6.25 \cdot 10^{-2}$	$5.00 \cdot 10^{-1}$	4	1.52
$1.56 \cdot 10^{-2}$	$2.87 \cdot 10^{-1}$	4	1.74
$3.91 \cdot 10^{-3}$	$1.44 \cdot 10^{-1}$	4	2
$9.77 \cdot 10^{-4}$	$4.42 \cdot 10^{-2}$	4	3.25
$2.44 \cdot 10^{-4}$	$1.70 \cdot 10^{-2}$	4	2.6
$6.10 \cdot 10^{-5}$	$8.20 \cdot 10^{-3}$	4	2.07
$1.53 \cdot 10^{-5}$	$3.91 \cdot 10^{-3}$	4	2.1
$3.81 \cdot 10^{-6}$	$1.95 \cdot 10^{-3}$	4	2
$9.54 \cdot 10^{-7}$	$9.77 \cdot 10^{-4}$	4	2

```
%  requires \usepackage{array}
\pgfplotstableset{%  configuration, for example, in preamble:
    create on use/quot1/.style={create col/quotient=error1},
    create on use/quot2/.style={create col/quotient=error2},
    columns={error1,error2,quot1,quot2},
    %
    %  display styles:
    columns/error1/.style={sci,sci zerofill},
    columns/error2/.style={sci,sci zerofill},
    columns/quot1/.style={dec sep align},
    columns/quot2/.style={dec sep align}
}

\pgfplotstabletypeset{pgfplotstable.example1.dat}
```

This style employs methods of the floating point unit, that means it works with a relative precision of about 10^{-7} (7 significant digits in the mantissa).

/pgfplots/table/create col/iquotient={⟨*column name*⟩}

Like `create col/quotient`, but the quotient is inverse.

/pgfplots/table/create col/dyadic refinement rate={⟨*column name*⟩}

A style for use in \pgfplotstablecreatecol which computes the convergence rate α of the data in column ⟨*column name*⟩. The contents of ⟨*column name*⟩ is assumed to be something like $e_i(h_i) = O(h_i^\alpha)$. Assuming a dyadic refinement relation from one row to the next, $h_i = h_{i-1}/2$, we have $h_{i-1}^\alpha/(h_{i-1}/2)^\alpha = 2^\alpha$, so we get α using

$$c_i := \log_2\left(\frac{e_{i-1}}{e_i}\right).$$

The first value c_0 is kept empty.

error1	error2	rate1	rate2
$2.50 \cdot 10^{-1}$	$7.58 \cdot 10^{-1}$		
$6.25 \cdot 10^{-2}$	$5.00 \cdot 10^{-1}$	2	0.6
$1.56 \cdot 10^{-2}$	$2.87 \cdot 10^{-1}$	2	0.8
$3.91 \cdot 10^{-3}$	$1.44 \cdot 10^{-1}$	2	1
$9.77 \cdot 10^{-4}$	$4.42 \cdot 10^{-2}$	2	1.7
$2.44 \cdot 10^{-4}$	$1.70 \cdot 10^{-2}$	2	1.38
$6.10 \cdot 10^{-5}$	$8.20 \cdot 10^{-3}$	2	1.05
$1.53 \cdot 10^{-5}$	$3.91 \cdot 10^{-3}$	2	1.07
$3.81 \cdot 10^{-6}$	$1.95 \cdot 10^{-3}$	2	1
$9.54 \cdot 10^{-7}$	$9.77 \cdot 10^{-4}$	2	1

```
%  requires \usepackage{array}
\pgfplotstabletypeset[%  here, configuration options apply only to this single statement:
    create on use/rate1/.style={create col/dyadic refinement rate={error1}},
    create on use/rate2/.style={create col/dyadic refinement rate={error2}},
    columns={error1,error2,rate1,rate2},
    columns/error1/.style={sci,sci zerofill},
    columns/error2/.style={sci,sci zerofill},
    columns/rate1/.style={dec sep align},
    columns/rate2/.style={dec sep align}]
    {pgfplotstable.example1.dat}
```

This style employs methods of the floating point unit, that means it works with a relative precision of about 10^{-6} (6 significant digits in the mantissa).

/pgfplots/table/create col/idyadic refinement rate={⟨*column name*⟩}

As create col/dyadic refinement rate, but the quotient is inverse.

/pgfplots/table/create col/gradient={⟨*col x*⟩}{⟨*col y*⟩}
/pgfplots/table/create col/gradient loglog={⟨*col x*⟩}{⟨*col y*⟩}
/pgfplots/table/create col/gradient semilogx={⟨*col x*⟩}{⟨*col y*⟩}
/pgfplots/table/create col/gradient semilogy={⟨*col x*⟩}{⟨*col y*⟩}

A style for \pgfplotstablecreatecol which computes piecewise gradients $(y_{i+1} - y_i)/(x_{i+1} - x_i)$ for each row. The y values are taken out of column ⟨*col y*⟩ and the x values are taken from ⟨*col y*⟩.

The logarithmic variants apply the natural logarithm, $\log(\cdot)$, to its argument before starting to compute differences. More precisely, the `loglog` variant applies the logarithm to both x and y, the `semilogx` variant applies the logarithm only to x and the `semilogy` variant applies the logarithm only to y.

dof	error1	error2	slopes1	slopes2
4	$2.50 \cdot 10^{-1}$	$7.58 \cdot 10^{-1}$		
16	$6.25 \cdot 10^{-2}$	$5.00 \cdot 10^{-1}$	−1	−0.3
64	$1.56 \cdot 10^{-2}$	$2.87 \cdot 10^{-1}$	−1	−0.4
256	$3.91 \cdot 10^{-3}$	$1.44 \cdot 10^{-1}$	−1	−0.5
1,024	$9.77 \cdot 10^{-4}$	$4.42 \cdot 10^{-2}$	−1	−0.85
4,096	$2.44 \cdot 10^{-4}$	$1.70 \cdot 10^{-2}$	−1	−0.69
16,384	$6.10 \cdot 10^{-5}$	$8.20 \cdot 10^{-3}$	−1	−0.52
65,536	$1.53 \cdot 10^{-5}$	$3.91 \cdot 10^{-3}$	−1	−0.54
262,144	$3.81 \cdot 10^{-6}$	$1.95 \cdot 10^{-3}$	−1	−0.5
1,048,576	$9.54 \cdot 10^{-7}$	$9.77 \cdot 10^{-4}$	−1	−0.5

```
%  requires \usepackage{array}
\pgfplotstableset{%  configuration, for example in preamble:
    create on use/slopes1/.style={create col/gradient loglog={dof}{error1}},
    create on use/slopes2/.style={create col/gradient loglog={dof}{error2}},
    columns={dof,error1,error2,slopes1,slopes2},
    %  display styles:
    columns/dof/.style={int detect},
    columns/error1/.style={sci,sci zerofill},
    columns/error2/.style={sci,sci zerofill},
    columns/slopes1/.style={dec sep align},
    columns/slopes2/.style={dec sep align}
}
\pgfplotstabletypeset{pgfplotstable.example1.dat}
```

level	error1	slopes1
1	2.50_{-1}	
2	6.25_{-2}	-1.39
3	1.56_{-2}	-1.39
4	3.91_{-3}	-1.39
5	9.77_{-4}	-1.39
6	2.44_{-4}	-1.39
7	6.10_{-5}	-1.39
8	1.53_{-5}	-1.39
9	3.81_{-6}	-1.39
10	9.54_{-7}	-1.39

```
%  requires \usepackage{array}
\pgfplotstableset{%  configuration, for example in preamble:
    create on use/slopes1/.style={create col/gradient semilogy={level}{error1}},
    columns={level,error1,slopes1},
    %  display styles:
    columns/level/.style={int detect},
    columns/error1/.style={sci,sci zerofill,sci subscript},
    columns/slopes1/.style={dec sep align}
}
\pgfplotstabletypeset{pgfplotstable.example1.dat}
```

This style employs methods of the floating point unit, that means it works with a relative precision of about 10^{-6} (6 significant digits in the mantissa).

/pgfplots/table/create col/linear regression={⟨*key-value-config*⟩}

Computes a linear (least squares) regression $y(x) = a \cdot x + b$ using the sample data (x_i, y_i) which has to be specified inside of ⟨*key-value-config*⟩.

x	y	regression
1	1	-2.33
2	4	4.67
3	9	11.67
4	16	18.67
5	25	25.67
6	36	32.67

The slope is ‘7.0e0’.

```
%  load table from somewhere:
\pgfplotstableread{
    x y
    1 1
    2 4
    3 9
    4 16
    5 25
    6 36
}\loadedtbl

%  create the ‘regression’ column:
\pgfplotstablecreatecol[linear regression]
    {regression}
    {\loadedtbl}
%  store slope
\xdef\slope{\pgfplotstableregressiona}

\pgfplotstabletypeset\loadedtbl\\

The slope is ‘\slope’.
```

The example above loads a table from inline data, appends a column named ‘`regression`’ and typesets it. Since no ⟨*key-value-config*⟩ has been provided, `x=[index]0` and `y=[index]1` will be used. The

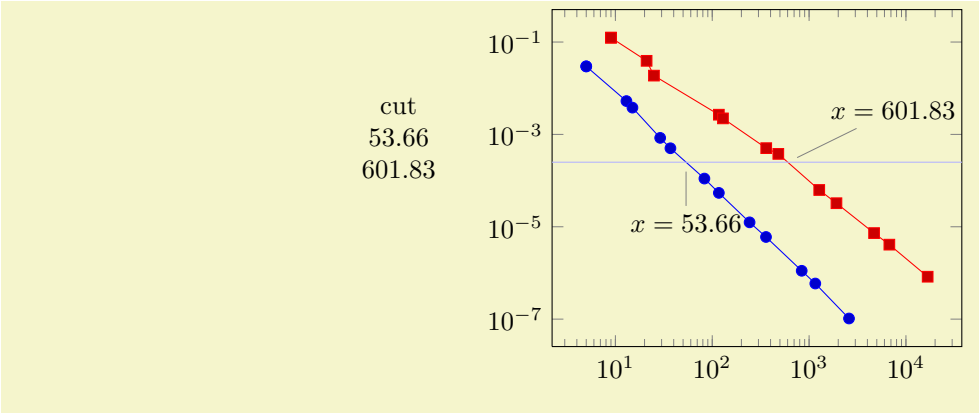
`\xdef\slope{...}` command stores the 'a' value of the regression line into a newly defined macro '`\slope`'[10].

The complete documentation for this feature has been moved to PGFPLOTS due to its close relation to plotting. Please refer to the PGFPLOTS manual coming with this package.

/pgfplots/table/create col/function graph cut y={⟨*cut value*⟩}{⟨*common options*⟩}{⟨*one key-value set for each plot*⟩}

A specialized style for use in `create on use` statements which computes cuts of (one or more) discrete plots $y(x_1), \dots, y(x_N)$ with a fixed ⟨*cut value*⟩. The x_i are written into the table's cells.

In a cost–accuracy plot, this feature allows to extract the cost for fixed accuracy. The dual feature with `cut x` allows to compute the accuracy for fixed cost.




```
% Preamble: \pgfplotsset{width=7cm,compat=1.7}
\pgfplotstablenew[
    create on use/cut/.style={create col/function graph cut y=
        {2.5e-4} %  search for fixed L2 = 2.5e-4
        {x=Basis,y=L2,ymode=log,xmode=log} %  double log, each function is L2(Basis)
        %  now, provide each single function f_i(Basis):
        {{table=plotdata/newexperiment1.dat},{table=plotdata/newexperiment2.dat}}
    },
    columns={cut}]
    {2}
    \loadedtable

%  Show the data:
\pgfplotstabletypeset{\loadedtable}

\begin{tikzpicture}
\begin{loglogaxis}
    \addplot table[x=Basis,y=L2] {plotdata/newexperiment1.dat};
    \addplot table[x=Basis,y=L2] {plotdata/newexperiment2.dat};
    \draw[blue!30!white] (axis cs:1,2.5e-4) -- (axis cs:1e5,2.5e-4);
    \node[pin=-90:{$x=53.66$}] at (axis cs:53.66,2.5e-4) {};
    \node[pin=45:{$x=601.83$}] at (axis cs:601.83,2.5e-4) {};
\end{loglogaxis}
\end{tikzpicture}
```

In the example above, we are searching for x_1 and x_2 such that $f_1(x_1) = 2.5{\cdot}10^{-4}$ and $f_2(x_2) = 2.5{\cdot}10^{-4}$. On the left is the automatically computed result. On the right is a problem illustration with proper annotation using PGFPLOTS to visualize the results. The ⟨*cut value*⟩ is set to `2.5e-4`. The ⟨*common options*⟩ contain the problem setup; in our case logarithmic scales and column names. The third argument is a comma-separated-list. Each element i is a set of keys describing how to get $f_i(\cdot)$.

During both ⟨*common options*⟩ and ⟨*one key-value set for each plot*⟩, the following keys can be used:

- `table={`⟨*table file or \macro*⟩`}`: either a file name or an already loaded table where to get the data points,
- `x={`⟨*col name*⟩`}`: the column name of the x axis,
- `y={`⟨*col name*⟩`}`: the column name of the y axis.

[10] The `\xdef` means "global expanded definition": it expands the argument until it can't be expanded any further and assigns a (global) name to the result. See any TEX book for details.

- `foreach={⟨\foreach loop head⟩}{⟨file name pattern⟩}` This somewhat advanced syntax allows to collect tables in a loop automatically:

cut
53.66
601.83

```
\pgfplotstablenew[
    %  same as above...
    create on use/cut/.style={create col/function graph cut y=
        {2.5e-4}%  search for fixed L2 = 2.5e-4
        {x=Basis,y=L2,ymode=log,xmode=log,
         foreach={\i in {1,2}}{plotdata/newexperiment\i.dat}}%
        {}%  just leave this empty.
    },
    columns={cut}]
    {2}
    \loadedtable
%  Show the data:
\pgfplotstabletypeset{\loadedtable}
```

PGFPLOTSTABLE will call `\foreach` ⟨*\foreach loop head*⟩ and it will expand ⟨*file name pattern*⟩ for every iteration. For every iteration, a simpler list entry of the form

`table={`⟨*expanded pattern*⟩`},x={`⟨*value of x*⟩`},y={`⟨*value of y*⟩`}`

will be generated.

It is also possible to provide `foreach=` inside of ⟨*one key-value set for each plot*⟩. The `foreach` key takes precedence over `table`. Details about the accepted syntax of `\foreach` can be found in the PGF manual.

The keys `xmode` and `ymode` can take either `log` or `linear`. All mentioned keys have the common key path

`/pgfplots/table/create col/function graph cut/`.

`/pgfplots/table/create col/function graph cut x={`⟨*cut value*⟩`}{`⟨*common options*⟩`}{`⟨*one key-value set for each plot*⟩`}`

As above, just with x and y exchanged.

5 Miscellaneous

5.1 Writing (Modified) Tables To Disk

`/pgfplots/table/outfile={`⟨*file name*⟩`}` (initially `empty`)

Writes the completely processed table as TEX file to ⟨*file name*⟩. This key is described in all detail on page 21.

`\pgfplotstablesave[`⟨*options*⟩`]{`⟨*\macro or input file name*⟩`}{`⟨*output file name*⟩`}`

This command takes a table and writes it to a new data file (without performing any typesetting).

If the first argument is a file name, that file is loaded first.

This command simply invokes `\pgfplotstabletypeset` with cleared output parameters. That means any of the column creation methods apply here as well, including any postprocessing steps (without the final typesetting).

`\pgfplotstablesave` uses the keys `reset styles` and `disable rowcol styles` to clear any typesetting related options.

Furthermore, it sets `string type` to allow verbatim output. You may want to use `numeric as string type` instead in case you only have numerical data – this will display integers resulting from arithmetics not in scientific notation[11].

[11] Note however, that `string type` does not round or truncate integers either, even though they are displayed as floats.

```
\pgfplotstablesave[
    create on use/postproc1/.style={create col/dyadic refinement rate=error1},
    columns={dof,error1,postproc1}
]
    {pgfplotstable.example1.dat}
    {pgfplotstable.example1.out.dat}
```

Now, `pgfplotstable.example1.out.dat` is

```
dof      error1  postproc1
4        2.50000000e-01  {}
16       6.25000000e-02  1.99998
64       1.56250000e-02  1.99998
256      3.90625000e-03  1.99998
1024     9.76562500e-04  1.99998
4096     2.44140625e-04  1.99998
16384    6.10351562e-05  1.99998
65536    1.52587891e-05  1.99998
262144   3.81469727e-06  1.99998
1048576  9.53674316e-07  1.99998
```

You can use the `col sep` key inside of ⟨*options*⟩ to define a column separator for the output file. In case you need a different input column separator, use `in col sep` instead of `col sep`.

Remarks

- Empty cells will be filled with `{}` if `col sep=space`. Use the `empty cells with` style to change that.
- Use `disable rowcol styles=false` inside of ⟨*options*⟩ if you need to change column/row based styles.

5.2 Miscellaneous Keys

`/pgfplots/table/disable rowcol styles=true|false` (initially `false`)

Set this to `true` if `\pgfplotstabletypeset` shall *not* set any styles which apply only to specific columns or only to specific rows.

This disables the styles

- `columns/`⟨*column name*⟩,
- `display columns/`⟨*column index*⟩,
- `every col no` ⟨*column index*⟩,
- `every row no` ⟨*row index*⟩.

`/pgfplots/table/reset styles` (no value)

Resets all table typesetting styles which do not explicitly depend on column or row names and indices. The affected styles are

- `every table`,
- `every even row`, `every odd row`, `every even column`, `every odd column`,
- `every first column`, `every last column`, `every first row`, `every last row`,
- `every head row`,
- `postproc cell content`, `preproc cell content`.

In case you want to reset all, you should also consider the key `disable rowcol styles`.

5.3 A summary of how to define and use styles and keys

This section summarizes features of `pgfkeys`. The complete documentation can be found in the PGF manual, [2].

Key handler ⟨*key*⟩`/.style={`⟨*key-value-list*⟩`}`
Defines or redefines a style ⟨*key*⟩. A style is a normal key which will set all options in ⟨*key-value-list*⟩ when it is set.
Use `\pgfplotstableset{`⟨*key*⟩`/.style={`⟨*key-value-list*⟩`}}` to (re)define a style ⟨*key*⟩ in the namespace `/pgfplots/table`.

Key handler ⟨*key*⟩`/.append style={`⟨*key-value-list*⟩`}`
Appends ⟨*key-value-list*⟩ to an already existing style ⟨*key*⟩. This is the preferred method to change the predefined styles: if you only append, you maintain compatibility with future versions.
Use `\pgfplotstableset{`⟨*key*⟩`/.append style={`⟨*key-value-list*⟩`}}` to append ⟨*key-value-list*⟩ to the style ⟨*key*⟩. This will assume the prefix `/pgfplots/table`.

Key handler ⟨*key*⟩`/.initial={`⟨*value*⟩`}`
Defines a new ⟨*key*⟩ and assigns ⟨*value*⟩.

Key handler ⟨*key*⟩`/.add={`⟨*before*⟩`}{`⟨*after*⟩`}`
Changes ⟨*key*⟩ by prepending ⟨*before*⟩ and appending ⟨*after*⟩.

‘a column’; ‘a column,another’; ‘a column,another,and one more’.

```
\pgfplotstableset{columns={a column}}
‘\pgfkeysvalueof{/pgfplots/table/columns}’;
\pgfplotstableset{columns/.add={}{,another}}
‘\pgfkeysvalueof{/pgfplots/table/columns}’;
\pgfplotstableset{columns/.add={}{,and one more}}
‘\pgfkeysvalueof{/pgfplots/table/columns}’.
```

This can be used inside of `\pgfplotsinvokeforeach` or similar (ungrouped!) loop constructs.

Key handler ⟨*key*⟩`/.code={`⟨*TEX code*⟩`}`
Occasionally, the PGFPLOTS user interface offers to replace parts of its routines. This is accomplished using so called “code keys”. What it means is to replace the original key and its behavior with new ⟨*TEX code*⟩. Inside of ⟨*TEX code*⟩, any command can be used. Furthermore, the `#1` pattern will be the argument provided to the key.

This is a pgfkeys feature. Argument=‘is here’

```
\pgfplotsset{
    My Code/.code={This is a pgfkeys feature. Argument=‘#1’}}
\pgfplotsset{My Code={is here}}
```

The example defines a (new) key named `My Code`. Essentially, it is nothing else but a `\newcommand`, plugged into the key-value interface. The second statement “invokes” the code key.

Key handler ⟨*key*⟩`/.append code={`⟨*TEX code*⟩`}`
Appends ⟨*TEX code*⟩ to an already existing `/.code` key named ⟨*key*⟩.

Key handler ⟨*key*⟩`/.code 2 args={`⟨*TEX code*⟩`}`
As `/.code`, but this handler defines a key which accepts two arguments. When the so defined key is used, the two arguments are available as `#1` and `#2`.

5.4 Plain TEX and ConTEXt support

The table code generator is initialized to produce LATEX `tabular` environments. However, it only relies on ‘`&`’ being the column separator and ‘`\\`’ the row terminator. The `column type` feature is more or less specific to `tabular`, but you can disable it completely. Replace `begin table` and `end table` with appropriate TEX- or ConTEXt commands to change it. If you have useful default styles (or bug reports), let me know.

5.5 Basic Level Table Access and Modification

PGFPLOTSTABLE provides several methods to access and manipulate tables at an elementary level.

Please keep in mind that PGFPLOTSTABLE has been written as a tool for table visualization. As such, it has been optimized for the case of relatively few rows (although it may have a lot of columns). The runtime

for table creation and modification is currently $O(N^2)$ where N is the number of rows[12]. This is completely acceptable for tables with few rows because TEX can process those structures relatively fast. Keep your tables small! PGFPLOTSTABLE is *not* a tool for large-scale matrix operations.

Tables are always stored as a sequence of column vectors. Therefore, iteration over all values in one column is simple whereas iteration over all values in one row is complicated and expensive.

`\pgfplotstableforeachcolumn`⟨*table*⟩`\as{`⟨*\macro*⟩`}{`⟨*code*⟩`}`

Iterates over every column name of ⟨*table*⟩. The ⟨*\macro*⟩ will be set to the currently visited column name. Then, ⟨*code*⟩ will be executed. During ⟨*code*⟩, `\pgfplotstablecol` denotes the current column index (starting with 0).

```
column name is ‘level’; index is0;
column name is ‘dof’; index is1;
column name is ‘error1’; index is2;
column name is ‘error2’; index is3;
column name is ‘info’; index is4;
column name is ‘grad(log(dof),log(error2))’; index is5;
column name is ‘quot(error1)’; index is6;
```

```
\begin{minipage}{0.8\linewidth}
\pgfplotstableread{pgfplotstable.example1.dat}\loadedtable
\pgfplotstableforeachcolumn\loadedtable\as\col{%
    column name is ‘\col’; index is\pgfplotstablecol;\par
}
\end{minipage}
```

This routine does not introduce TEX groups, variables inside of ⟨*code*⟩ are not scoped.

`\pgfplotstableforeachcolumnelement`⟨*column name*⟩`\of`⟨*table*⟩`\as`⟨*\cellcontent*⟩`{`⟨*code*⟩`}`

Reports every table cell t_{ij} for a fixed column j in read-only mode.

For every cell in the column named ⟨*column name*⟩, ⟨*code*⟩ will be executed. During this invocation, the macro ⟨*\cellcontent*⟩ will contain the cell's content and `\pgfplotstablerow` will contain the current row's index.

```
I have now cell element ‘2.50000000e-01’ at row index ‘0’;
I have now cell element ‘6.25000000e-02’ at row index ‘1’;
I have now cell element ‘1.56250000e-02’ at row index ‘2’;
I have now cell element ‘3.90625000e-03’ at row index ‘3’;
I have now cell element ‘9.76562500e-04’ at row index ‘4’;
I have now cell element ‘2.44140625e-04’ at row index ‘5’;
I have now cell element ‘6.10351562e-05’ at row index ‘6’;
I have now cell element ‘1.52587891e-05’ at row index ‘7’;
I have now cell element ‘3.81469727e-06’ at row index ‘8’;
I have now cell element ‘9.53674316e-07’ at row index ‘9’;
```

```
\begin{minipage}{0.8\linewidth}
\pgfplotstableread{pgfplotstable.example1.dat}\loadedtable
\pgfplotstableforeachcolumnelement{error1}\of\loadedtable\as\cell{%
    I have now cell element ‘\cell’ at row index ‘\pgfplotstablerow’;\par
}
\end{minipage}
```

The argument ⟨*column name*⟩ can also be a column index. In that case, it should contain `[index]`⟨*integer*⟩, for example `[index]4`. Furthermore, column aliases and columns which should be generated on-the-fly (see `create on use`) can be used for ⟨*column name*⟩.

This routine does not introduce TEX groups, variables inside of ⟨*code*⟩ are not scoped.

`\pgfplotstablemodifyeachcolumnelement`⟨*column name*⟩`\of`⟨*table*⟩`\as`⟨*\cellcontent*⟩`{`⟨*code*⟩`}`

A routine which is similar to `\pgfplotstableforeachcolumnelement`, but any changes of ⟨*\cellcontent*⟩ which might occur during ⟨*code*⟩ will be written back into the respective cell.

[12] The runtime for `plot table` is linear in the number of rows using a special routine.

error1
#0: 2.50000000e-01
#1: 6.25000000e-02
#2: 1.56250000e-02
#3: 3.90625000e-03
#4: 9.76562500e-04
#5: 2.44140625e-04
#6: 6.10351562e-05
#7: 1.52587891e-05
#8: 3.81469727e-06
#9: 9.53674316e-07

```
\pgfplotstableread{pgfplotstable.example1.dat}\loadedtable
\pgfplotstablemodifyeachcolumnelement{error1}\of\loadedtable\as\cell{%
    \edef\cell{\#\pgfplotstablerow: \cell}%
}
\pgfplotstabletypeset[columns=error1,string type]{\loadedtable}
```

If ⟨*column name*⟩ is a column alias or has been created on-the-fly, a new column named ⟨*column name*⟩ will be created.

\pgfplotstablegetelem{⟨*row*⟩}{⟨*col*⟩}\of⟨*table*⟩

Selects a single table element at row ⟨*row*⟩ and column ⟨*col*⟩. The second argument has the same format as that described in the last paragraph: it should be a column name or a column index (in which case it needs to be written as `[index]`⟨*number*⟩).

The return value will be written to `\pgfplotsretval`.

The value (4,error1) is '9.76562500e-04'. The value (2,0) is '3'.

```
\pgfplotstableread{pgfplotstable.example1.dat}{\loadedtable}
\pgfplotstablegetelem{4}{error1}\of{\loadedtable}
The value (4,error1) is '\pgfplotsretval'.

\pgfplotstablegetelem{2}{[index]0}\of{\loadedtable}
The value (2,0) is '\pgfplotsretval'.
```

Attention: If possible, avoid using this command inside of loops. It is quite slow.

\pgfplotstablegetrowsof{⟨*file name or \loadedtable*⟩}
\pgfplotstablegetcolsof{⟨*file name or \loadedtable*⟩}

Defines `\pgfplotsretval` to be the number of rows in a table[13]. The argument may be either a file name or an already loaded table (the ⟨*\macro*⟩ of `\pgfplotstableread`).

\pgfplotstablevertcat{⟨*\table1*⟩}{⟨*\table2 or filename*⟩}

See page 44 for details about this command.

\pgfplotstablenew[⟨*options*⟩]{⟨*row count*⟩}{⟨*\table*⟩}

See Section 4 for details about this command.

\pgfplotstablecreatecol[⟨*options*⟩]{⟨*row count*⟩}{⟨*\table*⟩}

See Section 4 for details about this command.

\pgfplotstabletranspose[⟨*options*⟩]{⟨*\outtable*⟩}{⟨*\table or filename*⟩}
\pgfplotstabletranspose*[⟨*options*⟩]{⟨*\outtable*⟩}{⟨*\table or filename*⟩}

Defines ⟨*\outtable*⟩ to be the transposed of ⟨*\table of filename*⟩. The input argument can be either a file name or an already loaded table.

The version with '*' is only interesting in conjunction with the `columns` option, see below.

[13] It will also assign `\pgfmathresult` to the same value.

a	b	c	d
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23

```
\pgfplotstabletypeset[string type]{pgfplotstable.example3.dat}
```

colnames	0	1	2	3	4	5
a	0	4	8	12	16	20
b	1	5	9	13	17	21
c	2	6	10	14	18	22
d	3	7	11	15	19	23

```
\pgfplotstabletranspose\loadedtable{pgfplotstable.example3.dat}
\pgfplotstabletypeset[string type]\loadedtable
```

The optional argument ⟨*options*⟩ can contain options which influence the transposition:

/pgfplots/table/colnames from={⟨*colname*⟩} (initially `empty`)

Inside of \pgfplotstabletranspose, this key handles how to define output column names.

If ⟨*colname*⟩ *is* empty (the initial value), the output column names will simply be the old row indices, starting with 0.

If ⟨*colname*⟩ is not empty, it denotes an input column name whose cell values will make up the output column names:

colnames	2	6	10	14	18	22
a	0	4	8	12	16	20
b	1	5	9	13	17	21
d	3	7	11	15	19	23

```
\pgfplotstabletranspose[colnames from=c]\loadedtable{pgfplotstable.example3.dat}
\pgfplotstabletypeset[string type]\loadedtable
```

The argument ⟨*colname*⟩ won't appear as cell contents. It is an error if the cells in ⟨*colname*⟩ don't yield unique column names.

/pgfplots/table/input colnames to={⟨*name*⟩} (initially `colnames`)

Inside of \pgfplotstabletranspose, this key handles what to do with *input* column names.

This key will create a further column named ⟨*name*⟩ which will be filled with the input column names (as string type).

Input	0	1	2	3	4	5
a	0	4	8	12	16	20
b	1	5	9	13	17	21
c	2	6	10	14	18	22
d	3	7	11	15	19	23

```
\pgfplotstabletranspose[input colnames to=Input]\loadedtable{pgfplotstable.example3.dat}
\pgfplotstabletypeset[string type]\loadedtable
```

Set ⟨*name*⟩ to the empty string to disable this column.

0	1	2	3	4	5
0	4	8	12	16	20
1	5	9	13	17	21
2	6	10	14	18	22
3	7	11	15	19	23

```
\pgfplotstabletranspose[input colnames to=]\loadedtable{pgfplotstable.example3.dat}
\pgfplotstabletypeset[string type]\loadedtable
```

/pgfplots/table/columns={⟨*list*⟩} (initially `empty`)

Inside of `\pgfplotstabletranspose`, this key handles which input columns shall be considered for the transposition.

If ⟨*list*⟩ is empty, all columns of the input table will be used (which is the initial configuration).

If ⟨*list*⟩ is not empty, it is expected to be a list of column names. Only these columns will be used as input for the transposition, just as if the remaining ones weren't there. It is acceptable to provide column aliases or `create on use` arguments inside of ⟨*list*⟩.

colnames	0	1	2	3	4	5
a	0	4	8	12	16	20
b	1	5	9	13	17	21

```
\pgfplotstabletranspose[columns={a,b}]\loadedtable{pgfplotstable.example3.dat}
\pgfplotstabletypeset[string type]\loadedtable
```

Here is the only difference between `\pgfplotstabletranspose` and `\pgfplotstabletranspose*`: the version without '`*`' *resets* the `columns` key before it starts whereas the version with '`*`' simply uses the actual content of `columns`.

`\pgfplotstablesort`[⟨*options*⟩]⟨*resulttable*⟩⟨*table or filename*⟩

Sorts ⟨*table or filename*⟩ according to ⟨*options*⟩ and writes the sorted table to ⟨*resulttable*⟩.

Use the high level `sort` key to enable sorting automatically during `\pgfplotstabletypeset`.

a	b	c
−11	−9	[d]
−9	−9	[f]
−6	−14	[b]
1	13	[g]
4	−14	[c]
8	−10	[h]
11	14	[e]
16	18	[i]
19	2	[a]
19	−6	[j]

```
\pgfplotstablesort\result{%
    a      b  c
    19     2 [a]
    -6   -14 [b]
     4   -14 [c]
   -11    -9 [d]
    11    14 [e]
    -9    -9 [f]
     1    13 [g]
     8   -10 [h]
    16    18 [i]
    19    -6 [j]
}
\pgfplotstabletypeset[columns/c/.style={string type}]{\result}%
```

The sort key and comparison function can be customized using the following keys:

/pgfplots/table/sort key={⟨*column*⟩} (initially `[index]0`)

Specifies the column which contains the sort key. The argument ⟨*column*⟩ can be any of the columns of the input table, including `create on use`, `alias` or `[index]`⟨*integer*⟩ specifications. The initial setting uses the first available column.

/pgfplots/table/sort key from={⟨*table*⟩} (initially `empty`)

Allows to load the `sort key` from a different ⟨*table*⟩, which can be either a ⟨*macro*⟩ or a ⟨*file name*⟩.

/pgfplots/table/sort cmp={⟨*less than routine*⟩} (initially `float <`)

Allows to use a different comparison function.

/pgfplots/fixed < (no value)
/pgfplots/fixed > (no value)
/pgfplots/int < (no value)
/pgfplots/int > (no value)
/pgfplots/float < (no value)
/pgfplots/float > (no value)
/pgfplots/date < (no value)
/pgfplots/date > (no value)
/pgfplots/string < (no value)
/pgfplots/string > (no value)

These styles constitute the predefined comparison functions. The `fixed <`, `int <` and `float <` routines operate on numerical data where `int <` expects positive or negative integers and the other two expect real numbers. The `fixed <` has a considerably smaller number range, but is slightly faster than `float <`.

The `date <` compares dates of the form `YYYY-MM-DD`. The `string <` uses lexicographical string comparison based on the ASCII character codes of the sort keys. The `string <` routine also evaluates ASCII codes of control sequences or active characters[14].

Header
brown
dog
fox
jumps
lazy
over
quick
the
the

```
\pgfplotstablesort[sort cmp=string <]\result{%  'Header' is the column name:
    Header
    the
    quick
    brown
    fox
    jumps
    over
    the
    lazy
    dog
}
\pgfplotstabletypeset[string type]{\result}%
```

/pgfplots/iflessthan/.code args={##1##2##3##4}{⟨...⟩}

Allows to define custom comparison functions (a strict ordering). It compares `#1 < #2` and invokes `#3` in case the comparison is true and `#4` if not. The comparison will be evaluated in local scopes (local variables are freed afterwards).

5.6 Repeating Things: Loops

\foreach⟨*variables*⟩ in ⟨*list*⟩ {⟨*commands*⟩}

A powerful loop command provided by Ti*k*Z, see [2, Section Utilities].

Iterating 1. Iterating 2. Iterating 3. Iterating 4.

```
\foreach \x in {1,2,...,4} {Iterating \x. }%
```

A PGFPLOTS related example could be

```
\foreach \i in {1,2,...,10} {\addplot table {datafile\i}; }%
```

The following loop commands come with PGFPLOTS. They are similar to the powerful Ti*k*Z `\foreach` loop command, which, however, is not always useful for table processing: the effect of its loop body end after each iteration.

The following PGFPLOTS looping macros are an alternative.

\pgfplotsforeachungrouped⟨*variable*⟩ in ⟨*list*⟩ {⟨*command*⟩}

A specialized variant of `\foreach` which can do two things: it does not introduce extra groups while executing ⟨*command*⟩ and it allows to invoke the math parser for (simple!) $\langle x_0\rangle, \langle x_1\rangle, \ldots, \langle x_n\rangle$ expressions.

Iterating 1. Iterating 2. Iterating 3. Iterating 4. All collected = , 1, 2, 3, 4.

```
\def\allcollected{}
\pgfplotsforeachungrouped \x in {1,2,...,4} {Iterating \x. \edef\allcollected{\allcollected, \x}}%
All collected = \allcollected.
```

A more useful example might be to work with tables:

```
\pgfplotsforeachungrouped \i in {1,2,...,10} {%
    \pgfplotstablevertcat{\output}{datafile\i} %  appends 'datafile\i' -> '\output'
}%
%  since it was ungrouped, \output is still defined (would not work
%  with \foreach)
```

[14] As long as they haven't been consumed by TEX's preprocessing.

Remark: The special syntax ⟨*list*⟩=⟨x_0⟩,⟨x_1⟩,...,⟨x_n⟩, i.e. with two leading elements, followed by dots and a final element, invokes the math parser for the loop. Thus, it allows larger number ranges than any other syntax if `/pgf/fpu` is active. In all other cases, `\pgfplotsforeachungrouped` invokes `\foreach` and provides the results without TEX groups.

`\pgfplotsinvokeforeach{`⟨*list*⟩`} {`⟨*command*⟩`}`

A variant of `\pgfplotsforeachungrouped` (and such also of `\foreach`) which replaces any occurence of `#1` inside of ⟨*command*⟩ once for every element in ⟨*list*⟩. Thus, it actually assumes that ⟨*command*⟩ is like a `\newcommand` body.

In other words, ⟨*command*⟩ is invoked for every element of ⟨*list*⟩. The actual element of ⟨*list*⟩ is available as `#1`.

As `\pgfplotsforeachungrouped`, this command does *not* introduce extra scopes (i.e. it is ungrouped as well).

The difference to `\foreach \x in` ⟨*list*⟩`{`⟨*command*⟩`}` is subtle: the `\x` would *not* be expanded whereas `#1` is.

Invoke them: [a] [b] [c] [d]

```
\pgfkeys{
  otherstyle a/.code={[a]},
  otherstyle b/.code={[b]},
  otherstyle c/.code={[c]},
  otherstyle d/.code={[d]}}
\pgfplotsinvokeforeach{a,b,c,d}
    {\pgfkeys{key #1/.style={otherstyle #1}}}
Invoke them: \pgfkeys{key a} \pgfkeys{key b} \pgfkeys{key c} \pgfkeys{key d}
```

The counterexample would use a macro (here `\x`) as loop argument:

Invoke them: [d] [d] [d] [d]

```
\pgfkeys{
  otherstyle a/.code={[a]},
  otherstyle b/.code={[b]},
  otherstyle c/.code={[c]},
  otherstyle d/.code={[d]}}
\pgfplotsforeachungrouped \x in {a,b,c,d}
    {\pgfkeys{key \x/.style={otherstyle \x}}}
Invoke them: \pgfkeys{key a} \pgfkeys{key b} \pgfkeys{key c} \pgfkeys{key d}
```

Restrictions: You cannot nest this command yet (since it does not introduce protection by scopes).

Index

— T —

— U —

— V —

— W —

— Y —

— Z —

References

[1] D. Knuth. *Computers & Typesetting*. Addison Wesley, 2000.

[2] T. Tantau. Ti*k*Z and PGF manual. http://sourceforge.net/projects/pgf. *v.* ≥ 2.00.

[3] J. Wright and C. Feuersänger. Implementing keyval input: an introduction. http://pgfplots.sourceforge.net as .pdf, 2008.