numerica

version 3.0.0

Andrew Parsloe
(ajparsloe@gmail.com)

August 22, 2023

Abstract

The `numerica` package defines a command `\nmcEvaluate` (short-name form `\eval`) to wrap around mathematical expressions in the LaTeX form in which they are typeset and numerically evaluate them. For programs (like LyX) with a preview facility, or for compile-as-you-go systems, interactive back-of-envelope calculations and numerical exploration are possible within the document being worked on.

- This document applies to version 3.0.0 of `numerica`.
- `numerica` requires a LaTeX 2$_\varepsilon$ system from October 2020 or later (when `xparse` became available in LaTeX 2$_\varepsilon$ systems).
- The package requires `amsmath` and `mathtools`, is compatible with the `mleftright` and `xfrac` packages, and 'knows about' some symbols from `amssymb`.
- I refer many times in this document to *Handbook of Mathematical Functions*, edited by Milton Abramowitz and Irene A. Stegun, Dover, 1965, abbreviated to *HMF*, and often followed by a number like 1.2.3 to locate the actual expression referenced.
- Version 3.0.0 of `numerica`
  - dispenses with a configuration file (`numerica.cfg`) but adds three package options:
    - `comma` sets the comma as the decimal point; items in the variable=value list must then be separated by semicolons;
    - `rounding`=n sets the default rounding value to the integer n;
    - `approx` replaces the default `=` between formula and result in displays with `\approx` ($\approx$);
  - enables outputting results as (approximate) fractions with integer numerators and denominators in both slash and `\frac` forms;
  - allows use of LaTeX braces for delimiting arguments to functions like `\sin` and `\cos` to handle complicated arguments in (e.g.) Fourier series; the previous `()=0`, `1`, `2` setting for this is removed;
  - enables multiple formulas to be evaluated within the one `\eval` command; and
  - provides enhanced treatment of mathematical environments for the presentation of results, especially for such multiple evaluations, with the `env` key in the settings option (which makes the `*` key obsolete);
  - defines a `\degree` command and uses it as an alternative to the `o` setting for specifying angles in degrees;
  - allows nested commands to be evaluated to a specified rounding value (rather than insisting that they be evaluated to maximum precision);
  - accepts the use of spaces to group blocks of digits in numbers in the variable=value list and formula (with the setting `1s2=1`);
  - resolves the 'leading space' issue with the `\macros` command when a user-defined macro begins with an expandable token;
  - *continues . . .*

- Version 3.0.0 of `numerica` *(continued)*
  - reworks (again!) the `\reuse` command to simplify its use, and
  - removes the `reuse` *setting* of `\eval` (not the command); now *only* the numerical result is saved, either as a decimal or in scientific notation or in fraction form (but without math delimiters or variable=value list);
  - adds the LaTeX form of a result to the debug display (`dbg=11`);
  - adds warnings with line numbers to the LaTeX log file for `numerica` errors (which continue to be displayed in the pdf);
  - accepts the use of `a-`, `ar-` or `arc-` prefixes for the inverses of all six hyperbolic functions so that, for instance, `\asinh`, `\arsinh`, and `\arcsinh`, displaying as asinh, arsinh and arcsinh, can be used (rather than only `\asinh` as before);
  - accepts the `\sfrac` command from the `xfrac` package, producing elegant slash fractions like $^{355}/_{113}$;
  - accepts commands of the `mleftright` package;
  - accepts `\mkern` and `\mskip` commands in formulas;
  - defines the commands `\comma` and `\equals` (expanding to `,` and `=`) for use in the settings option, as distinct from the 'bare' marks used in formulas;
  - fixes bugs that could occur: (i) when raising an n-th root to a power; (ii) when using a dot with `\left`, `\right`; and (iii) when using a non-integer in the first argument of `\binom`;
  - amends and adds to documentation.

Contents

Chapter 1

Introduction

`numerica` is a LaTeX package offering the ability to numerically evaluate mathematical expressions in the LaTeX form in which they are typeset.

There are a number of packages which can do calculations in LaTeX,[1] but those I am aware of all require the mathematical expressions they operate on to be changed to an appropriate syntax. Of these packages `xfp` comes closest to my objective with `numerica`. For instance, given a formula

```
\frac{\sin (3.5)}{2} + 2\cdot 10^{-3}
```

(in a math environment), this can be evaluated using `xfp` by transforming the expression to `sin(3.5)/2 + 2e-3` and wrapping this in the command `\fpeval`. In `numerica` you don't need to transform the formula, just wrap it in an `\eval` command:

```
\eval{ \frac{\sin (3.5)}{2} + 2\cdot 10^{-3} }.
```

(For the actual calculation see §1.1.3.)

`numerica`, like `xfp` and a number of other packages, uses `l3fp` (the LaTeX3 floating point module in `l3kernel` and since February 2020 available in LaTeX 2_ε distributions) as its calculational engine. The main command of the package, `\nmcEvaluate`, short-name form `\eval`, in many ways acts as a pre-processor to `l3fp`, converting mathematical expressions written in the LaTeX form in which they will be typeset into an 'fp-ified' form that is digestible by `l3fp`. The aim is for the command to act as a wrapper around LaTeX formulas, processing them into a form that is digestible by `l3fp` and allows compilation to `pdf` to take place. Ideally, one should not have to make *any* adjustment to a formula,

[1] A simple search finds the venerable `calc` in the LaTeX base, `calculator` (including an associated `calculus` package), `fltpoint`, `fp` (*fixed* rather than floating point), `spreadtab` (using either `fp` or `l3fp` as its calculational engine) if you want simple spreadsheeting with your calculations, the elaborate `xint`, `pst-calculate` (a limited interface to `l3fp`), `l3fp` in the LaTeX3 kernel, and `xfp`, the LaTeX3 interface to `l3fp`. Other packages include a calculational element but are restricted in their scope. (`longdivision` for instance is elegant, but limited only to long division.)

although any text on Fourier series suggests that hope in full generality is delusional. Surprisingly often however it *is* possible. We shall see shortly that even complicated formulas like

$$\cos\tfrac{m}{n}\pi - (1-4\sin^2\tfrac{m}{3n}\pi)\frac{\sin\frac{1}{n}\pi\sin\frac{m-1}{n}\pi}{2\sin^2\frac{m}{3n}\pi},$$

and

$$\left(\frac{1-4\sin^2\frac{m}{3n}\pi}{2\sin^2\frac{m}{3n}\pi}\right)\sin\tfrac{2m-3}{3n}\pi\sin\tfrac{m-3}{3n}\pi,$$

can be evaluated 'as is' (see below, §1.1.6.3). There is no need to shift the position of the superscript 2 on the sines, no need to parenthesize the arguments of `\sin` and `\cos`, no need to insert asterisks to indicate multiplication, no need to change the `\frac` and `\tfrac`-s to slashes, `/`, and in the second expression no need to delete the `\left` and `\right` that qualify the big parentheses (in the underlying LaTeX). Of course, if there are variables in an expression, as in these examples, they will need to be assigned values; that is unavoidable. And how the result of the evaluation is displayed also requires specifying, but the aim is always: to evaluate mathematical expressions in LaTeX with as little adjustment as possible to the form in which they are typeset.

`numerica` is written in `expl3`, the programming language of the LaTeX3 project, now incorporated into the LaTeX kernel. It uses the LaTeX3 module `l3fp` (since early 2020 part of a standard LaTeX 2_ε distribution) as its calculational engine. This enables floating point operations to 16 significant figures, with exponents ranging between -10000 and $+10000$. Many functions and operations are built into `l3fp` – arithmetic operations, trigonometric, exponential and logarithm functions, factorials, absolute value, max and min. Others have been constructed for `numerica` from `l3fp` ingredients – binomial coefficients, hyperbolic functions, sums and products – but to the user there should be no discernible difference.

Associated packages provide for additional operations: iteration of functions, finding zeros of functions, recurrence relations, mathematical table building.

1.1 How to use `numerica`

The package is invoked in the usual way: put

```
\usepackage[<options>]{numerica}
```

in the LaTeX preamble. `numerica` requires the `amsmath` and `mathtools` packages and loads these automatically. `numerica` will also accept use of some relation symbols from the `amssymb` package (see §2.3.5), all commands from the `mleftright` package, and the `\sfrac` command from `xfrac` (part of the `l3packages` bundle), provided these last three packages have been loaded by the user.

1.1.1 Package options

Version 2 of `numerica` had no package options. The options available in version 1 that gave access to commands for iteration, finding zeros, math-table making, etc., were discontinued. That functionality became available in associated but separate LaTeX packages (see below §1.1.1.2). With version 3.0.0 some package options have been added and the possible use of a configuration file dispensed with. The current options available with version 3.0.0 are:

- `comma` If present, a decimal point is denoted by a comma (more exactly, an unspaced comma). If absent, a decimal point is denoted by a dot (period, full stop, also unspaced). The choice – `comma` present, `comma` absent – has consequences for the item separator in the variable=value list and n-ary functions (see §1.1.2), and the item separator in the main argument of the `\eval`, `\macros` and `\constants` commands; see below §1.1.2.
  - 'Out of the box' the `comma` option is not used and the decimal point is a dot.
- `rounding=<integer>` The rounding value. The value of `<integer>` determines how many digits after the decimal point are displayed in numerical results (see §2.3.1). 'Out of the box' the value is set to 6.
- `approx` sets the default relation linking *formula* and *result* in displays from `=` to `\approx` (displaying as ≈). (The `eq` setting (§5.3.2.2) is available to change the relation for individual calculations.)

Thus a possible invocation of `numerica` might be

```
\usepackage[comma,rounding=4,approx]{numerica}
```

meaning that the decimal point is an unspaced comma, the default rounding value is 4, and `\approx` is inserted between formula and numerical result in (some) displays. Alternatively,

```
\usepackage{numerica}
```

means the decimal point is an unspaced dot, the rounding value is 6, and the display of (some) results is in the form *formula=result*. **This is how `numerica` is invoked for the present document.**

1.1.1.1 `numerica.cfg`

Previous versions of `numerica` supported use of a configuration file for setting various default values. With version 3.0.0, this has been dispensed with. Now, `numerica` supports (currently) the three package options mentioned. On review, most of the `.cfg` settings did not feel like ones that realistically qualified as package-level settings. For calculation-level default values, see Chapter 5.

1.1.1.2 Associated packages

Currently there are two of these, `numerica-plus` and `numerica-tables`. They are loaded with the familiar `\usepackage` command in the document preamble and require `numerica` to be loaded. This is different from version 2 where calling `numerica-plus` or `numerica-tables` automatically loaded `numerica`. I think it is clearer to do this in two explicit steps. Neither package will function without `numerica` loaded. Thus putting

```
\usepackage[<options>]{numerica}
\usepackage{numerica-plus}
```

in the preamble of your document gives access to the commands `\nmcIterate`, `\nmcSolve`, and `\nmcRecur` of `numerica-plus` and of course also to the commands in `numerica`. `\nmcIterate` enables the iteration of functions of a single variable, including finding fixed points and, by means of Newton-Raphson iteration, finding zeros. `\nmcSolve` enables the solving of equations of the form $f(x) = 0$ (i.e. finding zeros) by bisection, or the finding of local maxima or minima of a function of one variable. `\nmcRecur` enables the calculation of terms in recurrence relations, like the terms of the Fibonacci series, or othogonal polynomials defined recurrently. In all three cases, see the associated document `numerica-plus.pdf` for details.

If you enter

```
\usepackage[<options>]{numerica}
\usepackage{numerica-tables}
```

in the preamble of your document you gain access to the command `\nmcTabulate`, which enables the creation of (possibly multi-column) tables of function values and makes available most of the table formats used in *HMF* (and also to the commands in `numerica`). See the associated document `numerica-tables.pdf` for details.

1.1.2 Decimal point and item separators

From version 3.0.0 of `numerica` the trio of marks `.,;` have different functions depending as the package is called without or with the `comma` option. Without the comma option, the decimal point is a dot (period, full stop) and the variable=value list (§2.2) is punctuated with commas. There is no ambiguity in a list like `[g=9.81,u=1,t=0.5]` nor in the arguments of n-ary functions (§3.5) like `\max`, `\min`, `\gcd`, e.g. `\gcd(63,231)`, although the presence of such functions in the vv-list needs protective braces, e.g. `[x={\min(\pi,e,\phi,\gamma)},y=2]`.

But if the decimal point is a comma, then its use as a separator in these lists is problematic. For that reason, with the `comma` package option, `numerica` uses a *semicolon* to punctuate the variable=value list and the argument lists of n-ary functions: `[g=9,81;u=1;t=0,5]`, `\max(6,1;2e;\gamma\pi^2)`. This is in line with ISO 80000 Part 2, section 3 which reads: 'A comma, semicolon or other appropriate symbol can be used as a separator between numbers or expressions.

The comma is generally preferred, except when numbers with a decimal comma are used.' However, rather than the gently permissive language of the standard, `numerica` *insists* on semicolons for separating items when the `comma` package option is used.

In summary, the trio of punctuation marks `.,;` are used in `numerica` like this: when the `comma` package option is *not* used, the marks function as

- `.` = decimal point
  - also LaTeX dot signifying 'no delimiter' when used with `\left`, `\right` etc.
- `,` = item separator in the variable=value list
  - also argument separator in n-ary functions (`\max`, `\min`, `\gcd`)
  - also formula separator in the main argument of `\nmcEvaluate` for multi-formula calculations

or, when the `comma` package option *is* used, as

- `.` = LaTeX dot signifying 'no delimiter' when used with `\left`, `\right` etc.
- `,` = decimal point
- `;` = item separator in the variable=value list
  - also argument separator in n-ary functions (`\max`, `\min`, `\gcd`),
  - also formula separator in the main argument of `\nmcEvaluate` for multi-formula calculations

Note, in both cases, that if a formula involves an n-ary function (at present only `\max`, `\min`, or `\gcd`) then it's argument will need to be hidden in braces to avoid being interpreted as containing multi-formula separators. Alternatively (and better) there is a setting that allows a different character to be used as the multi-formula separator for a calculation, e.g. `ff=|` ; see §1.1.5 and §5.2.3.

1.1.3 Basic procedure

A simple example of how `numerica` is used is provided by the document

```
\documentclass{article}
\usepackage{numerica}
\begin{document}
  \eval{$ mc^2 $}[m=70,c=299792458][8x]
\end{document}
```

There is a formula, `mc^2`, between math delimiters: `$ $`. A command `\eval{ }` is wrapped around these, and two square-bracketed optional arguments have been appended. In the first option numerical values are assigned to the quantities `m` and `c` occurring in the formula. The assignments are separated by a

comma since the dot is being used as the decimal point in this document. The second option contains a cryptic specification of the format of the numerical result – to 8 places of decimals, and in (proper) scientific notation – the `x`. Running `pdflatex` on this document generates a pdf displaying

$$mc^2 = 6.29128625 \times 10^{18}, \quad (m = 70, c = 299792458)$$

where the formula (mc^2) is equated to the numerical value resulting from substituting the given values of m and c. Those values are displayed in a list following the result. As specified, the result of the calculation is presented to 8 decimal places in scientific notation. (According to Einstein's famous equation $E = mc^2$ this is the enormous energy content, in joules, of what was once considered an average adult Caucasian male.)[2]

A second example is provided by the formula in earlier remarks:

```
\documentclass{article}
\usepackage{numerica}
\begin{document}
\begin{quote}
First, evaluate the expression when it sits between
textstyle delimiters,
\eval{\( \frac{\sin(3.5)}{2} + 2\cdot 10^{-3} \)},
and then, second, when it sits between
displaystyle delimiters:
\eval{\[ \frac{\sin(3.5)}{2} + 2\cdot 10^{-3} \]}
\end{quote}
\end{document}
```

Running `pdflatex` on this document produces the result:

> First, evaluate the expression when it sits between textstyle delimiters, $\frac{\sin(3.5)}{2} + 2 \cdot 10^{-3} = -0.173392$, and then, second, when it sits between displaystyle delimiters:
>
> $$\frac{\sin(3.5)}{2} + 2 \cdot 10^{-3} = -0.173392$$

(For a quick mental check of the result, note that $\sin(3.5) \approx -0.35$.) The `\eval` command used in these examples is the main command of the `numerica` package and is discussed fully in the next two chapters, but I first discuss different ways to display the results of calculations.

[2] In earlier versions of `numerica` this calculation evaluated incorrectly because spaces were used to make `c=299792458` more 'eye friendly'. Although numbers with spaces can now be read by `numerica`, this ability needs to be turned on by the user. 'Out of the box' it is off; see §2.1.2.

1.1.4 Display of the result

In what follows I shall write things like (but generally more complicated than)

$$\texttt{\$ \textbackslash eval\{ 1+1 \} \$} \Longrightarrow 2$$

to mean: run `pdflatex` on a document containing `$ \eval{ 1+1 } $` in the document body to generate a pdf containing the calculated result – 2 in this instance, as indicated by the arrow. The reader will note that I have used dollar signs to delimit the math environment. I could (and perhaps should) have used the more LaTeX-pure `\( \)`, but habit has won out.

In the example the `\eval` command is used *within* a math environment (delimited by the dollar signs).

- When the `\eval` command is used *within* a math environment, only the *numerical result,* followed possibly by the *variable=value list*, is displayed (within the given math environment).

For the *variable=value list* see the mc^2 example earlier where values were assigned to m and c in a trailing optional argument; this is discussed more fully in §2.2. As a simple example, I repeat the previous addition with variables x and y:

$$\texttt{\$ \textbackslash eval\{ x+y \}[x=1,y=1] \$} \Longrightarrow 2$$

(If the package option `comma` were being used, setting the comma as the decimal point, then the example would look like `$ \eval{ x+y }[x=1;y=1] $` with a semicolon separating the variable assignments.)

To my eye, display of the variable=value list in this example looks silly. It needs context, some prior commentary or statement of the formula like

$$\texttt{\$ x+y=\textbackslash eval\{ x+y \}[x=1,y=1] \$} \Longrightarrow x+y=2$$

Otherwise display of the variable=value list can be suppressed, most simply by appending a star (asterisk) to the `\eval` command; see §2.2.4.2 or later in this section. Environments may include the standard LaTeX inline (`$ $` or `\( \)` or `math`) environments, the `displaymath`, `\[ \]` and `equation` environments, the `eqnarray` environment, as well as the AMS environments which come into their own when multi-formula calculations are performed, or when long formulas with many variables are involved (`multline`). Examples will recur throughout this document.

`\eval` is not limited to use within a math environment. As we have already seen with the mc^2 example, it can also wrap *around* math delimiters:

$$\texttt{\textbackslash eval\{\$ x+y \$\}[x=1,y=1]} \Longrightarrow x+y=2, \quad (x=1, y=1)$$

When it does, the display that results is different, as you can see. The formula is *automatically* included in the display; it does not need to be written in 'by hand' as I did in the previous example.

- When the `\eval` command is wrapped *around* a math environment, the result is displayed within that environment in the form, *formula=numerical result*, followed possibly by the *variable=value list*.
  - If the formula is long or contains many variables then it may be desirable to split the display over two lines; see the `multline*` example below, and §2.2.4.1.
- An alternative to explicitly wrapping `\eval` around math delimiters is to use the *settings* option, an optional argument before the main (mandatory) argument, and enter `env=<environment>` there, for example like this:

  `\eval[env=$]{x+y}[x=1,y=1]` $\Longrightarrow x+y=2, \quad (x=1, y=1)$

which reproduces the previous display. Doing this is more convenient when more 'verbose' environment names than the brief `$ $`, `\( \)` or `\[ \]` are used (although you can write, say, `\eval{\begin{multline*}...\end{multline*}}` if so inclined). Here is an example of a `multline*` environment being used to 'tame' the display of a long unwieldy formula (the phantom is there so that the hanging + sign spaces correctly):

```
\eval[env=multline*]
   { 1+2+3+4+5+6+7+8+9+10+\phantom{0}\\
        11+12+13+14+15+16+17+18+19 }
```

$\Longrightarrow$

$$1+2+3+4+5+6+7+8+9+10+ \\ 11+12+13+14+15+16+17+18+19=190$$

Note how the phantom and the new line command `\\` are swallowed by `\eval` without complaint.

- It is also possible to dispense with math delimiters entirely, neither wrapped within the `\eval` command nor wrapped around it nor invoked with the `env` setting, in which case `numerica` displays the *numerical result* between `$` delimiters.

This is different from the behaviour in earlier versions of `numerica` when no math delimiters were involved. Then, `\[ \]` delimiters were wrapped around the *numerical result*, possibly followed by the *variable=value* list. Now only the *numerical result* is displayed and, since the absence of the variable=value list means no mathematical constructs like `\frac`-tions are present, inline delimiters seem more appropriate than the previously used `\[ \]`.

```
\begin{quote}
The result of subtracting $e^\pi$ from $\pi^e$ is
```

```
\eval{ \pi^e-e^\pi }
which is negative; hence $e^\pi>\pi^e$.
\end{quote}
```

is an example of `\eval` used in the absence of delimiters and produces the result:

> The result of subtracting e^π from π^e is -0.681535 which is negative; hence $e^\pi > \pi^e$.

Note that the minus sign displays correctly because of the `$` delimiters automatically inserted by `numerica`.

- What is displayed can be pared to the minimum by appending an asterisk to the `\eval` command. Then, only the *numerical result* is displayed, with no math delimters added; if `\eval*` is used within a math environment, the numerical result will be displayed accordingly, but otherwise the result will be displayed as text, a negative sign displaying as a hyphen.

Compare `\eval*{ \pi^e-e^\pi }` $\Longrightarrow$ -0.681535 with the previous example. In the present example only the numerical result is displayed – as text, with a hyphen depicting the minus sign. It is up to you, the user, to provide the surrounding math environment if you want a proper minus.

1.1.4.1 Punctuation: the p setting

To complete a display you may wish to add a punctuation mark – usually a comma or full stop – after the displayed expression. For inline use punctuation is easy: add the punctuation mark after the `\eval` command and its arguments: `\eval{$ 2\pi $},` $\Longrightarrow 2\pi = 6.283185,$ and `$\eval{ 1+x+y }[x=2,y=3]$.` $\Longrightarrow 6$. The mark appears in the right place.

For displaystyle environments punctuation is not so straightforward. When `\eval` is used *within* a displaystyle environment, say between `\[ \]` delimiters, it is easy to add a comma or full stop after the `\eval` command and its arguments but before the closing delimiter, `\[ x+1=\eval{ x+1 }[x=1],\]` and the punctuation mark will appear in the right place. But when the `\eval` command wraps *around* `\[ \]` delimiters or the `env=\[` option is used, a problem arises: a fullstop or comma after the `\eval` command and its arguments – `\eval{\[ 1+1 \]}.` – lies *beyond* the closing delimiter and will slide off to the start of the next line, *after* the displayed result. We want it to display as if it were the last element *before* the closing delimiter.

Explicitly putting it there, like `\eval{\[ 1+1.\]}`, means the punctuation mark becomes part of the formula. Potentially `numerica` then needs to check not just for a fullstop but also other punctuation marks like comma, semicolon, perhaps even exclamation and question marks. All these marks have roles in mathematics or `l3fp` or `numerica`, and the program responds to them accordingly. For instance a full stop is also the decimal point mark and is treated as such (giving the rather cryptic result `\eval{$ . $}` $\Longrightarrow . = 0$ since the solitary dot is interpreted as the number 0.0). An exclamation mark is the

factorial sign; `numerica` recognizes it as such: `\eval{$ 4! $}` $\Longrightarrow 4! = 24$. A comma is used to separate the arguments of n-ary functions like `\max` and `\min`: `$\eval{\min(\pi,e,\phi,\gamma)}$` $\Longrightarrow 0.577216$. And as we will see below, a semicolon is used by `numerica` to separate expressions in a multi-formula calculation. Distinguishing the punctuation role from the mathematical role of these marks would only complicate the code and slow evaluation.

Instead, `numerica` uses a key in the settings option to add punctuation. As already noted, the settings option is an optional argument preceding the main argument, already met in relation to the `env`-ironment key. A second setting is the punctuation key `p`. Indeed, simply entering `p`, as here,

```
\eval[p]{\[ 1+x+y+z+xy+yz+zx+xyz \]}[x=2,y=3,z=4]
```

$\Longrightarrow$

$$1 + x + y + z + xy + yz + zx + xyz = 60, \qquad (x = 2, y = 3, z = 4),$$

puts a comma in the correct place, after the closing parenthesis of the variable=value list. If a full stop is wanted use `p=.`:

```
\eval[env=\[,p=.]{ (1+x)(1+y)(1+z) }[x=2,y=3,z=4]
```

$\Longrightarrow$

$$(1 + x)(1 + y)(1 + z) = 60, \qquad (x = 2, y = 3, z = 4).$$

Again the mark appears in the right place. As you can see, the settings option is a comma-separated list of *key=value* pairs. This remains true of the settings option even when the `comma` package option is used, since the only numbers appearing in the settings are integers – ambiguity does not arise. That entering `p` alone sufficed is because the punctuation key `p` defaults to a comma. If you want some other mark – a semicolon or exclamation mark perhaps – equate `p` to that mark in the optional argument. The default value for `p` does not change if the `comma` package option is used.

The comma is chosen as the default not only because it is a commonly used mark at the end of equations but because the settings option is a comma-separated list. By making the default the comma, it suffices to write `p` when you want a comma – rather than the more awkward `p={,}` which would otherwise be required.

1.1.5 Multi-formula calculations

It is possible to evaluate more than one expression at a time in the one `\eval` command by means of a further setting `ff`, or `ff=<char>`. The default separator of one formula from the next is the same as that used in the variable=value list. If the decimal point is a dot, then the default separator is a comma; if the decimal point is a comma (with the `comma` package option) then the default

separator is a semicolon. In both cases this will generally be fine except when n-ary functions are involved since their arguments are delimited in the same way. Either one can wrap the arguments of these functions in braces or, better, choose a different separator by means of the setting `ff=<char>`. I give an example shortly below; also see §5.2.3. For this document the decimal point is a dot and the comma generally suffices.

Suppose we want the values of the main trigonometric functions at, say, $\pi/6$. In the example, I have entered the functions separated by commas, assigned the value $\pi/6$ to the variable x in the trailing optional argument and concluded the display with a full stop by means of the `p=.` setting. The point to note is the `ff` in the settings option, signalling a multi-formula calculation:

```
\eval[ff,p=.]{\[ \sin x, \cos x, \tan x \]}[x=\pi/6]
```

$\Longrightarrow$

$$\sin x = 0.5, \qquad (x = \pi/6)$$
$$\cos x = 0.866025, \qquad (x = \pi/6)$$
$$\tan x = 0.57735, \qquad (x = \pi/6).$$

To understand these values we might add

```
\eval[ff,p=.]{\[ \surd3/2, 1/\surd3 \]}
```
$\Longrightarrow$

$$\sqrt{3}/2 = 0.866025$$
$$1/\sqrt{3} = 0.57735.$$

The displays in both instances follow the default format for a multi-formula calculation in the `equation*` (`\[`) environment.

There are irritants: the failure of the equals signs to line up, the repetition of the variable=value list, the different numbers of digits displayed in the answers and, if you are a 'punctuator', you might like commas to terminate the intermediate rows. For back-of-envelope calculations, who cares? But for inclusion in more formal documents such things matter. All can be remedied: see §2.3.2 (and below) about padding numbers with zeros to a given number of decimal places; see §5.3.2 about use of AMS environments to guarantee alignment; see §5.3.2.6 about suppressing repetition of the variable=value list. Punctuating the intermediate rows I discuss next.

1.1.5.1 Punctuation: the `pp`

Like the `p` setting, there is a `pp` setting also entered in the settings argument of the `\eval` command that enables the insertion of punctuation at the end of intermediate results in a multi-formula calculation; the `p` setting still determines the terminating punctuation mark. Like `p`, `pp` defaults to a comma, which means that you need enter only `pp` to insert commas. This remains true if the `comma` package option is used. For any other mark you need to equate `pp` to that

mark – e.g. `pp=;`. Thus, repeating the first of the examples above, I've added `pp` to the settings option, and also added a star (asterisk) in a second trailing argument on the right. This triggers padding the numerical result with zeros, should it not display the default six decimal places – as happens with both $\sin x$ and $\tan x$ (because they round to five or fewer figures).

```
\eval[ff,pp,p=.]{\[ \sin x, \cos x, \tan x \]}[x=\pi/6][*]
```

$\Longrightarrow$

$$\begin{aligned}\sin x &= 0.500000, \qquad (x = \pi/6),\\ \cos x &= 0.866025, \qquad (x = \pi/6),\\ \tan x &= 0.577350, \qquad (x = \pi/6).\end{aligned}$$

The display is improved: commas terminate intermediate rows, a full stop is at the end, and by padding with zeros, all three results display six decimal places and now align vertically.

1.1.5.2 Multi-formula separator: the `ff` setting

That the last example displays as desired depends on all three results being positive. If $x = 5\pi/6$, $\cos x$ and $\tan x$ will be negative and minus signs will destroy the alignment. The secret then is to use an `alignat*` environment. Also, just to show how it's done, I've changed the multi-formula separator with the setting `ff=|`. (Even if a formula contained an absolute value it should not contain the `|` character but rather `\abs` or `\lvert`, `\rvert`; see §3.6.)

```
\eval[pp,p=.,env=alignat*,ff=|]
  {\[ \sin x | \cos x | \tan x \]}[x=5\pi/6][*]
```

$\Longrightarrow$

$$\begin{aligned}\sin x &= 0.500000, \qquad (x = 5\pi/6),\\ \cos x &= -0.866025, \qquad (x = 5\pi/6),\\ \tan x &= -0.577350, \qquad (x = 5\pi/6).\end{aligned}$$

Padding wit zeros and the `alignat*` environment have ensured alignment of the numerical results.

Multi-formula calculations can also be performed in an inline context. In the following example, the `p` setting has been dispensed with since a full stop can be inserted at the end 'by hand' without problem. Again, just to show how, I have used the `ff=<char>` setting (although `&` is a confusing character to use for anyone familiar with LaTeX):

```
$\eval[pp,ff=&]{ \pi & \pi/2 & 1/\pi & \surd\pi }$ ⟹
```

$$3.141593, \quad 1.570796, \quad 0.31831, \quad 1.772454.$$

By default, a quad of space is inserted between results of a multi-formula calculation in an inline (`$ $`, `\( \)` or `math`) context, as evident here. This can be changed by means of the `sep` (for separator) setting (§5.3.2.4). For more on environments and their tweaks, see §5.3.2.

1.1.6 Examples of use

To give a sense of how `numerica` can be used, I include some examples of actual use of the program. (The `numerica-plus` and `numerica-tables` packages contain others.)

1.1.6.1 Checking

Occasionally, just to reassure myself that age hasn't completely rotted my brain I like to tackle short mathematical problems I come across on the internet. One that caught my attention was to simplify $\sqrt{220-30\sqrt{35}}$. After some bumbling and fumbling, I let

$$x=\sqrt{220-30\sqrt{35}},\qquad y=\sqrt{220+30\sqrt{35}},$$

(which seems an obvious thing to do) so that

$$xy=10\sqrt{484-315}=10\sqrt{169}=10\sqrt{13^2}=130.$$

Since $x^2+y^2=440$ it was easy to form both $(x+y)^2$ and $(x-y)^2$, and by separating the resulting numbers into their prime factors, to find $y+x$ and $y-x$ and work out that $x=5\sqrt{7}-3\sqrt{5}$. Was I right, or had I made a mistake? Since

```
\eval[p,pp,ff]{\[\sqrt{220-30\sqrt{35}},
               5\sqrt{7}-3\sqrt{5}\]}
```

$\Longrightarrow$

$$\sqrt{220-30\sqrt{35}}=6.520553,$$
$$5\sqrt{7}-3\sqrt{5}=6.520553.$$

the simplification was correct. Indeed $y=5\sqrt{7}+3\sqrt{5}$:

```
\eval[p=.,pp,ff]{\[\sqrt{220+30\sqrt{35}},
                  5\sqrt{7}+3\sqrt{5}\]}
```

$\Longrightarrow$

$$\sqrt{220+30\sqrt{35}}=19.93696,$$
$$5\sqrt{7}+3\sqrt{5}=19.93696.$$

As a final flourish,

```
\eval{$xy$}[x=5\sqrt{7}-3\sqrt{5},
            y=5\sqrt{7}+3\sqrt{5}]
```

$\Longrightarrow xy=130,\quad (x=5\sqrt{7}-3\sqrt{5}, y=5\sqrt{7}+3\sqrt{5}).$

1.1.6.2 Exploring

When working on `numerica`'s predecessor package, I constantly tested it against known results to check for coding errors. One test was to ensure that

$$\left(1+\frac{1}{n}\right)^n$$

did indeed converge to the number e as n increased. Let's do that here. Try first $n = 10$:

```
\eval{$ e-(1+1/n)^n $}[n=10][x] ⟹
```

$$e-(1+1/n)^n = 1.245394\times10^{-1}, \quad (n=10).$$

(The default number of decimal places displayed is 6.) The difference between e and $(1+1/n)^n$ is about an eighth (0.125) when $n = 10$, which is encouraging but hardly decisive. The obvious thing to do is increase the value of n. I'll use an `align*` environment to 'prettify' the presentation of the results. Although looking like a solid block of typing, most of the following was done by copy-and-paste; I only had to change the exponent on the `10`:

```
\begin{align*}
  e-(1+1/n)^{n} &= \eval{e-(1+1/n)^n}[n=1\times10^5][*x],\\
  e-(1+1/n)^{n} &= \eval{e-(1+1/n)^n}[n=1\times10^6][*x],\\
  e-(1+1/n)^{n} &= \eval{e-(1+1/n)^n}[n=1\times10^7][*x],\\
  e-(1+1/n)^{n} &= \eval{e-(1+1/n)^n}[n=1\times10^8][*x].
\end{align*}
```

This gave the result

$$\begin{aligned}
e-(1+1/n)^n &= 1.359128\times10^{-5}, && (n=1\times10^5),\\
e-(1+1/n)^n &= 1.359140\times10^{-6}, && (n=1\times10^6),\\
e-(1+1/n)^n &= 1.359141\times10^{-7}, && (n=1\times10^7),\\
e-(1+1/n)^n &= 1.359141\times10^{-8}, && (n=1\times10^8).
\end{aligned}$$

Clearly $(1+1/n)^n$ converges to e, the difference between them being of order $1/n$, but that is not what catches the eye. There is an unanticipated regularity here. 1.35914? Double the number: `\eval{2\times 1.35914}[5]` ⟹ 2.71828 which looked like e to me and suggested a relationship, namely,

$$\lim_{n\to\infty} n\left(e-\left(1+\frac{1}{n}\right)^n\right) = \tfrac{1}{2}e.$$

I hadn't seen this before. Is it true? Since

$$\ln\left(1+\frac{1}{n}\right)^n = n\ln\left(1+\frac{1}{n}\right),$$

it followed from the familiar expansion of the logarithm that

$$\ln\left(1+\frac{1}{n}\right)^n = n\left(\frac{1}{n}-\frac{1}{2}\frac{1}{n^2}+\frac{1}{3}\frac{1}{n^3}-\dots\right)$$
$$= 1-\frac{1}{2n}\left(1-\frac{2}{3}\frac{1}{n}+\frac{2}{4}\frac{1}{n^2}-\right).$$

Write E_n for the bracketed series on the right. E_n is an alternating series and the magnitudes of the terms of the series tend to 0 monotonically. Hence $1 > E_n > 1 - 2/3n$ and $E_n \to 1$ as $n \to \infty$. Now exponentiate:

$$\left(1+\frac{1}{n}\right)^n = e \times e^{-E_n/2n},$$

so that

$$n\left(e-\left(1+\frac{1}{n}\right)^n\right) = ne\Big(1-e^{-E_n/2n}\Big).$$

The proposed limit, new to me, now followed from the standard inequality (see *HMF* 2.3.32), $x/(1+x) < 1 - e^{-x} < x$ when $x > -1$.

1.1.6.3 Reassuring

In the course of some hobbyist investigations in plane hyperbolic geometry I derived the formula

$$\Phi_1(m,n) = \cos\tfrac{m}{n}\pi - (1-4\sin^2\tfrac{m}{3n}\pi)\frac{\sin\frac{1}{n}\pi\sin\frac{m-1}{n}\pi}{2\sin^2\frac{m}{3n}\pi},$$

for $m = 2, 3, \dots$ and integral $n \geq 2m+1$. A key concern was: when is Φ_1 positive? Φ_1 itself was opaque; could I work it into an equivalent but more enlightening form? After an embarrassingly laborious struggle, I derived the expression

$$\Phi_2(m,n) = \left(\frac{1-4\sin^2\frac{m}{3n}\pi}{2\sin^2\frac{m}{3n}\pi}\right)\sin\tfrac{2m-3}{3n}\pi\sin\tfrac{m-3}{3n}\pi,$$

in which the conditions for positivity were now clear: with $n \geq 2m+1$, so that $m\pi/3n < \pi/6$, the first parenthesized factor is always positive; the second is positive for $m \geq 2$, and the third is positive for $m \geq 4$. All well and good, but given the struggle to derive Φ_2, was I confident that Φ_1 and Φ_2 really are equal? It felt all too likely that I had made a mistake.

The simplest way to check was to see if the two expressions gave the same numerical answers for a number of m, n values. First I checked for $m = 2, n = 5$: I wrote `\eval{\[ \]}[m=2,n=5]` twice and between the delimiters pasted the already composed expressions for Φ_1 and Φ_2, namely:

```
\eval{\[
        \cos\tfrac{m}{n}\pi-(1-4\sin^{2}\tfrac{m}{3n}\pi)
        \frac{\sin\tfrac{1}{n}\pi\sin\tfrac{m-1}{n}\pi}
        {2\sin^{2}\tfrac{m}{3n}\pi}
      \]}[m=2,n=5]
\eval{\[ \left(
           \frac{1-4\sin^{2}\tfrac{m}{3n}\pi}
           {2\sin^{2}\tfrac{m}{3n}\pi}
         \right)
         \sin\tfrac{2m-3}{3n}\pi\sin\tfrac{m-3}{3n}\pi
      \]}[m=2,n=5]
```

I have added some formatting – indenting, line breaks – to make the formulas more readable but otherwise left them unaltered. The `\eval` command can be used for even quite complicated expressions without needing to tinker with their LaTeX form, but you may wish – as here – to adjust white space to clarify their component parts. Running `pdflatex` on these expressions, the results were

$$\cos\tfrac{m}{n}\pi-(1-4\sin^{2}\tfrac{m}{3n}\pi)\frac{\sin\tfrac{1}{n}\pi\sin\tfrac{m-1}{n}\pi}{2\sin^{2}\tfrac{m}{3n}\pi}=-0.044193,\qquad(m=2,n=5)$$

$$\left(\frac{1-4\sin^{2}\tfrac{m}{3n}\pi}{2\sin^{2}\tfrac{m}{3n}\pi}\right)\sin\tfrac{2m-3}{3n}\pi\sin\tfrac{m-3}{3n}\pi=-0.044193,\qquad(m=2,n=5)$$

which was reassuring. (The result is negative since $m-3<0$.)

I could have avoided the double writing of `\eval` and `[m=2,n=5]` by putting a comma between the expressions and performing a multi-formula calculation. This time I've checked equality for `m=5` and `n=13`, which should give a positive result, and I've taken the opportunity to `align*` the results with the `env` setting:

```
\eval[p=.,pp,env=align*,ff]
  { \cos\tfrac{m}{n}\pi-(1-4\sin^{2}\tfrac{m}{3n}\pi)
    \frac{\sin\tfrac{1}{n}\pi\sin\tfrac{m-1}{n}\pi}
    {2\sin^{2}\tfrac{m}{3n}\pi}
  , \left(
      \frac{1-4\sin^{2}\tfrac{m}{3n}\pi}
      {2\sin^{2}\tfrac{m}{3n}\pi}
    \right)
    \sin\tfrac{2m-3}{3n}\pi\sin\tfrac{m-3}{3n}\pi
  }[m=5,n=13]
```

which evaluates to

$$\cos\tfrac{m}{n}\pi - (1-4\sin^2\tfrac{m}{3n}\pi)\frac{\sin\tfrac{1}{n}\pi\sin\tfrac{m-1}{n}\pi}{2\sin^2\tfrac{m}{3n}\pi} = 0.107546, \qquad (m=5, n=13),$$

$$\left(\frac{1-4\sin^2\tfrac{m}{3n}\pi}{2\sin^2\tfrac{m}{3n}\pi}\right)\sin\tfrac{2m-3}{3n}\pi\sin\tfrac{m-3}{3n}\pi = 0.107546, \qquad (m=5, n=13).$$

Thus reassured that there was *not* an error in my laborious derivation of Φ_2 from Φ_1, it was not difficult to work back from Φ_2 to Φ_1 then reverse the argument to find a straightforward derivation.

Chapter 2

\nmcEvaluate (\eval)

The main calculational command in `numerica` is `\nmcEvaluate`. Because this would be tiresome to write too frequently, particularly for back-of-envelope calculations, there is an equivalent short-name form, `\eval`, used almost exclusively in this document. But wherever you see `\eval` you can substitute `\nmcEvaluate` and obtain the same result. `\eval` is defined using `\ProvideDocumentCommand` from the `xparse` package. Hence if already defined in some other package already loaded, it will not be redefined by `numerica`. It will retain its meaning in the other package. Its consequent absence from `numerica` may be an irritant, but only that. `\nmcEvaluate` is unlikely to be defined elsewhere and should still be available.

2.1 Syntax of \nmcEvaluate (\eval)

`\nmcEvaluate` (or `\eval`) takes five arguments of which only the third is mandatory. All others are optional. If all are deployed the command looks like

```
\nmcEvaluate*[settings]{expr.}[vv-list][num. format]
```

I discuss the various arguments in the referenced sections.

1. `*` optional *number-only* switch; if present ensures display of only the numerical result, as text with no formatting; see §2.2.4.2;
2. `[settings]` optional comma-list of *key=value settings* of the calculational environment for this particular calculation; see §5;
3. `{expr.}` mandatory main argument, the mathematical *expression* in LaTeX form that is to be evaluated, or a list of such expressions; see §2.1.1;
4. `[vv-list]` optional list of *variable=value* entries; see §2.2;
5. `[num. format]` optional *number-format* specification for the numerical result (rounding, padding with zeros, scientific notation, boolean or fraction-form output); see §2.3.

Note that arguments 4 and 5 are both square-bracket delimited optional arguments. Should only one such argument be used, `numerica` determines which is intended by looking for an equals sign within the argument. Its presence indicates the argument is the vv-list; its absence indicates the argument is the number-format specification.

The vv-list and number-format specification are trailing optional arguments but do not need to be hard against their preceding arguments; intervening spaces are allowed. This means there is a possibility that should the `\eval` command be followed by a square-bracketed mathematical expression that expression might be interpreted as a trailing argument. Experience using `numerica` suggests that this will be a (very) rare occurrence and is easily prevented by inserting an intervening empty brace pair (`{}`). By allowing spaces between the arguments complicated expressions and large vv-lists can be formatted, in the interests of clarity, with new lines and white space without requiring the insertion of line-ending comment characters (`%`).

Recommended practice is to minimise the number of optional arguments used in LaTeX commands by consolidating them into a single *key=value* list. Although `numerica` uses such an argument (the settings optional argument), the vv-list does not fit naturally into that scheme. And practice suggests that separating out the elements of the number-format specification of the result and placing them in a trailing argument is both convenient and intuitive for the kind of back-of-envelope calculations envisaged for `numerica`.

2.1.1 Expressions

What kind of formula or expression can be 'digested' by `\nmcEvaluate`? As seen above (§1.1.3), a formula can be complicated, including components like $2\sin^2\frac{m}{3n}\pi$ or

$$\left(\frac{1-4\sin^2\frac{m}{3n}\pi}{2\sin^2\frac{m}{3n}\pi}\right),$$

but the underlying aim is always: if the meaning of a formula in the `pdf` is clear to a human reader, it ought to be clear to `numerica`. In a perhaps surprising number of cases this aim can be met. Mathematicians understand an expression like $\sin 2\pi x$ to mean the sine of the triple product $2\pi x$; so does `numerica`. Mathematicians casually use and understand logically wrong but customary notations like $\sin^2 x$, the square of the sine of x; `numerica` digests this without fuss. Mathematicians use a wide variety of formatting commands to clarify their intent: `\left` and `\right`, `\phantom`-s, spaces and new lines (`\quad`, `\\`), structural commands like `\mathstrut`, or environments like `align` or `multline`; all are grist to `numerica`'s mill.

2.1.1.1 Multi-formula expressions

From version 3.0.0 of `numerica` the main (and only mandatory) argument of the `\eval` command may contain more than one formula to be evaluated. 'Expression' can now mean a (generally short) list of formulas. The default punctuation

mark separating one formula from the next in the list is the same as that used in the vv-list – a comma if the decimal point is a dot, a semicolon if the decimal point is a comma. The only potential conflict is if a formula contains an n-ary function (`\max` or `\min` or `\gcd`), since their arguments are separated by the same separators in the two cases. There are two responses. One is to wrap the n-ary function in braces. The other is to specify a different multi-formula delimiter in the settings option. This is done by entering `ff=<char>` there (see §5.2.3), where `<char>` is some suitably 'neutral' character not otherwise present in any of the formulas to be evaluated – perhaps `@`, or `|`.

Examples of multi-formula expressions being evaluated within the one `\eval` command were seen earlier in the Introduction, especially at §1.1.5, and §1.1.6.3. A multi-formula calculation is a natural way to check identities – see for example the test of $\sinh 3x = 3\sinh x + 4\sinh^3 x$ in §3.4.5. Numerous other examples occur throughout this document.

2.1.1.2 LaTeX braces and mathematical arguments

There are mathematical braces, `\{ \}`, which display in the `pdf` and are used to delimit (generally larger) parts of mathematical expressions, and there are LaTeX braces, `{ }`, which do not display in the `pdf` and are used to delimit LaTeX arguments or groupings. This discussion is about LaTeX braces.

Generally, the LaTeX braces `\eval` encounters should be 'announced' by a preceding LaTeX command. Thus the braced argument in `\sqrt{x^2+1}`, displaying in a math environment as $\sqrt{x^2+1}$, is 'announced' by the square root command. Similarly, `\frac` and `\binom` each announce two braced arguments. The superscripting `^` or subscripting `_` also announce a braced argument (in general). In these cases `\eval` knows what to do with the braced argument because it is prepared by the preceding command.

Although there is no LaTeX requirement for them, braced arguments can also be used after unary functions like `\sin` or `\ln` or `\tanh`. Given the presence of the unary function, `\eval` knows what to do with an immediately following braced argument and will happily digest it – it has been announced by the unary function. Indeed, from version 3.0.0 of `numerica` this is the recommended way of handling (for instance) the more complicated arguments that frequently occur following `\sin` and `\cos` in the study of Fourier series; see §3.4.1.1.

Even *without* braces, `\eval` will happily digest an argument to a unary function that is the product of a number, a variable, a constant, a `\tfrac` (or an `\sfrac` if `xfrac` is loaded) or some subset thereof:

`\eval{$ \cos \tfrac1{12}2n\pi $}[n=2]` $\Longrightarrow \cos \frac{1}{12}2n\pi = 0.5, \quad (n = 2).$

LaTeX braces are for those situations where the reader sees the function's argument extending *beyond* the point where a programming rule would end the argument. For example, a reader knows that the argument of the sine in $\sin \frac{1}{2}(A+B)$ does not end with the $\frac{1}{2}$, nor with the right parenthesis in $\sin(n+\frac{1}{2})\pi$ nor with the first factor in $\sin(n+\frac{1}{2})(x-t)$. It is for situations like this that em-bracing the argument is recommended. It makes no difference to the visual appearance,

hence does not interfere with the reader's comprehension, but informs `numerica` of exactly where the argument ends.

Because they are invisible in the `pdf`, LaTeX braces should never be used to shorten what the reader sees as the argument of a function. `\eval` assumes that the braced part is the *whole* argument but the reader doesn't read that. For example, presented with `\sin{2n}\pi`, `\eval` assumes the sine's argument is $2n$ and does not extend to `\pi`. This is not what a human reads in the `pdf`. The compiled expression, $\sin 2n\pi$, is read as 'the sine of $2n\pi$'. If the intention really is to multiply π by $\sin 2n$ then the reader needs to *see* that this is so: $(\sin 2n)\pi$ perhaps or $\sin 2n \times \pi$ or, best, $\pi \sin 2n$, but *not* by means of LaTeX braces which leave no visual trace in the `pdf`.

2.1.1.3 Unannounced braces

Unannounced braced expressions should be used with care. When `\eval` meets an unnanounced brace group it is 'flying blind'. LaTeX braces are 'punctuation marks' for LaTeX code, not for mathematical formulas. How could they be, since they do not display in the `pdf`? For `numerica` it is *how things look in the* ***`pdf`*** that is the guide.

`\eval` converts an unannounced braced expression into its corresponding `l3fp` form and appends that to the overall expression that is to be evaluated. It does not do anything further. Note in particular that it does not first evaluate the braced expression and append the result to the overall expression, nor does it parenthesize the `l3fp` form of the braced expression, nor does it check to see if a multiplying asterisk `*` should be appended or prepended to the `l3fp` form. It simply converts the braced expression into its `l3fp` form and appends.

This works fine if, as suggested above, the braces surround a function like `\max(x,y,z)` so it can be included in a multi-formula calculation, but it can give unexpected results in other contexts. Thus (math braces; LaTeX braces),

```
\eval[ff]{\[ \{ 1+2 \}^2, { 1+2 }^2 \]} ⟹
```

$$\{1+2\}^2 = 9$$

$$1+2^2 = 5$$

In the first of these `eval` reads the math-braced expression, converts it into its `l3fp` form and appends that, which *includes the math braces*, to the (empty at this stage) overall expression. In the second of these, `\eval` reads the braced expression, converts that to its `l3fp` form and appends the result to the (empty at this stage) overall expression. In both cases, `\eval` proceeds to read and append `^2` to the overall expression so that the superscript acts on a bracketed expression in the first case and on 2 alone in the second.

A second example is

```
\eval[ff]{\[ 3\{ 2+1 \}, 3{ 2+1 } \]} ⟹
```

$$3\{2+1\} = 9$$

$$32+1 = 33$$

Similarly `{ 2+1 }\tfrac13` displays like $2+1\frac{1}{3}$ which one might naively expect to evaluate to 3.333333 but in fact it evaluates to 2.333333 since `l3fp` uses juxtaposition to mean *multiply*. Because LaTeX braces do not visibly display, this kind of result is all too likely. Unless there is some compelling reason to do otherwise avoid *unannounced* brace groups in expressions except to 'hide' functions containing problematic characters (like the argument separator in n-ary functions in some contexts).

There is also a quirk (or feature) of `l3fp` that could catch one out. An expression of the form `(1)1` produces a LaTeX error when `l3fp` tries to evaluate it. In `l3fp` a number can be placed directly *before* a parenthesized expression but not after; `1(1)` and `(1)1` are read very differently by the program. In normal use, 'behind the scenes' `numerica` takes care of this quirk; a user should never have to worry about it. But this internal coping mechanism is not activated when the `l3fp` form of a braced group is appended to the overall expression.

2.1.2 Numbers

In the present document the numerous examples presented mainly use ordinary decimal notation for numbers, but scientific notation is also available, both for input (see §5.2.1) and output (see §2.3.3).

2.1.2.1 Decimal point

Prior to version 3.0.0 the only decimal marker accepted was the dot (period, full stop). Now, by means of the `comma` package option, the comma can also be used as the decimal point. This means using semicolons in the variable=value list (see below §2.2) and for separating the arguments of n-ary functions (§3.5) in line with ISO 80000 Part 2, section 3 which reads: 'A comma, semicolon or other appropriate symbol can be used as a separator between numbers or expressions. The comma is generally preferred, except when numbers with a decimal comma are used.' When the `comma` package option is used, `numerica` *insists* on semicolons for separating items in the two contexts mentioned.

2.1.2.2 Grouping blocks of digits

Sometimes, to make numbers more readable, blocks of digits are grouped together, separated perhaps by spaces or by an explicit mark like a comma. `numerica` can digest numbers in which the grouping is by *spaces*, but not a mark. 'Out of the box', however, the package does not expect spaces in numbers. This needs to be turned on by the user by entering `1s2` (or `1s2=1`; `1s2` indicating a space 's' between digits) in the settings option:

`\eval[1s2]{ 12 345.678 901 }` $\Longrightarrow$ 12345.678901

`l3fp` works to 16 significant figures; so does `numerica`. But experience suggests that in 'everyday' use significantly fewer digits are generally used, both as input

and output. The cases in which grouping digits aids readability will be rare. For this reason, 'out of the box', `1s2=0`.

Note that `1s2=0` should not cause a LaTeX error nor raise a message in `numerica` if a number containing spaces is fed to `\eval`, but the result may be disconcerting:

`\eval[1s2=0]{ 12 34 }` $\Longrightarrow 408$

What has happened in the example is that 12 and 34 have been read separately as distinct numbers, an asterisk inserted between them, and multiplied. Sometimes the product may lead to a result sufficiently close to the expected one as to pass unnoticed. For this reason it is recommended that spaces in numbers be avoided as a matter of habit except in rare special circumstances (e.g. when entering a number like `0.7777 7777 7777 7777`).

Grouping blocks of digits with spaces is available through the `1s2` setting solely for *inputting* numbers in the variable=value list or formula. The `\eval` command does not *output* numbers in this form.

2.1.3 Variable names

In mathematical practice, variable names are generally single letters of the Roman or Greek alphabets, or occasionally from other alphabets, in a variety of fonts, and often with subscripts or primes or other decorations: x, $\boldsymbol{x}$, μ, x', α'', T^{iv}, Θ_n, $\boldsymbol{a}'_n$, β''_{mn}, $\vec{v}$, k^+, k_- are examples. `numerica` does not attempt to characterize variables by their 'internals'(alphabet, font, decoration, etc.). Rather, the program accepts as a variable *whatever lies to the left of the equals sign* in an item of the variable=value list (for which see §2.2 immediately below).

What lies to the left is a *LaTeX expression*. Different LaTeX almost always means different variable. For instance x and x are *different* variables since, in the underlying LaTeX, one is `x` and the other `\mathrm{x}`. I write '*almost* always' because there are exceptions. Since braces do not display in the `pdf`, names that look identical in the `pdf` may well be distinct in LaTeX. This is true particularly of superscripts and subscripts: `x_0` and `x_{0}` appear identical in the `pdf` but in the underlying LaTeX they are distinct, and will be treated as distinct variables by `numerica`. The user needs to be aware of this. Also, because equals signs and commas give structure to the variable=value list, a variable name should not contain a *naked* equals sign or a *naked* comma. Instead they should be decently wrapped in braces, like `R_{=}` displaying as $R_=$. These provisos aside, variables can be single- or multi-token, can be in different fonts, can be decorated with primes and indices – and may even contain spaces. (But please don't; such names are not part of mathematical practice.) If a variable is natural to the mathematical context, it will almost certainly be accepted as a variable in `numerica`. For the kind of back-of-envelope calculations envisaged for the package, most variables will be single letters from the Roman or Greek alphabets.

2.1.3.1 Multi-token variable handling

Although multi-token variables are perfectly acceptable, *internally* `numerica` works with single tokens. Variable names can be so different in structure, one from another, that to ease the parsing of formulas, all *internal* variable names are assumed to be single tokens. Hence a necessary initial step for the package is to map all multi-token variable names in the vv-list and the formula to single tokens. `numerica` does this by turning the multi-token variable names into control sequences with names in the sequence `\nmc_a`, `\nmc_b`, `\nmc_c`, etc., then searches through the vv-list and the formula for every occurrence of the multi-token names and replaces them with the relevant control sequences. It does this in order of decreasing size of name, working from the names that contain most tokens down to names containing only two tokens. (Doing the replacing in this order prevents *parts* of longer names possibly being mistaken for shorter variable names.)

The conversion process uses computer resources. Even if there are no multi-token variables present, `numerica` still needs to check that this is so – unless the user alerts the program to the fact with the setting `xx=0`; see §5.2.2.

2.2 The variable=value list

To evaluate algebraic, trigonometric and other formulas that involve variables we need to give those variables values. This is done in the *variable=value list* – or *vv-list* for short. This is the fourth argument of the `\nmcEvaluate` command and is a square-bracket delimited optional argument (optional because an expression may depend only on constants and numbers).

A vv-list is a comma-separated list (or, if the `comma` package option is used, semicolon-separated list) where each item is of the form *variable=value*. It might be something simple like `[g=9.81,t=2]` or something more complicated like

```
[V_S=\tfrac43\pi r^3,V_C=2\pi r^2h,h=3/2,r=2].
```

Spaces around the equals signs or the commas (resp., semicolons) are stripped away during processing so that `[g=9.81,t=2]` and `[ g = 9.81 , t = 2 ]` are the *same* variable=value list.

Math delimiters should never be used in the vv-list. If they are present they will cause errors. Math delimiters have a part to play only in the main argument, where their presence or absence can determine the form of display of the result, as discussed above in §1.1.4.

2.2.1 Evaluation from right to left

In these examples, with variables depending on other variables, there is an implication: that the list is evaluated *from the right*. Recall how a function of a function is evaluated, say $y = f(g(h(x)))$. To evaluate y, first x is assigned a value then $h(x)$ is calculated, then $g(h(x))$ then $f(g(h(x))) = y$. We work

from right to left, from the innermost to the outermost element. Or consider an example like calculating the area of a triangle by means of the formula

$$A = \sqrt{s(s-a)(s-b)(s-c)}.$$

First we write the formula; then we state how s depends on a, b, c, namely $s = \frac{1}{2}(a+b+c)$, then we give values to a, b, c. In `numerica` this is mirrored in the layout of the `\eval` command:

```
\eval{$ \sqrt{s(s-a)(s-b)(s-c)} $}
             [s=\tfrac12(a+b+c),a=3,b=4,c=5]
```

The formula in a sense is the leftmost extension of the vv-list. The entire evaluation occurs from right to left. This means that the rightmost variable in the vv-list can depend only on constants and numbers – although it may be a complicated expression of those elements. Other variables in the vv-list can depend on variables *to their right* but not to their left.

2.2.2 Constants

`numerica` has five built-in constants and can also accept user-defined constants. For the latter, see §6.3. The five built-in constants known to `numerica` are `\pi`, the ratio of circumference to diameter of a circle; `e`, the base of natural logarithms; Euler's constant `\gamma`, the limit of $\left(\sum_1^N 1/n\right) - \ln N$ as $N \to \infty$; the golden ratio `\phi`, equal to $\frac{1}{2}(1+\sqrt{5})$; and the utilitarian constant `\deg`, the size of a degree in radians. Using a comma list for a multi-formula calculation and an `align*` environment,

```
\eval[env=align*,pp,p,ff]{ \pi, e, \gamma, \phi, \deg } ⟹
```

$$\begin{aligned} \pi &= 3.141593, \\ e &= 2.718282, \\ \gamma &= 0.577216, \\ \phi &= 1.618034, \\ \deg &= 0.017453, \end{aligned}$$

so that `\eval{$ 180\deg $}` $\Longrightarrow 180\deg = 3.141593$ (as it should).

In some contexts it may feel natural to use any or all of `\pi`, `e`, `\gamma` and `\phi` as variables by assigning values to them in the vv-list. `numerica` does not object. The values assigned in this way override the built-in constant values.

For example, if instead of the usual ABC we label a triangle EFG with sides (note!) $e = 3, f = 4$ and $g = 5$, its area is

```
\eval{$ \sqrt{s(s-e)(s-f)(s-g)} $}
          [s=\tfrac12(e+f+g),e=3,f=4,g=5]
```

$\Longrightarrow$
$\sqrt{s(s-e)(s-f)(s-g)} = 6, \quad (s = \frac{1}{2}(e+f+g), e = 3, f = 4, g = 5).$

Clearly the value 3 assigned to `e` in the vv-list has been used in the calculation, not the value of the constant. But if `e` (or `\pi` or `\gamma` or `\phi`) is not assigned a value in the vv-list then it has, by default, the value of the constant. In the case of `e`, if you wish to use it as a variable, the constant is always available as `\exp(1)`. No similar alternative is available for `\pi`, `\gamma` or `\phi`.

2.2.3 Expressions in the variable=value list

Suppose our expression is $\frac{4}{3}\pi r^3$, the volume V_S of a sphere in terms of its radius r, and we want to calculate the volume for different values of r to get a sense of how rapidly volume increases with radius.

`$ V_S=\eval{ \tfrac43\pi r^3 }[r=1] $` $\Longrightarrow V_S = 4.18879.$

Having set up this calculation it is now an easy matter to change the value of r in the vv-list:

`$ V_S=\eval{ \tfrac43\pi r^3 }[r=1.5] $` $\Longrightarrow V_S = 14.137167.$

Or we could 'rephrase' the calculation like this:

`\eval{$ V_S $}[V_S=\tfrac43\pi r^3,r=2]` $\Longrightarrow$
$V_S = 33.510322, \quad (V_S = \frac{4}{3}\pi r^3, r = 2).$

As you can see, values in the vv-list are not limited to numbers. They can be expressions depending on constants, numbers or other variables to their right in the list. This calculation also shows a multi-token variable (`V_S`) being used.

Another example: to compute the volume $V_C = \pi r^2 h$ of a cylinder, we have two variables to assign values to:

`$ V_C=\eval{ \pi r^2h }[h=4/3,r=1] $` $\Longrightarrow V_C = 4.18879.$

Or we can divide the calculation up like this,

`$ V_C=\eval{ hA_C }[A_C=\pi r^2,h=4/3,r=1] $` $\Longrightarrow V_C = 4.18879,$

which emphasizes that the volume is 'base × height' (and again uses a multi-token variable).

A third instance is provided by the example above in which we calculated the area of a triangle by means of Brahmagupta's formula.

2.2.4 Display of the vv-list

By default, the vv-list is displayed with the numerical result. That and the format of the display can both be changed.

2.2.4.1 Changing the display format

In the example above where the area of a triangle is calculated using Brahmagupta's formula, display of the result is crowded. One remedy is to force display of the vv-list to a new line. In the default set-up, this happens automatically if the `env` setting is equated to `multline` or `multline*`. Let's do this when Brahmagupta's formula is used not for a triangle but in the more challenging case of a cyclic quadrilateral.

The cyclic quadrilateral in question is formed by a 45-45-90 triangle of hypotenuse 2 joined to a 30-60-90 triangle along its hypotenuse of the same length. Two triangles, six vertices but the two along the hypotenuses are shared, hence four vertices in all, lying on a circle. The sides of the cyclic quadrilateral are therefore $\sqrt{2}, \sqrt{2}, \sqrt{3}, 1$. Adding the areas of the two triangles, the area of the quadrilateral is $A = 1 + \frac{1}{2}\sqrt{3}$, or in decimal form, `$\eval{1+\tfrac12\surd3}$` $\Longrightarrow$ 1.866025. Let's check with Brahmagupta's formula:

```
\eval[env=multline*]{\sqrt{(s-a)(s-b)(s-c)(s-d)}}
    [s=\tfrac12(a+b+c+d),
       a=\surd2,b=\surd2,c=\surd3,d=1]
```

$\Longrightarrow$

$$\sqrt{(s-a)(s-b)(s-c)(s-d)} = 1.866025,$$
$$(s = \tfrac{1}{2}(a+b+c+d), a = \sqrt{2}, b = \sqrt{2}, c = \sqrt{3}, d = 1)$$

2.2.4.2 Suppressing display of the vv-list

Star (`*`) option If display of the vv-list is not wanted at all, only the numerical result, it suffices to attach an asterisk (star) to the `\eval` command, giving a 'naked' result:

`\eval*{ hA_C }[A_C=\pi r^2,h=4/3,r=1]` $\Longrightarrow$ 4.18879.

The star option for a multi-formula calculation evaluates each formula and presents the numerical result of each calculation with a space between; it is minimal and inelegant:

`\eval*[ff]{ \surd x, x, x^2 }[x=\pi]` $\Longrightarrow$ 1.772454 3.141593 9.869604

However, the `pp` setting is available to add a comma (for instance) and the `sep` setting can change the space – perhaps `sep=\quad`.

Note that with the star option a negative result will display with a hyphen for the minus sign.

`\eval*{ y }[y=ax+b,x=2,a=-2,b=2]` $\Longrightarrow$ -2,

In a math environment, the hyphen will display as a minus sign. Wrapping `\eval*` around math delimiters or using the `env` option has no effect – the `*` dominates.

No math environment: In the absence of a math environment, recall from §1.1.4, `\eval` (no star!) automatically presents the numerical result *without vv-list* between `$` delimiters, (so that negative values display with a proper minus sign):

`\eval{ y }[y=ax+b,x=2,a=-2,b=2]` $\Longrightarrow -2$.

Settings option: Display of the vv-list can also be suppressed through the settings option, by writing `vv=` there – i.e. by giving the key `vv` an empty value. If the example above is too bare, then perhaps

`\eval[vv=]{$ V_C $}[V_C=h\pi r^2,h=4/3,r=1]` $\Longrightarrow V_C = 4.18879$

is more acceptable? See §5.3.2.3 for a fuller discussion of the `vv` setting.

Scriptstyle contexts: In versions of `numerica` before version 3, display of the vv-list was automatically suppressed in scriptstyle (and scriptscriptstyle) contexts. This occurred by means of TEX's `\mathchoice` command. Version 3.0.0 of `numerica` has dispensed with `\mathchoice` and requires the user to suppress display in these contexts (if so wished), by starring the `\eval` command:

`$e^{\eval*{xy}[x=\pi,y=1/e]}$` $\Longrightarrow e^{1.155727}$

Empty vv-list suppressed: Should the vv-list be empty, or display of *all* variables be suppressed by wrapping each in braces (see next, §2.2.4.3), then *nothing* is displayed where the vv-list would normally be:

`$ V_C=\eval{ hA_C }[{A_C}=\pi r^2,{h}=4/3,{r}=1] $.` $\Longrightarrow$
$V_C = 4.18879$.

2.2.4.3 Suppressing display of items

You may wish to retain some variables in the vv-list display, but not all. For those variables you wish omitted from the display, wrap each variable (but not the equals sign or value) in braces. When calculating the volume of a cylinder in a previous example, the base area A_C has a different status from the 'fundamental' variables r and h. It is an intermediate value, one that we pass through on the way to the final result. To suppress it from display enclose the variable in braces:

`$ V_C=\eval{ hA_C }[{A_C}=\pi r^2,h=4/3,r=1] $` $\Longrightarrow V_C = 4.18879$.

As you can see, A_C no longer appears in the displayed vv-list. Of course the name and its value are still recorded 'behind the scenes' and can still be used in calculations. Note that the braces enclose only the variable name, not the equals sign or the value.

2.2.5 Abusing multi-token variable names

A variable name is what lies to the left of the equals sign of an item in the vv-list. Since multi-token variables are converted to single tokens before any calculating is done, it is possible to sin. We can, for instance, make $\sin\pi$ a variable name and produce an absurdity:

`\eval{$ \sin\pi $}[{\sin\pi}=1]` $\Longrightarrow \sin\pi = 1;$

or similarly treat a number as a variable name and produce another:

`\eval{$ -1 $}[{-1}=1]` $\Longrightarrow -1 = 1.$

What is happening here is that the multi-token 'variables' `\sin\pi` and `-1` are converted, right at the start of proceedings, to single tokens like `\nmc_a`, `\nmc_b`. These TEX macros expand to their respective multi-token variable names when *displayed*, but for calculating within `numerica` the single token is used. By this means it is easy to construct whatever grotesqueries you like.

Should `numerica` try to check variable names to avoid consequences like this? I see no reasonable way of doing so. Digits, and symbols like `(` and `+`, can easily be part of valid variable names – k^+, x_1, $C_n^{(0)}$ and so on. It is left to the user, in any *public* document, to avoid such sins (which can easily be constructed in LATEX without recourse to `\eval` at all, should the user so wish). See also §6.2.5.1 where a similar issue arises with user-defined macros.

2.3 Formatting the numerical result

Internally, values are stored to 16 significant figures (if available), calculations are carried out to 16 significant figures, but only rarely are results displayed to 16 figures. Generally, they are rounded to some smaller number of figures. The default 'out of the box' rounding value is 6, meaning at most 6 decimal places are shown. (It can of course be reset with the `rounding` package option, e.g. `rounding=4`; see §§1.1.1, 2.3.1.) So far, nearly all results have been rounded to the default figure, although not all digits are always displayed – for instance if the sixth one is 0, or the result is an integer.

But `numerica` is not limited to the decimal presentation of results. Scientific notation is also available, as is fraction-form output with numerator and denominator both integers, and boolean output when the formula being evaluated is a comparison, treated by the `\eval` command not as a statement like *'A is less than B'* but rather as a question: *'Is A less than B?'*.

The appearance of the result in any of these various formats can be chosen and customized by means of a square-bracketed optional argument following the vv-list – or the formula if there is no vv-list. This optional argument may contain a number of juxtaposed items from these possibilities:

- a question mark `?` which gives boolean output, 1 for true, 0 for false; see §2.3.5;
  - a second question mark, `??`, which gives boolean output T for true, F for false;
  - a third question mark, `???`, which gives boolean output `T` for true, `F` for false;
- a slash `/` which gives output in slash-fraction form with integers for both numerator and denominator; see §2.3.4; if followed by `s` the fraction will be formed with the `\sfrac` command from the `xfrac` package if loaded, or in scriptstyle slash fraction form if not;
  - a second slash, `//`, which gives output in fraction form using `\frac` with integers for both numerator and denominator; if followed by `t` or `d` the fraction will be formed with, respectively, the `\tfrac` or `\dfrac` commands from `amsmath`;
- an integer, the *rounding value*, positive, negative or zero, specifying how many decimal places to display the result to, or to how many zeros after the decimal point fraction-form output approximates the result; see §2.3.1 and §2.3.4;
- an asterisk, *, which pads the result with zeros should it not display as many decimal places as the rounding value specifies; see §2.3.2;
- the character `x` (lower case!) which presents the result in 'proper' scientific notation (a form like 1.2345×10^5 for 123450) except for numbers in the interval `[1,10)`; see §2.3.3.1;
  - the character `x` repeated, `xx`, which extends the notation to numbers in the interval `[1,10)`;
- the character `t` (lower case!) which presents the result in a style of scientific notation useful in tables (a form like (5)1.2345 for 123450) except for numbers in the interval `[1,10)`; see §2.3.3.1;
  - the character `t` repeated, `tt`, which extends the notation to numbers in the interval `[1,10)`;
- a letter other than `x` or `t`, usually one of the letters `e`, `d`, `E`, `D`, which presents the result in scientific notation with that letter as the exponent mark (a form like 1.2345e5 for 123450); see §2.3.3.1;
  - the letter repeated – say `dd` – which extends the notation to numbers in the interval `[1,10)`.

If you use `?` in the same specification as some other character, the `?` prevails; if you use `/` in the same specification as some other character except for `?`, the `/` prevails; if you use `x` in the same specification as some other character except

for `?` or `/`, the `x` prevails; if you use `t` in the same specification as some other character except for `?`, `/` or `x`, the `t` prevails: `?`≻`/`≻`x`≻`t`≻... (where ≻ means 'prevails over').

2.3.1 Rounding value

If the number is displayed as a decimal, the rounding value specifies the number of decimal places displayed. If a number is displayed in scientific notation (see below §2.3.3) that is still true, but it can mean differences in the overall number of digits displayed. For the moment, I show the effect of rounding in a purely decimal display:

`\eval{ 1/3 }[4],\qquad \eval{ 1/3 }` ⟹ 0.3333, 0.333333.

In the first case `4` was entered in the number-format option and the result is displayed to four decimal places; in the second, the default rounding value of 6 takes effect.

Following the default behaviour in `l3fp`, which is the calculational engine used by `numerica`, 'ties' are rounded to the nearest *even* digit. Thus a number ending 55 with a 'choice' of rounding to 5 or 6 rounds up to the even digit 6, and a number ending 65 with a 'choice' of rounding to 6 or 7 rounds down to the even digit 6:

`\eval[pp,ff]{ 0.1234555, 0.1234565 }` ⟹ 0.123456, 0.123456.

`l3fp` works to 16 significant figures and never displays more than that number (and often fewer).

- In the following, although I have specified a rounding value of 19, only 16 decimal places are displayed (with the final digits rounded up):

  `\eval{ 0.12345678912345678912 }[19]` ⟹ 0.1234567891234568

  `\eval{ 1.12345678912345678912 }[19]` ⟹ 1.123456789123457

- Now I add ten zeros after the decimal point, meaning that all 19 decimal places specified by the rounding value can be displayed in the first since the ten zeros do not contribute to the significant figures, but in the second, by changing the figure before the decimal point to 1, the ten zeros added *do* contribute to the significant figures:

  `\eval{ 0.0000000000123456789 }[19]` ⟹ 0.0000000000123456789

  `\eval{ 1.0000000000123456789 }[19]` ⟹ 1.000000000012346

- Lastly, I have added 9 digits before the decimal point:

  `\eval{ 987654321.123456789 }[19]` ⟹ 987654321.1234568

In all cases, no more than 16 *significant* figures are displayed, although the number of decimal places displayed may exceed 16.

It is possible to use *negative* rounding values. Such a value replaces the specified number of digits *before* the decimal point with zeros.

```
\eval{ 987654321.123456789 }[-4]
```
$\Longrightarrow 987650000$

A rounding value of 0 rounds to the nearest integer:

```
\eval{ 987654321.123456789 }[0]
```
$\Longrightarrow 987654321$

If you wish to change the *default* rounding value from 6 to some other value, this can be done by means of the package option `rounding`. For example, calling `numerica` like this,

```
\usepackage[rounding=4]{numerica}
```

will make 4 the new default rounding value.

The rounding value also plays a part in how closely fraction-form output approximates the calculated result (§2.3.4), and how many terms are used to evaluate 'infinite' sums (§3.1).

2.3.2 Padding with zeros

A result may contain fewer decimal places than the rounding value specifies, the trailing zeros being suppressed by default (this is how `l3fp` does it). Sometimes, perhaps for reasons of presentation like aligning columns of figures, it may be desirable to pad results with zeros – we have already met an example at §1.1.5.1. Padding is achieved by inserting an asterisk into the final optional argument of the `\eval` command:

```
\eval{ 1/4 }[4]
```
$\Longrightarrow 0.25,$

```
\eval{ 1/4 }[4*],\quad \eval{ 1/4 }[*4]
```
$\Longrightarrow 0.2500, \quad 0.2500.$

As you can see, it doesn't matter if there is a rounding value also present (the example at §1.1.5.1, using the default rounding value, had none), nor the order in which the asterisk and rounding value are entered.

2.3.3 Scientific notation

`numerica` can output numbers in various scientific notations. Entering `x` (lower case) in the number-format option produces the 'pure' form:

```
\eval[pp,ff]{ 2^{256}, 2^{-256} }[x]
```
$\Longrightarrow$
$1.157921 \times 10^{77}, \quad 8.636169 \times 10^{-78}.$

With the default rounding value, as here, one digit is shown before the decimal point and 6 digits after. Alternatively, you can enter some other letter in the number-format option, generally `e`, to generate a less elegant, more utilitarian output:

`\eval[pp,ff]{ 3^{81}, e^{-81} }[e]` $\Longrightarrow$ 4.434265e38, 2.255165e-39.

Other letters sometimes used for this purpose are `E`, `d` and `D`, but `numerica` will accept *any* letter (although `x` and `t` produce distinctive output). The 'e' in the output is the *exponent mark*, separating the *significand* (1.234) from the *exponent* (3). In scientific notation, the significand always has one *non-zero* digit before the decimal point (except for 0 itself).

For scientific notation the rounding value still means the number of decimal places displayed – by default, 6. This can mean different numbers of digits being shown from those displayed in 'ordinary' decimal form. Compare `\eval{ 123.456789 }` $\Longrightarrow$ 123.456789, where nine significant figures are displayed, with `\eval{ 123.456789 }[e]` $\Longrightarrow$ 1.234568e2 where only seven are, one before the decimal point and six after. In the other direction, only five significant figures are shown in the 'ordinary' decimal form `\eval{ 0.0123456789 }` $\Longrightarrow$ 0.012346, whereas in scientific notation `\eval{ 0.0123456789 }[e]` $\Longrightarrow$ 1.234568e-2 once again seven are on display.

Negative rounding values are pointless for scientific notation.

Sometimes letters other than 'e' are used for the exponent mark, particularly 'E' or 'd' or 'D'.

`\eval{$ 1/123456789 $}[4d]` $\Longrightarrow 1/123456789 = 8.1\text{d-}9.$

But when `x` is inserted in the trailing optional argument, the output is in the form $d_0.d_1 \ldots d_m \times 10^n$ (except when $n = 0$), where each d_i denotes a digit.

`\eval{$ 1/123456789 $}[4x*]` $\Longrightarrow 1/123456789 = 8.1000 \times 10^{-9}$.

As you can see, padding with zeros still applies in scientific notation and is activated, as before, with an asterisk in the number-format option.

The requirements of tables leads to another form of scientific notation. Placing `t` in the trailing argument turns on this table-ready form:

`\eval{$ 1/123456789 $}[4t*]` $\Longrightarrow 1/123456789 = (-9)\,8.1000.$

This is discussed more fully in the documentation for the `numerica-tables` package.

The order in which items are entered in the number-format option doesn't matter:

`\eval{$ 1/125 $}[*e4]` $\Longrightarrow 1/125 = 8.0000\text{e-}3,$

`\eval{$ 1/125 $}[4e*]` $\Longrightarrow 1/125 = 8.0000\text{e-}3.$

2.3.3.1 Numbers in the interval `[1,10)`

Usually when scientific notation is being used, numbers with magnitude in the interval $[1, 10)$ are rendered in their normal decimal form, 3.14159 and the like. Occasionally it may be desired to present numbers in this range in scientific notation (this can be the case in tables where the alignment of a column of figures

might be affected). `numerica` offers a means of extending scientific notation to numbers in this range by repeating the letter chosen as the exponent mark in the trailing optional argument.

$$\texttt{\textbackslash eval\{\$ \textbackslash pi \$\}[4xx]} \Longrightarrow \pi = 3.1416 \times 10^{0}$$

2.3.3.2 \eval* and scientific notation

Starring the `\eval` command gives a number-only result which can be presented in scientific notation:

$$\texttt{\textbackslash eval*\{ \textbackslash pi\^{}\textbackslash pi \}[e]} \Longrightarrow 3.646216\text{e}1$$

There is one catch: if you substitute `x` for `e` here, compilation will halt and LaTeX will state `Missing $ inserted.` This is because an `x` in the number-format option means a `\times` in the output and that requires a math environment to display. It is up to you, as the user, to provide the necessary delimiters *outside* the `\eval*` command. (Alternatively, use `\eval` without the star and without any delimiters whatever. Dollar signs are automatically placed around the result.)

2.3.4 Fraction-form output

The `\eval` command can output numbers in fraction form by including in the number-format specification either one or two slashes:

$$\texttt{\textbackslash eval[pp,ff]\{0.333333, 1.875\}[4/]} \Longrightarrow 1/3, \quad 15/8;$$

$$\texttt{\textbackslash eval[pp,ff]\{\textbackslash pi, e\}[2//]} \Longrightarrow \tfrac{22}{7}, \quad \tfrac{19}{7}.$$

A single slash results in a textstyle slash fraction; two slashes produce a `\frac`tion. Always, numerator and denominator are integers. The rounding value determines how close the fraction is to the calculated result. Consider

```
\eval{\pi}[1//],\quad \eval{\pi}[2//],
\eval{\pi}[3//],\quad \eval{\pi}[6//].
```

$\Longrightarrow \frac{19}{6}, \quad \frac{22}{7}, \quad \frac{267}{85}, \quad \frac{355}{113},$

and compare these results against π:

```
\eval[env=alignat*,pp,p=.,ff]{ \pi-\tfrac{19}6,
    \pi-\tfrac{22}7, \pi-\tfrac{267}{85},
    \pi-\tfrac{355}{113} }[8*]
```

$\Longrightarrow$

$$\begin{aligned}
\pi - \tfrac{19}{6} &= -0.02507401,\\
\pi - \tfrac{22}{7} &= -0.00126449,\\
\pi - \tfrac{267}{85} &= 0.00041618,\\
\pi - \tfrac{355}{113} &= -0.00000027.
\end{aligned}$$

In each case the fraction approximates the correct result to at least as many decimal places as specified in the rounding value – the difference between pi and the fraction is zero to that rounding value. Sometimes, as evident with the familiar approximations 22/7 and 355/113, the difference is zero to twice the number of digits in the denominator – these are exceptionally good approximations.

The example illustrates the use of an `alignat*` environment to ensure the results line up, including taking account of the minus signs.

What happens if I seek a still more accurate approximation – say to (at least) seven places of decimals? The result is a message:

`\eval{\pi}[7//]` $\Longrightarrow$!!! No result to 7 zeros with $1 \leq$ `denom` ≤ 200 in: formula. !!!

By default `numerica` searches denominators from 1 to 200 before halting the search and producing the message. The program offers two settings that may assist here. The first is the `/max` setting which allows the user to specify the maximum denominator tried, e.g. putting `/max = 1000` in the settings option of the `\eval` command would allow the search to continue up to a denominator of 1000. But perhaps that means waiting too long? The user can also specify an initial value to start searching from with the `/min` setting, by entering in the settings option, say, `/min = 500`, which would mean searching from an initial denominator of 500. By default `/min = 1` and `/max = 200`.

To test the settings equate both to 113:

`\eval[/min=113,/max=113]{\pi}[6//]` $\Longrightarrow \frac{355}{113}$.

Performing a similar exercise with other irrational numbers like e, ϕ and $\sqrt{2}$, one realizes just how good this particular approximation is.

2.3.4.1 Refining the form of fraction

The form of the fraction output can be refined by adding a qualifying letter to the single or double slash in the number format option.

By adding an `s` after a single slash, `/s`, it is possible to get scriptstyle numbers in a slash fraction. If the `xfrac` package[1] has been loaded (as it has been for the present document), then the fraction takes this form:

`\eval{$ \pi $}[2/s]` $\Longrightarrow \pi = {}^{22}\!/_{7}$.

This is the result of the `\sfrac` command from the `xfrac` package. If that package had not been loaded then the fraction would have been presented in the form 22/7.

By adding a `t` or `d` after two slashes, `//t` or `//d`, it is possible to force textstyle or displaystyle output by means of `amsmath`'s `\tfrac` or `\dfrac` commands (in place of `\frac`). For example, even in a displaystyle environment,

[1] `xfrac` is included in the `l3packages` bundle.

`\eval[p]{\[ \pi \]}[6//t]` $\Longrightarrow$

$$\pi = \tfrac{355}{113},$$

with the `//t` specification, and similarly, in a textstyle environment you can force a displaystyle fraction with the `//d` specification.

2.3.5 Boolean output

`l3fp` can evaluate comparisons, outputting 0 if the comparison is false, 1 if it is true. By entering a question mark, `?`, in the trailing optional argument, you can force `numerica` to do the same depending as the result of a calculation is zero or not. (The expression being evaluated does not need to be a comparison, `\eval{\pi}[?]` $\Longrightarrow 1$, but comparisons are what this is designed for.)

Possible comparison relations are `=`, `<`, `>`, `\ne`, `\neq`, `\ge`, `\geq`, `\le`, `\leq`. Although programming languages use combinations like `<=` or `>=` and, from version 3.0.0 `\eval` will accept these without raising an error, this is not part of mathematical practice.

An example of boolean output where the relation is equality exhibits a numerological curiosity:

`\eval[p=.]{\[ \frac1{0.0123456789}=81 \]}[5?]` $\Longrightarrow$

$$\frac{1}{0.0123456789} = 81 \to 1.$$

The expression on the left is to be read as a *question*: 'Is 1/0.0123456789 equal to 81?', not as a statement; the arrow points to the answer, in this instance 1, meaning *true*. But notice the 5 alongside the question mark in the trailing argument. That is critical. Change it to 6 (or omit it since the default rounding value is 6) and the outcome is different:

`\eval[p=.]{\[ \frac1{0.0123456789}=81 \]}[6?]` $\Longrightarrow$

$$\frac{1}{0.0123456789} = 81 \to 0.$$

Now the relation is *false*. Evaluating the fraction to more than 6 places, say to 9, we can see what is going on:

`\eval{$ 1/0.0123456789 $}[9]` $\Longrightarrow 1/0.0123456789 = 81.000000737.$

In other words, the question posed by the `?` specification is not 'Is 1/0.0123456789 equal to 81?' but 'Is 1/0.0123456789 equal to 81 *to the specified number of decimal places*?' To 5 decimal places it is; to 6 decimal places it is not.

2.3.5.1 Outputting `T` or `F`

To my eye, outputting 0 or 1 in response to a question like $1/0.0123456789 = 81$ is confusing. It is easy to change the boolean output from $0, 1$ to a more appropriate F, T, or `F,T` by duplicating (F, T) or triplicating (`F,T`) the question mark in the number-format option.

```
\eval[p=.]{\[ \frac1{0.0123456789}=81 \]}[6???] ⟹
```

$$\frac{1}{0.0123456789} = 81 \to \texttt{F}.$$

The default boolean output format is chosen to be $0, 1$ in case an `\eval` command is used within another `\eval` command ('nesting'– see Chapter 7.1). The inner command needs to output a *numerical* answer.

2.3.5.2 Rounding error tolerance

If at least one of the terms in a comparison is the result of a calculation, then it's value is likely to contain rounding errors. What level of rounding error can we tolerate before such errors interfere with the comparison being made? `l3fp` tolerates none. It decides the truth or falsity of a comparison to all 16 significant figures: 1.000 0000 0000 0000 and 1.000 0000 0000 0001 are *not* equal in `l3fp`. But for most purposes this will be far too severe a criterion.

Suppose our comparison relation is ϱ, denoting one of $=$, $<$, $>$, `\le`, etc. If $X \varrho Y$ then $X - Y \varrho Y - Y$, i.e. $X - Y \varrho 0$. This is what `numerica` does. It takes the right-hand side of the relation from the left-hand side and then compares the *rounded* difference under ϱ to 0. The rounding value used is the number specified with the question mark in the trailing argument of the `\eval` command or, if no number is present, the default rounding value ('out of the box' this is 6). Thus, in a recent example, $1/0.0123456789 - 81$ when rounded to 5 decimal places is 0.00000, indistinguishable from zero at this rounding value; hence the equality $1/0.0123456789 = 81$ is true. But when rounded to 6 places it is 0.000001 which *is* distinguishable from zero and so the equality is false. Truth or falsity depends on the rounding value.

When dealing with numbers generated purely mathematically, rounding values of 5 or 6 are likely to be too small. More useful would be rounding values closer to `l3fp`'s 16 – perhaps 14? – depending on how severe the calculations are that generate the numbers. However if the numbers we are dealing with come from outside mathematics, from practical experiments perhaps, then even a rounding value of 5 or 6 may be too large.

Mathematically, the claim that $X = Y$ at a rounding value n is the claim that

$$|X - Y| \leq 5 \times 10^{-(n+1)}.$$

since this rounds *down* to zero at n places of decimals. This gives a more accurate test of equality than doing things in the opposite order – rounding each number first and then taking the difference. One might, for instance, have numbers like $X = 0.12345$, $Y = 0.12335$. Rounding to $n = 4$ places, both round to 0.1234 and yet the difference between them is 0.0001 – they are distinguishable numbers to 4 places of decimals. This is why `numerica` forms the difference *before* doing the rounding.

2.3.5.3 And, Or, Not

For logical And LaTeX provides the symbols `\wedge` and `\land`, both displaying as $\wedge$, but `numerica` adds thin spaces (`\,`) around the symbol for `\land` (copying the package `gn-logic14.sty`). For logical Or LaTeX provides the symbols `\vee` and `\lor`, both displaying as $\vee$, but again `numerica` adds thin spaces around the symbol for `\lor`.

`\eval{$ 1<2 \wedge 2<3 $}[??]` $\Longrightarrow 1 < 2 \wedge 2 < 3 \to T$,
`\eval{$ 1<2 \land 2<3 $}[???]` $\Longrightarrow 1 < 2 \land 2 < 3 \to \mathtt{T}$.

To my eye the second of these with its increased space around the wedge symbol displays the meaning of the overall expression better than the first. Both And and Or have equal precedence; in cases of ambiguity the user needs to parenthesize as necessary to clarify what is intended.

LaTeX provides two commands for logical Not, `\neg` and `\lnot`, both displaying as $\neg$. Not binds tightly to its argument:

`\eval{$ \lnot A \land B $}[A=0,B=0]` $\Longrightarrow \neg A \land B = 0, \quad (A = 0, B = 0)$.

Here `\lnot` acts only on the A; if it had acted on $A \land B$ as a whole the result would have been different:

`\eval{$ \lnot(A \land B) $}[A=0,B=0]` $\Longrightarrow$
$\neg(A \land B) = 1, \quad (A = 0, B = 0)$.

For a little flourish, I evaluate a more complicated logical statement:[2]

```
\eval{$(A\lor\lnot C)\land(C\lor B)\land
      (\lnot A\lor\lnot B)$}[A=1,B=0,C=1][???]
```

$\Longrightarrow (A \lor \neg C) \land (C \lor B) \land (\neg A \lor \neg B) \to \mathtt{T}, \quad (A = 1, B = 0, C = 1)$.

[2] Quoting from an article in *Quanta Magazine* (August 2020) by Kevin Hartnett: 'Let's say you and two friends are planning a party. The three of you are trying to put together the guest list, but you have somewhat competing interests. Maybe you want to either invite Avery or exclude Kemba. One of your co-planners wants to invite Kemba or Brad or both of them. Your other co-planner, with an ax to grind, wants to leave off Avery or Brad or both of them. Given these constraints, you could ask: Is there a guest list that satisfies all three party planners?' I have written C for Kemba, A and B for Avery and Brad.

2.3.5.4 Chains of comparisons

`numerica` can handle chains of comparisons like $1 < 2 < 1 + 2 < 5 - 1$. 'Behind the scenes' it inserts logical And-s into the chain, $1 < 2 \wedge 2 < 1{+}2 \wedge 1{+}2 < 5{-}1$, and evaluates the modified expression:

`\eval{$ 1<2<1+2<5-1 $}[???]` $\Longrightarrow 1 < 2 < 1 + 2 < 5 - 1 \rightarrow$ `T`.

2.3.5.5 `amssymb` comparison symbols

`numerica` accepts some alternative symbols for the basic comparison relations from the `amssymb` package provided that package is loaded, i.e. the preamble of your document includes the statement

```
\usepackage{amssymb}
```

The variants from this package are: `\leqq` ($\leqq$), `\leqslant` ($\leqslant$), `\geqq` ($\geqq$), and `\geqslant` ($\geqslant$).[3] There are also negations: `\nless` ($\nless$), `\nleq` ($\nleq$), `\nleqq` ($\nleqq$), `\nleqslant` ($\nleqslant$), `\ngtr` ($\ngtr$), `\ngeq` ($\ngeq$), `\ngeqq` ($\ngeqq$), `\ngeqslant` ($\ngeqslant$).

[3] No, that is not `eggplant`.

Chapter 3

Calculational details

3.1 Arithmetic

Addition, subtraction, multiplication, division, square roots, n-th roots, and exponentiating (raising to a power) are all available.

Multiplication can be rendered explicitly with an asterisk,

`\eval{$ 9*9 $}` $\Longrightarrow 9 * 9 = 81,$

but that's ugly. More elegant is to use `\times`:

`\eval{$ 9\times9 $}` $\Longrightarrow 9 \times 9 = 81.$

`\cdot` is also available and in many cases juxtaposition alone suffices:

`\eval{$ \surd2\surd2 $}` $\Longrightarrow \sqrt{2}\sqrt{2} = 2,$
`\eval{$ ab $}[a=123,b=1/123]` $\Longrightarrow ab = 1, \quad (a = 123, b = 1/123).$

Division can be rendered in multiple ways too. Using a comma list for a multi-formula evaluation (and the `ff` setting),

`\eval[p=.,ff]{\[ 42/6, 42\div6, \frac{42}6 \]}` $\Longrightarrow$

$$42/6 = 7$$

$$42 \div 6 = 7$$

$$\frac{42}{6} = 7.$$

In a displaystyle environment, `\frac` displays as shown. In a textstyle environment it displays as $\frac{42}{6}$. If you want to force a textstyle display, even in a displaystyle environment, use `\tfrac` (from `amsmath`) and, conversely, if you want to force a displaystyle display, even in a textstyle environment, use `\dfrac`. If the package `xfrac` is loaded, then slash fractions are rendered with scriptstyle numbers, `\eval[p]{\[ \sfrac{42}{6} \]}` $\Longrightarrow$

$$^{42}/_{6} = 7,$$

even in a displaystyle environment. If `xfrac` is not loaded, then `numerica` defines `\sfrac` to expand to a scriptstyle slash fraction, e.g. 42/6.

Note that since juxtaposition means multiplication, a combination like $42\frac{1}{6}$ also evaluates to 7 within an `\eval` command; it does not mean 'forty two and a sixth'. Such fractions need to be entered as improper fractions to evaluate correctly – for instance, 'two and a half' entered as $\frac{5}{2}$ (as one does anyway in mathematical expressions because of the ambiguity in a form like $2\frac{1}{2}$).

Powers are indicated with the superscript symbol `^`. It is clear from the braced LaTeX grouping that a 'tower' of superscripts is evaluated from the top down. Thus 3^{2^3} is 3^8 $(= 9^4)$, not $9^3 = 729$:

```
\eval{$ 3^{2^3} $}
```
$\Longrightarrow 3^{2^3} = 6561$.

3.2 Square roots and n-th roots

Let us check that 3, 4, 5 and 5, 12, 13 really are Pythagorean triples (I use `\sqrt` in the first, `\surd` in the second):

```
\eval[ff]{\[ \sqrt{3^2+4^2}, \surd(5^2+12^2) \]}
```
$\Longrightarrow$

$$\sqrt{3^2 + 4^2} = 5$$

$$\surd(5^2 + 12^2) = 13$$

The `\sqrt` command has an optional argument which can be used for extracting n-th roots of a number. In `numerica`, when used with the `\sqrt` command, n is assumed to be a *positive integer*, in practice generally a *small* positive integer like 3 or 4.

```
\eval{$ \sqrt[4]{81} $}
```
$\Longrightarrow \sqrt[4]{81} = 3,$

```
\eval{$ \sqrt[n]{125} $}[n=\floor{\pi}]
```
$\Longrightarrow \sqrt[n]{125} = 5, \quad (n = \lfloor \pi \rfloor).$

For displaystyle expressions, the `\sqrt` sign grows to accommodate the extra vertical height; the `\surd` sign doesn't. Here is an example which uses the `\mleft`, `\middle`, `\mright` commands from the package `mleftright` (requiring `\usepackage{mleftright}` in the preamble of the present document). In the formula I have enlarged the 3 of the cube root from the default `\scriptscriptstyle` visible in the examples above to a more appropriately sized `\scriptstyle`.

```
\eval[p=.]{\[ \sqrt[\scriptstyle3]{\!
      \mleft(\frac AD\middle/\frac BC\mright)}
  \]}[A=729,B=81,C=9,D=3]
```

$\Longrightarrow$

$$\sqrt[3]{\left(\frac{A}{D}\Big/\frac{B}{C}\right)} = 3, \qquad (A = 729, B = 81, C = 9, D = 3).$$

As implemented in `numerica`, n-th roots found using `\sqrt[n]` are restricted to positive integral `n`. This raises an interesting question: if the 'n' of an n-th root is the result of a calculation, what happens with rounding errors? The calculation may not produce an *exact* integer. (This problem also arises with factorials; see §3.7.) The solution employed in version 3.0.0 of `numerica` is simply to round to the nearest integer. This is simpler than in previous versions where an error could be raised in some rare situations. But it does mean that, e.g. $\sqrt[\pi]{27} = 3$, since π rounds to 3. In such cases, for the sake of the reader, a user should make the rounding explicit – as I did in an example above, wrapping π in the `\floor` command.

3.2.1 n-th roots of negative numbers

Odd (in the sense of 'not even') positive integral roots of *negative* numbers are available with `\sqrt`,

```
\eval[p=.,ff]{\[ \sqrt[3]{-125}, \sqrt[3]{-3.375} \]}
```
$\Longrightarrow$
$$\sqrt[3]{-125} = -5$$
$$\sqrt[3]{-3.375} = -1.5.$$

3.2.2 Powers of n-th roots

In previous versions of `numerica`, raising an n-th root to a power when $n \neq 2$ gave a false answer unless the n-th root was parenthesized before raising to the power. From version 3.0.0, the parentheses are unnecessary (but notice the thin space inserted before the 3 in the second example to improve the visual appearance):

```
\eval[p=.,ff]{\[ \bigl(\sqrt[3]{-8}\,\bigr)^3, \sqrt[3]{-8}^{\,3} \]}
```
$\Longrightarrow$
$$\left(\sqrt[3]{-8}\right)^3 = -8$$
$$\sqrt[3]{-8}^{\,3} = -8.$$

3.2.3 Inverse integer powers

Of course to find an n-th root we can also raise to the inverse power,

```
\eval{$ 81^{1/4} $}
```
$\Longrightarrow 81^{1/4} = 3.$

However, raising a *negative* number to an inverse power generates an error even when, mathematically, it should not. This matter, which is a product of floating point representation of numbers, is discussed below in §4.1.7.2.

3.3 Precedence and parentheses

The usual precedence rules apply: multiplication and division bind equally strongly and more strongly than addition and subtraction which bind equally stongly. Exponentiating binds most strongly. Evaluation occurs from the left.

`\eval{$ 4+5\times6+3 $}` $\Longrightarrow 4+5\times6+3=37,$

`\eval{$ 6\times10^3/2\times10^2 $}` $\Longrightarrow 6\times10^3/2\times10^2=300000,$

which may not be what was intended. Parentheses (or brackets or braces) retrieve the situation:

`\eval{$ (4+5)(6+3) $}` $\Longrightarrow (4+5)(6+3)=81,$

`\eval{$ (6\times10^3)/(2\times10^2) $}` $\Longrightarrow (6\times10^3)/(2\times10^2)=30.$

When one writes -4^2 it is not clear what is intended: is it $-(4^2)$ or $(-4)^2$? In `numerica` exponentiating binds most strongly; negative values must therefore be parenthesized when raised to a power. Thus

`\eval[pp,ff]{$ -4^2, (-4)^2 $}` $\Longrightarrow -4^2=-16, \quad (-4)^2=16.$

3.3.1 Command-form brackets

Note that brackets of all three kinds are available also in command form: `\lparen \rparen` (from `mathtools`) for `( )`, `\lbrack \rbrack` for `[ ]`, and `\lbrace \rbrace` for `\{ \}`.

3.3.2 Modifiers (`\left`, `\right`, `\big`, etc.)

The `\left` and `\right` modifiers and also the series of `\big...` modifiers (`\bigl \bigr`; `\Bigl \Bigr`; `\biggl \biggr`; `\Biggl \Biggr`) are available for use with all brackets (parentheses, square brackets, braces). If you feel `\left`, `\right` give too much space around your formulas, you can use `\mleft`, `\mright` from the `mleftright` package.

```
\eval[p=.,ff]{\[ \exp\left(
      \dfrac{\ln2}{4}+\dfrac{\ln8}{4}
    \right),
    \exp\mleft(
      \dfrac{\ln2}{4}+\dfrac{\ln8}{4}
    \mright)\]}
```

$\Longrightarrow$

$$\exp\left(\frac{\ln 2}{4}+\frac{\ln 8}{4}\right)=2$$

$$\exp\Big(\frac{\ln 2}{4}+\frac{\ln 8}{4}\Big)=2.$$

`numerica` also accepts the use of left-right modifiers with `.` (dot) and with `/`, but if parentheses are not wanted it can be simpler just to use a `\big` command:

`\eval[p=.]{\[ \dfrac{3+4}{2+1}\bigg/\dfrac{1+2}{4+5} \]}` $\Longrightarrow$

$$\frac{3+4}{2+1}\Big/\frac{1+2}{4+5} = 7.$$

Modifiers with their accompanying brackets etc. can be nested.

3.4 Unary functions

The unary functions catered for in `numerica` (at present) are the trigonometric and hyperbolic functions, the various logarithms, the exponential function, and the signum function. Mathematicians delimit the arguments of these functions not only with parentheses, but also with square brackets and (mathematical) braces (`\{ \}`). In LaTeX these are available both in explicit character form and also in the command form of §3.3.1. Of whatever kind, brackets can be qualified with `\left \right`, `\bigl \bigr`, etc., and `\mleft` and `\mright` from the `mleftright` package.

3.4.1 Trigonometric functions

LaTeX provides all six trignometric functions, `\sin`, `\cos`, `\tan`, `\csc`, `\sec`, `\cot`. Their arguments are assumed to be in radians unless degrees are explicitly ordered, either by entering `o` (lowercase letter 'o', reminiscent of a degree symbol) in the settings option, or by appending `\degree` to a number. The command `\degree` is defined in `numerica` (using `\ProvideDocumentCommand`) and expands to ° in both text and math modes.)

`\eval{$ \sin(\pi/3) $}` $\Longrightarrow \sin(\pi/3) = 0.866025,$

`\eval[o]{$ \sin 60 $}` $\Longrightarrow \sin 60 = 0.866025,$

`\eval{$ \sin 60\degree $}` $\Longrightarrow \sin 60^\circ = 0.866025.$

LaTeX also provides the three main trigonometric inverses: `\arcsin`, `\arccos`, `\arctan`. The three missing inverses – `\arccsc`, `\arcsec`, `\arccot` – are provided by `numerica`. In the example, the `p` setting has been used to attach a degree symbol to the answer:

`\eval[p=\degree]{$ (\arccot 1)/1\deg $}` $\Longrightarrow (\operatorname{arccot} 1)/1\,\mathrm{deg} = 45^\circ.$

Alternatively, you can manually append a `\degree` command after the `\eval` command. Repeating the last example, everything is clearer if the `o` option is used:

`\eval[o]{$ \arccot 1 $}\degree` $\Longrightarrow \operatorname{arccot} 1 = 45^\circ.$

Inverses can also be constructed using the '−1' superscript notation. Thus

`\eval[p=\degree,o]{$ \sin^{-1}(1/\surd2) $}` $\Longrightarrow \sin^{-1}(1/\sqrt{2}) = 45^\circ.$

3.4.1.1 Complicated arguments

A general function f of x is usually written $f(x)$: the argument of the function is delimited by parentheses. In practice, with familiar functions, mathematicians often don't bother with the parentheses, even when the argument includes more than one factor: $\sin\frac{1}{2}\pi$, $\cos 2\pi t$, $\ln xy$ $(= \ln x + \ln y)$, and so on. So long as the argument is composed of numbers, constants, variables or `\tfrac`-s or `\sfrac`-s, `numerica` parses the argument without difficulty and without requiring parentheses to be inserted. As function arguments become more complicated, parentheses can become necessary to clarify just what expression the function is acting on. But trigonometric identities like

$$\sin A + \sin B = 2\sin\tfrac{1}{2}(A+B)\cos\tfrac{1}{2}(A-B)$$

and especially Fourier series where expressions like

$$\cos\frac{2\pi}{T}nt, \quad \cos\frac{2\pi}{T}n(t+\tfrac{1}{2}T), \quad \sin(N+\tfrac{1}{2})\frac{2\pi\tau}{T},$$

$$\sin 2\pi\left(\frac{x}{\lambda}-\frac{t}{T}\right), \quad \sin(n+\tfrac{1}{2})(x-t),$$

are a common occurrence, show that in practice parentheses that enclose the whole argument, even for complicated expressions, are often omitted. Context makes clear where the function argument ends and mathematicians read the expressions accordingly.

What should `numerica` do? Insist that the whole argument be parenthesized? But that results in expressions that are generally less pleasing to the eye and require more concentration to read, to disentangle the enclosing from the enclosed parentheses:

$$\sin\left(\tfrac{1}{2}(A+B)\right), \quad \sin\left(2\pi\left(\frac{x}{\lambda}-\frac{t}{T}\right)\right), \quad \sin\left((n+\tfrac{1}{2})(x-t)\right).$$

Admittedly square brackets and mathematical braces can help here, but mathematicians don't (generally) do this. The examples above are culled from a number of different texts that I had to hand – I didn't need to go searching for them. To insist that formulas be written in a 'forced' or pedantic way, goes against the underlying idea behind `numerica`: to evaluate expressions in the form in which they are typeset.

Rather, from version 3.0.0 the recommended way of handling such expressions is to put the *whole* argument of the function between LaTeX braces. (This applies not only to the trigonometric functions but also to *any* unary function.) Yes, inserting LaTeX braces does involve modifying the formula, but it fits naturally within LaTeX practice and, crucially, it makes *no change* to the `pdf` display. The formula retains its 'natural' appearance at the same time as the full argument is delimited so that `numerica` knows what to operate on. Thus

`\eval{$ \sin{\tfrac16(m+n)\pi} $}[m=1,n=2]`, $\Longrightarrow$
$\sin\frac{1}{6}(m+n)\pi = 1, \quad (m=1, n=2),$

which is $\sin\frac{1}{2}\pi$, and

```
\eval{\[
          \sin{2\pi\mleft(\frac{x}{\lambda}
             -\frac{t}{T}\mright)}
       \]}[x=1,\lambda=2,t=3,T=4]
```

$\Longrightarrow$

$$\sin 2\pi\left(\frac{x}{\lambda}-\frac{t}{T}\right)=-1, \qquad (x=1,\lambda=2,t=3,T=4)$$

which is the sine of $-\frac{1}{2}\pi = 2\pi\times(-\frac{1}{4})$.

In earlier versions of `numerica` there was a setting `()=0`, `1`, `2` (see Chapter 5 for a discussion of settings) which changed the way parentheses were parsed and allowed (most of) these usages. But it was difficult to document and remember exactly what was and was not allowed at each setting value, meaning the result of a calculation might not reflect what a user intended. Besides, dealing with the different setting values complicated the code. From version 3.0.0, with the use of braces to delimit such arguments, this setting has been withdrawn and now produces a `numerica` error message. The use of LaTeX braces to delimit the arguments of *mathematical* functions is more generally discussed at §2.1.1.2.

3.4.2 Hyperbolic functions

Four of the six hyperbolic functions: `\sinh`, `\cosh`, `\tanh`, and `\coth` are provided by LaTeX, and *no* inverses. `numerica` fills the gaps, providing the missing hyperbolic functions, `\csch` and `\sech`, and all missing inverses. There is no agreed notation in common use for the hyperbolic inverses. *HMF* writes arcsinh, arccosh, ..., ISO recommends arsinh, arcosh, ..., `l3fp` uses asinh, acosh, ... as do the computer algebra system `maxima` and the spreadsheet `LibreOffice Calc`. `numerica` makes no attempt to decide the issue. From version 3.0.0, it accepts all three forms for all six functions. All can be used within an `\eval` command:

```
\eval[pp,p=.,ff]{\[ \atanh\tanh 3, \sinh\arsinh 3,
                    \arcsech\sech 3 \]}
```

$\Longrightarrow$

$$\operatorname{atanh}\tanh 3 = 3,$$

$$\sinh\operatorname{arsinh} 3 = 3,$$

$$\operatorname{arcsech}\operatorname{sech} 3 = 3.$$

As for the trig. inverses, hyperbolic inverses can also be constructed using the '-1' superscript notation. Thus

`\eval{$ \coth\coth^{-1}1.5 $}` $\Longrightarrow \coth\coth^{-1} 1.5 = 1.5.$

3.4.2.1 Absence from l3fp

Please note that l3fp does not (as yet) provide *any* hyperbolic functions natively. The values numerica provides for these functions are *calculated* values using familiar formulas involving exponentials (for the direct functions) and natural logarithms and square roots for the inverses. Rounding errors mean the values calculated may not have 16-figure accuracy. The worst 'offenders' are likely to be the least used, \acsch and \asech. For instance,

$$\operatorname{acsch} x = \ln\left[\frac{1}{x} + \left(\frac{1}{x^2}+1\right)^{1/2}\right],$$

\eval{$ \csch \acsch 7 $}[15] $\Longrightarrow \operatorname{csch}\operatorname{acsch} 7 = 6.999999999999983.$

3.4.3 Logarithms

The natural logarithm \ln, base 10 logarithm \lg, and binary or base 2 logarithm \lb are all recognized, as is \log, preferably with a subscripted base:

\eval{$ \log_{12}1728 $} $\Longrightarrow \log_{12} 1728 = 3$

If there is no base indicated, base 10 is assumed. (The notations \ln, \lg, and \lb follow ISO 80000-2 recommendation, which frowns upon the use of the unsubscripted \log although only \ln appears to be widely used.) The base need not be explicitly entered as a number. It could be entered as an expression or be specified in the vv-list:

\eval*{$ \log_b c $}[b=2,c=1024] $\Longrightarrow 10,$

the log to base 2 in this case. It is possible to use the unadorned \log with a base different from 10; see §5.2.8.

3.4.4 Other unary functions

Other unary functions supported are the exponential function \exp, and signum function \sgn, equal to 1, −1, or 0, depending as its argument is positive, negative or zero.

```
\eval{$ \sgn(\exp(x)-e^x) $}[x=1],\quad
\eval{$ \sgn(e^x-\exp(x)) $}[x=2].
```

$\Longrightarrow \operatorname{sgn}(\exp(x) - e^x) = 0, \quad (x = 1), \quad \operatorname{sgn}(e^x - \exp(x)) = -1, \quad (x = 2).$

The first of these is expected, the second probably not. $\exp x$ is provided by l3fp, a built-in function; e^x is calculated by numerica, a number (e) raised to a power. They differ by 1 in the 15-th decimal place:

\eval[ff]{$ \exp 2, e^2 $}[15*] $\Longrightarrow$
$\exp 2 = 7.389056098930650 \quad e^2 = 7.389056098930649.$

3.4.5 Squaring, cubing, ... unary functions

`\eval` happily digests a familiar but 'incorrectly formed' expression like

$$\sin^2 1.234 + \cos^2 1.234.$$

You do not have to parenthesize like $(\sin 1.234)^2 + (\cos 1.234)^2$ or (heaven forbid) $(\sin(1.234))^2 + (\cos(1.234))^2$; the everyday usage is fine:

```
\eval{$ \sin^2\theta+\cos^2\theta $}[\theta=1.234] ⟹
```

$$\sin^2\theta + \cos^2\theta = 1, \quad (\theta = 1.234)\ .$$

Equally `\eval` has no difficulty reading the 'correct' pedantic form

```
\eval{$ (\sin(\theta))^2+(\cos(\theta))^2 $}[\theta=1.234] ⟹
```

$$(\sin(\theta))^2 + (\cos(\theta))^2 = 1, \quad (\theta = 1.234)\ .$$

A hyperbolic identity is corroborated in this example:

```
\eval[ff]{\[ \sinh 3x, 3\sinh x+4\sinh^3x \]}[x=1] ⟹
```

$$\sinh 3x = 10.017875, \qquad (x = 1)$$

$$3\sinh x + 4\sinh^3 x = 10.017875, \qquad (x = 1)$$

In fact all named unary functions in `numerica` can be squared, cubed, etc., in this 'incorrect' but familiar way, although the practice outside the trigonometric and hyperbolic contexts seems rare.

When the argument of the function is parenthesized and raised to a power – like $\sin(\pi)^2$ – it is read by `\eval` as the 'sine of the square of pi', $\sin(\pi^2)$, and *not* as the 'square of the sine of pi', $(\sin\pi)^2$:

```
\eval{$ \sin(\pi)^2 $} ⟹
```

$\sin(\pi)^2 = -0.430301$.

Things are done like this in `numerica` above all to handle the logarithm in a natural way. Surely (see *HMF* 4.1.11) $\ln x^n = n \ln x$? I.e. $\ln x^n = \ln(x^n)$ rather than $(\ln x)^n$. And if we wish to write (as we do) $\ln(1+1/n)^n = n\ln(1+1/n) = 1 - 1/2n + 1/3n^2 - \ldots$ to study the limiting behaviour of $(1+1/n)^n$, then we are committed to $\ln(x)^n = n\ln(x) = \ln(x^n)$ too.

3.5 *n*-ary functions

The functions of more than one variable (n-ary functions) that `numerica` supports are `\max`, `\min` and `\gcd`, greatest common divisor. The comma list of arguments (semicolon list if the `comma` package option has been used) to `\max`, `\min` or `\gcd` can be of arbitrary length. The arguments themselves can be expressions or numbers.

As implemented in `numerica`, for `\gcd` non-integer arguments are rounded to integers. Hence both y and $3y$ are independently rounded in the following example – to 81 and 243 respectively:

```
\eval{$ \gcd(12,10x^2,3y,y,63) $}[y=1/0.0123456789,x=3] ⟹
```

$$\gcd(12, 10x^2, 3y, y, 63) = 3, \quad (y = 1/0.0123456789, x = 3).$$

The rounding occurs within the greatest common divisor routine, not in the vv-list; the variable retains its original value. Modifying the example, this becomes evident in the sixth decimal place of the new result:

```
\eval{$ \gcd(12,10x^2,3y,y,63) + y $}[y=1/0.0123456789,x=3] ⟹
```

$$\gcd(12, 10x^2, 3y, y, 63) + y = 84.000001, \quad (y = 1/0.0123456789, x = 3).$$

For n-ary functions, squaring, cubing, etc. follow a different pattern from that for unary functions. The argument of these functions is a comma list. Squaring it makes no sense and we understand the superscript as applying to the function as a whole. (Consistency is not the point here; it is how a person reads the expression that `numerica` tries to accommodate.)

```
\eval{$ \gcd(3x,x,\arcsin 1/\deg)^2 $}[x=24] ⟹
```

$$\gcd(3x, x, \arcsin 1/\deg)^2 = 36, \quad (x = 24)\ .$$

3.6 Absolute value, floor & ceiling functions

It is tempting to use the | key on the keyboard for inserting an absolute value sign. `numerica` accepts this usage, but it is strongly deprecated. The spacing is incorrect – compare $|-l|$ using |, against $|-l|$ using `\lvert \rvert`. Also, with |, the identity of the left and right delimiters makes parsing nested absolute values difficult. `numerica` does not attempt to do so. `\lvert \rvert` are better in every way except ease of writing. To aid such ease `numerica` provides the `\abs` function (using the `\DeclarePairedDelimiter` command of the `mathtools` package). This takes a mutually exclusive star (asterisk) or square bracketed optional argument, and a mandatory braced argument. The starred form wraps `\left\lvert`, `\right\rvert` around the mandatory argument:

```
\eval[p=.]{\[ 3\abs*{\frac{\abs{n}}{21}-1} \]}[n=-7] ⟹
```

$$3\left|\frac{|n|}{21} - 1\right| = 2, \qquad (n = -7).$$

The optional argument provides access to the `\big...` modifiers:

```
\eval[p=.]{\[
    \abs[\Big]{\abs{a-c}-\abs[\big]{A-C}}
  \]}[A=12,a=-10,C=7,c=-5]
```

⟹

$$\Big||a - c| - \big|A - C\big|\Big| = 0, \qquad (A = 12, a = -10, C = 7, c = -5).$$

The form without either option dispenses with modifiers altogether:

\eval{$ \tfrac12(x+y)+\tfrac12\abs{x-y} $}[x=-3,y=7]. $\Longrightarrow$
$\frac{1}{2}(x+y)+\frac{1}{2}|x-y| = 7, \quad (x=-3, y=7).$

As noted, the star and square bracket options are mutually exclusive.

`numerica` also provides the functions `\floor` and `\ceil`, defined in the same way, taking a mutually exclusive star or square bracketed optional argument, the starred forms wrapping `\left\lfloor`, `\right\rfloor` or `\left\lceil`, `\right\rceil` around the mandatory argument, and the square bracket option forms replacing the `\left` and `\right` with the corresponding `\big` commands (see the `\abs` example above). The form without star or square-bracket option dispenses with any modifier at all.

\eval[pp,ff]{$ \floor{-\pi}, \ceil{\pi} $} $\Longrightarrow \lfloor -\pi \rfloor = -4, \quad \lceil \pi \rceil = 4.$

The floor function, $\lfloor x \rfloor$, is the greatest integer $\leq x$; the ceiling function, $\lceil x \rceil$ is the smallest integer $\geq x$. Like the absolute value, the floor and ceiling functions, can be nested:

\eval{$ \floor{-\pi+\ceil{e}} $} $\Longrightarrow \lfloor -\pi + \lceil e \rceil \rfloor = -1.$

3.6.1 Squaring, cubing, ... absolute values, etc.

These three functions can be raised to a power *without* extra parentheses:

\eval[pp,ff]{$ \ceil{e}^2, \floor{e}^2 $} $\Longrightarrow \lceil e \rceil^2 = 9, \quad \lfloor e \rfloor^2 = 4,$

\eval{$ \abs{-4}^2 $} $\Longrightarrow |-4|^2 = 16.$

3.7 Factorials, binomial coefficients

Factorials use the familiar trailing `!` notation:

\eval{$ 7! $} $\Longrightarrow 7! = 5040,$
\eval{$ (\alpha+\beta)!-\alpha!-\beta! $}[\alpha=2,\beta=3] $\Longrightarrow$
$(\alpha+\beta)! - \alpha! - \beta! = 112, \quad (\alpha = 2, \beta = 3).$

The examples illustrate how `numerica` interprets the argument of the factorial symbol: it 'digests'

- a preceding (possibly multi-digit) integer, or
- a preceding variable, or
- a bracketed expression, or
- a bracket-like expression.

A bracket-like expression is an absolute value, floor or ceiling function, since they delimit arguments in a bracket-like way:

`\eval{$ \abs{-4}!+\floor{\pi}!+\ceil{e}! $}` $\Longrightarrow$
$$|-4|! + \lfloor \pi \rfloor ! + \lceil e \rceil ! = 36.$$

The result of feeding the factorial an expression different in kind from one of these four cases may give an error message or an unexpected result. Use parentheses around such an expression; for example write $(3^2)!$, rather than $3^2!$.

Nesting of brackets for factorials is accepted:

`\eval{$ ((5-2)!+1)! $}` $\Longrightarrow ((5-2)!+1)! = 5040.$

The factorials of negative integers are not defined and raise a `numerica` error. It simplifies the code to treat the factorial of a positive non-integer as the factorial of the integer it rounds to, rather than raising an error. For the sake of the reader, in such circumstances, an author should make the rounding explicit:

`\eval{$ \floor{\pi}! $}` $\Longrightarrow \lfloor \pi \rfloor ! = 6.$

This rounding to an integer is different from the behaviour in earlier versions of `numerica` but should make no noticeable difference.

3.7.1 Double factorials

The double factorial, written $n!!$, is the product $n(n-2)(n-4)\ldots\times 4\times 2$ when n is even, and the product $n(n-2)(n-4)\ldots\times 3\times 1$ when n is odd:

`$\eval[pp,ff]{6!!, 5!!}$` $\Longrightarrow$ $48, \quad 15.$

As with factorials, the double factorial sign can be appended to a (possibly multi-digit) number, a variable, a bracketed expression or a bracket-like expression.

`\eval[env=\[,ff]{ n!!, (n-1)!!, \abs{2-n}!! }[n=\sqrt{49}]` $\Longrightarrow$

$$n!! = 105, \qquad (n = \sqrt{49})$$
$$(n-1)!! = 48, \qquad (n = \sqrt{49})$$
$$|2-n|!! = 15, \qquad (n = \sqrt{49})$$

Since $n! = n!!(n-1)!!$,

$$n!! = \frac{n!}{(n-1)!!} = \frac{(n+1)!}{(n+1)!!},$$

on multiplying top and bottom by $n+1$. Putting $n=0$ in the left and right expressions shows that $0!! = 1$. Now put $n=0$ in the left and middle expressions. We deduce that $(-1)!! = 1$. It follows that double factorials are defined for integers ≥ -1.

3.7.2 Binomial coefficients

Binomial coefficients are entered in LaTeX with the `\binom` command. It takes two arguments, `\binom{a}{b}` and scales like `\frac`: inline it displays as $\binom{a}{b}$, and in displaystyle as

$$\binom{a}{b}.$$

One can force textstyle with `\tbinom` and force displaystyle with `\dbinom`. As implemented in `numerica`, these are *generalised* binomial coefficients:

$$\binom{x}{k} = \frac{x(x-1)\dots(x-k+1)}{k(k-1)\dots 1}, \quad (x \in \mathbb{R},\ k \in \mathbb{N}),$$

where x need not be a positive or zero integer, and where $\binom{x}{0} = 1$ by definition.

`\eval[pp,p=.,ff]{$ \tbinom53, \tbinom70 $}` $\Longrightarrow \binom{5}{3} = 10, \quad \binom{7}{0} = 1.$

The first (or upper) argument can be any real number; it does not need to be an integer or positive: recalling that $\pi^2 \approx 9.87$,

`\eval[pp,p=.,ff]{$ \tbinom94, \tbinom{\pi^2}4, \tbinom{10}4 $}[3]` $\Longrightarrow$

$$\binom{9}{4} = 126, \quad \binom{\pi^2}{4} = 197.187, \quad \binom{10}{4} = 210.$$

If the second (or lower) argument of `\binom` is negative, `numerica` responds with a message:

`\eval{$ \binom 5{-3} $}` $\Longrightarrow$!!! Integer ≥ 0 required in {arg2} of: binomial coeff. !!!.

If the second argument is positive but not an integer, `numerica` rounds it to the nearest integer before calculating the binomial coefficient:

`\eval[pp,ff]{$ \binom 5e, \binom 53 $}` $\Longrightarrow \binom{5}{e} = 10, \quad \binom{5}{3} = 10.$

This differs from previous versions of `numerica` which would raise an error in this case. Although positive non-integers are now rounded to the nearest integer, out of consideration for the reader, an author should make explicit the fact that an integer has been used. In the example I should have written `\ceil{e}` rather than `e`.

3.8 Sums and products

`numerica` recognizes `\sum`, displaying as $\sum$, and `\prod`, displaying as $\prod$, and expects both symbols to have lower and upper summation or product limits specified. The lower limit must be given in the form *variable=initial value* where *variable* is the summation or product variable; the upper limit requires only the final value to be specified (although it can also be given in the form

variable=final value). The values may be expressions depending on other variables and values, and are rounded to integers. This differs from earlier versions of `numerica` where, if the result of a calculation differed too much from an integer, it prompted an error message. Now the rounding happens automatically, whatever the value. As in other similar contexts, for the sake of the reader an author should ensure that the integer value is explicit:

```
\eval[p]{\[ \sum_{n=\floor{\pi/e}}^{\ceil{\pi e}}n \]} ⟹
```

$$\sum_{n=\lfloor \pi/e \rfloor}^{\lceil \pi e \rceil} n = 45,$$

(which is $\sum_{n=1}^{9} n$). If the upper limit is less than the lower limit the result is zero. Notice that there is no vv-list. The summation variable does not need to be included there unless there are other variables that depend on it. However, in the case

```
\eval[p]{\[ \sum_{k=1}^N\frac1{k^3} \]}[N=100][4] ⟹
```

$$\sum_{k=1}^{N} \frac{1}{k^3} = 1.202, \qquad (N = 100),$$

the upper limit N is necessarily assigned a value in the vv-list.

To the author it seems natural to enter the lower limit first, immediately after the `\sum` command (the sum is *from* something *to* something), but no problem will accrue if the upper limit is placed first (after all, the appearance of the formula in the `pdf` is the same):

```
\eval[p=.]{\[ \sum^N_{k=1}\frac1{k^3} \]}[N=100][4] ⟹
```

$$\sum_{k=1}^{N} \frac{1}{k^3} = 1.202, \qquad (N = 100).$$

Another example of a sum, using binomial coefficients this time, is

```
\eval[p]{\[ \sum_{m=0}^5\binom{5}{m}x^m y^{5-m} \]}[x=0.75,y=2.25]
                                    ⟹
```

$$\sum_{m=0}^{5} \binom{5}{m} x^m y^{5-m} = 243, \qquad (x = 0.75, y = 2.25),$$

which is just

```
\eval{$(x+y)^5$}[x=0.75,y=2.25] ⟹
```

$(x+y)^5 = 243, \ \ (x = 0.75, y = 2.25),$

or 3^5. Now let's calculate a product:

```
\eval[p]{\[
  \prod_{k=1}^{100}
    \biggl(\frac{x^2}{k^2\pi^2} +1\biggr)
      \]}[x=1][3]
```

$\Longrightarrow$

$$\prod_{k=1}^{100}\left(\frac{x^2}{k^2\pi^2}+1\right) = 1.174, \qquad (x=1),$$

to be compared with `\eval{$ \sinh 1 $}[3]` $\Longrightarrow \sinh 1 = 1.175$. Obviously more terms than 100 are required in the product to achieve 3-figure accuracy.

3.8.1 Infinite sums and products

How many more? Let's 'go the whole hog' and put ∞ in the upper limit of this product:

```
\eval[p=.]{\[
  \prod_{k=1}^{\infty}
    \biggl(\frac{x^2}{k^2\pi^2} +1\biggr)
      \]}[x=1][3]
```

$\Longrightarrow$

$$\prod_{k=1}^{\infty}\left(\frac{x^2}{k^2\pi^2}+1\right) = 1.174, \qquad (x=1).$$

Disappointingly, we still get the same result, deficient by 1 in the third decimal place. Obviously `numerica` has not multiplied an infinite number of terms and, just as obviously, the finite number of terms it *has* multiplied are too few. How `numerica` decides when to stop evaluating additional terms in an infinite sum or product is discussed below in §3.8.2.

For this particular product the problem is that it converges slowly. Any criterion for when to stop multiplying terms or, for an infinite sum adding terms, seems bound to fail whenever convergence is sufficiently slow. Presumably any stopping criterion must measure smallness in some way. But terms of the divergent harmonic series $\sum(1/n)$, for example, can always be found smaller than any value we care to specify. It is not surprising that a stopping criterion will fail when convergence is slow enough. However, the default criterion can be changed: again, see below in §3.8.2.

Other infinite sums converge more rapidly, and the default settings work admirably for them. For example

`\eval{$ (1+0.1234)^{4.321} $}` $\Longrightarrow (1+0.1234)^{4.321} = 1.653329.$

Using binomial coefficients we can express this as an infinite sum:

```
\eval[p=.]{\[
    \sum_{n=0}^{\infty}\binom{\alpha}{n}x^{n}
      \]}[\alpha=4.321,x=0.1234]
```

Table 3.1: Stopping criterion settings

key	type	meaning	default
`S+`	int	extra rounding for sums	`2`
`S?`	int ≥ 0	number of query terms for sums	`0`
`P+`	int	extra rounding for products	`2`
`P?`	int ≥ 0	number of query terms for products	`0`

$\Longrightarrow$

$$\sum_{n=0}^{\infty} \binom{\alpha}{n} x^n = 1.653329, \qquad (\alpha = 4.321, x = 0.1234).$$

3.8.2 The stopping criterion

There are ways of tweaking various parameters to nudge infinite sums and products to a correct limit. These tweaks are applied via the settings option of the `\eval` command.

The normal convergence criterion used by `numerica` to determine when to stop adding/multiplying terms in an infinite sum/product is *when the next term added/multiplied leaves the total unaltered when rounded to 2 more digits than the specified rounding value.* Suppose L_k is the partial sum/product after k terms, and r is the rounding value.[1] Let $(L_k)_r$ denote L_k rounded to r figures. *The infinite sum or product stops at the* $(k+1)$*-th term (and the value is attained at the* k*-th term) when* $(L_{k+1})_{r+2} = (L_k)_{r+2}$. The hope is that if this is true at rounding value $r+2$ then at rounding value r the series or product will have attained a stable value at that smaller rounding value.

For a series of monotonic terms converging quickly to a limit, this stopping criterion works well, less so if convergence is slower, as seen earlier with the infinite product for $\sinh 1$. The criterion can fail completely when terms behave in a non-monotonic manner. Terms of a Fourier series, for example, may take zero values so that $L_{k+1} = L_k$ and, *a fortiori*, $(L_{k+1})_{r+2} = (L_k)_{r+2}$; the criterion is necessarily satisfied but the series may still be far from its limit. In a product the equivalent would be a term taking unit value. Sometimes the initial terms of series or products are 'irregular' and take these 'stopping' values meaning sum or product would stop after only one or two additions/multiplications and far from any limit.

To cope with these possibilities, `numerica` offers two settings for sums, two for products, summarized in Table 3.1. These are entered in the settings option of the `\eval` command.

- `S+=<integer>` `(P+=<integer>)` *additional* rounding on top of the specified (or default) rounding for the calculation; the larger `<integer>` is, the

[1] E.g. if $T(n)$ is the n-th term then the partial sum $L_k = \sum_{n=1}^{k} T(n)$.

more likely that sum or product has attained a stable value at the specified rounding value r; default = 2

- `S?=<integer>` ≥ 0 `(P?=<integer>` ≥ 0`)` the number of terms to *query* after the stopping criterion has been achieved to confirm that it is not an 'accident' of particular values; default = 0

  – once the stopping criterion has been met, we add/multiply these next few terms to the result and check at each step whether the result is unchanged at the specified rounding value. Suppose the additional rounding (`S+` or `P+`) is δr on top of the specified rounding r and let the number of query terms be q. (By default $\delta r = 2$ and $q = 0$.) Suppose L_{k_0} is the first term at which the stopping criterion is achieved. That means $(L_{k_0})_{r+\delta r} = (L_{k_0+1})_{r+\delta r}$. What we require of the query terms is that $(L_{k_0})_{r+\delta r} = (L_{k_0+1+j})_{r+\delta r}$ for $j = 0, 1, \ldots, q$.

Earlier we found that the infinite product for $\sinh 1$ with the default settings gave the wrong value, 0.174, deficient by 1 in the last digit. We now have the means to tweak the stopping criterion by increasing the additional rounding:

```
\eval[p,P+=3]{\[
  \prod_{k=1}^{\infty}
  \biggl(\frac{x^2}{k^2\pi^2} +1\biggr)
\]}[x=1][3]\nmcInfo{prod}.
```

$\Longrightarrow$

$$\prod_{k=1}^{\infty}\left(\frac{x^2}{k^2\pi^2}+1\right) = 1.175, \qquad (x = 1),$$

350 factors. To obtain that last item of information (350 factors), I've anticipated a little and used the command `\nmcInfo` with the argument `prod` (see §6.1). The product now produces the correct three-figure value, but it takes 350 factors to do so.

Knowing how many terms or factors have been needed helps assess how trustworthy the result from an infinite sum or product is. For example, for the exponential series,

```
\eval[p]{\[
  \sum_{k=0}^\infty \frac1{k!}
\]}[9]\nmcInfo{sum}.
```

$\Longrightarrow$

$$\sum_{k=0}^{\infty}\frac{1}{k!} = 2.718281828,$$

To 9 places of decimals, using the default value `S+=2`, the exponential series arrives at the right sum after only 15 terms. Convergence is rapid. We can trust this result (and it is in fact the correct nine-figure value). By contrast, if we didn't know the value of $\sinh 1$ beforehand, noting the number of factors

required would make us justly cautious about accepting the result of the infinite product calculation.

One way to gain confidence in a result is to choose a possibly unrealistic rounding value r – say the default 6 for the infinite product – then use *negative* values for the additional rounding, `S+=-5`, `S+=-4`, ... , so that the stopping criterion applies at rounding values $s = r + S_+$ of $6+(-5) = 1$ decimal place, $6+(-4) = 2$ decimal places, and so on, but the result is always presented to 6 decimal places. You can then see how the 6-figure results behave relative to the number of terms it takes to meet the stopping criterion. A little experimenting shows that for the infinite product for $\sinh 1$ the number of factors N_s required at a stopping rounding value s increases in geometric proportion with a scale factor of about 3: $N_s \approx \text{const} \times 3^s$. This rapidly becomes large ($3^4 = 81, 3^5 = 243 \ldots$). For the exponential series on the other hand $N_s = 4 + s$, the number of terms increases only slowly, in direct proportion to the stopping rounding value.

Similar experiments with the sums of inverse fourth, third and second powers of the integers, using `\nmcInfo` to find how many terms are required at each stopping rounding value, show that at least over the rounding value range 1 to 8, for inverse fourth powers $N_s \approx \text{const} \times 1.7^s$, for inverse third powers $N_s \approx \text{const} \times 2^s$ and for inverse squares $N_s \approx \text{const} \times 3^s$. All are geometric rather than arithmetic progressions, but for inverse fourth powers the scale factor (≈ 1.7) is sufficiently small that for these low values of s the number of terms required does not grow too quickly (e.g. $1.7^6 \approx 24$).

It is a standard result (Euler) that the inverse fourth power series sums to $\pi^4/90$: `$ \eval{ \pi^4/90 } $` $\Longrightarrow$ 1.082323 to six places, and indeed, with the default rounding value 6 and default extra rounding `S+=2`,

```
\eval[p]{\[ \sum_{k=1}^\infty \frac1{k^4} \]} ⟹
```

$$\sum_{k=1}^{\infty} \frac{1}{k^4} = 1.082323.$$

3.8.2.1 Premature ending of infinite sums

All the series considered so far have been monotonic. Trigonometric series will generally not be so, nor even single-signed.

Trigonometric sums are computationally intensive and so, for the following example, I have specified a rounding value of 2. The series

$$\sum_{n=1}^{\infty} \frac{4}{n^2\pi^2}(1 - \cos n\pi)\cos 2\pi nt$$

is the Fourier series for the triangular wave function ⋀⋀⋀ ... of period 1,

symmetric about the origin where it takes its maximum value 1, crossing the t-axis at $t = 0.25$ and descending to its minimum -1 at $t = 0.5$, before ascending to a second maximum at $t = 1$ (and so on). In the interval $[0, 0.5)$ the series should sum to $1-4t$. The problem is that the summand $\frac{4}{n^2\pi^2}(1-\cos n\pi)\cos 2\pi nt$ vanishes both when n is even and when $4nt$ is an odd integer. If $t = 0.1$ then $4nt$ is never an odd integer so the summand vanishes only for n even, every second term. We expect the result to be $1 - 4 \times 0.1 = 0.6$.

```
\eval[p]{\[
  \sum_{n=1}^{\infty}
    \frac{4}{n^{2}\pi^{2}}
    (1-\cos n\pi)\cos2\pi nt
\]}[t=0.1][2]\nmcInfo{sum}.
```

$\Longrightarrow$

$$\sum_{n=1}^{\infty} \frac{4}{n^2\pi^2}(1 - \cos n\pi)\cos 2\pi nt = 0.66, \qquad (t = 0.1),$$

1 term. Only one term? Of course – in the second term $n = 2$ is even so the term vanishes and the stopping criterion is satisfied. The way around this problem is to query terms *beyond* the one where the stopping criterion is achieved, i.e. to set `S?` to a nonzero value. We try `S?=1`:

```
\eval[p,S?=1]{\[
  \sum_{n=1}^{\infty}
    \frac{4}{n^{2}\pi^{2}}
    (1-\cos n\pi)\cos2\pi nt
\]}[t=0.1][2]\nmcInfo{sum}.
```

$\Longrightarrow$

$$\sum_{n=1}^{\infty} \frac{4}{n^2\pi^2}(1 - \cos n\pi)\cos 2\pi nt = 0.6, \qquad (t = 0.1),$$

65 terms.

Table 3.2: Partial sums

N	Σ
63	0.6001
64	0.6001
65	0.5999
66	0.5999
67	0.5999

Table 3.2 lists the results of evaluating the partial sums from $n = 1$ to $n = N$ for values of N around 65. Since the specified rounding value is 2 for the calculation, the stopping criterion applies at a rounding value of $2 + 2 = 4$. Since $N = 64$ is even, the 64th term is zero and the sum takes the same value as for $N = 63$. The 65th term is the query term and the sum differs, so the summation continues. The 66th term vanishes, so the stopping criterion is met. This time for the query term, the 67th, the sum retains the same 4-figure value, and the summation

stops. The result was attained at the 65th term. Should we be confident in the result? Increase the number of query terms to 3 (there is no point in increasing `S?` to 2 because of the vanishing of the even terms); the sum stops after 113 terms, with the same 0.6 result.

For a final example, consider the error function

$$\operatorname{erf} z = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2}\, dt$$

which can also be rendered as an infinite sum (*HMF* 7.1.5):

$$\operatorname{erf} z = \sum_{n=0}^{\infty} (-1)^n \frac{z^{2n+1}}{n!\,(2n+1)}.$$

(`\erf` expanding to `erf` has been defined in the preamble to this document using `\DeclareMathOperator`.) We calculate this sum when $z = 2$ to 10 places of decimals. Although this is an alternating series, it is obvious that the summand never vanishes when $z \neq 0$ as here. Hence there seems no need to change the default value `S?=0`.

```
\eval[p]{\[
  \frac2{\sqrt{\pi}}
    \sum_{n=0}^\infty(-1)^n
      \frac{z^{2n+1}}{n!\,(2n+1)}
\]}[z=2][10*]\nmcInfo{sum}.
```

$\Longrightarrow$

$$\frac{2}{\sqrt{\pi}} \sum_{n=0}^{\infty} (-1)^n \frac{z^{2n+1}}{n!\,(2n+1)} = 0.9953222650, \qquad (z = 2),$$

26 terms. According to *HMF* Table 7.1, this calculated value of $\operatorname{erf} 2$ is correct to all 10 places. But beyond $z = 2$ errors will begin to interfere with the result. Note that 26 terms means $n = 26$ was the last value of n for which the summand was evaluated. (The sum stops at the 26th term, $n = 25$, but the next term $n = 26$ needs to be calculated for the stopping criterion.) Fortuitously, $2^{2\times 26+1} = 2^{53}$ is the greatest power of 2 that can be *exactly* rendered to the 16 significant figures that `l3fp` uses. But $n!$ exceeds the 16-significant figure limit of `l3fp` when $n > 21$, so despite the 10-figure result, errors have already begun to occur in the denominator of the summand and accrue in the sum when $z = 2$. For larger z values the errors can only get worse and at some point will render the calculated value worthless at any meaningful rounding value. For example, when $z = 7$ the sum apparently 'evaluates' to over 929 whereas we know that

$$\operatorname{erf} z < \frac{2}{\sqrt{\pi}} \int_0^{\infty} e^{-t^2}\, dt = 1.$$

3.8.2.2 Double sums or products

Sums or products can be iterated. For instance, the exponential function can be calculated this way:

```
\eval[p]
  {\[ \sum_{k=0}^{\infty}
    \prod_{m=1}^{k}\frac{x}{m} \]}[x=2]
```

$\Longrightarrow$

$$\sum_{k=0}^{\infty}\prod_{m=1}^{k}\frac{x}{m} = 7.389056, \qquad (x=2),$$

which is `\eval{$ e^2 $}` $\Longrightarrow 7.389056$.

A second example is afforded by Euler's transformation of series (*HMF* 3.6.27). To calculate e^{-1} we use

```
\eval[p]
  {\[ \sum_{n=0}^{\infty}
          \frac{(-1)^{n}}{n!} \]}[3]\info{sum}.
```

$\Longrightarrow$

$$\sum_{n=0}^{\infty}\frac{(-1)^n}{n!} = 0.368,$$

9 terms. Following Euler, this series can be transformed to the form

```
\eval[p,S?=1]{\[
  \sum_{k=0}^\infty \frac{(-1)^k}{2^{k+1}}
  \sum_{n=0}^k(-1)^n\binom kn \frac1{(k-n)!}
\]}[3]\nmcInfo{sum}.
```

$\Longrightarrow$

$$\sum_{k=0}^{\infty}\frac{(-1)^k}{2^{k+1}}\sum_{n=0}^{k}(-1)^n\binom{k}{n}\frac{1}{(k-n)!} = 0.368,$$

16 terms. Note the setting `S?=1`. Without it, the summation stops after 1 term, the $k=0$ term, because the $k=1$ term vanishes. With `S?=1` it takes 16 terms of the *outer* sum to reach the stopping criterion. Since that sum starts at 0, that means that changing the upper limit from ∞ to 15 should give the same result – which it does, taking $\frac{1}{2}\times 16\times 17 = 136$ terms in total to get there, to be compared with the 9 terms of the earlier simpler sum, and the terms are more complicated. Obviously such double sums are computationally intensive.

3.9 Formatting commands

There are many formatting commands which change the layout of a formula on the page but do not alter its calculational content. `numerica` copes with a great

many of these, although there will surely be some that have been overlooked[2] and which will trigger an 'Unknown token' message; see §4.

3.9.1 Spaces, phantoms, struts

These include cryptic forms like `\,` `\:` and `\>`, `\;` and the corresponding 'verbose' forms, `\thinspace`, `\medspace` and `\thickspace` and their negative equivalents `\!` or `\negthinspace`, `\negmedspace` and `\negthickspace`:

`\eval{$ 1\negthickspace+\negthickspace 1 $}` $\Longrightarrow 1+1=2$

which is a tiny bit tighter than the text spacing, 1+1, and much tighter than the usual math spacing $1+1$ – but it doesn't affect the result of the calculation.

Other spacing commands are `\quad` and `\qquad`, and `\hspace{arg}` and `\mspace{arg}`. For `\hspace` there is also a starred form, `\hspace*{arg}`. Phantoms similarly take an argument: `\phantom{arg}`, `\hphantom{arg}` and `\vphantom{arg}`.

`\eval{$ 1\hphantom{mmm}+\hphantom{mmm}1 $}` $\Longrightarrow 1 \qquad + \qquad 1=2.$

Like `\vphantom`, struts allow vertical spacing adjustments. `numerica` should digest both `\xmathstrut[optarg]{arg}` from `mathtools` and its 'baby cousin' `\mathstrut` from TEX. An example from *The TEX book* demonstrating the use of `\mathstrut` is

```
\eval{$\sqrt{\mathstrut a}+\sqrt{\mathstrut d}+
      \sqrt{\mathstrut y}$}[a=4,d=9,y=16]
```

$\Longrightarrow \sqrt{a}+\sqrt{d}+\sqrt{y}=9, \quad (a=4, d=9, y=16)$.

And here is an evaluation of a somewhat ridiculous expression modified from the `mathtools` documentation that uses `\xmathstrut`:

```
\eval{\[ \frac{ \frac{\xmathstrut{0.1} 2\ceil x-1}
       { \xmathstrut{0.25} \ceil x-\sin x } }
         {\xmathstrut{0.4} \sqrt{10-\ceil x} } \]}
           [x=\pi/6]
```

$\Longrightarrow$

$$\frac{\frac{2\lceil x\rceil-1}{\lceil x\rceil-\sin x}}{\sqrt{10-\lceil x\rceil}}=0.666667, \qquad (x=\pi/6)$$

3.9.1.1 `\mkern`, `\mskip`

From version 3.0.0, both `\mkern` and `\mskip` are recognized by `numerica`. `\mkern` should be followed either by an explicit space specification in `mu` (math

[2] Please contact the author in that case: ajparsloe@gmail.com

units), like `3 mu` (or `3mu`), or a control sequence containing such a specification; `\mkern` should be followed by an explicit 'glue' specification or a control sequence containing such a specification. A glue spec. is a distance in `mu` possibly followed by some stretch and shrink, e.g. `3 mu plus 1 mu minus 2 mu` (or `3muplus1muminus2mu`) with or without the `plus` and `minus` parts. A silly example of the use of `\mkern` and `\mskip` is the following:

```
\def\negvmu{-5mu}
\eval[env=$]{ 1 \mkern \negvmu +
                \mskip 18mu plus 6mu minus 9mu 1 }
```

$\Longrightarrow$ 1+ 1 = 2.

3.9.2 \splitfrac, \splitdfrac

The `mathtools` package provides `\splitfrac` and `\splitdfrac` to aid handling of clumsy fractions. I've mangled the example in the `mathtools` documentation illustrating this command to produce an even more ridiculous illustration, adding to the mess an enormous square root, `\left` and `\right` modifiers, and command-form parentheses; also the use of `\dfrac`. In the other direction, the `vv=` in the settings option suppresses the vv-list (see §5.3.2.3). A little mental arithmetic will convince that we are evaluating the square root of $(9\times 7)^2$ which indeed is what we get:

```
\eval[p=.,vv=]
  {\[ \sqrt{ \left\lparen
      \frac{ \splitfrac{xy + xy + xy + xy + xy}
            {+ xy + xy + xy + xy}
           }
           { \dfrac z7}
             \right\rparen \left\lparen
      \frac{ \splitdfrac{xy + xy + xy + xy + xy}
             {+ xy + xy + xy + xy}
             }
             {\dfrac z7}\right\rparen}
   \]}[x=2,y=5,z=10]
```

$\Longrightarrow$

$$\sqrt{\left(\frac{\begin{array}{c}xy+xy+xy+xy+xy\\ +xy+xy+xy+xy\end{array}}{\dfrac{z}{7}}\right)\left(\frac{\begin{array}{c}xy+xy+xy+xy+xy\\ +xy+xy+xy+xy\end{array}}{\dfrac{z}{7}}\right)} = 63.$$

3.9.3 Colour

(Anglicised spelling at least for the heading!) If you add to the preamble of your document the line

```
\usepackage{color}
```

two commands become available, `\textcolor[optarg]{arg1}{arg2}` and the declaration form of command, `\color[optarg]{arg}`. `numerica` readily accepts the former in a formula to be evaluated:

```
\eval{$ \sin \tfrac\pi6n\textcolor{red}{T}+1 $}[T=9,n=3] ⟹
```
$$\sin \tfrac{\pi}{6}nT + 1 = 2, \quad (T = 9, n = 3)$$

(assuming you had some wish to highlight the time T). You can even colour the T in the vv-list too, but it adds a lot of typing for a small gain:

```
\eval{$ \sin \tfrac\pi6n\textcolor{red}{T}+1 $}[\textcolor{red}{T}=9,n=3]
```
$$\Longrightarrow \sin \tfrac{\pi}{6}nT + 1 = 2, \quad (T = 9, n = 3).$$

However `\color` is a *declaration* form of command. It has effect until the end of the current group or environment. If you want to restrict it to only part of that group you need to em-brace the command and what it is to apply to,

```
<pre-stuff>{\color{red}<red-stuff>}<post-stuff}
```

but that is where the problem arises. 'Unannounced' brace groups (see §2.1.1.2) can easily result in unexpected results or LaTeX errors. Writing

```
\eval{$ \color{red} \sin \tfrac\pi6nT+1 $}[T=9,n=3]⟹
```
$$\sin \tfrac{\pi}{6}nT + 1 = 2, \quad (T = 9, n = 3)$$

is fine. So too, because the `\color` command is 'trapped' within the braces defining the first argument of the `\frac`, is

```
\eval{$ \frac{\color[gray]{0.5}A}B $}[A=12,b=4]
```
$$\Longrightarrow \tfrac{A}{B} = 3, \quad (A = 12, B = 4).$$

(Both arguments of the `\color` command have been used for grayscale output.) Also fine is

```
\eval{$ 3{\color{red}x}+1 $}[x=2] ⟹ 3x + 1 = 7, (x = 2)
```

because juxtaposition in this case means multiplication, but substituting the actual value `2` for `x` produces

```
\eval{$ 3{\color{red}2}+1 $} ⟹ 32 + 1 = 33.
```

Recall the discussion at §2.1.1.2. An *unannounced* brace group is simply appended to what has gone before. The resulting juxtaposition of 3 and x means multiply which was what was intended, but the juxtaposition of 3 and 2 has quite a different meaning.

3.9.4 \text, \mbox and font commands

The content of a \text or \mbox command is *invisible* to the \eval command and is ignored in calculations,

\eval*{ 1/0.0123456789 \mbox{approx.\ 81} }[5] $\Longrightarrow$ 81,

even when the \text or \mbox contains mathematical content. The author's view is that these commands are likely to be used for comments, annotations, or remarks, rather than to modify symbols of calculational significance.

Conversely, *specific* font commands, like \mathbf or \textrm, may well apply to such symbols and so there content should be *visible* to \eval. This becomes useful should numbers be input in scientific notation (see §5.2.1) where \textrm or \texttt may be useful so that a number in scientific notation like 2e-1 appearing in a formula or the vv-list can display correctly rather than inappropriately as the algebraic expression $2e - 1$.

The complete list of font commands where the content is visible to \eval is

1. \mathrm{}, \mathit{}, \mathtt{}, \mathbf{}, \mathsf{}, \mathcal{}, \mathbb{}, \mathfrak{}, \mathscr{}
2. \textrm{}, \textsf{}, \texttt{}, \textit{}, \textsl{}, \textbf{}, \textsc{}

The commands \mathbb{} and \mathfrak{} require the amsfonts package to be loaded; \mathscr requires the euscript package to be loaded with the mathscr package option.

3.10 Environment precedence

Math environments are relevant particularly for multi-formula calculations. For a single formula evaluation, either an inline ($) or equation* (\[) environment is probably all that is needed.

The environment of a calculation is determined in one or two of three ways: first, \eval may sit within an environment,

```
\begin{env}
\eval{...}
\end{env}
```

Second, the env setting may be used, \eval[env=...]{...}. And third, \eval may wrap around an environment, e.g. \eval{\[...\]}. Should a user, inadvertently or otherwise, specify 'extra' environments in a calculation, the environment wrapping around \eval takes precedence over the env-setting environment which takes precedence over the environment wrapped within \eval:

$\eval[env=align,pp,ff]{\[\pi,e,\phi,\gamma \]}[3]$ $\Longrightarrow$
3.142, 2.718, 1.618, 0.577.

As you can see, the inline `$` delimiters have prevails over the `align` environment and `\[ \]` delimiters.

Of course for the `-ed` environments of `amsmath` a surrounding math environment is necessary – as it is for `cases`, `dcases` and `array` environments. If `\eval` wraps around an `-ed` or like environment, the surrounding environment may either be `env`-specified or wrap around `\eval` (and if both are used, inadvertently or otherwise, the latter takes precedence).

Having these three different ways of specifying the environment may seem like overkill, but they are likely to arise in different contexts. The author envisaged a user writing an expression within an environment and then wondering what it evaluates to. Surely it is natural in this case just to wrap `\eval` around the lot, environment and expression, and let *it*, `\eval`, do the *formula=result* typesetting. Or, one may enter an expression into `\eval` *before* deciding on the environment. How much easier just to write `env=align*` (or whatever the environment chosen is) in the settings option than to insert `\begin{align*}` before the expression and `\end{align*}` after. Finally, `\eval` can find itself within a surrounding environment in many ways. It may be necessary in order to use an AMS `-ed` environment, or it may occur in a context like that in §1.1.6.2. The upshot is that all three ways of specifying environments should be – and are – catered for as of version 3.0.0.

Chapter 4

Error messages

There are two kinds of error in `numerica`: those in the underlying LaTeX which are reported in the LaTeX log and shown on the terminal, and generally halt compilation, and specifically `numerica`-related errors which do not halt compilation and produce messages displayed in the pdf where one would expect the result of the calculation to be. The original reason for doing things this way was to enable `numerica` to be used effectively with the instant preview facility of the document processor LyX. More philosophically, one might view such errors as similar to errors of grammar or spelling mistakes in text. It is not clear that they should halt compilation.

Hence strictly `numerica`-related errors leave brief messages in the pdf at the offending places. From version 3.0.0 of `numerica` they also leave messages in the LaTeX log like

```
numerica error on line <n> in <location>
```

where `<location>` will usually be `formula` or `variable=value list` or possibly some more specific location like `sum` or `product`. `<n>` is the line number in the `.tex` file where the error occurs. These messages in the log do not halt compilation. They allow the user to pinpoint – especially helpful in a long document – the actual line in the `.tex` file where the `numerica` error occurs.

Before discussing specific error messages, note that there is a debug facility (of a sort) discussed below in §5.1.

4.1 Specific messages

`numerica` error messages that appear in the `pdf` in place of the expected result are in two parts: a *what* part and a *where* part.

4.1.1 Mismatched brackets

An unmatched left parenthesis or other left bracket (in this case a missing right parenthesis) usually results in a `numerica` error:

`$\eval{\sin(\pi/(1+x)}[x=1]$` ⟹ !!! Unmatched (in: formula. !!!

For the same error in the vv-list, the *what*-part remains unchanged but the *where*-part is altered:

`$\eval{ 1+y }[x=1,y=\sin(\pi/(1+x)]$` ⟹
!!! Unmatched (in: variable=value list. !!!

An unmatched right parenthesis or other right bracket (in this case a missing *left* parenthesis) usually results in a similar `numerica` error:

`$\eval{2((x+y)/(y+z)))^2}[x=1,y=2,z=3]$` ⟹
!!! Unmatched) in: formula. !!!

But note that an unmatched modifier like `\left` or `\right` is a LaTeX error and is caught by LaTeX before `numerica` can respond and so results in a terminal and logfile message.

4.1.2 Unknown tokens

An 'Unknown token' message can arise in a number of ways. If an expression involves a number of variables, some of which depend on others, their order in the vv-list matters:

`$\eval{\tfrac12 vt}[t=2,v=gt,g=9.8]$` ⟹ !!! Unknown token `t` in:
variable=value list. !!!

The vv-list is evaluated from the *right* so that in this example the variable `v` depends on a quantity `t` that is not yet defined. Hence the message. The remedy is to move `t` to the right of `v` in the vv-list.

Similarly, if we use a variable in the formula that has not been assigned a value in the vv-list, we again get the 'Unknown token' message, but this time the location is the formula:

`$\eval{\pi r^2h}[r=3]$` ⟹ !!! Unknown token `h` in: formula. !!!

The remedy obviously is to assign a value to `h` in the vv-list.

The same message will result if a mathematical operation or function is used that has not been implemented in `numerica`:

`$\eval{u \bmod v }[v=7,u=3]$` ⟹ !!! Unknown token `\bmod` in: formula.
!!!

A missing comma in the vv-list will generally result in an unknown token message:

`$\eval{axy}[a=3 y=2,x=1]$` ⟹ !!! Unknown token `y` in: variable=value list.
!!!

Because of the missing comma, `numerica` sees only two variables in the vv-list, `x` and `a` and assumes `a` has the 'value' `3y=2`, an expression which it then tries to evaluate, but `y` has not been assigned a value. Parenthetically, if you give `y` a value, say `y=2/3`, the expression evaluates, treating the combination `3 y=2` as a comparison evaluating to true for this value of `y`, meaning `a=1` and

`$\eval{axy}[a=3 y=2,y=2/3,x=1]$` $\Longrightarrow$ 0.666667.

Extra commas in the vv-list should cause no problems:

`$\eval{axy}[,a=3,,y=2,x=1,]$` $\Longrightarrow$ 6 .

The presence of multi-token variables can also cause an unknown token message if the check for such variables is turned off; see §5.2.2.

4.1.3 Overlooked value assignments

Perhaps if one is evaluating a long formula with a number of variables and assigning different experimental values to them to see the effect, a variable might be overlooked (I have done this). The example is too simple to be a likely candidate for this error but shows the message:

`$\eval{axy}[a=3,y=,x=1]$` $\Longrightarrow$!!! No value for y in: variable=value list. !!!

In the example the variable `y` has been overlooked. The remedy is obvious.

4.1.4 Negative integers in the wrong place

Factorials (apart from the double factorial $(-1)!! = 1$), binomial coefficients, and n-th roots, require positive or at least non-negative integers.

`\eval{$ \sqrt[-1]{2} $}` $\Longrightarrow$!!! Integer > 0 required for [arg] of: \sqrt. !!!

`\eval{$ (-3)! $}` $\Longrightarrow$!!! Integer ≥ 0 required in: factorial. !!!

`\eval{$ \binom{7}{-3} $}` $\Longrightarrow$!!! Integer ≥ 0 required in {arg2} of: binomial coeff. !!!

4.1.5 Invalid base for \log

ISO recommends using `\log` only with a subscripted base specified, a recommendation honoured in the breach rather than the observance. `numerica` assumes that when `\log` is used unsubscripted, the base is 10 and that `\ln` is used for base `e`. Suppose you want to make 12 the base, but forget to put braces around the 12:

`$\eval{ \log_12 1728 }$` $\Longrightarrow$
!!! Valid base required for \log in: formula. !!!

Here, `numerica` has taken 1 as the base (and 2 as the argument) of the logarithm and responds accordingly.

4.1.6 Environment errors

Errors can arise from environments wrongly used, although environmental precedence (§3.10) sidesteps a number of apparent problems. Some environments – `aligned`, `alignedat`, `gathered`, `cases`, `dcases` and `array` – can be used only within another math environment. Thus

```
\eval[env=aligned,ff]{ \sin x, \cos x, \tan x }[x=\pi/6]
```
$\Longrightarrow$!!! Math mode needed for `aligned` environment in: settings. !!!

The remedy, obviously, is to put the `\eval` command between, say, `\[`, `\]` delimiters. In the other direction, because of environment precedence, forgetting the `ed` at the end of the `env`-ironment does not result in an error but displays the result in the outer, wrapping environment:

```
\[ \eval[env=align,p=.,ff]
      { \sin x, \cos x, \tan x}[x=\pi/6] \]
```

$\Longrightarrow$

$$\begin{aligned} 0.5, &\qquad (x = \pi/6)\\ 0.866025, &\qquad (x = \pi/6)\\ 0.57735, &\qquad (x = \pi/6). \end{aligned}$$

An unknown environment produces a message:

```
\eval[env=foo]{ \pi }\par
\eval{\begin{foo} \pi \end{foo}}
```

$\Longrightarrow$!!! Unknown math environment `foo` in: settings. !!!
!!! Unknown math environment `foo` in: formula. !!!

4.1.7 `l3fp` errors

Some errors arising at the `l3fp` level are trapped and a message displayed.

4.1.7.1 Dividing by zero

`$\eval{1/\sin x}[x=0]$` $\Longrightarrow$!!! `l3fp` error 'Division by zero' in: formula. !!!

Note however that

`$\eval{1/\sin x}[x=\pi]$` $\Longrightarrow$ 4193528956200936,

because of rounding errors in distant decimal places. No doubt this is true for other functions as well.

4.1.7.2 Invalid operation

Finding inverse integer powers of *positive* numbers should always be possible, but raising a *negative* number to an inverse power generates an error even when – mathematically – it should not:

`\eval{$ (-125)^{1/3} $}` ⟹ !!! `l3fp` error 'Invalid operation' in: formula. !!!

This is a feature of floating point arithmetic. When a number is raised to a rational power, say p/q where p and q are non-zero integers, then the result is the pth power of the q-th root of the number. Can a q-th root be taken? If our floating point system used (for ease of illustration) only 4 significant digits, $p/q = 1/3$ would be the fraction $3333/10^4$, an odd numerator over an even denominator. But a negative number does not possess an even (10^4th) root. The user needs to take care of the minus sign, in this case simply by omitting the parentheses.

Trying to evaluate a function like a factorial or square root or inverse trig. function outside its domain of definition also produces this error:

`$\eval{\arccos x}[x=2]$` ⟹
!!! `l3fp` error 'Invalid operation' in: formula. !!!

In this case the inverse cosine, which is defined only on the interval $[-1, 1]$, has been fed the value 2.

Trying to evaluate an expression that resolves to 0/0 also produces this message:

`$\eval{\frac{1-y}{x-2}}[x=2,y=1]$` ⟹ !!! `l3fp` error 'Invalid operation' in: formula. !!!

4.1.7.3 Overflow/underflow

The factorial (discussed in §3.7) provides an example of overflow:

`$\eval{3249!}$` ⟹ !!! `l3fp` error 'Overflow' in: formula. !!!

This is hardly surprising since

`$\eval{3248!}[x]$` ⟹ $1.973634 \times 10^{9997}$.

There is a limit on the size of exponents that `l3fp` can handle. A number in the form $a \times 10^b$ must have $-10001 \leq b < 10000$. If this is not the case an overflow or underflow condition occurs. As the examples show, an overflow condition generates a `numerica` error.

For underflow, where the number is closer to 0 than 10^{-10001}, `l3fp` assigns a zero value to the quantity. `numerica` accepts the zero value and the error is ignored.

4.1.8 Obsolete settings

(For settings, see the next chapter.) Some settings in earlier versions of `numerica` may be superseded by later developments and rendered obsolete. With version 3.0.0 there are two of these obsolete settings, and some deprecations. The superseded settings are these:

- `()` previously a setting for handling complicated arguments to trigonometric functions. Now LaTeX braces are recommended; see §2.1.1.2 and §3.4.1.1.
- `reuse` previously a setting determining what is saved with the `\nmcReuse` command (§6.4). Now only the numerical result is saved – although it can be saved in a variety of forms, depending on the result-format specification in the `\eval` command.

Use of either setting generates a similar message, changing only the content between the quote marks in the following:

```
\eval[()=2]{$ \sin\frac12\bigl(A+B\bigr) $}[A=\pi/5,B=\pi/7] ⟹
      !!! See the documentation; ‘()’ key discontinued in: settings. !!!
```

Chapter 5

Settings

The first square-bracketed optional argument (and second argument overall) of the `\nmcEvaluate` command (see Chapter 2) is the settings option preceding the main argument that contain the expression or expressions to be evaluated. The settings option is a comma-separated[1] *key=value* list. Such lists tend to be wordy. For back-of-envelope calculations one wants to be able to 'dash off' the calculation; hence short, cryptically named keys have been used. Many settings are generic, applicable not only to `\nmcEvaluate` but also to other commands that are available in `numerica` (see Chapter 6) and the packages `numerica-plus` and `numerica-tables` – briefly described in §1.1.1.

A calculation is effected against a background of assigned values for various quantities – the calculational environment. For a particular calculation, these values may not be appropriate; or you may have different preferences. The complete list of such settings available for `\nmcEvaluate` (or `\eval`) is shown in Tables 5.1 and 5.2, separated into *functional* settings (which affect the calculation) and *display* settings (which don't).

- The *initial* values listed are the values assigned to the settings initially at each use of the `\eval` (and other) commands.
- A *default* value is the value assigned to the setting if you simply enter its name (without assigning a value to it) in the settings option.
- 0/1 alternatives are interpreted as 1 meaning ON and 0 meaning OFF.

For example, entering `o` in the settings option is equivalent to entering `o=1`, meaning angles are assumed to be in degrees, but unless `o` is entered, `\eval` uses `o=0`, the initial value, meaning angles are assumed to be in radians.

[1] Including when the `comma` package option is used. No *decimal* number is required in the settings, only integers.

Table 5.1: Functional settings parameters

key	type	meaning	default	initial
`dbg`	int	debug data		`0`
`view`		`dbg=1`	`dbg=1`	
`^`	char	exp. mark for sci. notation input	`e`	
`xx`	int (0/1)	accept multi-token variables		`1`
`ff`	char	main arg. multi-formula delimiter	`,` (if decimal dot) `;` (if decimal comma)	
`1s2`	int (0/1)	allow spaced digit groups in numbers	`1`	`0`
`/min`	int ≥ 1	fraction form denominator search start		`1`
`/max`	int ≥ 1	fraction form denominator search end		`200`
`vv@` `vvmode`	int (0/1)	vv-list calculation mode		`0`
`o`	int (0/1)	trig. function args in degrees	`1`	`0`
`log`	num	base of logs for `\log`		`10`
`S+`	int	extra rounding, sums		`2`
`S?`	int ≥ 0	number of query terms, sums		`0`
`P+`	int	extra rounding, products		`2`
`P?`	int ≥ 0	number of query terms, products		`0`
`()`	obsolete; see §5.4			
`reuse`	obsolete; see §5.4			
`*`	obsolete; see §5.4			

Table 5.2: Display settings parameters

key	type	meaning	default	initial
f	int (0/1)	show/hide formula		
p	token(s)	concluding punctuation	,	
pp	token(s)	multi-formula inter-result punctuation	,	
env	token(s)	math environment	see Table 5.3	
arg	token(s)	arg. for -at, array envs	see Table 5.3	
eq	token(s)	relation symbol	see Table 5.3	
vv	token(s)	vv-list specification	see Table 5.3	
sep	token(s)	separator between multi-formula results	see Table 5.3	
\}	token(s)	right bracket for inner math environments	\ \}	
vvi	deprecated; use vv			
vvd	deprecated; use vv			

5.1 'Debug' facility

It is rather grandiose to call this a debug facility, but if a calculation goes wrong or produces a surprising result, numerica offers a means of examining various quantities at some intermediate stages on the way to the final result. To use the facility, enter

```
dbg = <integer>
```

into the settings option. (White space around the equals sign is optional.)

- dbg=0 turns off the debug function, displays the result or error message (this is the initial setting);
- dbg=1 equivalent to dbg=2*3*5*7*11 for \eval;

The 'magic' integers are the first few prime numbers and their products

- dbg=2 displays the formula after multi-token variables have been converted to their single token form, \nmc_a, \nmc_b, etc.;
- dbg=3 displays the vv-list after multi-token variables have been converted to their single token form;
- dbg=5 displays the stored variables and their values *after* evaluation (dbg=3 lists the values as expressions);
- dbg=7 displays the formula after it has been fp-ified but before it has been fed to l3fp to evaluate;

  - When interpreting the fp-form, be aware that differences in the ways `numerica` and `l3fp` read formulas can lead to more or fewer parentheses than seem strictly necessary. In particular be aware that in `l3fp` function calls bind most tightly so that, for example, `sin 2pi` evaluates not to zero but to $(\sin 2) \times \pi$, and `sin x^2` evaluates to $(\sin x)^2$. `numerica` takes care of the former by inserting extra parentheses and exploits the latter by not inserting parentheses.

- `dbg=11` displays the LaTeX form of the final display; it will contain, *inter alia*, the numerical result.

To display two or more of the debug elements simultaneously, use the product of their debug numbers for the magic integer. This can be entered either as the multiplied-out product, or as the 'waiting to be evaluated' product with asterisks (stars) between the factors. Thus `dbg=6` and `dbg=2*3` each display both the vv-list and formula after multi-token variables have been converted to single token form; `dbg=77` or `dbg=7*11` each display both the form of the expression that is fed to `l3fp` (the 'fp-ified' form) and the LaTeX form of the final display (including the numerical result). And generally, if an integer is divisible by 2, 3, 5, 7, or 11 the corresponding element of the debug display will be shown. Both `dbg=2310` and `dbg=2*3*5*7*11` display all five elements, but rather than remembering this product or typing all those digits and asterisks, it suffices to enter `dbg=1`. This is equivalent and displays all elements.

The debug option uses an `aligned` or `align*` environment to display its wares, depending on whether `\eval` lies within or around a math environment. The following uses `align*` and shows how multi-token variables are handled and how a chain of comparisons is evaluated (§2.3.5):

```
\eval[dbg=1]{ a < 2a' < 3a'' }
  [a=\pi,a'=\phi,a''=e\gamma][4???]
```

$\Longrightarrow$

formula: a < 2\nmc_k < 3\nmc_j
vv-list: a=\pi , \nmc_k =\phi , \nmc_j =e\gamma
stored: a=3.141592653589793, \nmc_k =1.618033988749895, \nmc_j =1.569034853003742
fp-form: round((3.141592653589793)-(2(1.618033988749895)),4)<0&&round(2(1.618033988749895)-(3(1.569034853003742)),4)<0
LaTeX: $\texttt {T}$

The various items are displayed in chronological order. First comes the formula after conversion of multi-token to single-token variables, then the vv-list in those single-token variables; these are created essentially at the same time. Next the stored values of the variables are displayed. These are the values *after* vv-list

evaluation. (Even if the `comma` package option is being used and the decimal point is a comma, the stored values will display with a decimal dot because this is what `l3fp` uses.) The fourth element both in the display and chronologically is the fp-ified formula. Often this can be a thicket of parentheses, especially if unary functions or fractions are involved. The final element of both the display and chronologically is the LaTeX form of the display. In the example it is skimpy, because no environment was specified. Putting, say, `env=$` in the settings option results in a much fuller final line:

```
\eval[dbg=11,env=$]{ a < 2a' < 3a'' }
  [a=\pi,a'=\phi,a''=e\gamma][4???]
```

$\Longrightarrow$

LaTeX: $a < 2\nmc_m < 3\nmc_l \rightarrow \texttt {T},\mskip 12muplus6muminus9mu(a=\pi ,a'=\phi ,a''=e\gamma)$

By using `dbg=11` in the settings option I have limited the display to the LaTeX form, since the other elements are unchanged.

Mathematical operations that have no direct counterpart in `l3fp` contribute only their numerical value to the fp-form. This applies to sums and products, double factorials, partly to binomial coefficients, and also to `\eval` and other commands when nested one within another (see Chapter 7.1). The following (ridiculous) example illustrates the matter:

```
\eval[dbg=1]{\[
  \sum_{n=1}^k n + \binom{2k}{m} - \frac1{4k} +
    \prod_{n=2}^k (1-1/n) + m!! \]}[m=6,k=5]
```

$\Longrightarrow$

formula: \sum _{n=1}^k n + \binom {2k}{m} - \frac 1{4k} + \prod _{n=2}^k (1-1/n) + m!!
vv-list: m=6, k=5
stored: m=6, k=5
fp-form: 15+(151200/720)-((1)/(4(5)))+0.2+(48)
LaTeX: \[\sum _{n=1}^k n + \binom {2k}{m} - \frac 1{4k} + \prod _{n=2}^k (1-1/n) + m!!=273.15,\mskip 36muminus24mu(m=6,k=5)\]

(0° C in kelvin!) In the `fp-form` line, the various contributions to the overall result are displayed simply as numbers because `l3fp` does not (at least as yet) handle these elements natively.

5.1.1 Multi-formula calculations

Using `dbg=1` on a multi-formula calculation displays the formula and fp-form of the last formula 'digested' by `\eval`. Other elements of the debug display are

not limited in this way. If all formulas are successfully evaluated then this will be the final formula entered in the multi-formula calculation. But should there be an error when evaluating one of the component formulas, *that* will be the last formula evaluated; the debug information will pertain to that formula. For example,

```
\eval[dbg=1,ff]{$ \sin x, \arccos x, \tan x $}[x=\pi/3]
```

$\Longrightarrow$!!! `l3fp` error 'Invalid operation' in: formula. !!!

formula: \arccos x
vv-list: x=\pi /3
stored: x=1.047197551196598
fp-form: acos((1.047197551196598))
LaTeX: $\sin x=0.866025,\mskip 12muplus6muminus9mu(x=\pi /3)\quad $

Both error message and debug information are present, the debug information pertaining to the formula, `\arccos x`, where the error occurred.

5.1.2 Negative `dbg` values

Had the `$` delimiters been placed around rather than within the `\eval` command in the last example, both error message and debug display would have been crowded onto the one line and would generally exceed the dimensions of the paper. The remedy is to turn off the error message by using a *negative* debug number of the same numerical value, in this case `dbg=-1` (note that this use of *negative* debug numbers differs from their use in previous versions of `numerica`):

```
$\eval[dbg=-1,ff]{ \sin x, \arccos x, \tan x }[x=\pi/3]$
```

formula: \arccos x
vv-list: x=\pi /3
$\Longrightarrow$ stored: x=1.047197551196598
fp-form: acos((1.047197551196598))
LaTeX: 0.866025\quad

5.1.3 `view` setting

Putting `dbg=1` in the settings option may seem somewhat obscure in order to view internal values of `numerica`. Writing `view` instead (it does not need to be equated to anything) is equivalent and should be easier to remember.

5.2 Other functional settings

5.2.1 Inputting numbers in scientific notation

Outputting numbers in scientific notation is controlled by the final trailing argument of the `\eval` command. Such output is turned off by default and needs

to be explicitly ordered. Similarly, *inputting* numbers in scientific notation is turned off by default and needs to be explicitly ordered. To turn it on, write

```
^ = <char>
```

in the settings option, where `<char>` is any single character, usually `e` or `d` or their upper-casings, but not restricted to them: `^=@` for instance is perfectly possible, and has the advantage over `e` or `d` that it doesn't conflict with the use of the character as a variable or constant.

`$ \eval[^=@]{ 0.123 + 1.23@-1 } $` $\Longrightarrow 0.246$.

The example shows that numbers can still be input in ordinary decimal form at the same time as scientific notation is used.

The default exponent mark is `e` so that entering `^` in the settings option is equivalent to entering `^=e` and will suffice to turn on the inputting of numbers in scientific notation using this mark:

`$ \eval[^]{ 0.123 + 1.23e-1 } $` $\Longrightarrow 0.246$.

With letters for the exponent mark – say `d` or `e` – there is a problem in interpreting forms like `8d-3` or `2e-1`. Does such a form denote a number in scientific notation or an algebraic expression? In `numerica`, if the settings option shows `^=d`, then a form like `8d-3` is treated as a number in scientific notation. Similarly for `e` or any other letter used as the exponent mark for the input of scientific numbers. (But only one character can be so used at a time.) Note that the number *must start with a digit*: `e-1` for instance does not and will be treated as an algebraic expression involving the exponential constant:

`$ \eval[^]{ x+e-1 }[x=1] $` $\Longrightarrow 2.718282$

but

`$ \eval[^]{ x+1e-1 }[x=1] $` $\Longrightarrow 1.1$.

A problem of appearance arises if scientific numbers appear in the vv-list or formula and either is displayed in the result. A number like `2e-1` will display as $2e-1$, as if it were an algebraic expression. In version 1 of `numerica` the cure was to wrap `2e-1` in a `\text` or `\mbox` command. In version 2 of `numerica` the behaviour of `\text` and `\mbox` was re-thought; see §3.9.4. Their contents are invisible to the `\eval` command. The solution is to wrap `2e-1` in a `\textrm` or `\textsf` or `\texttt` command. These commands were not recognized by `\eval` in version 1 but in versions from 2.0.0:

`\eval[^=e]{$  5x $ }[x=\texttt{2e-1}]` $\Longrightarrow 5x = 1$, $(x = \texttt{2e-1})$,
`\eval[^=e]{$ 5\texttt{2e-1} $ }` $\Longrightarrow 5(\texttt{2e-1}) = 1$.

If you use a particular character as the exponent marker for inputting numbers in scientific notation, it is good practice *not* to use that character as a variable, not because it will cause an error but because it makes expressions harder to read.

5.2.2 Multi-token variables

Variables need not consist of a single character or token (like x or α). Multi-token symbols like x' or t_i or var are perfectly acceptable. For its internal operations, `numerica` converts such multi-token names to single tokens (as discussed in §2.1.3). This conversion takes time. Even if there are no multi-token variables used at all, `numerica` still needs to check that that is so. There is a setting that allows a user to turn off or turn on the check for such variables by entering

```
xx = <integer>
```

into the settings option. If `<integer>` is `0`, the check for (and conversion of) multi-token variables is turned off; if `<integer>` is `1` (or any other *non-zero* integer), the check, and conversion if needed, goes ahead. `numerica` assumes multi-token variables may be used (primed or subscripted variables are common) so `xx=1` is assumed initially. (The name for the key, `xx`, is chosen because `x` is the most familiar variable of all, introduced in elementary algebra, and doubling it like this suggests multi-token-ness.)

If checking is turned off when a multi-token variable is present, an error results. We don't need to turn on the check in the first of the following examples because that is the initial pre-set state. Explicitly turning it off in the second produces an error.

`\eval{$ x_0^{\,2} $}[x_0=5]` $\Longrightarrow x_0^2 = 25, \quad (x_0 = 5),$

`\eval[xx=0]{$ x_0^{\,2} $}[x_0=5]` $\Longrightarrow$
!!! Unknown token x in: formula. !!!

5.2.3 Multi-formula separator

The `\eval` command can evaluate more than one formula at a time. This is activated with the `ff` setting. Entering

```
ff = <char>
```

in the settings option means multiple formulas separated by `<char>` in the main argument of `\eval` will be evaluated. For example, using `ff=|`:

`\eval[ff=|,pp]{ \pi^e | e^\pi | 2\phi^{\phi\pi} }` $\Longrightarrow$
22.459158, 23.140693, 23.086428.

When `ff` is used without the `= <char>` part, it defaults to the vv-list separator – a comma if the decimal point is a dot (period, full stop), a semicolon if the decimal point is a comma (i.e. if the `comma` package option is used). If `ff=` is used, with nothing on the right of the equals sign, to prevent a LaTeX error being raised, the vv-list separator is retained as the multi-formula separator.

5.2.4 Spaced digit grouping

Numbers containing many digits can be easier to read if the digits are grouped into blocks. `numerica` accepts *as input* numbers with such grouped digits provided the intervening character is a space. This needs to be explicitly turned on with the setting `1s2`:

`\eval[1s2]{ 12 3456.7890 1234 }` $\Longrightarrow$ 123456.789012.

Without the `1s2` setting, the separate blocks of digits are multiplied together:

```
\eval[dbg=77]{ 12 3456.7890 1234 }
```

$\Longrightarrow$

fp-form: 12*3456.7890*1234
LaTeX: $12 3456.7890 1234=51188131.512$

5.2.5 Fraction-form denominator limits

To restrict the size of the denominator in fraction-form ouput `\eval` has two settings

```
/min = <integer1>
/max = <integer2>
```

(Those are *not* backslashes!) `<integer1>` is the value to start searching from for a denominator for the fraction; `integer2` is the value to search to and then stop the search if none has been found (to the requested accuracy). The initial values are `/min=1` and `/max=200`. See the discussion at §2.3.4.

5.2.6 Calculation mode

A variable may change in the course of a calculation. This is certainly true of sums and products. If a parameter in the vv-list depends on the variable then that parameter will need to be recalculated, perhaps repeatedly, in the course of a calculation. By entering either

```
vv@ = <integer>
```

or (as in version 1 of `numerica`),

```
vvmode = <integer>
```

in the settings option it is possible to turn on or off the ability to repeatedly evaluate the vv-list; `<integer>` here takes two possible values, `0` or `1`. `vv@=0` (or `vvmode=0`) means the vv-list is evaluated once at the start of the calculation; `vv@=1` (or `vvmode=1`) means the vv-list is recalculated every time the relevant variable changes.[2]

[2] In version 1 of `numerica` only the `vvmode` name for this setting was available. To the author's eye, the `@` sign seems suggestively close to a symbol like ↻ to mean 'redo' (and is generally available on keyboards).

For example, in a sum it may be desirable to place the summand, or some part of it, in the vv-list. Since the summation variable obviously changes during the course of the calculation, we need to enter `vv@=1` in the settings option. Repeating an earlier sum,

```
\eval[p=.,vv@=1]{\[ \sum_{k=1}^N f(k) \]}
  [N=100,f(k)=1/k^3,{k}=1][4]
```

$\Longrightarrow$

$$\sum_{k=1}^{N} f(k) = 1.202, \qquad (N = 100, f(k) = 1/k^3).$$

As you can see, the summand `f(k)` has been given explicit form in the vv-list – equated to `1/k^3`. That means we need to give a preceding value to `k` in the vv-list to avoid an unknown token message, hence the rightmost entry. But we don't want `k=1` appearing in the final display, so we wrap `k` in braces (see §2.2.4.3). Since the value `k=1` applies only to the first term in the sum, to ensure it is not used for all terms, we enter `vv@=1` in the settings option. This turns vv-recalculation mode on and ensures `k=1` is overwritten by `k=2`, `k=3` and so on, and the vv-list recalculated each time. The final result is the same as before, although recalculating the vv-list at each step is a more resource-hungry process. The difference may not be marked for this example; with more complicated expressions it noticeably takes longer.

Because it is necessary to activate this switch when using *implicit* notations – like $f(k)$ in the example – rather than the explicit form of the function in the main argument, it seems natural to call `vv@=1` *implicit* mode and `vv@=0` (the default) *explicit* mode. Most calculations are explicit mode – the vv-list is evaluated only once.

5.2.7 Using degrees rather than radians

You may find it more convenient to use degrees rather than radians with trigonometric functions. This can be switched on simply by entering a lowercase `o` in the settings option. (The author hopes the charitable eye sees a degree symbol in the `o`.) Thus

`\eval[o]{$ \sin 30 $}` $\Longrightarrow \sin 30 = 0.5,$
`\eval[o]{$ \arcsin 0.5 $}` $\Longrightarrow \arcsin 0.5 = 30.$

This is a `0/1` switch, `0` signifying `off` or 'don't use degrees', `1` signifying `on` or 'do use degrees'. Out-of-the-box `numerica` assumes radians are being used, `o=0`. As noted in §3.4.1, one can also append `\degree` to a number (or a variable), making the `o` switch unnecessary for the *direct* functions (but always necessary if you want the *inverse* functions to produce an answer in degrees).

5.2.8 Specifying a logarithm base

If you wish to use `\log` without a subscripted base in a particular calculation, then add an entry like

```
log = <positive number>
```

where `<positive number>` $\neq 1$ to the settings option of the `\eval` command. The `<positive number>` does not need to be an integer. It could be `e` (if you object to writing `\ln`) but is more likely to be 2 or another small integer. If no value is specified, `numerica` assumes `log=10`.

5.2.9 'Infinite' sum and product settings

These settings, `S+=<integer>`, `P+=<integer>`, `S?=<integer>`, `P?=<integer>`, where the latter two are non-negative, determine when the stopping criterion for 'infinite' sums and products applies. They are discussed in §3.8.2.

5.3 Display-related settings

The following settings have no effect on the calculation but do change how the result is displayed.

5.3.1 Show/hide formula, `f`

`\eval` automatically shows the formula in the result display if the `env` setting is used to define the environment or `\eval` wraps around the environment, but if the environment wraps around `\eval`, display of the formula is suppressed. Either circumstance can be changed by entering `f=0` to suppress display of the formula or `f=1` to display the formula.

5.3.2 Environment settings, `env` etc.

The math environment in which a calculation is presented can be specified in three different ways: externally, with `\eval` lying within the environment, or internally, either through the `env` setting

```
env=<environment>
```

or by explicitly writing the environment into the main argument containing the formula, although this last method is clumsy for anything other than `$...$` or `\(...\)` or `\[...\]`. If multiple environments are – inadvertently or otherwise – specified, the outer environment takes precedence over the `env` setting which takes precedence over the main-argument environment; this matter was discussed with examples in §3.10.

Table 5.3 lists the environments that `env` can be set to and that `\eval` recognizes. It also lists the initial values of the settings `arg`, `eq`, `vv` and `sep`,

Table 5.3: Initial values for environments

env	rem/arg	eq	vv	sep
$ \(math		=	,\mskip 12muplus 6muminus9mu(vv)	\quad
\[		=	,\mskip 36mu minus24mu(vv)	\]\[
displaymath equation equation*		=	,\mskip 36mu minus24mu(vv)	\end{*env*} \begin{*env*}
multline multline*	\eval in m'line	=	,\mskip 36mu minus24mu(vv)	\hfill\\
multline multline*	m'line in \eval	=	,\\(vv)	\end{*env*} \begin{*env*}
eqnarray eqnarray*		&=&	,\mskip 36mu minus24mu(vv)	\\
align align* aligned		&=	,\mskip 36mu minus24mu(vv)	\\
flalign flalign*		&=	,&(vv)	\\
gather gather* gathered		&=&	,\mskip 12muplus 6muminus9mu(vv)	\\
alignat alignat* alignedat	2	&=\;&	,\qquad&(vv)	\\
array	rcrl	&=&	,&(vv)	\\
cases dcases		=	,\quad\hfill(vv)	\\

designed to give sensible displays, without further intervention by the user, in many circumstances. (When the `comma` package option is used, the comma leading the entries in the `vv` column is replaced by a semicolon.) If you are using the `env` setting, and want to change a value for one or more of these settings, then the changed value must be entered *after* the `env` setting to have any effect.

There are other settings which inter-play with these environment settings but are not given initial values by the `env` setting as `arg`, `eq`, `vv` and `sep` are, and so do not need to follow it in the settings option, but it is clearer and therefore good practice if they do. These are the punctuation settings `p` and `pp`, and the `\}` setting applying only to `aligned`, `alignedat`, `gathered`, `cases`, `dcases`, and `array` environments – ones that require an enclosing math environment. The `\}` setting gathers the results of a multi-formula display within a large right delimiter (right brace by default); it is usually followed by the vv-list and allows the results of a multi-formula calculation to be collectively numbered rather than individually.

5.3.2.1 `arg`

The `array` and `alignat` group of environments require an argument, as in `\begin{array}{arg}...` and `\begin{alignat}{arg}...`. For recognized environments the pre-set value of `arg` is shown in the second column of Table 5.3. Note that the use of `arg` environments is possible with `\eval` *only* when used with the `env` setting; otherwise the `{arg}` is misinterpreted and will generally cause an error, as in the following, with its peculiar error message

```
\[
  \eval[ff]{ \begin{array}{rcll}
    \sin x, \cos x, \tan x
  \end{array}}[x=\pi/6]
\]
```

$\Longrightarrow$

!!! Unmatched `r` in: formula. !!!

The problem does not arise when the `env` setting is used:

```
\[
  \eval[env=array,ff]{
    \sin x, \cos x, \tan x }[x=\pi/6][*]
\]
```

$\Longrightarrow$

$$\begin{array}{rcll} \sin x & = & 0.500000, & (x = \pi/6) \\ \cos x & = & 0.866025, & (x = \pi/6) \\ \tan x & = & 0.577350, & (x = \pi/6) \end{array}$$

In this case, with the pre-set `arg=rcrl`, the formulas are right-aligned, the equality signs are centred, the numerical results are right-aligned and the vv-lists are left-aligned. These alignments can be changed with the `arg` setting, but this must *follow* the `env` setting for the changes to have any effect.

5.3.2.2 `eq`

Throughout this document, *formula=result* displays have used the equality sign between the two sides. Most of the results however have been approximate. If you want a different relation symbol in the display, enter

```
eq=<char(s)>
```

in the settings option. For example,

```
\eval[eq=\approx]{$ \pi $}[4]
```
$\Longrightarrow \pi \approx 3.1416.$

If the `approx` package option is being used, note that `eq` defaults to `=` , so that – say – for a calculation with an integer answer for which ≈ would be inappropriate it suffices just to enter `eq` to obtain = between formula and result.

A main use of this setting is with environments with alignment where `&=` and `&=&` are natural values for `eq`; see below, §5.3.2. If the `env` setting is being used then the `eq` setting must *follow* it to have any effect.

5.3.2.3 `vv`

In many of the examples in this document the vv-list has been displayed following the result. It is wrapped in parentheses following a comma and a space. (A semicolon replaces the comma if the `comma` package option is used.) These elements – comma, space, parentheses – can all be changed by entering

```
vv=<tokens>
```

in the settings option. For example

```
vv=;\mskip 12mu plus 6mu minus 9mu(vv)
```

will result in a semicolon immediately following the numerical result followed by an elastic space followed by the actual vv-list – replacing the placeholder `vv` – enclosed in parentheses. No full stop is inserted after the closing parenthesis because the `\eval` command may occur in the middle of a sentence (and the `p` setting is available for such punctuation in displaystyle contexts). For inline use, the elasticity of the space becomes relevant when TEX is adjusting individual lines to fit sentences into paragraphs and paragraphs into pages. If a comma were to replace the semicolon in the suggested vv-list spec., it would need to be entered in braces or as `\comma`, since the settings option is a comma-separated list. In displaystyle settings, some shrink in the space is a good idea to accommodate a long formula or many variables in the vv-list.

For a particular calculation with a surprising result, one might specify `vv=?!` with no vv-list shown (since the `vv` placeholder is not used on the right):

`\eval[vv=?!]{$ \pi $}[\pi=3]` $\Longrightarrow \pi = 3?!$

Different formats for vv-list display are appropriate for different environments. Those with alignment, like the AMS environments, may have an alignment token `&` following a comma (or semicolon) before display of the vv-list proper; see below, §5.3.2 and Table 5.3. If the `env` setting is being used then the `vv` setting must *follow* it to have any effect.

5.3.2.4 `sep`

Displaying the results of a multi-formula calculation means separating the displays for each result. The `sep` setting specifies the tokens inserted between each display:

```
sep = <tokens>
```

where `<tokens>` may be something like `\quad` for an inline environment or `\\` in a multi-line environment, see Table 5.3. For `multline` and `multline*` different values are assigned depending as `\eval` is wrapped by or wraps the environment. The function of the `sep` setting overlaps that of the `pp` setting, but it proves more flexible to keep them separate. Since `env`-ironments come with pre-set values of `sep`, if both settings are being used, the `sep` setting must *follow* `env` for it to have any effect.

5.3.2.5 `p`, `pp`

As noted in the discussion at §1.1.4.1, punctuation of the display of a result is straightforward when the `\eval` command is used in inline contexts but requires use of a setting in displaystyle environments. The setting

```
p = <char(s)>
```

places `<char(s)>` after the display of everything else but within the environment delimiters. The default punctuation mark is the comma so that entering `p` alone will produce a comma in the appropriate place. Since the settings option is a comma-separated list, this saves having to write `p={,}`. Note that `<char(s)>` need not be a single-character punctuation mark:

`\eval[p=\ \text{(but no 8!)}]{\[ \frac{1}{81} \]}[9]` $\Longrightarrow$

$$\frac{1}{81} = 0.012345679 \text{ (but no 8!)}$$

For multi-formula commands punctuation may be desired not just after the final result (the `p` setting) but also after the intermediate results. For these, use the setting

```
pp = <char(s)>
```

Like `p`, the `pp` setting defaults to a comma to avoid having to write `pp={,}`. See the examples at §1.1.5.1 and throughout this document.

5.3.2.6 \}

Some environments must lie within another math environment to be used. These environments – the AMS `-ed` environments, the `cases` and `dcases` environments, and the `array` environment – allow one to 'collect' results of a multi-formula calculation within a large right delimiter – usually a right brace – and follow that with the vv-list. The setting for this is `\}`:

```
\} = <space-spec><right-delimiter>
```

Here `<space-spec>` is either a single token or a brace group defining a space between the numerical results and the right delimiter. `<right-delimiter>` is any right delimiter that responds to the command `\right` of a `\left \right` pair; that includes a dot, `.` , if you want no delimiter. The default specification is `\}=\ \}`. In the following example an `alignedat` environment is used to align numerical results containing a minus sign. The `pp` and `p` settings place commas after the results, not after the vv-list. The full stop there is manually inserted after the number-format option (and before the `\]`). The example shows how to suppress the needless repetition of the vv-list that otherwise accompanies display of a multi-formula result. If so wished, the three calculations could be collectively numbered by a single equation number.

```
\[
   \eval[env=alignedat,pp,p,\}=.,vv=\qquad(vv),ff]
     { \sin nx, \cos (n+1)x, \tan x }
        [n=2,x=\pi/6][*].
 \]
```

$\Longrightarrow$

$$\begin{aligned} \sin nx &= 0.866025, \\ \cos(n+1)x &= -0.518359, \qquad (n=2, x=\pi/6). \\ \tan x &= 0.577350, \end{aligned}$$

5.4 Deprecated and obsolete settings

`vvi, vvd` These settings have been superseded by the `vv` setting as a result of the enhanced treatment of environments in version 3.0.0 of `numerica`; see immediately above. Both `vvi` and `vvd` settings are now treated as alternative (but deprecated) names for `vv`.

`*` In version 2 of `numerica` this key was used to suppress equation numbering when the `vvd` specification contained a newline character. If `\\` was present in `vvd`, it triggered replacement of whatever math delimiters were enclosed by the `\eval` command with a `multline` environment. The star * then converted the `multline` into a `multline*`. This is all now superseded by the enhanced treatment of environments.

`()` Earlier versions of `numerica` used this setting to adjust how 'expansively' the arguments of trigonometric functions were read, with the arguments of such functions in Fourier series in mind. Version 3.0.0 of `numerica` has dispensed with the setting and now recommends using LaTeX braces to delimit such arguments. See the general discussion at §2.1.1.2, and more specific discussion at §3.4.1.1.

`reuse` In earlier versions of `numerica` this setting determined exactly what was saved with the next `\reuse` command. Now only the 'naked' numerical result is saved, although it may be in scientific notation or fraction form; see §6.4.5.

Chapter 6

Supplementary commands

This chapter introduces four commands, `\nmcInfo` (met already in relation to infinite sums), `\nmcMacros`, `\nmcConstants` and `\nmcReuse`, supplementary to the principal command `\nmcEvaluate`. They use the same machinery as `\nmcEvaluate` and so have the same syntax. If all arguments are present it is

```
\nmc<cmd>*[settings]{main arg}[vv-list][rounding]
```

where `<cmd>` is one of `Info`, `Macros`, `Constants` and `Reuse`. All four commands have short-name forms: `\info`, `\macros`, `\constants`, `\reuse`.

Generally the final two optional arguments will not be used. The user should be aware of this if following a command with a square bracketed expression – the expression will be absorbed without trace unless it is preceded by, for example, an empty brace pair. Because the commands share the machinery of `\nmcEvaluate`, the `dbg`, `view` and functional settings discussed in the previous chapter (Chapter 5) for the `\eval` command are also available for these commands, although many will be without effect.

The starred form of command is available in all four cases and in all cases produces a pure number. If both star (`*`) and `view` are used at the same time, the `view` setting prevails over starring.

6.1 Feedback on 'infinite' processes: `\nmcInfo`

Used after the evaluation of an 'infinite' process, the `\nmcInfo` command, or its short-name form `\info`, will tell you how many terms or factors or other operations[1] were needed to arrive at the result.The main argument contains an identifier for the 'infinite' process:

```
\nmcInfo{<arg>}
```

[1] It also applies to the commands `\nmcIterate` and `\nmcSolve` from the `numerica-plus` package.

(or, using the short-name form, `\info{<arg>}`) where, at this stage, `<arg>` is either `sum` or `prod`. The display, as we have seen in earlier examples, is a number followed by a space then a descriptor. For `sum` and `prod` the descriptors are `terms` and `factors` respectively. Starring `\nmcInfo` – `\nmcInfo*{arg}` or `\info*{arg}` – suppresses the descriptor and leaves only the number.

As an example, let's test 'the hard way' a standard identity, $\cosh^2 x - \sinh^2 x = 1$. We know that $\cosh x = \sum_{n=0}^{\infty} \frac{x^{2n}}{(2n)!}$ and $\sinh x = x \prod_{k=1}^{\infty} \left(1 + \frac{x^2}{k^2\pi^2}\right)$. The difference of their squares should be 1:

```
\eval{\[
  \left[\sum_{n=0}^{\infty}
    \frac{x^{2n}}{(2n)!}
  \right]^2-
    \left[x\prod_{k=1}^{\infty}
      \left(1+\frac{x^{2}}{k^{2}\pi^{2}}\right)
    \right]^2
  \]}[x=1][3] \info{sum},\quad \info{prod}.
```

$\Longrightarrow$

$$\left[\sum_{n=0}^{\infty} \frac{x^{2n}}{(2n)!}\right]^2 - \left[x \prod_{k=1}^{\infty} \left(1 + \frac{x^2}{k^2\pi^2}\right)\right]^2 = 1.002, \qquad (x = 1)$$

5 terms, 119 factors.

Nearly right. Obviously the product converges only slowly which is where the error comes from (see the discussion in §3.8.2, where we needed the extra rounding setting `P+=3` and 350 factors to get a correct 3-figure value). The point of the example is to show the information command being used for both sum and product in the one evaluation. One does not exclude the other.

The starred form of the `\info` command suppresses the descriptor ('terms' or 'factors') and gives a purely numerical result:

```
\eval{$
  \sum_{k=0}^{\infty}\binom \alpha k x^k
    $}[x=1/2,\alpha=3],\\ \\
requiring \eval{$ \info*{sum}-1 $}\  additions.
```

$\Longrightarrow \sum_{k=0}^{\infty} \binom{\alpha}{k} x^k = 3.375, \quad (x = 1/2, \alpha = 3),$

requiring $4 - 1 = 3$ additions. In fact the starring is unnecessary in this example since all nested commands are now treated as if they were starred.

6.1.1 Errors

Should the *wrong* argument be used in the `\nmcInfo` command, no harm is done:

```
\eval{$ \sum_{k=0}^{\infty}\binom \alpha k x^k $}
     [x=1/2,\alpha=3], \ \info{prod}
```

$\Longrightarrow \sum_{k=0}^{\infty} \binom{\alpha}{k} x^k = 3.375, \quad (x = 1/2, \alpha = 3)$, 119 factors.

119 *factors*? We were evaluating a sum but used `prod` as the argument of the `\info` command which remembers a previous result, the last time `prod` was used as its argument. Changing the argument from `prod` to `sum` reveals the correct number of *terms*.

Should a non-existent argument be used, an error message is generated:

```
\eval{$
  \sum_{k=0}^{\infty}\binom \alpha k x^k
     $}[x=1/2,\alpha=3], \\ \info{Fred}
```

$\Longrightarrow \sum_{k=0}^{\infty} \binom{\alpha}{k} x^k = 3.375, \quad (x = 1/2, \alpha = 3),$

!!! Unknown process `Fred` in: info command. !!!

6.1.2 `view` setting

Only two settings seem relevant for `\info`: the `dbg` and `view` settings. Rather than use the obscure `dbg=<integer>` (which is possible), it suffices to enter `view` in the settings option:

`\info[view]{}` $\Longrightarrow$

process: sum {4}, prod {119}

The result is a display of all the current values of all the 'infinite' processes available. (Further processes become available if the `numerica-plus` package is used.) The mandatory but empty braces can be filled with `sum`, `prod` or even `Fred` but it makes no difference; the display is always of the current values of all 'infinite' processes `numerica` knows about. If there has been no sum or product evaluated, the displayed value is conventionally shown as zero.

6.2 User-defined macros: `\nmcMacros`

The `\nmcMacros` command was prompted by a question on TEX Stack Exchange.[2] Some time later the maintainer of the `mandi` package approached me with a similar problem. Suppose one has defined a macro to contain a value, say

- `\def\myvalue{0.35}`, or
- `\newcommand\myvalue{0.35}`, or
- `\NewDocumentCommand\myvalue{}{ 0.35 }`.

[2] See https://tex.stackexchange.com/questions/602993/use-macros-in-numerica-vv-list/602998#602998

(If you're using the document processor LyX then there is good reason to prefer `\gdef` to define your macro, `\gdef\myvalue{0.35}`; see Chapter 7.3). After one of these commands, `\myvalue` is now known to LaTeX, but it is not known to `numerica`. The quantities `numerica` *does* know about are variables in the vv-list of an `\eval` command, and those LaTeX (and `amsmath` and `mathtools`) commands used for writing mathematical expressions. These quantities are stored in `numerica` in structures called property lists. Since `\myvalue` is not – yet – recorded in these lists putting `x=\myvalue` in the formula or vv-list of an `\eval` command will produce an 'Unknown token' error message:

```
\NewDocumentCommand \myvalue {} { 0.35 }
\eval{ \myvalue }
```

$\Longrightarrow$!!! Unknown token \myvalue in: formula. !!!

From version 2 of `numerica`, a command `\nmcMacros` is available to register macros and their values with the property lists used internally by `numerica`. The macro must have been defined earlier in the document or in a supporting package. Using `\nmcMacros` (or `\macros` in its short-name form) is simple.

If you have a list of macros you wish to use in `numerica`, enter them in a comma (or a semicolon list if the decimal comma is being used) in the mandatory argument of the command. The `ff` setting necessary for a multi-formula calculation in `\eval` is implicit for `\nmcMacros` – it does not need to be added:

```
\nmcMacros{ \macro1, \macro2, ... }
\nmcMacros{ \macro1; \macro2; ... }
```

Since no decimal numbers are involved, no ambiguity would arise from using a comma separator even when the `comma` package option is used but consistency with multi-formula calculations in `\eval` suggests that a semicolon should be – and is – insisted upon unless explicitly countermanded by entering `ff=<char>` in the settings option of the `\macros` command.

Multiple `\macros` commands can be used in a document. If the command is placed in the preamble (*after* the definition of the macros) then the user-defined macros and their values are available throughout the document, otherwise they are available from the position of the `\macros` statement. However, it is not necessary for macros to be defined in the *current* document provided they are defined and accessible from some other loaded LaTeX package. But always an `\nmcMacros` (or `\macros`) command is required to register them with `numerica` for use in `\eval`.

6.2.1 What can be stored in a macro?

Generally a user-defined macro will store a number. This macro might well be defined in an external package – for example the `mandi` package defines a large number of macros containing the values of physical constants, some fundamental

like the speed of light, others contingent like the earth–moon distance. If the `mandi` package is loaded[3] then writing, for instance,

```
\macros{ \electronmassprecisevalue,
         \protonmassprecisevalue }
```

will make these two macros available for use in `numerica`. One could then write in the vv-list of an `\eval` command

```
m_e=\electronmassprecisevalue,
m_p=\protonmassprecisevalue
```

which would allow (among other things) calculation of the mass ratio m_p/m_e of proton to electron. (The length of name of some of the macros in the `mandi` package has a pedagogical purpose, but makes them unwieldy for direct use in mathematical calculations.)

In version 2 of `numerica`, where the `\macros` command was introduced, if the first token in a macro definition was expandable (like `\sin` expanding to sin or `\sum` expanding to $\sum$) it was necessary to ensure there was a preceding space. If the first token was a digit, there was no problem. From version 3.0.0 this is no longer the case. A leading space may or may not be included; it doesn't matter. I thank David Carlisle for the routine that solved this irritant.[a]

[a] https://tex.stackexchange.com/questions/683578/storing-the-unknown-contents-of-a-macro-in-a-clist

6.2.1.1 Macros containing formulas

Numbers are not the only quantities that can be stored in a macro for use in `numerica`. In fact any mathematical expression that can be `\eval`-uated can be stored in a macro:

```
\NewDocumentCommand \mysumC {}
    { \sum_{n=1}^{100}1/n - \ln 100 }
\macros{ \mysumC }
\eval{$ \mysumC $}[4]
```

$\Longrightarrow \quad \sum_{n=1}^{100} 1/n - \ln 100 = 0.5822,$

(to be compared with Euler's constant $\gamma = 0.5772$ – obviously many more terms are needed). The `\eval` command wraps around math delimiters in the example. Hence the result is presented in the form *formula=result*. In that presentation, note how `\mysumC` displays as the formula it contains.

[3] Maintainer Joe Heafner explains that 'mandi' abbreviates 'matter and interactions' after a physics textbook of that name.

6.2.1.2 vv-list

In the example it would be nice to be able to vary the number of terms summed. This is easily done by using a vv-list in the `\macros` statement:

```
\NewDocumentCommand \mysumN {}
    { \sum_{n=1}^{N}1/n - \ln N }
\macros{ \mysumN }[N=150]
\eval{$ \mysumN $}
```

$\Longrightarrow \quad \sum_{n=1}^{N} 1/n - \ln N = 0.580545.$

`numerica` needs a definite value to store; it does not store the formula as such. To give `\mysumN` a definite value, give the variable `N` a value. This is done in the vv-list added to the `\macros` statement: `N=150`. In this way a definite value is stored in `numerica` against the macro `\mysumN`. The definition of the macro is unaffected. If a new value is given to `N` in the `\macros` statement (which is the point of using a variable), the old value is overwritten and the new value is used in subsequent calculations.

6.2.2 Seeing what macros are available

Perhaps your document has a number of `\nmcMacros` statements scattered through it and you want to remind yourself of what exactly has been stored. `\nmcMacros` has the `view` setting for this purpose. Writing

```
\macros[view]{} ⟹
```

```
macros:  \mysumN { \sum _{n=1}^{N}1/n - \ln N },\mysumC { \sum
         _{n=1}^{100}1/n - \ln 100 }
stored:  \mysumN =0.580545294547621, \mysumC =0.582207331651529
```

produces a list of all macros registered with `numerica` and their values, as you can see.

If the braced argument is not empty, the display is slightly modified:

```
\def\mydef{ \sin(m\pi/n) }
\newcommand\mynewcmd{ \cos(m\pi/n) }
\macros[view]{ \mydef,\mynewcmd }[m=3,n=18]
```

⟹

```
added:   \mydef { \sin (m\pi /n) },\mynewcmd { \cos (m\pi /n)}
vv-list: m=3, n=18
stored:  \mydef =0.4999999999999999, \mynewcmd =0.8660254037844387, \mysumN
         =0.580545294547621, \mysumC =0.582207331651529
```

`\mydef` and `\mynewcmd` have been added to those available for use in `numerica`.

6.2.3 Freeing macros from storage

Rather than cluttering `numerica`'s property lists with no-longer-needed macros, it is possible to remove them from there with the `free` setting. This has no effect on the LaTeX definition of the macro. It merely 'de-registers' the macro with `numerica`.

`\macros[free,view]{ \mysumC }` $\Longrightarrow$

freed: \mysumC { \sum _{n=1}^{100}1/n - \ln 100 }
stored: \mydef =0.4999999999999999, \mynewcmd =0.8660254037844387, \mysumN =0.580545294547621

If you want to free *all* macros registered with `numerica` use an empty main argument with the `free` setting. For an example, see just below.

6.2.4 Counting how many macros are available

You can count how many macros are currently registered with `numerica` by starring the `\nmcMacros` command:

`\macros*{}` $\Longrightarrow 3.$

If the braced argument is not empty, the list of macros it contains will be added to those registered with `numerica` and included in the overall count.

Note that the `view` setting prevails over starring if both are used.

The star can also be used with the `free` setting. As mentioned above, if the main argument is empty, then *all* macros are freed:

`\macros*[free]{}` $\Longrightarrow 0.$

6.2.5 Errors

If a macro is used in a `\macros` statement but has not been defined earlier in the document or a supporting package it will cause an error:

`\macros{ \mymacro }` $\Longrightarrow$!!! Undefined macro \mymacro in: macros command. !!!

An undefined macro used in an `\eval`-uation will cause an 'Unknown token' message in `numerica`. The solution in both these cases is (obviously) to define the macro.

If a macro is defined but the `\macros` statement overlooked and the macro used in an `\eval`-uation, it will generate an 'Unknown token' message. If your macro stores a formula with variables, and you forget to give those variables values in the `\macros` statement that will produce a message, for instance:

```
\def\mysumk{ \sum_{n=1}^k n }
\macros{ \mysumk }
```

$\Longrightarrow$!!! Unknown token k in: sum. !!!
In this case the 'where' part of the message is specific, but more usually will be `macros command`. And of course there can be 'all the usual suspects' discussed in Chapter 4.

6.2.5.1 Display of macros

Once a macro is known to LaTeX it can be used as a variable name. If it is entered as such on the left of an equals sign in the vv-list, `\eval` will treat it as a variable name. As with multi-token variables (§2.2.5), this fact can be abused. In the following example the macro is defined in LaTeX but there is no `\macros` statement.

```
\def\mymac{1}
\eval[vv=\,???]{$ \mymac+\mymac $}[\mymac=2]
```

$\Longrightarrow$ $1+1=4$???
Do not define a macro containing some value and then use it as a variable name for a different value. Macros display as their content. The value (2) assigned to a variable name (`\mymac`) for calculational purposes in `\eval` and how that variable name displays in LaTeX are separate things. It is up to the user not to abuse this fact.

6.2.6 Rounding value

Values are stored to 16 significant figures (if available). In most cases appending a rounding value to a `\macros` statement has no effect on the value stored. In the following example note the `o` setting, meaning the sine reads angles in degrees:

```
\NewDocumentCommand\testi{}{ \sin 60 }
\NewDocumentCommand\testii{}{ \sin 60 }
\macros[o]{ \testi }[10]
\macros[o]{ \testii }[3]
\macros[view]{}
```

$\Longrightarrow$

macros: \testii { \sin 60 },\testi { \sin 60 }
stored: \testii =0.8660254037844386, \testi =0.8660254037844386

Despite the different rounding values in the `\macros` statements the same 16 figures are stored in both `\testi` and `\testii`.

For the `\eval` command, rounding values specify how results are *displayed*. In general the rounding value matters *after*, not during, the calculation. Only for infinite sums or products is this otherwise. (Although fraction-form output from `\eval` also depends on the rounding value, that is not relevant here.) For infinite sums or products the rounding value is used to determine when to stop

adding further terms or factors. The same is true of the `\macros` command. Only if a macro contains an infinite sum or product does the rounding value become relevant. Sixteen figures are still stored, but most of them will be 'wrong' since the infinite sum or product has stopped early, after only a finite number of terms or factors. Exactly how many of the first few figures are correct depends on the rounding value. An example may clarify the matter.

```
\macros[free]{}
\def\zetaiii{ \sum_{n=1}^\infty 1/n^3 }
\macros[view]{ \zetaiii }[3]
\info{sum}
\macros[view]{ \zetaiii }[6]
\info{sum}
```

$\Longrightarrow$

added: \zetaiii { \sum _{n=1}^\infty 1/n^3 }
stored: \zetaiii =1.201844363305174

47 terms

added: \zetaiii { \sum _{n=1}^\infty 1/n^3 }
stored: \zetaiii =1.202054634870939

468 terms

HMF Table 23.3 tells me that $\zeta(3) = 1.202056903159594\ldots$ The different rounding numbers have restricted the infinite sums to the very finite 47 and 468 terms respectively. Although 16 figures are stored, only the first few are correct, three more for rounding value 6 than for rounding value 3.

6.3 User-defined constants: `\nmcConstants`

As noted much earlier in this document (§2.2.2), there are five built-in constants: `\pi`, `e`, `\phi`, `\gamma` and `\deg`, but a user may well want to define their own constant or constants. There are contexts where it would make sense to permanently record fundamental constants like the speed of light or Planck's constant, or more down-to-earth constants like the acceleration due to gravity or the viscosity of water, rather than having to enter them in the vv-list for each calculation. Or a parameter might be held constant for a particular problem or class of problems where other variables change – for example triangles of constant perimeter but varying sides. This is the purpose of the `\nmcConstants` command.

The symbols used to denote constants are subject to exactly the same constraints and freedoms as the symbols used to denote variables. They might be single latin letters like `c` (e.g. $c = 3 \times 10^8$), or greek letters like `\alpha` (e.g. $\alpha = 1/137$), or multi-token combinations like the Rydberg constants `R_\infty`

or `R_{\mathrm{H}}` from atomic physics, or `\mu_0` and `\epsilon_0` used to denote the permeability and permitivity of free space, or personal constants like `total` of no wider significance. `numerica` handles all these different forms of constant with the command `\nmcConstants`, where the main argument is a comma-list of *const=value* statements:

```
\nmcConstants{  const1=value1, const2=value2, ... ,
                const-n=value-n }
```

When the `comma` package option is used and the decimal point is a comma, the default item separator in `\nmcConstants` is the semicolon. Alternatively and irrespective of the choice of decimal point, a user can specify a separator with the setting `ff=<char>`, e.g. `ff=|`, and use that.

This is the simplest use – each constant is assigned a numerical value. But it is easy to envisage situations where it would be convenient to have a constant with value $1/\sqrt{2\pi}$ say, or another with value $e^{\frac{\pi}{2}}$, and so on. That is easy: simply put the expession for the value on the right:

```
\constants{ a=1/\sqrt{2\pi},b=e^{\tfrac\pi2} }
```

where the short(er)-name form `\constants` has been used. Or the values could be expressions depending on parameters:

```
\constants{ s=\tfrac12(a+b+c) }[a=3,b=5,c=7]
```

Some constants might depend on earlier constants in the list:

```
\constants{ A=\sqrt{s(s-a)(s-b)(s-c)},
            s=\tfrac12(a+b+c) }[a=3,b=5,c=7]
```

Or the values could involve an 'infinite' process, requiring a rounding value:

```
\constants{ \zeta=\sum_{n=1}^\infty(1/n^k) }[k=4][5]
```

In this, although 16 figures will be stored, only the first few will be accurate, the precise number depending on the value of `k` and the rounding value (`5` in the example); see the discussion on this issue for user-defined macros above.

6.3.1 New list *replaces* old

A particular group of constants may be relevant only to a particular part of a document. Another part of the document may use other constants. By default, a second list of constants *replaces* the first list. Thus each of the `\constants` statements above would replace the previous one.

There is a technical reason for this. For each calculation all *multi-token* constants are added internally to the start of the vv-list of the `\eval` command. Even if the vv-list is empty, this is still the case since the formula might well use constants. As for multi-token variables (see §2.1.3), multi-token constants are mapped internally to single tokens. This occurs afresh for each calculation which will require not only this mapping from multi- to single tokens but the evaluation of a vv-list that includes the multi-token constants. It seems safer to make the default behaviour replacement of one constant list by another, rather than cumulating them.

6.3.2 Adding constants to a list

Despite which there will be occasions when adding new constants to an existing list is desired. This is easily done with the `add` setting. For instance,

```
\nmcConstants[add]{ \sigma=5.67\times10^{-8},
                    k_B = 1.381\times10^{-23} }
```

would add `\sigma` and `k_B` to the current list. The presence of the `add` setting triggers appending rather than replacement.

6.3.3 Examples of use

6.3.3.1 Example 1: atomic constants

In the following example, the values of various atomic constants are taken from the `mandi` package. I use two `\constants` statements in order to show the use of the `add` setting. I've also included a `view` setting in the second `\constants` statement.

The constants are used to calculate the fine-structure constant `\alpha` in the vv-list of the `\eval` command, and its well-known reciprocal (close to 137) in the main argument. Note that the constants do not need to be entered in the vv-list of the `\eval` command. Their values are available from the `\constants` statements.

```
\constants{ c=2.99792458\times10^{8},
            h=6.62607015\times10^{-34},
            e=1.602176634\times10^{-19} }
\constants[view,add]
  { \epsilon_0=8.854187817\times10^{-12} }
\eval{$ 1/\alpha $}[\alpha=e^2/2\epsilon_0hc]
```

$\Longrightarrow$

added: \nmc_p =8.854187817\times 10^{-12}

constants: e=1.602176634e-19, h=6.62607015e-34, c=299792458, \epsilon_0=0.000000000008854187817

$1/\alpha = 137.035999, \quad (\alpha = e^2/2\epsilon_0 hc)$.

The `view` setting produces a now familiar kind of display. It shows that the three-token `\epsilon_0` (the control sequence `\epsilon`, the underscore `_` and the digit `0`) has been replaced by `\nmc_p` – which may look as if it is also three tokens but is in fact a single control sequence.

6.3.3.2 Example 2: local constants

Long ago, when there were such creatures as reference librarians, I was asked about a school physics project along these lines.

A *car is travelling at 50 km/hr when it hits a lamppost. The bonnet crumples 1 metre and the car comes to an immediate halt. Although she herself is wearing a seat-belt, a woman in the passenger cabin is holding her 5 kg baby. Does the baby survive?*

The enquirer was familiar with the equations describing constant acceleration,

$$x = ut + \tfrac{1}{2}at^2, \quad \text{and} \quad v^2 - u^2 = 2ax,$$

and Newton's second law, $F = ma$, force equals mass times acceleration. The question was really about understanding these laws and how to think with them. Here, s is the distance travelled in time t, with initial speed u at $t = 0$, speed v at time t, and constant acceleration a – a deceleration in this case.

The given data provide our constants: distance $x = 1$ metre, initial speed $u = 1000 * 50/(60 * 60) = (10/36) * 50$ metres per second, final speed $v = 0$. To estimate whether the woman can hold on to her baby, we will need to make a comparison with forces we have personally experienced. Most of us have tried lifting someone else, so let's use a characteristic human weight as our test mass. Thus, we have the (baby's) mass $m = 5$ kilograms, and a test mass, M say, which we will leave as a variable. But dealing with weight, we will need the acceleration due to gravity. For the kind of rough estimating we are doing, $g = 10$ metres per second per second will be an adequate approximation.

```
\constants{ x=1,v=0,u=(10/36)50,m=5,g=10 }
```

The deceleration experienced by the woman is found from the second equation of constant acceleration, $a = (v^2 - u^2)/2x$. Even if the deceleration isn't constant this will give an estimate of its magnitude. (If some of the deceleration is less than this a, some must be greater.) This is also the deceleration experienced by the baby as long as the woman holds onto her. Hence the magnitude of the force exerted by the baby on the woman's arms is $ma = m(v^2 - u^2)/2x = -mu^2/2x$ which we want to compare with our test force, say that required to lift $M = 70$ kilograms, which was once considered the mass of an average western adult male (but is doubtless a considerable underestimate now). Hence the test force is Mg. Let's do the calculations. (I have altered the `\constants` statement to allow for a later comparison with the effect of a small increase in speed.)

```
\constants{ x=1,u=(10/36)U,m=5,g=10 }[U=50]
\eval{$ mu^2/2x $}[0], \par
```

```
\eval{$ Mg $}[M=70].
```

$\implies$ $mu^2/2x = 482$,
$Mg = 700$, $(M = 70)$.

The force required to hold on to the baby is noticeably less than that required to lift a 70 kg person – in fact about the same as that needed to lift a 50 kg person. But we have ignored the force experienced by the mothers forearms – perhaps doubling m (baby plus forearms) would give a better estimate of the force she experiences. In that case $mu^2/2x$ obviously doubles and the total force required by the woman to retain her baby – now 964 newtons – is significantly more than that required to lift a 70 kg person. I think it almost certain that the baby is torn from her arms.

What difference does increasing the speed to 60 km/hr make?

```
\constants{ x=1,u=(10/36)U,m=5,g=10 }[U=60]
\eval{$ mu^2/2x $}[1], \par
\eval{$ Mg $}[M=70].
```

$\implies$ $mu^2/2x = 694$,
$Mg = 700$, $(M = 70)$.

Now the force of baby alone is comparable to that required to lift a 70 kg person. Including the woman's forearms in m, doubling m say, will result in a force twice as great – like that required to lift two 70 kg people or one 140 kg person. There is no chance of the woman holding on to her baby. The force is too great.

6.3.3.3 Example 3: macros and constants

Constants can depend on previously defined and registered user macros. Suppose I have defined two macros

```
\NewDocumentCommand\electronmassprecisevalue {}
                   {9.1093837015\times10^{-31}}
\NewDocumentCommand\protonmassprecisevalue {}
                   {1.672621898\times10^{-27}}
```

(I have taken both the names and the values from the `mandi` package.) The long explicit names of the macros has a pedagogic purpose, but they are too cumbersome to use in calculations. For that purpose we need, first, a `\macros` statement registering the two macros with `numerica`, and then a `\constants` statement like

```
\nmcConstants{ m_e=\electronmassprecisevalue,
               m_p=\protonmassprecisevalue }
```

With that `m_e` and `m_p` could be entered in formulas, taking the values contained in the macros. Let's do it:

```
\NewDocumentCommand\electronmassprecisevalue {}
                    {9.1093837015\times10^{-31}}
\NewDocumentCommand\protonmassprecisevalue {}
                    {1.672621898\times10^{-27}}
\nmcMacros{ \electronmassprecisevalue,
            \protonmassprecisevalue }
\nmcConstants{ m_e=\electronmassprecisevalue,
               m_p=\protonmassprecisevalue }
\eval{$ m_p/m_e $}
```

$\Longrightarrow \quad m_p/m_e = 1836.152645,$
the familiar mass ratio of proton and electron.

6.3.4 Viewing, counting constants

To see all constants currently 'in play', use the `view` setting in the `\constants` command. The main argument can be empty,

`\constants[view]{}` $\Longrightarrow$

constants: m_p=0.00000000000000000000000001672621898,
m_e=0.00000000000000000000000000000091093837015

or contain a list of constants. In the latter case, the display is of the above form but featuring the constants of the new list or, if the `add` setting is used, featuring the joined lists, old and new:

`\constants[view,add]{X=42}` $\Longrightarrow$

added: X=42
constants: m_p=0.00000000000000000000000001672621898,
m_e=0.00000000000000000000000000000091093837015, X=42

To count how many constants are currently in play, star the `\constants` command. The number will depend on whether the main argument is empty or not, and whether the `add` setting is active:

`\constants*{}` $\Longrightarrow 3.$

If the `view` setting is being used at the same time as the star, the `view` prevails.

6.3.5 Errors

When contemplating error messages from `numerica` it needs to be remembered that *multi-token* constants are added to the vv-list for every calculation. Hence an error may not be in the vv-list as indicated in the message but in the `\constants` statement, specifically, the multi-token constants.

6.4 Saving and reusing results: `\nmcReuse`

You may want to use at some place in a document a result calculated earlier. It would be good to be able to do so without having to do the calculation again at the new location. `numerica` offers a command `\nmcReuse` (short-name form, `\reuse`) which saves the numerical result of the most recent `\eval`-uation to a control sequence (a macro) that can then be used elsewhere in the document, expanding to the saved result. The control sequence and its content are also saved to file, allowing the possibility of using the result in other documents.

For those familiar with earlier usage, in version 2 of `numerica` the `\nmcReuse` command was completely rewritten and was no longer compatible with how the command was used in version 1. In version 3 of `numerica` the command has, again, been reworked with an eye to greater consistency in its use (and simplifying the code). The command created difficulties for itself by mixing two functions: (i) saving and retrieving a result, and (ii) making the retrieved result usable within another calculation (an `\eval`-uation). That latter function is the purpose of the `\macros` command and in version 3 of `numerica` has been removed from `\recur`. For a saved result to be used within an `\eval` command it must first be 'registered' by means of the `\macros` command – like any other LaTeX macro. Further, *only the numerical result* is now saved, although that can be saved as a decimal or in scientific notation or fraction form, but saving associated elements like the vv-list or math delimiters with the numerical result has been discontinued.

6.4.1 Use of `\nmcReuse`

As noted, all the supplementary commands share the syntax of the `\eval` command, so that `\nmcReuse` has an optional settings argument preceding a mandatory main argument, followed by two trailing optional arguments. `\nmcReuse` does not use the last two. The command is used to save the numerical result of the last `\eval` command to file and to load saved results from file. The results are saved as pairs, `\foo {<result>}`, where `foo` is a control sequence name (or macro name) chosen by the user. When loaded from file, `\foo` is globally defined to expand to `<result>`. The file that is used to store control sequences and their values is named `<\jobname>.nmc` where `\jobname` is the LaTeX macro that expands to the filename of the current document. The file is organized as a comma list.

Because of past practice,

- `\nmcReuse{}`, or `\reuse{}`, loads all saved macros and globally defines them to expand to their saved values;
- `\nmcReuse{foo}`, or `\reuse{foo}`, saves the numerical result from the most recent `\eval`-uation to file in the form `\foo {<result>}` and globally defines `\foo` to expand to `<result>`.

In light of the first of these you may want to put `\reuse{}` in the preamble of your document (*after* `\usepackage{numerica}` of course). In that way, saved control sequences are available from the start.

But these are somewhat inconsistent default behaviours. Version 3 of `numerica` offers options to make more explicit what is being done:

- `save`
  - `\reuse[save]{foo}` saves the numerical result from the latest `\eval` command to file `<\jobname>.nmc` as the pair `\foo {<result>}`;`\foo` is globally defined to expand to `<result>`; if `\foo` is defined elsewhere in LaTeX or already exists in `<\jobname>.nmc` with a *different* value then the save fails and a message is generated;
  - `\reuse[save]{}` is equivalent to `\reuse{}`, see above;
- `renew`
  - `\reuse[renew]{foo}` saves the numerical result from the latest `\eval` command to file `<\jobname>.nmc` as the pair `\foo {<result>}`, if necessary overwriting any previously saved value for `\foo`; `\foo` is globally defined to expand to `<result>`; if `\foo` already exists elsewhere in LaTeX then the save fails and a message is generated;
  - `\reuse[renew]{}` is equivalent to `\reuse{}`, see above;
- `load`
  - `\reuse[load]{foo}` loads `\foo` from file `<\jobname>.nmc` and defines it globally to expand to its saved value; if `\foo` is defined elsewhere in LaTeX or if it does not exist in file `<\jobname>.nmc` nothing happens;
  - `\reuse[load]{}` is equivalent to `\reuse{}`, see above;
- `delete`
  - `\reuse[delete]{foo}` undefines `\foo` in the current LaTeX session and deletes `\foo` and its value from file `<\jobname>.nmc` should it be there;
  - `\reuse[delete]{}` deletes all contents from the file `<\jobname>.nmc` and undefines all macros previously saved there from the current LaTeX session;

Only one of the options should be used at a time; if more than one is, it is the rightmost (in the settings option of the `\reuse` command) which prevails.

As examples of use, the following illustrate first the default behaviour and then the explicit use of the `save` option.

```
\eval{$ x+y $}[x=-1,y=3] \par
\reuse{two} >> \two \ <<
```

$\Longrightarrow x + y = 2, \quad (x = -1, y = 3)$
» 2 «
Only the numerical result is saved, not the math environment, so that if you want a minus sign to display correctly (rather than as a hyphen) when you use a saved control sequence you will need to ensure it is used in a math environment:

```
\eval{$ -1-1 $}. \reuse[save]{negtwo}
Now use the control sequence: $\negtwo$.
```

$\Longrightarrow -1 - 1 = -2$. Now use the control sequence: -2.

There may be occasions when you wish to change a previously saved value and yet, irritatingly, the control sequence name will now be known to LaTeX and so will generate an 'already defined' message. And if you choose a different name for the control sequence to save the new value to, do you want the old name cluttering the `.nmc` file? The settings `delete` and `renew` provide for such situations.

The following example shows the use of these options. In the example a control sequence `\testing` is defined in the first line. Being unable to recall what is in the `.nmc` file, I try deleting it from there in the second line. In the third line the sum of the first ten integers is evaluated. In the fourth line I try to save the sum to the control sequence `\testing`. That generates a message: `\testing` is defined elsewhere (in LaTeX) and therefore was *not* deleted. Okay, I resort to a new control sequence `\test`. In the fifth line, in case it is present in the `.nmc` file, I try deleting `\test` from there. The lack of a message means that either it was successfully deleted or was not present in the `.nmc` file in the first place. In the following lines, I calculate the sum of the first 20 integers and save the result to `\test`. The final line shows that this was successful.

```
\NewDocumentCommand\testing{}{1}
\reuse[delete]{testing}
\eval[env=\[ ]{\sum_{n=1}^{10}n}
\reuse{testing}\par
\reuse[delete]{test}\par
\eval[env=\[ ]{\sum_{n=1}^{20}n}
\reuse{test} Now testing
\textbackslash test: \test.
```

$\Longrightarrow$

$$\sum_{n=1}^{10} n = 55$$

!!! `\testing` defined elsewhere; failed save in: reuse command. !!!

$$\sum_{n=1}^{20} n = 210$$

Now testing \test: 210.

If a control sequence `\foo` is already known to LaTeX from elsewhere (perhaps it is a LaTeX command) then writing `\reuse{foo}` or `\reuse[save]{foo}` or even `\reuse[renew]{foo}` will produce a message and the result of the latest `\eval`-uation will *not* be saved :

```
\eval*{\sum_{n=1}^{10}n}\par
\reuse[renew]{sigma} \par
>> $\sigma$ <<
```

$\Longrightarrow$ 55
!!! `\sigma` defined elsewhere; failed save in: reuse command. !!!
» σ «
As you can see, `\sigma` is unchanged.

If a control sequence is already saved to file from a previous `\reuse` command trying to save a different value to it produces a message:

```
\eval{$-1-1-1$} \par \reuse{negtwo}
```

$\Longrightarrow -1-1-1=-3$
!!! Saved value for `\negtwo` already exists in file: numerica.nmc. !!!

If you really do want to change the stored value, use the `renew` option.

Empty results (from an `\eval`-uation) are *not* saved. A following `\reuse` command which attempts to do so generates another message:

```
\eval{1/0} \par \reuse{oops}
```

$\Longrightarrow$!!! `l3fp` error 'Division by zero' in: formula. !!!
!!! Nothing to save to `\oops` in: reuse command. !!!

The `\eval`-uative error has produced an error message and an empty result which generates the `\reuse` message. However, if `\oops` had already existed in `<\jobname>.nmc` the stored macro and its value would have been loaded to prevent LaTeX later complaining about an undefined control sequence, should `\oops` have been used later in the document, and halting compilation.

6.4.1.1 Seeing what is saved

The `view` setting is available for `\reuse`, as for other commands.

- `\reuse[view]{}` displays all saved commands and their values;
- `\reuse[view]{foo}` displays only `\foo` and its contents.

`view` is independent of the other `\reuse` options and can be used freely with them: they do their thing, and `view` shows what results. For instance, the option combination `[view,delete]` results *first* in the deletion and only after in the viewing. Note that because `\reuse{foo}` is equivalent to `\reuse[save]{foo}`, so `\reuse[view]{foo}` is equivalent to `\reuse[save,view]{foo}`.

6.4.1.2 Saving in other number formats

`numerica` can output numbers in scientific notation and in fraction form and save them in that form. In the example, to check that both results have indeed been saved, I use the `view` option:

```
\eval{ \pi }[//t6],  \reuse[renew]{fracpi} \quad
\eval{ \pi }[xx]. \reuse[renew]{scipi}
\reuse[view]{}
```

$\Longrightarrow \frac{355}{113}, \quad 3.141593 \times 10^{0}.$

saved: \scipi {3.141593\times 10^{0}}, \fracpi {\tfrac {355}{113}}, \test {210}, \negtwo {-2}, \two {2}

6.4.2 Using saved macros in calculations

To use a saved value in an `\eval`-uation requires first 'registering' the control sequence (the macro) containing the value with `numerica`. This is hardly surprising: saved control sequences are macros and `numerica` needs to be alerted to their presence as with other macros. The macros `\two` and `\negtwo` are active in the current document but, in the following example, the initial attempt to use them in an `\eval` command fails; the second time, after registering with the `\macros` command, it succeeds:

```
\eval{ \two \times \negtwo }\par
\macros{ \two, \negtwo }
\eval{ \two \times \negtwo }
```

$\Longrightarrow$!!! Unknown token \two in: formula. !!!
$2 \times -2 = -4$

6.4.3 The `.nmc` file

The file that control sequences are saved to has a filename composed of the document name with the extension `.nmc`. If your document is `mydoc.tex` (so that the LaTeX command `\jobname` expands to `mydoc`) then the file to which results are saved is `mydoc.nmc`, located in the document directory.

The `.nmc` file is a comma list of pairs of the form `\csname {value}`. Thus, the contents of `mydoc.nmc` might be `\csname1 {value1}`,`\csname2 {value2}`,..., `\csname-n {value-n}`. If `mydoc.nmc` does not already exist then `\reuse{csname}` will create it in the document directory, and `\csname {value}` will becomes its first element.

6.4.3.1 Editing the `.nmc` file externally

The `.nmc` file is a text file and can be edited in a text editor. Thus it is possible to externally add control sequences and values to it provided the (simple) structure of the file is strictly adhered to: pairs like `\csname {value}` separated by

commas. It is also possible to delete items from it or rename control sequences or edit values by the same mechanism. Editing the file externally like this, or renaming it, or transferring items from one `.nmc` file to another, provides a way of using saved values in multiple documents.

6.4.4 Counting, viewing all saved control sequences

The `\reuse` command has a starred form, `\reuse*` (or `\nmcReuse*`) which, like other starred commands, produces a purely numerical result. The number is the count of how many control sequences have been saved:

$$\texttt{\textbackslash reuse*\{\}} \Longrightarrow 5$$

The star does not interfere with the other functions of `\reuse`; the main argument does not need to be empty.

6.4.5 Obsolete `reuse` setting of `\eval` command

In version 2 of `numerica` the `\eval` command had a setting `reuse` that gave some choice as to what was saved. That has been discontinued. `numerica` evaluates mathematical expressions; the numerical result is what is important; the rest is cosmetics. Adding the extra capability was confusing both as to what was saved and where and how it could be used – and complicated the code. For the added complexity there was little gain.

Chapter 7

Miscellaneous matters

In this chapter I consider the nesting of commands, the parsing of mathematical arguments, the use of different mathematical environments and the use of `numerica` in the document processor LyX.

7.1 Nesting commands

The `\eval` command and the supplementary commands of the previous chapter can be *nested* – used within other `\eval` or supplementary commands. Nesting may occur in the main argument, or the vv-list, or the settings option, or some combination of all three. With the commands currently introduced, nesting is unlikely to be a major concern, but it becomes significant for the commands defined in the associated package `numerica-plus` (see §1.1.1). Since those additional commands are not available for this document, the examples below use the commands introduced earlier: `\eval`, `\info`, `\macros`, `\constants` and `\reuse`.

7.1.1 In the formula

Consider a statement like `\eval{...\eval...}`. There is an inner `\eval` and an outer `\eval`. The inner `\eval` 'digests' *its* LaTeX formula to produce an `l3fp`-readable expression which is fed to `l3fp` to evaluate. The result is then fed to the outer `\eval` to be inserted into the outer formula to be evaluated and then displayed. In version 1 of `numerica` that meant the inner command *had* to be starred, `\eval*`, so that no display formatting was fed to the outer command to try to digest (and cause an error). From version 2 of `numerica` this is no longer the case. `numerica` detects whether a command is inner or outer, and if inner, suppresses all display formatting, producing only a number, as if the command had been starred:

`\eval{$ \sin(\eval{\sin x}[x=\pi/6]\pi) + 1 $}` $\Longrightarrow \sin(0.5\pi) + 1 = 2.$

In the presentation of the overall result, the inner `\eval` command is evaluated, displaying as a number.

In the example, the inner vv-list could be attached to the outer `\eval x=\pi/6` since outer variables are available to the inner command – and it aids clarity:

`\eval{$ \sin(\eval{\sin x}\pi) + 1 $}[x=\pi/6]` $\Longrightarrow$
$$\sin(0.5\pi) + 1 = 2, \quad (x = \pi/6).$$

Or both inner and outer commands could have their own vv-lists – indeed, the same variable could appear in both and be assigned different values in each, without conflict:

`\eval{$ \sin(\eval{\sin x}[x=\pi/6]\pi) + x $}[x=1]` $\Longrightarrow$
$$\sin(0.5\pi) + x = 2, \quad (x = 1).$$

Just to show that it is possible, the next example shows `\eval` being used in a `\constants` command. The `o` setting in the `\constants` command pervades its argument; hence it needs to be explicitly turned off for the `\eval` if `\sin(\pi/6)` is to evaluate as expected.

```
\constants[o]{ y=\sin 30, x=\eval[o=0]{\sin(\pi/6)} }
\eval{$ x+y $}
```

$\Longrightarrow \; x + y = 1.$

7.1.1.1 Math delimiters and double evaluations

Any math delimiters in the inner `\eval` are ignored. (This also differs from version 1 of `numerica` where they caused an error.) Obviously it is simpler to omit them as I have done in the examples.

However, math delimiters in the outer `\eval` command still have their normal effect and produce a *formula = result, (vv-list)* display. One consequence of such a display is that the formula in the inner `\eval` command is evaluated *twice* – once when the overall result is being calculated (i.e. the formula of the outer `\eval`) and later when the overall display of the result is created. In the *formula* part of the *formula = result, [vv-list]* display, the tokens in the *formula* are expanded to their display form. For example, `\sin` is expanded to sin, `\pi` is expanded to π – and the inner `\eval` is expanded to the numerical result of its evaluation – which means a second evaluation. If the inner formula is simple, this will be of little moment, but should the inner formula contain, say, a slowly converging infinite series, then evaluating it twice is a bad idea and it would be better to remove the delimiters from the outer `\eval`. That prevents the second evaluation.

The problem does not arise if the outer `\eval` lies within a math environment (e.g. `$ \eval{...} $`) since that produces a display of the form *result, (vv-list).* The formula is not displayed and so the second evaluation does not occur. The inner `\eval` is evaluated once only to calculate the result.

7.1.2 In the vv-list

The inner `\eval` can be placed in the vv-list of the outer command. If the vv-list of the inner `\eval` contains a comma then the entire inner `\eval` and its LaTeX arguments needs to be wrapped in braces to hide *its* comma from the vv-list of the outer `\eval`. To show the effect of not doing so, I have slightly complicated a previous example by adding a second (unnecessary) variable. The first example is with braces, the second without:

`\eval{$ \sin k\pi + 1 $}[k={\eval{y\sin x}[x=\pi/6,y=1]}]` $\Longrightarrow$
$$\sin k\pi + 1 = 2, \quad (k = 0.5).$$
`\eval{$ \sin k\pi + 1 $}[k=\eval{y\sin x}[x=\pi/6,y=1]]` $\Longrightarrow$!!!
Unmatched] in: variable=value list. !!!.

The vv-list of the outer `\eval` is parsed as containing two entries, first `k=\eval {y\sin x}[x=\pi/6` and second `y=1]` (containing the right bracket). Both will cause errors but since the vv-list is evaluated from the right, it is `y=1]` which actually does so.

7.1.3 In the settings option

This will be rare, but commands can occur in the settings option of the outer command. The `\info` command provides a good example. I have included it in the punctuation setting of an `\eval`-uation.

```
\eval[p=\mbox{,\qquad\info{sum} terms.}]
  {\[ \sum_{n=0}^{\infty}\frac{(-1)^{n}}{n!} \]}[3]
```

$\Longrightarrow$

$$\sum_{n=0}^{\infty} \frac{(-1)^n}{n!} = 0.368, \qquad 9 \text{ terms.}$$

Because of the `\[ \]` math delimiters, if the `\info` command had been placed *after* the `\eval` command, it would have slid down to the next line. Used in the settings, as here, the display is *inside* the `\[ \]` delimiters, on the same line as the expression. This may be significant for adjusting vertical spacing of later parts of the document – widow and orphan control for instance.

A point to note is the explicit writing of the 'terms' descriptor. Normally `\info{sum}` would automatically supply the descriptor, but as noted earlier, nesting of one command in another suppresses all elements of display of the inner command beyond the numerical result. It is as if the inner command is starred. Because the `\info` command is nested in the `\eval` command, the 'terms' descriptor is suppressed and has had to be explicitly supplied by hand.

7.1.4 Rounding and display

In the display of the overall result, it is the numerical outcome of the inner command which is shown, not the formula that the inner command acts on.

In previous versions of `numerica` that result was *always* evaluated to 16 figures which were then fed into the formula of the outer command. From version 3.0.0, the number-format option of the inner command is heeded, *if it is present*, and only the specified number of decimal digits of the floating point result are fed to the the outer command. In its absence, the inner command feeds 16 significant figures to the the outer command. For example,

```
\eval[env=alignat*,ff]{ \pi - \eval{ \pi },
                        \pi - \eval{ \pi }[4] }[15*]
```

$\Longrightarrow$

$$\begin{aligned} \pi - 3.141592653589793 = &\quad 0.000000000000000 \\ \pi - 3.1416 = &-0.000007346410207 \end{aligned}$$

In the first instance, with no explicit number-format specification for the inner `\eval`, the outer result is zero because all 16 significant figures of the inner result have been passed to the outer `\eval`; in the second instance, with an explicit rounding number specified, only 4 decimal digits have been passed to the outer `\eval` and as a consequence the overall result does not vanish.

7.1.4.1 '-ed' environments

With 'inner' environments like `cases` and `dcases`, `array` and the AMS '`-ed`' environments which are used *within* a math environment, there can be a conflict between display and result when commands are nested. Consider the example

```
\[ \eval[env=array,pp,p=.,ff]{ \pi - \eval{ \pi },
                               \pi - \eval{ \pi }[15] }[15*] \]
```

$\Longrightarrow$

$$\begin{aligned} \pi - 3.141593 &= 0.000000000000000, \\ \pi - 3.141592653589793 &= 0.000000000000000. \end{aligned}$$

In both instances the result is zero to all 16 significant figures, but only the second instance, where an explicit rounding number is specified, displays correctly. The *implicit* rounding number in the first instance is used in the display but not in the calculation.

To see what is going on in this case, put `dbg=7*11` in the settings option. This will display both the fp-form and the LaTeX form of the result:

```
\[ \eval[env=array,dbg=7*11]{ \pi - \eval{ \pi } }[15*] \]
```

$\Longrightarrow$

fp-form: (pi)-(3.141592653589793)

LaTeX: \begin {array}{rcrl}\pi - \eval { \pi }&=&0.000000000000000\end {array}

In the *formula* part of the *formula=result* display, the *formula* contains the unevaluated `\eval{ \pi }`, whereas the *result* has long since been evaluated.

When LaTeX comes to display the whole thing, the `\eval` in the *formula* is finally expanded, which means evaluated. The rounding value is the implicitly specified 6 decimal places of *this* `\eval`, not the explicitly specified 15 of the outer `\eval`. This is a problem for this group of 'inner' environments. The solution is to explicitly specify rounding numbers for inner `\eval` commands.

7.1.5 Error messages

Errors in an inner command create a small change in error message display.

```
\eval{ 1 + \eval{ 1 + \eval{ k } } } ⟹
    !!! Unknown token k in: formula (3). !!!
\eval{ x + \eval{ k }[k=\arcsin 2] }[x=1] ⟹
!!! l3fp error 'Invalid operation' in: variable=value list (2). !!!
```

An integer is added to the 'where' part of the error message. The integer indicates the *depth* (or *level*) where the error occurs.

For an error at the top level the integer is suppressed, even though there may be nesting elsewhere in the overall expression. This is in the interests of straightforwardness when nesting is absent, which will be overwhelmingly the most common situation.

```
\eval{ k + \eval{ x }[x=1] }[k=\arcsin 2] ⟹
!!! l3fp error 'Invalid operation' in: variable=value list. !!!
```

7.1.6 Debugging

It is worth looking at the debug display when `\eval` commands are nested. For the outer `\eval` command:

```
\eval[dbg=1]{$ \sin \eval{\sin x}[x=\pi/6]\pi + 1 $} ⟹
```

formula: \sin \eval {\sin x}[x=\pi /6]\pi + 1
vv-list:
stored:
fp-form: sin((0.4999999999999999)(pi))+1

There is no vv-list for the outer command whence the two empty slots in the display but when the inner `\eval` is in the vv-list, they are filled:

```
\eval[dbg=1]{$ \sin k\pi + 1 $}[k={\eval{\sin x}[x=\pi/6]}] ⟹
```

formula: \sin k\pi + 1
vv-list: k={\eval {\sin x}[x=\pi /6]}
stored: k=0.4999999999999999
fp-form: sin((0.4999999999999999)(pi))+1
LaTeX: $\sin k\pi + 1=2,\mskip 12muplus6muminus9mu(k={\eval {\sin x}[x=\pi /6]})$

Contrary to their behaviour (which was accidental) in earlier versions of `numerica` debug numbers have no effect when used with an inner `\eval` command:

```
\eval{$ \sin{\left(
        \eval[dbg=1]{ \sin x }[x=\pi/6]
        \right)\pi} + 1 $}
```

$\Longrightarrow \sin(0.5)\,\pi + 1 = 2$

In the example, note the use of braces to define the argument of the outer `\sin` function; see §§2.1.1.2, 3.4.1.1.

7.2 Parsing mathematical arguments

A main aim of the `numerica` package is to require minimal, preferably no, adjustment to the LaTeX form in which an expression is typeset in order to evaluate it. But when writing formulas mathematicians do not follow codified rules of the kind programming languages insist on – like parenthesizing the arguments of functions, or inserting explicit multiplication signs (*) between juxtaposed terms. For the package to attain its aim, the question of where the arguments of mathematical functions end is acute.

Before discussing the rules `numerica` uses to answer this question, I discuss the tools the package provides to handle exceptions to those rules, when an author *does* need to make some adjustment to a formula for it to be evaluated correctly.

7.2.1 LaTeX braces

So, one has a complicated argument to a mathematical function. It is clear to a person reading the compiled expression in the `pdf` where the argument ends. The simplest way to make it clear to `numerica` also is by enclosing the argument in LaTeX braces. This is the recommended practice from version 3.0.0 of `numerica`, both because braces do not alter the visual appearance of the formula in the `pdf` and because it is consonant with general LaTeX practice.

The following expression, $\sin(n+\frac{1}{2})(x-t)$, or one like it, occurs in multiple texts on Fourier series. The human reader understands that the argument of the sine includes *both* parenthesized factors. But asking `numerica` to evaluate it 'as is' produces

```
\eval{$ \sin(n+\tfrac12)(x-t) $}[n=3,x=t+\pi,t=1.234]
```
$\Longrightarrow \sin(n+\frac{1}{2})(x-t) = -1.102018, \quad (n=3, x=t+\pi, t=1.234),$

which is $(\sin\frac{7}{2})\times\pi$ and not what was intended. Enclosing the whole argument – both factors – in braces rescues the situation,

```
\eval{$ \sin{(n+\tfrac12)(x-t)} $}[n=3,x=t+\pi,t=1.234]
```
$\Longrightarrow \sin(n+\frac{1}{2})(x-t) = -1, \quad (n=3, x=t+\pi, t=1.234),$

which is $\sin(\frac{7}{2}\pi)$, as intended. Another example is

```
\eval{\[\cos{\frac{2\pi}T n(t+\tfrac12T)}\]}[T=2,t=1,n=3]
```
$$\Longrightarrow$$
$$\cos\frac{2\pi}{T}n(t+\tfrac12T)=1,\qquad (T=2,t=1,n=3)$$

which otherwise evaluates to -6.

See §3.4.1.1 for further examples. The point is that the braces do not alter the visual appearance in the `pdf` but do delimit the whole argument for the `\eval` command.

7.2.2 The cleave commands \q and \Q

An alternative to using braces is to use `numerica`'s 'cleave' commands. The word *cleave* has two opposed meanings: to adhere or cling to, and to split apart or separate. `numerica` defines two commands, `\q` and `\Q` to achieve these opposite effects. When a mathematical argument is being parsed, the `\q` command joins the next token to the argument (cleaves *to*); the `\Q` command severs the next token from the argument (cleaves *apart*). Neither command leaves a visible trace in the output or has any other effect on the calculation beyond joining or severing the tokens on either side.

Repeating the penultimate example with `\q` between the bracketed factors rather than enclosing all in braces,

```
\eval{$ \sin(n+\tfrac12)\q(x-t) $}[n=3,x=t+\pi,t=1.234]
```
$$\Longrightarrow \sin(n+\tfrac12)(x-t)=-1,\quad (n=3,x=t+\pi,t=1.234).$$

which again is $\sin(\frac72\pi)$.

The `\Q` command splits an argument. Without it, we have

```
\eval{$ \ln n\,X $}[n=2,X=e^2]
```
$$\Longrightarrow \ln n\,X=2.693147,\quad (n=2,X=e^2);$$

with `\Q` inserted before the fraction,

```
\eval{$ \ln n\Q\,X $}[n=2,X=e^2]
```
$$\Longrightarrow \ln n\,X=5.121703,\quad (n=2,X=e^2).$$

However, an author could be kinder to the reader by not using a thin space (`\,`) to indicate separation of the terms but rather parentheses, $((\ln n)X$, or rearrangement, $X\ln n$, and avoid the need for `\Q` entirely.

7.2.2.1 Mnemonic

As mnemonic, best seen in sans serif for the Latin Modern fonts used in this document, think of the letter `q` as a circle cleaving *to* a vertical descender; think of the letter `Q` as a circle cleaved *asunder* by the diagonal stroke.

7.2.3 Parsing groups

A formula is a sequence of tokens and brace groups. When evaluating a formula, `\eval` digests the formula from the left, LaTeX argument by LaTeX argument, where *argument* here means either a token (an N-type argument in `expl3`-speak) or a brace group (an n-type argument). Some mathematical functions have arguments which correspond to LaTeX arguments – think `\sqrt`, `\frac`, `\binom`; also `^`. But for mathematical functions like `\surd` or `\sin` or `\ln`, this is not so; nor is it for sums and products, nor for comparisons. There need be no direct translation of LaTeX argument (an 'L-arg') to mathematical argument (an 'M-arg').

Different mathematical functions, operations, relations – for convenience call them collectively *functions* – have different reaches when it comes to how we read subsequent tokens. We might think of a function as casting a shadow over subsequent tokens; what lies in the shadow is part of its argument; what lies beyond the shadow is not. Table §7.1 lists the different *parsing groups* (alternatively *shading groups*) that `numerica` takes account of. Those functions with the lowest group number have the shortest reach, cast the shortest shadows which progressively lengthen as the group number increases.

Table 7.1: Parsing groups

group	function/operation
I	surd, `Not`
II	unary functions, `/`
III	sums, products
IV	comparisons
V	`And`, `Or`

To justify the groupings in Table 7.1, I look at examples, particularly from *HMF* but also from G. H. Hardy, *A course of pure mathematics*, and other sources and use the following terminology. A mathematical argument may end *at* an L-arg, meaning immediately before the L-arg, or end *with* the L-arg, meaning immediately after the L-arg. Ending or not will in general depend on whether the argument is in *first position* – the position immediately following a function token like `\sin` or `\surd` – or in *general position* – any later position. Formatting elements do not change the position count. This applies to things like spaces or phantoms (and their arguments) or modifiers like `\left` or `\biggl`. Multi-token numbers (in decimal or scientific formats) are treated as single items; they advance the position count by exactly one. Finally, a *naked* sign (like a plus or minus sign) is one that is *not* enclosed in brackets lying wholly to the right of the function whose argument is being determined.

7.2.3.1 Parsing group I

The only functions in this category are the surd and logical `Not`.

Why distinguish the surd from other unary functions? Hardy writes (§13) $\surd(pq)$ but (§206) $\log xy = \log x + \log y$ – the surd does not have the reach of the logarithm. The logarithm extends to both members of a product; the surd

does not and the factors need to be parenthesized if the surd is to apply to both. Surely we all agree that `\sin2\pi`, displaying as $\sin 2\pi$, vanishes? The argument of the sine extends beyond the 2 to include the π. But `\surd2\pi`, displaying as $\surd2\pi$, we understand to be $\sqrt{2} \times \pi$. The argument of the surd ends with the 2. The surd binds more tightly to its argument than is true of unary functions generally.

For parsing group I

1. if a left bracket is in first position, the mathematical argument ends with the matching right bracket; otherwise
2. the argument ends with the item in first position and any L- or M-args required by that item.

If the factorial sign `!` *preceded* its argument, it too would belong in this parsing group, for it also binds tightly like the surd. This means that an expression like $\surd4!$ is intrinsically ambiguous. Is it the square root of 24 or the factorial of 2? In `numerica` it produces the (perhaps rather odd) error

`\eval{$ \surd 4! $}` $\Longrightarrow$!!! Empty argument to fp-ify in: factorial. !!!

Since `\eval` digests a formula from the left, the surd seizes the argument `4`; there is then nothing for the factorial to operate on. Parenthesizing like either `(\surd 4)!` or `\surd(4!)` repairs the situation. Because other unary functions (like the sine or logarithm) do not bind as tightly, this ambiguity does not arise for them.

Exponents cause no problem because taking square roots and raising to a power are commutative operations – the result is the same whichever is performed first.

`\eval[pp,ff]{$ \surd 3^4, \surd(3^4), (\surd3)^4 $}` $\Longrightarrow$
$$\sqrt{3^4} = 9, \quad \sqrt{(3^4)} = 9, \quad (\sqrt{3})^4 = 9.$$

7.2.3.2 Parsing group II

This category includes the trigonometric and hyperbolic functions, their inverses, the various logarithms and the exponential functions, and the signum function `\sgn`. It also includes the denominators of slash fractions `/`.

- In parsing group II we wish to accommodate usages like $\ln z^n = n \ln z$ (*HMF* 4.1.11), or $2 \arctan e^z$ (*HMF* 4.3.117), meaning an exponent is included in the argument. This must include exponents of parenthesized arguments to accommodate
$$\ln\left(1 + \frac{x}{n}\right)^n = n \ln\left(1 + \frac{x}{n}\right).$$
- An approximation to Stirling's formula for the factorial is often written $\ln N! \approx N \ln N - N$ (widely used in texts on statistical mechanics). Hence the factorial sign should also be considered part of the argument.

- $\ln xy = \ln x + \ln y$ means the argument must reach over a product of variables. Identities like $\sin 2z = 2\sin z\cos z$ mean the argument also reaches over numbers, and expressions like $\sin\frac{1}{2}\pi x$ (*HMF* 4.3.104) mean that it further reaches over `\tfrac`-s and constants.

- Essentially *anything* can be in first position, and without parentheses; e.g.
  - unary functions: $\ln\ln z$ (*HMF* 4.1.52), $\ln\tan\dfrac{z}{2}$ (*HMF* 4.3.116),
  - fractions: $\ln\dfrac{z_1}{z_2}$ (*HMF* 4.1.9), $\arcsin\dfrac{(2ax+b)}{(b^2-4ac)^{1/2}}$ (*HMF* 3.3.36), $\ln\dfrac{\tan z}{z}$ (*HMF* 4.3.73),
  - absolute values: $\ln\left|\dfrac{a+x}{a-x}\right|$ (*HMF* 3.3.25),
  - square roots: $\arctan\sqrt{\dfrac{\nu_1}{\nu_2}F}$ (*HMF* 26.6.8)

With these examples in mind, for parsing group II

1. if a left bracket is in first position, the mathematical argument ends with the matching right bracket and any attached exponent, or factorial or double factorial sign; otherwise

2. the mathematical argument includes the item in first position and any L- or M-args required by that item;

   (a) if the item in first position is a number, variable, constant or `\tfrac`

      i. the argument appends the next item if it is a number, variable, constant or `\tfrac`, and so on recursively; or

      ii. the argument appends the next item if it is an exponent, or facorial or double factorial sign, and ends there; otherwise

      iii. the argument ends.

   (b) if the item in first position is not a number, variable, constant or `\tfrac`

      i. the argument appends the next item if it is an exponent, or factorial or double factorial sign, and ends there; otherwise

      ii. the argument ends.

Thus an argument may extend over numbers, constants, variables and `\tfrac`-s, as instanced by $\sin 2\frac{p}{q}\pi x$ which exhibits all elements.

There are still areas of ambiguity. Consider the different outcomes here (the logarithms are to base 10):

```
\eval[pp,ff]{$ \log m^3n, \log nm^3 $}[m=10,n=5] ⟹
```

$\log m^3n = 15, \quad (m=10, n=5), \quad \log nm^3 = 3.69897, \quad (m=10, n=5).$

In both instances the argument stops with the exponent, which means n is *not* part of the argument in the first. Any criterion is going to miss some instances where a different outcome might be anticipated. Where an argument ends is affected by visual appearance in the `pdf`. Anything that breaks the 'visual flow' of juxtaposed numbers, variables, constants and `\tfrac`-s is considered by `numerica` to end the argument. An exponent does that. If you feel ambiguity is possible, parenthesize or rearrange to clarify.

Similarly, a `\dfrac` breaks the visual flow in the following:

`\eval[pp,ff]{$ \sin\dfrac12\pi, \sin\tfrac12\pi $}` $\Longrightarrow$

$$\sin \frac{1}{2}\pi = 1.50616, \quad \sin \tfrac{1}{2}\pi = 1.$$

Nearly always, someone writing an expression like this intends the π to be part of the argument. In that case, a `\tfrac` should be used since the `\dfrac` breaks the 'visual flow' of the argument.

`\frac` The problem comes with `\frac` which in an inline environment displays like `\tfrac`. I considered making the argument behaviour of `\frac` the same as `\tfrac` for textstyle contexts, and the same as `\dfrac` for displaystyle contexts, but that would have meant the same expression evaluating to different results depending on whether it lay in an inline or displaystyle environment. That was unacceptable and for argument determination `\frac` is treated like `\dfrac` in *all* environments.

Slash fraction denominators When using `/` to write fractions or indicate division, it is easy to write ambiguous expressions. How should $\pi/2n$ be interpreted? With from-the-left evaluation and calculator precedence rules (which give equal precedence to multiplication and division), this would be interpreted as $(\pi/2)\times n$, but most people will instinctively interpret it as $\pi/(2n)$ and this is what `numerica` does. It places `/` in parsing group II and treats the denominator of the slash *as if it were the argument of a unary function.* This means that $1/2\sin(\pi/6)$ is parsed as $(1/2)\sin x$ rather than as $1/(2\sin x)$. It also means that $1/2\exp(1)$ and $1/2e$ give different results. In the author's view this is acceptable since they display differently and are not instinctively read in the same way. (But again authors should paranthesize or rearrange to avoid ambiguity.)

Trigonometric functions Trigonometric functions are set to parsing group II. This accommodates many instances of how arguments are used with these functions, but trigonometrical identities and Fourier series in particular make a nonsense of the restrictions. I find $\tan\frac{1}{2}(A+B)$ (*HMF* 4.3.148) and $\sec\pi(\frac{1}{4}+\frac{1}{2}az)$ (*HMF* 19.3.3), $\cos(2m+p)z$ (*HMF* 20.2.3) and $\sin(2n+1)v$ (*HMF* 16.38.1), and, on looking through various texts on Fourier series,

$$\cos\frac{2\pi}{T}nt, \quad \cos\frac{2\pi}{T}n(t+\tfrac{1}{2}T),$$

$$\cos(N+\tfrac{1}{2})\frac{2\pi\tau}{T}, \quad \sin 2\pi\left(\frac{x}{\lambda}-\frac{t}{T}\right).$$

Previous versions of `numerica` employed a setting `()=0,1,2` to parse these outliers from usual practice. From version 3.0.0, this setting has been removed and authors are advised to enclose such arguments in LaTeX braces. The braces do not affect the visual appearance in the `pdf` but unambiguously tell `numerica` where the argument begins and ends.

7.2.3.3 Parsing group III

The only members of this group are `\sum` and `\prod`. A 'naked' plus or minus sign is one that is not enclosed by brackets. For parsing group III

1. the argument ends
   (a) at the first naked plus or minus sign encountered, or
      i. first naked comparison sign (or command) encountered, or
      ii. first naked `And` or `Or` sign encountered, or
   (b) at the end of the formula.

Almost always this means (a) or (b), and seems to be the instinctive practice. *HMF* has multiple examples in multiple chapters of the argument to a sum ending at a naked plus sign: 7.3.12 & 7.3.14, 9.1.11 & 9.1.77, 9.6.35 & 9.6.43, 11.1.9, ... (at that point I stopped looking). They were all of the form

$$\sum \text{argument} + \ldots$$

A minus sign serving the same purpose was harder to find but *HMF* 10.4.65 & 10.4.67 are two instances. I considered whether a `\times` or slash fraction sign `/` might end the argument of a sum, but surely we need to allow things like $\sum 1/n^2$ which rules out the slash and *HMF* 9.9.11 provides two of a number of instances of sum arguments continuing past explicit `\times` signs (at line breaks when a summand spills onto a second line).

Please note that parenthesizing to clarify the argument of a sum has limitations. Writing

$$\sum (< \texttt{stuff} >) < \texttt{more stuff} >$$

does not necessarily end the summand at the right parenthesis: it ends at the first naked + or − sign encountered (if it is not forced by a `\Q` command or LaTeX braces). Sums *have to* include brackets to cope with factors like $(-1)^n$.

Because they are evaluated using the same code as sums I (at first unthinkingly) placed products with sums but doubts later intruded. In *HMF* products occur only occasionally and are almost all of the form

$$\prod (\text{argument})$$

where the argument is bracketed (often with `\left \right` modifiers) and the multiplicand ends with the right bracket. At least twice (*HMF* 6.1.25 and 24.2.2.1) an exponent (-1) is attached to the right bracket and the argument ends there. Looking further afield, a text on number theory has examples where the argument of the product extends to *three* parenthesised factors, $\prod$ (arg1) (arg2) (arg3) and a number of others where it extends to two. A text on theory of functions has

$$\prod_{n=1}^{\infty} \left(1 + \frac{z}{n}\right) e^{z/n}$$

although *HMF*, for the same expression, encloses the two factors within (large) square brackets, as if some ambiguity existed as to how far the reach of the `\prod` extended.

Tentatively I retain products here in the same group as sums.

7.2.3.4 Parsing group IV

Comparison symbols form this group: `=`, `<`, `>`, `\ne`, `\le`, `\ge`, `\leq`, `\geq`, and the various comparison commands from the `amssymb` package listed in §2.3.5.5. It is the argument on the right-hand side of the relation that needs determining (the argument on the left ends at the comparison). For parsing group IV

1. the argument ends at
   (a) the first naked `And` or `Or` encountered, or
   (b) the first naked comparison sign or command encountered, or
   (c) the end of the formula.

7.2.3.5 Parsing group V

Logical `And` and logical `Or` are the sole members of this group. It is the right-hand side of the `And` or `Or` command that needs determining. For parsing group V

1. the argument ends at
   (a) the first naked `And` or `Or` encountered, or
   (b) the end of the formula.

7.2.3.6 Disclaimer

The parsing rules of the different groups are not normative; they are not statements of how mathematical formulas should be written. Rather they attempt to capture regularities in how mathematicians write formulas. It is *how things look and are read in the* `pdf`, not LaTeX, that is the guide. You are always free to parenthesize as you see fit, to insert cleave commands (`\q` or `\Q`) or to use LaTeX braces to force outcomes.

The rule should always be to write expressions that are clear to the reader of the `pdf`. An expression that is ambiguous to the reader, even if it fits within the parsing rules, is to be deprecated. The *intent* is that `\eval` can parse unambiguous expressions of the kind that mathematicians, scientists, engineers *do* write, rather than falter over 'corner cases'.

7.3 Using `numerica` with LyX

The document processor LyX has a facility that enables snippets from a document to be compiled separately and the results presented to the user without having to compile the entire document. The present document was written in LyX. The demonstration calculations were evaluated using this *instant preview* facility.

To use `numerica` in LyX go to Document ▷ Settings ▷ LaTeX Preamble and enter

```
\usepackage{numerica}
```

then click OK. You may wish to follow the above line in the preamble with `\nmcReuse{}`:

```
\nmcReuse{}
```

In that case, type the extra line and *then* click OK. The additional line ensures all saved values are available in your document from the outset.

7.3.1 Instant preview

The instant preview facility of LyX performs mini-LaTeX runs on selected parts of a document (for instance, the mathematical parts) and displays the results in LyX while the user continues to work on the surrounding document. `numerica` uses these mini-LaTeX runs to do its evaluations and display their results. That means you get feedback on your calculations almost immediately.

To use this facility first ensure that instant preview is turned on. This means selecting Tools ▷ Preferences ▷ Look & Feel ▷ Display, ensuring that the Display graphics checkbox is checked, and against Instant preview selecting On, then clicking OK.

7.3.1.1 Document location

It also matters where your document is located. You may have your own local or personal texmf tree. If your document is located there, perhaps in the `doc` folder, then not all features of preview will work as expected. Presumably this is because both LyX and your LaTeX distribution (e.g. TeXLive or MiKTeX) are interacting with the location and interfere. Move your document to another location which your LaTeX distribution has no interest in, and open it in LyX there.

7.3.1.2 Global vs local previewing

Compilation of previews occurs in two distinct modes.

Global preview generation: When a document is opened (and preview is *on*), all previews in the document are formed in sequence in the one LaTeX run. This is the global mode. The mini-LaTeX run may well be substantial. It compiles a `.tex` file that begins with the document's preamble with some additions then comes `\begin{document}`. That is followed by a sequence of preview environments,

```
\begin{preview}
<stuff>
\end{preview}
```

one for each preview in the document. Finally there is an `\end{document}` statement. Critically, all previews are between the same `\begin{document}`, `\end{document}` statements, and so earlier previews in the sequence can communicate with later ones.

Local preview generation: The other mode in which preview operates is local. Suppose you have your document open and want to add to it, for instance with a simple evaluation, `\eval{x+y}[x=1,y=2]` in an ERT inset in a preview inset. The resulting mini-LaTeX run is of the form

```
<preamble>
\begin{document}
\begin{preview}
\eval{x+y}[x=1,y=2]
\end{preview}
\end{document}
```

The preamble is as before but only a solitary preview sits between the `\begin{document}`, `\end{document}` statements. That preview is isolated from all other, previous previews and will be isolated from all other, later previews.

This has implications for the supplementary commands of Chapter 6 and means that if you want to transfer information (a macro, a constant, a result) from one preview to another, you need to do it through the preamble or by means of an external file or, in some cases, by forcing a global preview run in which all previews are recompiled between the same `\begin{document}`, `\end{document}` statements.

Forcing a global preview run Closing then opening a document is one way to force a global preview compilation. Another is to change the zoom level. This causes LyX to recompile all previews at the new zoom level. But you may not want to work at the new zoom level. Going back to the old zoom level will force a second recompilation of all previews. For a large document *two* recompilations

is too heavy a burden. The secret is to combine a zoom in and a zoom out into one command and attach it to a shortcut.

If you go to Tools ▷ Preferences ▷ Editing ▷ Shortcuts, click on the New button and enter

```
command-sequence buffer-zoom-in; buffer-zoom-out
```

then assign a shortcut to it (`Alt+Z` for zoom?) you will gain a simple means of forcing a global recompilation of previews.

7.3.1.3 Mathed

(Mathed = the LyX mathematics editor.) If you have instant preview *on* then one way to use `numerica` in LyX is to enter an `\eval` command in mathed. Clicking the cursor outside the editor with the mouse or moving it outside with the space bar or arrow keys will then trigger formation of a preview of the editor's contents – a snippet of what will be shown in the `pdf`. This will be displayed in mathed's place after a generally short 'pause for thought' as the mini-LaTeX run progresses behind the scenes.

The original expression can be recovered by clicking on the preview. The content of mathed is immediately displayed and can be edited.

LaTeX braces { } LyX does not support `numerica`'s `\eval` command 'out of the box' as it does, say, `\frac` or `\sqrt`. To use the `\eval` command in mathed you will need to supply the braces used to delimit its mandatory argument. (For `\frac` and `\sqrt` by contrast, LyX supplies these automatically in the form of blue-outlined boxes.) Unfortunately the `{` key[1] does not insert a left brace into the document but rather an escaped left brace `\{` as you can see by looking at View ▷ Code Preview Pane. Escaped braces like this are used for grouping terms in *mathematics*; they are not the delimiters of a LaTeX argument.

The brace delimiters for LaTeX arguments are entered in mathed by typing a backslash \ then a left brace { – two separate key presses rather than a single combined press. This enters a balanced pair of (unescaped) braces with the cursor sitting between them waiting for input. Alternatively, if you have already written an expression that you want to place between braces, select it, then type \ then {.

7.3.1.4 Preview insets

There are problems with using mathed for calculations.

- Expressions entered in mathed are necessarily of the form `$ \eval... $` or more generally `delimiter \eval... delimiter`. But you may wish to wrap the `\eval` command *around* the math delimiters to produce a *formula=result* form of display. In mathed the only way to effect such a display is to write the *formula=* part yourself – which may involve no more than copy and paste but is still additional mouse work/key pressing.

[1] Shift+[on my keyboard.

- Mathed does not accept carriage returns. If you want to format a complicated expression for readability by breaking it into separate lines, you can't. The expression is jammed into the one line, along with the settings option content and the vv-list, often extending well beyond the edge of the screen.

For these reasons I have come to prefer *not* using mathed for calculations but instead to use preview insets wrapped around TEX-code (ERT) insets. LYX uses the shortcut Ctrl+L to insert an ERT inset. Since LYX now does no printing itself, the shortcut Ctrl+P that was formerly used for printing is available for other purposes. On my keyboard, the P key lies diagonally up and to the right but adjacent to the L key. I suggest assigning Ctrl+P to inserting a preview inset. Then typing Ctrl+P Ctrl+L – which means holding the Ctrl key down and tapping two diagonally adjacent keys, P followed immediately by L – will insert an ERT inset inside a preview inset with the cursor sitting inside the ERT inset waiting for input. In the ERT inset you can enter carriage returns, and so format complicated expressions. You can place the vv-list on a separate line or onto consecutive lines. And when you have finished, clicking outside the preview inset will trigger preview into doing its thing and present the result 'before your eyes'.

To assign the suggested shortcut, go to Tools ▷ Preferences ▷ Editing ▷ Shortcuts. Under Cursor, Mouse and Editing Functions in the main window on the right, scroll down until you come to preview-insert, select it, then click Modify. Now press Ctrl+P. The shortcut will magically appear in the greyed, depressed key. Click OK and then OK in the Preferences window to close it. (Most of the examples in this document have been evaluated in this way, using Ctrl+P Ctrl+L.)

7.3.1.5 Errors

Instant preview will display error messages generated by `numerica` in LYX just as it does the results of calculations. Clicking on the message will show the underlying expression which can then be edited. However LATEX errors will *not* produce a preview; formation of the preview will stall. To find precisely what has gone wrong, you will need to look at the LATEX log, but not the log of the overall document; rather the *preview* log.

Temporary directory of LYX Unfortunately this is tucked away in a temporary directory and is not immediately accessible in LYX (unlike the main LATEX log from Document ▷ LATEX Log). When LYX is started, it sets up a temporary directory in which to perform various tasks. On Windows systems this will be located in `C:\Users\<your name>\AppData\Local\Temp` and will have a name like `lyx_tmpdir.XOsSGhBc1344`.

One of the tasks LYX uses this temporary directory for is to create preview images when a document is opened. If you look inside LYX's temporary directory when a document is first loaded, you will see a subdirectory created, with a

name like `lyx_tmpbuf0`. There may already be such directories there, in which case the number on the end will be greater than `0` – it depends on whether other documents are or have been open in the current instance of LyX. Inside the appropriate `lyx_tmpbuf`*n* folder will be the preview log with a name like `lyxpreviewZL1344.log`. It will usually be accompanied by other files with extensions like `.dvi`, `.tex`, and – depending on the number of previews in your document – a number, perhaps a lot, of image files with the extension `.png`, each one of which is a preview. For a document just loaded there will be only the one preview log, but if you have added preview insets or math insets to your document in the current editing session there will be a number of such logs and you will need to determine the relevant one by the time stamp.

The log files are text files and can be opened in a text editor. The relevant part of the log is towards the end (just before the final statistical summary) where you will find a list of entries like `Preview: Snippet 1 641947 163840 7864588`. If there is an error, it will be noted here among these snippets and will generally make clear what needs remedying.

CPU usage, LaTeX processes It is possible when a preview stalls that the LaTeX process associated with the preview will continue to run, using CPU cycles, slowing overall computer performance, and perhaps resulting in extra fan use giving a different sound to the computer. In Windows 10, the Task Manager (Ctrl+Shift+esc) under the Details tab shows the current executables running. The CPU column will show which processes are preoccupying the CPU. Check whether one or more of these processes looks LaTeX-related (e.g. `latex.exe` or `pdflatex.exe`, or `miktex-pdftex.exe` if using MiKTeX). Click the Name column to sort the processes by name and look for the relevant name in the list, select it, and end the process (click the End Task button).

I am not familiar with the corresponding situation on Linux or Mac.

7.3.1.6 Hyperref support vs speed

If you want the `pdf` produced from your document to support hyperref links and show an outline window in your `pdf` viewer (generally placed on the left in the viewer) then you need to ensure the checkbox at Document Settings ▷ PDF Properties ▷ Use Hyperref Support is indeed checked. But you don't need to do this until the final compilation of the document. The advantage of leaving this until the last is that in a large document with many previews the time for preview generation is essentially halved. If hyperref support is enabled, preview generation not only creates all the individual image files that are the previews (files of extension `.png`) but also requires the compilation of a single `pdf` document showing all the previews in sequence. (Like the previews, the `pdf` document 'hides' in the termporary directory where LyX does its work.) In other words, *two* images are created for each preview, the `.png` image which is the one LyX displays, and another image buried inside the `pdf` of all images. That second step does not occur if hyperref support is disabled. In a small document, this is not going to matter; in a large document it becomes significant. It is well

worth temporarily turning off hyperref support and then, when the time for final compilation comes, turning it back on.

7.3.2 Supplementary commands in LyX

There are some difficulties using the supplementary commands successfully with instant preview.

7.3.2.1 Reuse of earlier previews

One is that whenever LyX has generated a preview image for a particular LaTeX expression, it will use that same image whenever it meets that same LaTeX expression later. That means that a statement like `\macros[view]{}` and the same statement later will display the same image, even though there may have been macros defined or freed in between. The same goes for all the other supplementary functions, including, for example, `\info{sum}`. A second instance of `\info{sum}` will display the image generated by the first instance even though further infinite sums may have been evaluated between the `\info` statements.

The remedy is to make some small but insignificant difference to the LaTeX expression in the second instance – generally a change in white space will do. For example: first time `\macros[view]{}`, second time `\macros[view]{ }` where a space has been inserted between the braces; or: first time `\info{sum}`, second time `\info{ sum}` where a space has been inserted before `sum`. This will ensure LyX doesn't fall back on the previously generated image.

7.3.2.2 'Stalled' previews

It is possible to put content into an ERT inset inside a preview inset (Ctrl+P Ctrl+L) and for nothing to happen. The preview has apparently stalled. Certainly this can be the case if there is an error in the input (e.g. a missing brace) but it also occurs if there is no output to display. For instance `\constants { c=300000000 }` does not produce any visual output. There is nothing for the preview to display and so the preview inset sits there, apparently stalled. This is a security measure for previews in LyX to provide at least some guard against malicious code being run in the preview. If the preview resolved, it would disappear completely from view in the LyX window.

If you find the visual appearance of such apparently stalled previews distracting, the addition of some displayable content to the preview will result in it resolving to that content; the content could be as small as a full stop.

7.3.2.3 Using `\nmcMacros`

As noted earlier, previews are mini-LaTeX runs, either local or global. Each local preview is of the form<preamble>

```
<preamble>
\begin{document}
```

```
\begin{preview}
<stuff>
\end{preview}
\end{document}
```

Whatever goes into or comes out of the preview is isolated from any other local preview, unless it is through the preamble or an external file. Sometimes a global preview run can overcome this problem for then all the previews lie between the same `\begin{document}`, `end{document}` statements. However, this does not help with macro definitions. `\def`, `\newcommand`, `\NewDocumentCommand` all provide *local* definitions which remain trapped within their own `\begin{preview}`, `\end{preview}`) statements. Another preview, say containing an `\eval` command, between a different pair of `\begin{preview}`, `\end{preview}`) statements, will not know about the macro definition.

There are (at least) three ways out:

1. Confine everything to the same preview inset: the definition of a macro, the `\macros` statement, and the use of the macro in an `\eval` command.
2. Confine macro definitions to the preamble (Document ▷ Settings ▷ LaTeX Preamble).
3. Within previews use `\gdef` (or `\global\def`) exclusively for making your macro definitions; this makes the macro available to all later previews.

7.3.2.4 Using `\nmcConstants`

Because `\nmcConstants` doesn't use `\def` or `\newcommand` or `\NewDocumentCommand` it is not subject to the same localisation problem as `\nmcMacros`, but the reach of a `\constants` command will still be confined to its own preview unless a *global* preview run is forced; see above §7.3.1.2.

7.3.2.5 Using `\nmcReuse`

As noted earlier, LyX creates its previews in a temporary directory, not the document directory. If you want to save values from your current document – say, `mydoc.lyx` – to `mydoc.nmc` then you do so as described earlier (§6.4), but the file `mydoc.nmc` containing the saved results will be located in the temporary directory. When LyX is closed the file will be deleted along with all the other contents of that directory.

Fortunately LyX has a copying mechanism for getting files out of the temporary directory and into the document directory. When a document is exported – say to `pdf` – it is possible to specify a *copier* to automatically copy back to the document directory or subdirectory various files in the temporary directory. We want the `.nmc` file containing the saved values to be copied back. Go to Tools ▷ Preferences ▷ File Handling ▷ File Formats and find PDF (pdflatex) (assuming export to `pdf` by this route) in the list of formats. In the Copier slot of the dialogue insert the following line of code:

```
python -tt $$s/scripts/ext_copy.py -e nmc,pdf -d $$i $$o
```

`ext_copy.py` is a python script that is supplied with LyX. The `-e nmc,pdf -d` part of the line tells `ext_copy.py` that on export to `pdf` by the `pdflatex` route to copy any files with the extensions `.nmc` or `.pdf` from the temporary directory where LyX does its work back to the document directory – the `-d` option (which became available with LyX 2.3.0).

But if you have a complex document, it may take too much time to want to export to pdf before closing LyX, particularly if there are a lot of evaluations in the document. Much faster is to export to *plain text*, not because you want a plain text version of your document but because it too can be used to trigger the copier mechanism. Go to Tools ▷ Preferences ▷ File Handling ▷ File Formats and find Plain text in the list of formats. In the Copier slot enter

```
python -tt $$s/scripts/ext_copy.py -e nmc -d $$i $$o
```

The only difference from the previous copier command is the absence of `pdf`.[2] This will copy `mydoc.nmc` with its saved values from the temporary directory back to the document directory. To effect the export, go to File ▷ Export and find Plain text in the list of formats and click on it.

A shortcut would be nice. For that go to Tools ▷ Preferences ▷ Editing ▷ Shortcuts, click on New, enter `buffer-export text` in the Function: slot, click on the blank key against Shortcut: and type your shortcut. You may have to try a number before you find one that hasn't already been assigned. (I'm using Ctrl+; for no particular reason beyond the fact that it fits under the fingers easily and saving values to the document directory has a punctuation-like feel to it, a pause in the process of writing.) It is now an easy matter to press the shortcut at the end of a LyX session to copy all the values saved in `mydoc.nmc` back to a file of the same name in the document directory. And it is brisk, not least because plain text export ignores ERT insets (and hence preview insets wrapped around ERT insets), nor does it evaluate `\eval` commands in math insets.

7.3.2.6 A final tweak?

But one still needs to *remember* to press the shortcut. The thought arises: can *closing* the current document trigger the copying process? LyX provides a means of linking two commands and assigning a keyboard shortcut to them with its `command-sequence` LyX function. I suggest assigning a shortcut to

```
command-sequence buffer-export text; view-close
```

Indeed, why not reassign the current shortcut for `view-close`, which is Ctrl+W on my system, to this command sequence? (I use the `cua` key bindings – check the Bind file: slot in Tools ▷ Preferences ▷ Editing ▷ Shortcuts.)

[2] I'm assuming that you don't actually want the plain text version of the file copied back. If you do, then change `-e nmc` to `-e nmc,txt`.

Please note, however, that *this will work as intended only from LyX 2.4.0.*[3] For LyX 2.3 and earlier, the command sequence will generally fail because of 'asynchronous' processing – `buffer-export` and `view-close` use different threads and the latter may well start before the former is complete. From LyX 2.4.0 this defect has been fixed. You press your shortcut, the export to plain text occurs and the `.nmc` file is copied back to the document directory, then the current view is closed.

Note that in the other direction, the `.nmc` file in your document directory is *automatically* copied to the temporary directory when needed. Nothing needs to be done by you, the user.

7.3.3 Use of LyX notes

The central fact about a LyX note is that it does not contribute to the pdf. But instant preview still works there. This suggests a possibility: that a calculation be performed within a LyX note and the result saved using `\nmcReuse` within the same note. The saved value is now available *from file* for use elsewhere in the document. In this way, some selected content from a LyX note *can* find its way into the pdf when the document is compiled.

[3] Long-planned and initially due for release in 2021 but still (August 2023) not released. But a beta3 version is now available from https://ftp.lip6.fr/pub/lyx/devel/lyx-2.4/

Chapter 8

Reference summary

8.1 Package options

- `comma` sets the comma as the decimal point; items in the variable=value list must then be separated by semicolons; arguments in n-ary functions must also be separated by semicolons; by default formulas in a multi-formula calculation should also be separated by a semicolon;
- `rounding`=n sets the default rounding value to the integer n;
- `approx` replaces the default `=` between formula and result in (some) displays with `\approx` (displaying as ≈; the `eq=<char(s)>` setting is still available to change the symbol for individual calculations).

8.2 Commands defined in `numerica`

1. `\nmcEvaluate, \eval`
2. `\q, \Q` ('cleave' commands)
3. `\nmcInfo, \info`
4. `\nmcMacros, \macros`
5. `\nmcConstants, \constants`
6. `\nmcReuse, \reuse`

Provided they have not already been defined when `numerica` is loaded, the following commands are defined in `numerica` using `\DeclareMathOperator` from `amsmath`:

1. `\arccsc, \arcsec, \arccot`
2. `\csch, \sech`

3. `\asinh, \acosh, \atanh, \acsch, \asech, \acoth`
4. `\arsinh, \arcosh, \artanh, \arcsch, \arsech, \arcoth`
5. `\arcsinh, \arccosh, \arctanh, \arccsch, \arcsech, \arccoth`
6. `\sgn, \lb`

Provided they have not already been defined, the following commands are defined in `numerica` using `\DeclarePairedDelimiter` from `mathtools`:

`\abs, \ceil, \floor`

Provided they have not already been defined, the following commands are defined in `numerica` using `\ProvideDocumentCommand`

`\comma, \equals, \degree`

The following commands have been redefined in `numerica` to give more spacing around the underlying `\wedge` and `\vee` symbols:

`\land, \lor`

8.3 'Digestible' content

`numerica` knows how to deal with the following content, meaning that any of these elements occurring within an `\eval` command should not of itself cause a `numerica` error. Not all formatting commands affect display of the output.

1. variable names (sequences of tokens given values in the variable=value list)
2. digits, decimal point
   (a) `1, 2, 3, 4, 5, 6, 7, 8, 9, 0, .` or `,` (if `comma` package option used)
3. constants
   (a) `e, \pi, \gamma, \phi, \deg, \infty`
4. arithmetic operators
   (a) `+, -, *, /, ^, \times, \cdot, \div`
5. logical operators
   (a) `\wedge, \land, \vee, \lor, \neg, \lnot`
6. degree symbol/switch
   (a) `\degree` (possibly from package `gensymb`)

7. comparisons
   (a) `=, <, >, \ne, \neq, \le, \leq, \ge, \geq`
   (b) (if `amssymb` loaded) `\nless, \ngtr, \geqq, \geqslant, \leqq, \leqslant, \ngeq, \ngeqq, \ngeqslant, \nleq, \nleqq, \nleqslant`
8. brackets, bracket-like elements, modifiers
   (a) `( ), [ ], \{ \}; \lparen \rparen` (from `mathtools`); `\lbrack \rbrack, \lbrace \rbrace`
   (b) `\lvert \rvert, \lfloor \rfloor, \lceil \rceil`
   (c) `| |` (cannot be nested, deprecated)
   (d) `\left \right; \mleft \mright, \mleftright, \mleftrightrestore` (if `mleftright` loaded)
   (e) `\bigl \bigr, \Bigl \Bigr, \biggl \biggr, \Biggl \Biggr`
   (f) `\middle` (if `mleftright` loaded);`\big, \Big, \bigg, \Bigg`
   (g) `. / |` (used with a modifier)
   (h) `\abs[]{}, \abs*{}, \floor[]{}, \floor*{}, \ceil[]{}, \ceil*{}`
9. unary functions (in the mathematical sense)
   (a) `\sin, \cos, \tan, \csc, \sec, \cot`
   (b) `\arcsin, \arccos, \arctan, arccsc, \arcsec, \arccot`
   (c) `\sin^{-1}, \cos^{-1}, \tan^{-1}, \csc^{-1}, \sec^{-1}, \cot^{-1}`
   (d) `\sinh, \cosh, \tanh, \csch, \sech, \coth`
   (e) `\asinh, \acosh, \atanh, \acsch, \asech, \acoth, \arsinh, \arcosh, \artanh, \arcsch, \arsech, \arcoth, \arcsinh, \arccosh, \arctanh, \arccsch, \arcsech, \arccoth`
   (f) `\sinh^{-1}, \cosh^{-1}, \tanh^{-1}, \csch^{-1}, \sech^{-1}, \coth^{-1}`
   (g) `\exp, \lb, \lg, \ln, \log, \log_{}, \sgn, \surd`
   (h) `\sqrt{}, \abs[]{}, \abs*{}, \floor[]{}, \floor*{}, \ceil[]{}, \ceil*{}`
   (i) `!, !!` (both with a prepended argument)
10. binary functions
   (a) `\tfrac{}{}, \frac{}{}, \dfrac{}{}, \sfrac` (if `xfrac` loaded)
   (b) `\tbinom{}{}, \binom{}{}, \dbinom{}{}`
   (c) `\sqrt[]{}`
11. n-ary functions

(a) `\min, \max, \gcd`

12. sum, prod

    (a) `\sum_{}{}^{}, \prod_{}{}^{}`

13. formatting commands

    (a) `,` (comma, in n-ary functions)

    (b) `{}, \\, &, \to`

    (c) `\begin{}, \end{}, $, \[, \]`

    (d) `\dots, \ldots, \cdots`

    (e) `\ , \, , \;, \:, \!, \>`

    (f) `\thinspace, \medspace, \thickspace,`

    (g) `\negthinspace, \negmedspace, \negthickspace,`

    (h) `\hspace*{}, \mspace{},`

    (i) `\quad, \qquad , \hfill, \hfil`

    (j) `\phantom{}, \vphantom{}, \hphantom{}`

    (k) `\xmathstrut[]{} , \splitfrac{}{}, \splitdfrac{}{}` (from `mathtools`), `\mathstrut`

    (l) `\displaystyle, \textstyle, \scriptstyle, \scriptscriptstyle`

    (m) `\label{}, \ensuremath{}, \text{}, \mbox{}, \smash{}`

    (n) `\color[]{}, \textcolor[]{}{}`

    (o) `\mkern, \mskip`

14. font commands

    (a) `\mathrm{}, \mathit{}, \mathcal{}, \mathtt{}, \mathbf{}, \mathbb{}, \mathsf{}, \mathfrak{}, \mathscr{}`

    (b) `\mathnormal{}, \boldsymbol{}`

    (c) `\textrm{}, \textsf{}, \texttt{}, \textit{}, \textsl{}, \textbf{}, \textsc{}`

8.4 Settings

8.4.1 Functional settings

key	type	meaning	default	initial
`dbg`	int	debug data		`0`
`view`		`dbg=1`	`dbg=1`	
`^`	char	exp. mark for sci. notation input	`e`	
`xx`	int (0/1)	accept multi-token variables		`1`
`ff`	char	main arg. multi-formula delimiter	`,` (if decimal dot) `;` (if decimal comma)	
`1s2`	int (0/1)	allow spaced digit groups in numbers	`1`	`0`
`/min`	int ≥ 1	fraction form denominator search start		`1`
`/max`	int ≥ 1	fraction form denominator search end		`200`
`vv@` `vvmode`	int (0/1)	vv-list calculation mode		`0`
`o`	int (0/1)	trig. function args in degrees	`1`	`0`
`log`	num	base of logs for `\log`		`10`
`S+`	int	extra rounding, sums		`2`
`S?`	int ≥ 0	number of query terms, sums		`0`
`P+`	int	extra rounding, products		`2`
`P?`	int ≥ 0	number of query terms, products		`0`
`()`	obsolete; see §5.4			
`reuse`	obsolete; see §5.4			
`*`	obsolete; see §5.4			

8.4.1.1 Debug settings

- `dbg=0` turns off the debug function, displays the result or error message (this is the initial value);
- `dbg=1` equivalent to `dbg=2*3*5*7*11` for `\eval`;
- `dbg=2` displays the formula after multi-token variables have been converted to their single token form, `\nmc_a`, `\nmc_b`, etc.;

- `dbg=3` displays the vv-list after multi-token variables have been converted to their single token form;
- `dbg=5` displays the stored variables and their values *after* evaluation (`dbg=3` lists the values as expressions);
- `dbg=7` displays the formula after it has been fp-ified but before it has been fed to `l3fp` to evaluate;
- `dbg=11` displays the LaTeX form of the final display; it will contain, *inter alia*, the numerical result.

8.4.2 Display settings

key	type	meaning	default	initial
`f`	int (0/1)	show/hide formula		
`p`	token(s)	concluding punctuation	`,`	
`pp`	token(s)	multi-formula inter-result punctuation	`,`	
`env`	token(s)	math environment	see Table 5.3	
`arg`	token(s)	arg. for `-at`, `array` envs	see Table 5.3	
`eq`	token(s)	relation symbol	see Table 5.3	
`vv`	token(s)	vv-list specification	see Table 5.3	
`sep`	token(s)	separator between multi-formula results	see Table 5.3	
`\}`	token(s)	right bracket for inner math environments	\ `\}`	
`vvi`	deprecated; use `vv`			
`vvd`	deprecated; use `vv`			

8.4.3 Environment settings

env	rem/arg	eq	vv	sep
$ \(math		=	,\mskip 12muplus 6muminus9mu(vv)	\quad
\[		=	,\mskip 36mu minus24mu(vv)	\]\[
displaymath equation equation*		=	,\mskip 36mu minus24mu(vv)	\end{*env*} \begin{*env*}
multline multline*	\eval in multline	=	,\mskip 36mu minus24mu(vv)	\hfill\\
multline multline*	multline in \eval	=	,\\(vv)	\end{*env*} \begin{*env*}
eqnarray eqnarray*		&=&	,\mskip 36mu minus24mu(vv)	\\
align align* aligned		&=	,\mskip 36mu minus24mu(vv)	\\
flalign flalign*		&=	,&(vv)	\\
gather gather* gathered		&=&	,\mskip 12muplus 6muminus9mu(vv)	\\
alignat alignat* alignedat	2	&=\;&	,\qquad&(vv)	\\
array	rcrl	&=&	,&(vv)	\\
cases dcases		=	,\quad\hfill(vv)	\\

8.4.4 Settings for supplementary commands

In principle the settings

dbg, view, ^, xx, ff, 1s2, /min, /max, vv@, o, log, S+, S?, P+, P?

are available for \nmcInfo, \nmcMacros, \nmcConstants, \nmcReuse but most will be of little relevance in most cases.

- \nmcInfo

  - `view` equivalent to `dbg=2`;

- `\nmcMacros`
  - `view` equivalent to `dbg=2*3*5`;
  - `free` 'deregister' a macro from `numerica`;

- `\nmcConstants`
  - `view` equivalent to `dbg=2*3*5`;
  - `add` add the new list of constants to the current one;

- `\nmcReuse`
  - `view` equivalent to `dbg=3`;
  - `save` the control sequence formed from the supplied name to the `.nmc` file with the numerical result from the latest `\eval` command and define it in LaTeX;
  - `renew` if necessary overwrite the value of a control sequence in the `.nmc` file and redefine it in LaTeX.
  - `load` load the saved control sequences and define them globally in LaTeX;
  - `delete` remove the listed control sequence and value from the `.nmc` file and undefine it in LaTeX;