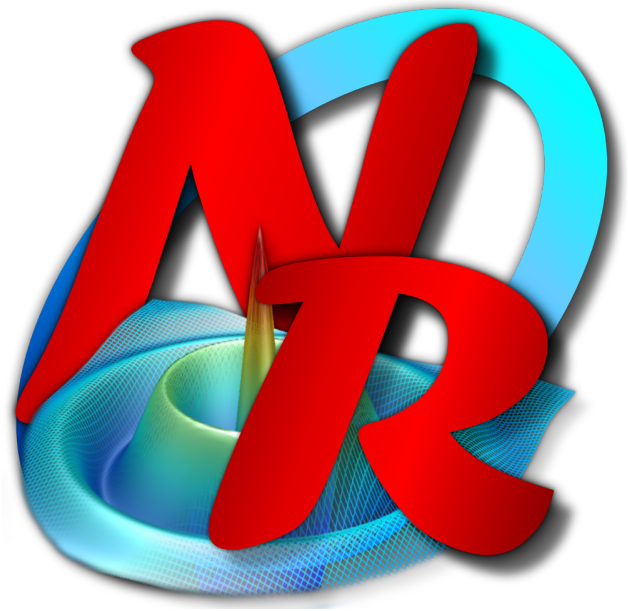
Documentation

NumeRe: Framework für Numerische Rechnungen

1.1.x-Series
Free numerical software released under the GNU GPL v3

Erik Hänel et al.

January 22, 2018



For the friends of science

CONTENTS

LEGAL INFORMATION

GNU Free Documentation Licence v1.2

— Licence of this document —

GNU General Public Licence v3

— Licence of the application and the displayed code fragments —

Team

- **Project lead:** Erik HÄNEL
- **Concept/UI:** Erik HÄNEL, Chameleon Team
- **Mathematical parser:** Ingo BERG, Erik HÄNEL (muParser)
- **Plotting:** Alexey BALAKIN (MathGL)
- **Numerical algorithms:** GNU Scientific Library, Erik HÄNEL, Alexey BALAKIN

- **Tokenizer:** Boost-Library
- **Matrix algorithms:** Eigen-Library
- **XML parser:** Lee THOMASON (TinyXML-2)
- **Excel-(97-2003)-Functionality:** YAP CHUN Wei (BasicExcel)
- **Testing:** C. ALONSO, D. BAMMERT, J. HÄNEL, R. HUTT, K. KILGUS, E. KLOSTER, K. KURZ, M. LÖCHNER, L. SAHINOVĆ, D. SCHMID, V. SEHRA, G. STADELMANN, R. WANNER, A. WINKLER, F. WUNDER, J. ZINSSER

INSTALLATION

Since the version v1.0.7 »Bose« NUMERE is released as an installer on SourceForge. To install NUMERE on your own computer, just download the installer and execute it. After following the corresponding steps, NUMERE will be installed on your machine and may be executed afterwards.

To install a newer version of NUMERE just install the newer version of NUMERE ontop of the previous one. Do *not* uninstall the previous version, otherwise all your settings will get lost.

Note The installer nominates »C:/Software/NumeRe« as default installing path. This path may be changed to your needs, however you should avoid using the standard »C:/Program Files/...« or »C:/Program Files (x86)/...«, respectively. NUMERE probably cannot be executed correctly at these locations.

System Preconditions

NUMERE may be executed on all desktop version of MS-Windows from Windows XP to Windows 10. For Windows 8.1 and 10 the additional component compatibility has to be installed. Otherwise there are possible crashes during the plotting process.

NUMERE also requires a keyboard for correct execution. The on-screen keyboard may be used as well, though resulting in quite uncomfortable interaction.

Portable and Stable Version

As default the installation of the stable version is recommended, which will create the file links for NUMERE. This requires admin permissions. The portable version may be executed without any admin permissions, however it won't create any file links. NUMERE-Portable may be started out of any directory, moved to an arbitrary location and simply can be deleted if it shall be uninstalled.

Note The »Stable Version« installer only writes the file links and the link to the uninstaller into the windows registry. No configuration values or other relevant data for execution are written to the registry.

Part I.

BASIC USAGE

1. First Steps

Most beginnings are not really easy. This is quite obvious, therefore we gathered all relevant first things, which shall make the beginning of the work with NumeRe as easy as possible. But this documentation doesn't claim that it might by complete. Not all of the possible options and their values are described here. Their description may be found in the integrated documentation.

Note This documentation is supported with marginal texts, which shall point to actions in NumeRe. Marginal texts printed in red are syntax elements, which may be entered into the NumeRe console. Bordered and dark blue marginal texts name articles in the integrated NumeRe documentation.

1.1. The User Interface

The user interface of NumeRe is separated in five central elements. Most important of them is the *console* (bottom center) and the *editor* (top right). Using the console one may interact directly with NumeRe and the editor may be used to edit text files, NumeRe scripts and NumeRe procedures. The *input history* (bottom right) is an additional element, which protocols all inputs in the console, so that they may be repeated by dragging them to the console or by double clicking in them. The *file tree* (left sidebar, first tab) and the *symbol tree* (left sidebar, second tab) support the navigation in the files or the commands and functions, respectively. The editor itself also contains tabs, so that multiple files may be opened and edited.

During the first start the NumeRe editor displays a starting page, which shall make the first task in the application more easy. Users, who are already knowing similar software (such as MATLAB), will get familiar to NumeRe with these information really fast. However, we will add some first words.

First, we'll look at the interaction over the console. There's an arrow <- at the beginning of the line, which emphasises that an input is expected (Figure 1.1). The program's output is prefixed with an arrow, pointing to the opposite direction ->. You may *only* enter something, if such an input arrow (or an explicit claim) appears.

You may enter mathematical-numerical expressions as well as commands into the NumeRe console. The syntax of the mathematical-numerical expressions is quite intuitive and will be described in the next section. The syntax of the commands has to be described more precisely.

We tried to create NumeRe as intuitive as possible. This target should apply to the command syntax as well, but because NumeRe is a long-term and somehow »grown« application, we could

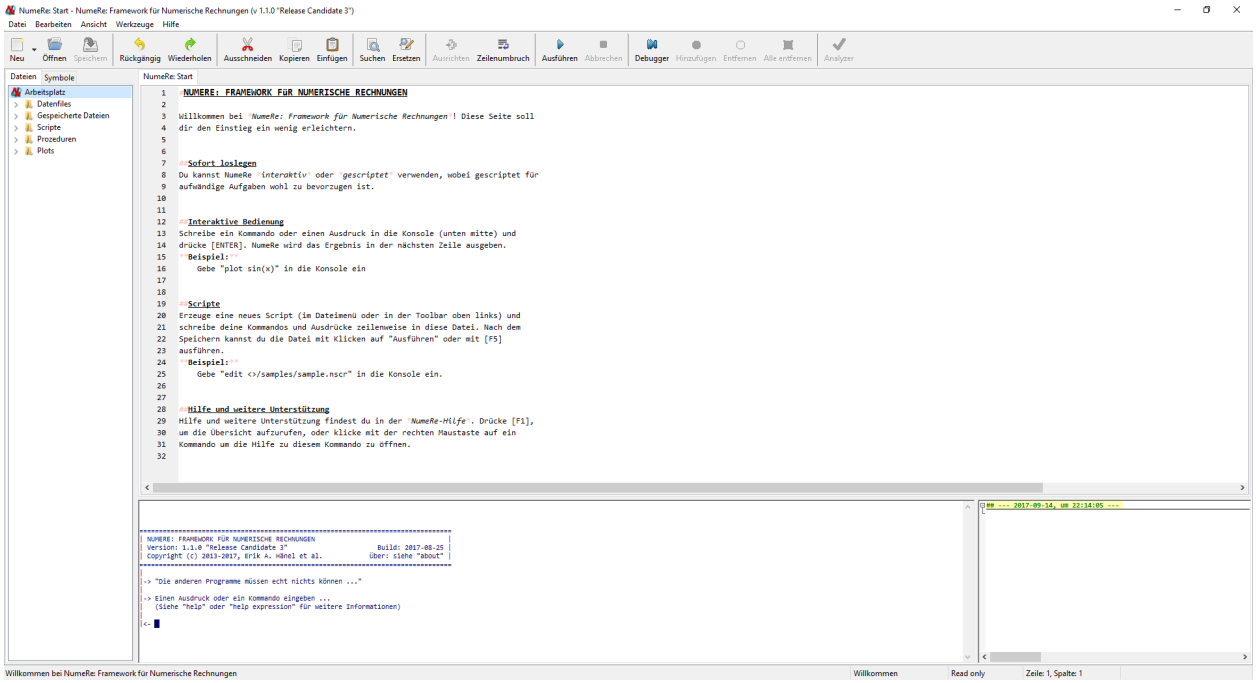


Figure 1.1.: The starting view of NUMERE with the described interface elements. The starting page of NUMERE, which describes some basics, is also visible. *Note: All screenshots shown in this document are in german language but if you've installed the english version, the interface language will be English, of course*

not fulfill the target everywhere. However, all commands are following the same scheme:

```
COMMAND [EXPRESSION] [-PARAMETER[=VALUE] [OPTIONS[=VALUE]]]
```

Syntax elements in brackets are sometimes optional or for some commands not present, respectively. Examples for the command syntax are

```
help
help expression
plot sin(x)
mesh sin(x)+cos(y) -set box
copy <>/samples/data* -target=<loadpath>/*
```

You'll recognize that (some) commands are not requiring an expression or a parameter.

This command syntax is illustrated in the following sentence:

> *»Apply an action [on something] by using the follwing parameters.«*

Applied to one of the upper examples, you can possibly say

> *»Open the documentation article, which has »expression« as topic.«*

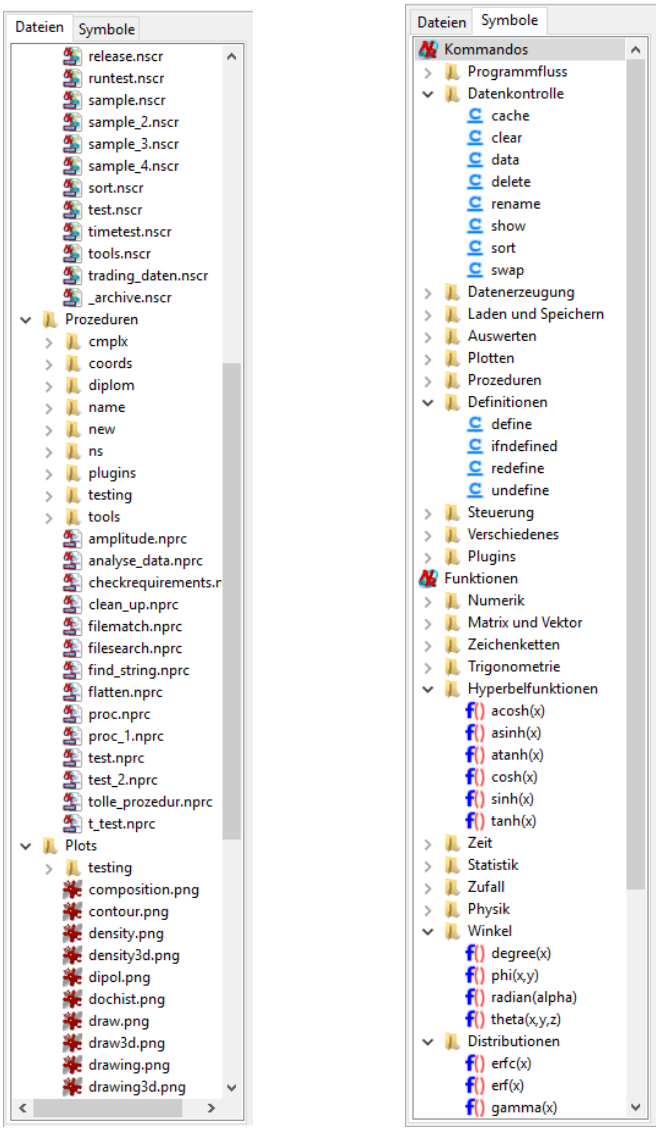


Figure 1.2.: File and symbol tree in the NUMERE interface

or

»Create a meshgrid plot of the function »sin(x)+cos(y)« by using the surrounding box.«

syntax ▶ **In the NUMERE documentation at:** `help syntax`

All commands are listed in the symbol tree and they are separated into different categories, so that the needed command may be found easily. If you dwell your mouse over a command, a tooltip will be displayed showing a short explanation. If this information is not enough, you may display
Syntax element help the documentation article concerning the command, e.g. through

```
1 help plot
```

All parameters and syntax information for correct execution are listed in these articles, as well as an example of the syntax.

numere ▶ **In the NUMERE documentation at:** `help numere`

If an entry is probably erroneous, NUMERE will display an corresponding message in the console. If a mathematical-numerical expression is erroneous, the position of the error—if possible—will be displayed as well. If the error results from the command syntax, NUMERE will display also a reference to the corresponding article in the NUMERE documentation.

Errors in expressions or commands will abort all calculations. As a consequence all NUMERE scripts and NUMERE procedures will be aborted with an corresponding message as well.

1.2. Preferences

All preferences for NUMERE are set in the options dialogue, which may be found in the tools menu. Some settings may additionally be changed with the command `set`. The desired setting
Syntax element set and its new value have to follow this command:

```
set -SETTING=VALUE
```

If the value of the setting shall contain whitespaces (e.g. a file path), it has to be passed surrounded with quotation marks:

```
1 set -SETTING="VALUE WITH WHITESPACES"
```

Syntax element The command `list` may display a list of all preferences:

```
list -settings
```

Changed settings are saved at application termination and available again after a restart.

1.3. Simple Calculations

As a framework for numerical calculations, NUMERE may of course calculate numerically. The equations, which shall be evaluated, may be entered similar to a pocket calculator. Spaces between

operators and values don't play any role, but there are *no spaces* allowed between function names and their argument parentheses. Lower- and uppercase letters are of course different and the multiplication dot * mustn't be omitted:

```
5*cos(_2pi)
```

multiplies cos 2π with 5 and returns the result in the next line. The value `_2pi` is a built-in constant for 2π and `cos()` is of course the cosine function.

The built-in constants and functions may be found in the symbol tree. They may be found as well by entering

```
list -const
list -func
```

into the console. `list -func` may also be restricted further.

▶ **In the NUMERE documentation at:** `help list` — `list`

NUMERE cannot only handle numbers but also variables. The variables `x`, `y`, `z`, `t` and `ans` are already predefined. However, you may define more variables for the current session. This is done either automatically (if NUMERE stumbles upon an unknown variable) or through an explicit assignment:

```
neue_variable = 5
```

(*This variable's name is german for staying consistent with the screenshots.*) This line declares the new variable `neue_variable` and assigns the value 5 to it. The upper equation may now be written as follows:

```
neue_variable*cos(_2pi)
```

NUMERE may as well evaluate multiple expressions simultaneous. The expressions have to be separated by a comma `,`. In the whole (multiple) expression line, NUMERE will evaluate the values from left to right. The line

```
neue_variable = 5, neue_variable*cos(_2pi), neue_variable = 1
```

assigns 5 to `neue_variable`, calculates 5 cos 2π and finally assigns 1 to `neue_variable` (Figure 1.3). This way of writing a calculation may be speed up the evaluation inside of loops significantly.

▶ **In the NUMERE documentation at:** `help expression` — `expression`

1.4. Important Commands

You will interact with NUMERE mainly with commands. The most important commands are

Syntax element
help
find
list
quit

```
help
help TOPIC
find TERMS
list -OBJECT
```

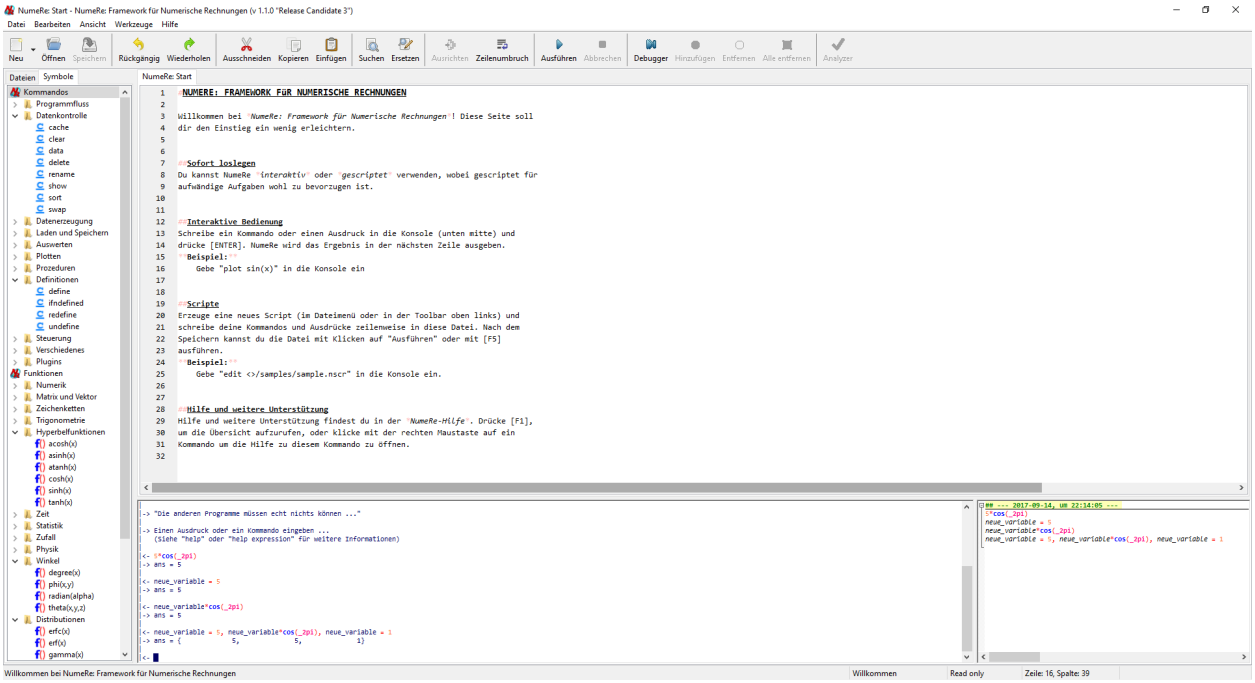


Figure 1.3.: NUMERE after the evalution of the last calculation. You may notice that the input is as well stored in the entry history

These commands call the integrated documentation (`help` or `help TOPIC` for the desired topic, respectively), the keyword search (`find TERMS`) and list objects (`list -OBJECT`).

All commands have an entry in the NUMERE documentation. This article may be read by entering `help COMMAND` into the console or by using the corresponding item in the context menu and contains all necessary information about the command and—if applicable—references to further important articles. In the case that an information may not be found directly in the NUMERE documentation, you may use the keyword search (`find`), which will also return the corresponding articles in the NUMERE documentation.

Note By using a semicolon »;« you may also enter multiple commands and expressions, which shall be evaluated successively, in a single line:

```
COMMAND1; COMMAND2; EXPRESSION1; COMMAND3; EXPRESSION2; ...
```

2. Loading and Using Data

One of the main tasks in scientific work is data analysis. Therefore it is necessary to use applications, which may load and process the corresponding large amount of data in a reasonable time frame. NumeRe transforms all read data into a table and may calculate statistics, histograms and more elaborate evaluations.

2.1. File Types

In addition to simple text files in ANSI format (using the file extensions *.txt and *.dat) NumeRe can load the following file formats:

- **CassyLab file (*.labx).** This file type is only used by CassyLab and contains a complete CassyLab experiment. However, NumeRe will only extract the data table.
- **CSV file (*.csv).** Comma-Separated-Values files are a nearly-standard for data exchange. However, there is no real »CSV standard« defined, so that though tested it is possible that NumeRe cannot read single CSV files correctly. (Please notify the programmer if you face such a problem and forward him the file.)
- **JCAMP-DX file (*.dx, *.jdx and *.jcm).** JCAMP-DX files (meaning *Joint Committee on Atomic and Molecular Physical data – Data eXchange*) are a standard for file exchanges of spectroscopy data. NumeRe will only extract the data table out of these files.
- **OpenDocument spreadsheet (*.ods).** These are data tables, which are created for example by OpenOffice Calc. NumeRe can only extract the real numerical and string values but not the equations. Please note also that files opened by OpenOffice Calc are locked: these spreadsheets are not readable by NumeRe.
- **Excel workbooks (*.xls and *.xlsx).** These are data tables, which are created by Excel. NumeRe will only extract the numerical and string values, not the equations. The legacy Excel format also doesn't provide the values calculated by the equations in the data tables, so that these are not available in this case. Please note also that files opened with Excel are locked: these workbooks are not readable by NumeRe.
- **IGOR Binary Waves (*.ibw).** IGOR Binary Waves contain the data of a (1, 2 or 3 dimensional) IGOR wave.

- **NumeRe data file (*.ndat).** NumeRe data files are a data format by NUMERE, which is used for fast saving and loading of data sets.

load ▶ **In the NUMERE documentation at:** `help load`

However, NUMERE may not write all of these formats. The following output formats are available:

- **Text file (*.dat or *.txt).** NUMERE creates a text file in ANSI encoding, which is following the formatting standards in the next section. Most of the other applications should be able to read these files
- **NumeRe data file (*.ndat).** This file format can only be read by NUMERE. However, this file format is documented online and may implemented in other programs.
- **CSV file (*.csv).** NUMERE creates a CSV file, where commas , are used as column and dots . are used as decimal separators. (Some table calculations, especially those, which are using the comma as decimal separator, probably cannot read these files: if you replace the commas with semicolons ; and the dots with commas, this problem should be fixed. The programmer does not understand this behavior.)
- **Excel Workbook (*.xls).** NUMERE may write the data in the legacy Excel-(97-2003)-Format (*.xls) for further processing in Excel and similar applications.
- **TEX file (*.tex).** NUMERE writes a table using the TEX standard. The comments in this file contain the preconditions for including the table in a TEX document (*booktabs* and *longtable* Packages).

save ▶ **In the NUMERE documentation at:** `help save`

2.2. Formatting of Text Files

Although NUMERE uses the dot . in the console as decimal separator, it is not necessary for files. NUMERE may even read files, which are using the comma and the dot mixed. All commas are transformed to dots internally.

However, text files have to be formatted as a table. It is not relevant, if the columns of this file are separated using whitespaces, tabulators or a mixture of both.

If you have text inside of the data table, it will be ignored. If you want to provide a column header, you have to put these headers in the last line before the actual data. Whitespaces in a single column header have to be replaced with underscores _. If you want to provide some comments in this file, prefix that with a # at the beginning of the line (Column headers are found even it they are prefixed with a #). You may also use a separating line between the column headers and the data table made out of =.

```
# COMMENT
# COMMENT
# HEAD_1  HEAD_2  HEAD_3 [...]
# ======================[...]
   0,225      12       0 [...]
   0,245    12.5      .5 [...]
```

▶ **In the NUMERE documentation at:** `help data`

data

2.3. Loading and Using of Files

You may load the file in the previously defined formats, if you drag and drop them on the console, use the corresponding item in the context menu of the file tree and through

Syntax element
load

```
load FILEPATH/FILE.EXT
```

File paths and file names containing whitespaces have to be surrounded with quotation marks. The exemplary `FILEPATH` may be omitted, if the file is located in the default directory `<loadpath>` (which is—as default—the folder »data« in the NUMERE root directory). If the file extension `.EXT` is omitted, NUMERE will use the first file matching to `FILE` and determine the filetype by itself. There is further information on this topic in the NUMERE documentation at `help load`.

There are some example files in the subdirectory »samples« in a default NUMERE installation. You may load them for example through

```
load <>/samples/data
```

The symbol <> is a path placeholder, which points to the NUMERE root directory. This line loads the file »data.dat« to NUMERE's memory. The data is now available as a table in the data object `data()` and may be used (It is a coincidence that the file and the data object carry the same name. The data of loaded files is always stored to `data()`).

You may display the data table with the command `show`:

Syntax element
show

```
show data()
```

This will display the table in a separate window.

You may access the data in `data()` using the so-called *interval syntax* `a:b` (for values ranging from `a` to `b`). This is used to define the column and row ranges, from where the data should be taken. You pass these indices inside of the argument parentheses of `data()`:

Syntax element
data()

```
data(3,1)
data(:,1)
data(4:55,4)
data(3,3:)
```

In these examples a single element is selected (`data(3,1)` for third row and first column), a whole column (`data(:,1)` for all rows in the first column), a subselection of a column (`data(4:55,4)` for

fourth to 55th row and fourth column) or a subsection of a row (`data(3,3:)` for third row and third to last column).

You only may extract values using columns or rows inside of calculations, i.e. the interval syntax may only be used for rows or columns. It is not possible to perform calculations using subtables. Instead, you may use the matrix operations out of a later chapter to achieve this behaviour. However, you may use different elements of `data()` in a single expression:

```
(cos(data(:,1)) + sin(data(:,2))) * sqrt(data(1,3:))
```

This expression will return as many elements as the longest interval nominates. There are also functions, which can handle an arbitrary number of elements and calculate a single result:

```
1 avg(), cmp(), cnt(), max(), med(), min(), norm(), num(), prd(), std(), sum(),
    to_char()
```

These functions calculate statistics of the elements (average, median, standard deviation, minimal and maximal value), summarize or multiply their arguments, search for elements, count elements or transform the values to ASCII character codes (the usage of strings is explained in the part »Advanced Usage« and is not relevant at this point).

Note Users, who are already familiar with Matlab or Octave, will know the difference between `OPERATOR` and `.OPERATOR`. This doesn't exist in NUMERE, because NUMERE was completely designed as a table calculation. Elements from different columns or lines will always be processed element by element and not following a matrix or vector algebra (a matrix-matrix or a matrix-vector multiplication is provided by the framework in the context of the `matop` command using the ** operator).

Some commands, like `fit`, `fft` and `plot` (and similar) can handle whole subtables. In the context of this commands you may use the interval syntax in both, columns and rows.

The table in `data()` is a so-called *read-only* data set. This means that the data in this table may not be overwritten or modified in other means. In a later chapter we will explain the caches as tables, which may be manipulated.

`data` ▶ **In the NUMERE documentation at:** `help data`

2.4. Data Analysis

The data analysis is probably the main reason, why someone should search for a fast and simple numerics application. NUMERE provides many predefined functions for analysis, which may be used fast and mostly uncomplicated. The needs for the most common statistics should be fulfilled satisfyingly.

Syntax element

You may either calculate the statistics all at once with the command `stats`

`stats`

```
stats data(i1:i2,j1:j2)
```

or single with the statistical functions, which where mentioned in the previous section:

```
std(data(i1:i2,j1))
avg(data(i1:i2,j1))
[...]
```

The command stats returns nearly all reasonable statistical value of the table, which is for example the average, the standard deviation and the standard error, RMS, skewness, excess and the student factor (for a two-fold 95 % confidence interval). Most of them can also be calculated through single functions.

▶ **In the NUMERE documentation at:** `help stats`

stats

A very common way of analyzing a data set is generating a histogram. Histograms are a way to visualize the frequency of a measured quantity. This may either be the period of an oscillation, the life time of elementary particles, the distribution of measured length values or another measured quantity.

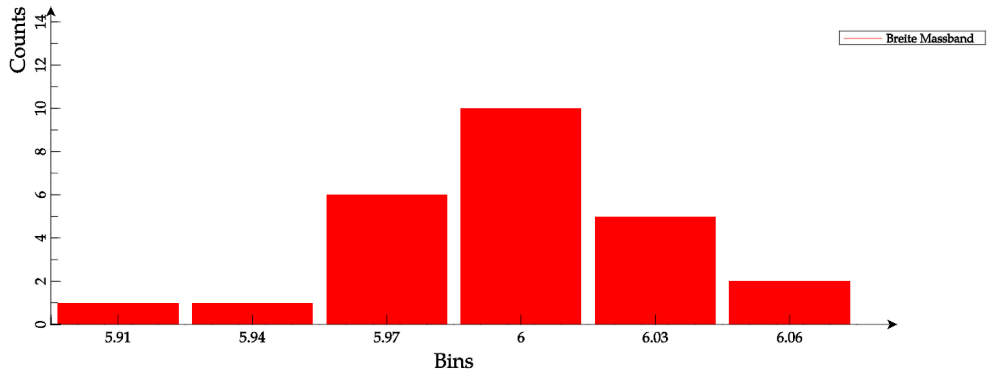


Figure 2.1.: Histogram of a data set, which was created using hist

Histograms of the loaded data are created using

Syntax element
hist

```
hist data(i1:i2,j1:j2)
```

The command hist supports a large number of parameters, which may modify the created histogram even further, though the results without any parameters are already good (Figure 2.1). The number of used bins is determined by the *Rule of Sturges*, but may also be set directly or based upon the *Rule of Scott* or the *Rule of Freedman and Diaconis*.

A histogram is calculated for each column of the data set and the results are presented in a common graph. Columns, which don't contain any reasonable data, have to be excluded using the interval syntax.

NUMERE may also calculate histograms out of a two-dimensional data set. You either have to pass the option `grid` to hist, then a single histogram of only the z values ignoring their spatial information is calculated. Or you can use the command hist2d for histograms in two spatial

Syntax element
hist2d

directions (like projections along the x and y axes). Further details are listed in the corresponding documentation article.

hist ▶ **In the NUMERE documentation at:** `help hist`

In addition to histograms and statistics it's very common to fit a function (sometimes called *model* or *regression curve*) to the measured (and already somehow processed) data.

Syntax element

To fit a function `FUNCTION()` to data in `data()`, you can use the command `fit`:

fit

```
fit data(:,j1:j2) -with=FUNCTION(x,PARAMS) params=[PARAMS=INITIALW.]
fit data(:,1:2) -with=A*sin(B*x+C) params=[A=1,B=3,C=0]
```

NUMERE will now try to alter the parameters `PARAMS` in a way that the `FUNCTION()` matches the data points best. The fitted values are stored in the parameters so that you can continue your calculations with these values directly. (In newer NUMERE versions it is not necessary to provide the parameters through `params`. NUMERE will detect them on its own.)

If NUMERE reaches an optimal set of parameters, it will cancel the algorithm and return an overview of the parameters, which is presented as an example in the following:

```
Function: 0.646875*sin(3.00223*x+0.00837336)
Data points:                              101 without weighting factors
Degrees of freedom:                       98
Parameters for the algorithm:             TOL=0.0001, MAXITER=500
Iterations:                               7
Weighted sum of the residuals (chi^2):    2.33825
Variance of the residuals (red. chi^2):   0.0238597
Standard deviation of the residuals:      0.1544658

Parameter  Initial value        Fitted        Asymptotic standard error
-------------------------------------------------------------------------
A                      1     0.6468754    ± 0.02188576          (3.383%)
B                      3      3.002228   ± 0.005649476         (0.1882%)
C                      0   0.008373364    ± 0.06579646          (785.8%)
-------------------------------------------------------------------------

Correlation matrix of the fitted parameters:

/          1     0.0159    -0.0164 \
|     0.0159          1     -0.862 |
\    -0.0164     -0.862          1 /

Fitting result analysis:
The fitted function could describe the trend of the data points, but
there is some room for optimisation.
```

This overview contains the important quantity χ^2, which contains the sum of the quadratic de-

viations of the data points to the fitted function. In addition, you'll find the result values of the parameters and the calculated error values, which may inserted in the concluding error progression. The correlation matrix of the parameters contains an information, how the parameters are connected to each other and the fitting result analysis is a summary of what can be learned from the value of χ^2. This overview will also be stored in the fit log file at `<savepath>/numerefit.log`.

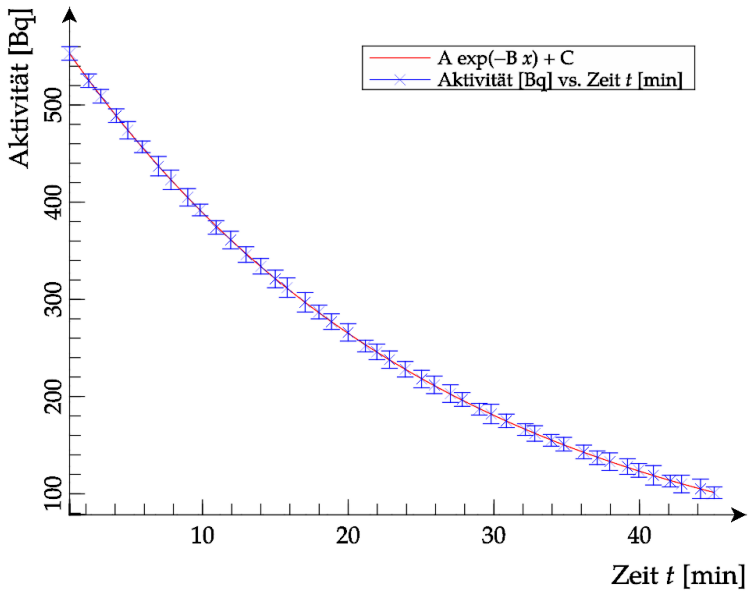


Figure 2.2.: Fitting a model for an exponential decay with `fitw`

Note If the quantity χ^2 is an indicator on the quality of the fit, can be discussed. In common, one assumes that a fit with the smallest possible χ^2 value has the best possible parameter set. More precise: it's the so-called reduced χ^2 value, which is the important quantity: if it is noticeable smaller than 1, the resulting parameter set seems to fit the data points quite well. If you perform a fit, which will consider the error estimations (see below), this value should be in the range of 1. NUMERE will consider all these conditions and return a corresponding fitting result analysis. But note that the analysis may only support and not fully replace the optical comparation of fitted function and data points.

After the fitting procedure, the fitted function may be displayed graphically using the command `plot` (see next chapter) (Figure 2.2). This is a good possibility to check the results of the fitting procedure (and should always be performed).

If you can estimate the measurement errors, NUMERE may consider these values. Just use

Syntax element
`fitw`

```
fitw data(:,1:2:3) -with=A*sin(B*x+C) params=[A=1,B=1,C=0]
```

fitw (for *weighted fit*) uses the error estimations in the third column to calculate the weighting factors for the data points: data points with higher errors are considered less important than those with small errors.

If a fitting procedure does not converge, it may help varying the initial values of the parameters (which always shall be passed). It also may help to restrict the fitting interval. This may be done either through restricting the rows in `data()` or through passing the option `x=a:b`:

```
fit data(:,1:2) -with=A*sin(B*x+C) params=[A=1,B=1,C=0] x=0:4
```

In addition the maximal number of iterations, the precision and further restrictions of the parameters can be set. The last option may have a large influence on the stability of the algorithm.

fit ▶ **In the NUMERE documentation at:** `help fit`

All mentioned commands are acting along the columns. If the data is organized in rows, one has to perform the following line

Syntax element

copy

```
1 copy data(:,:) -target=cache(:,:) transpose
```

and replace `data` with `cache` in all previous examples.

3. Creating Graphs

A very important functionality of NumeRe is the creation of graphs of functions and data. You may visualize the behaviour of functions and data and analyze them more easy. NumeRe can store all created graphs in image files, if you specify that in the options list of the graph.

3.1. Types of Graphs

NumeRe knows a large number of different plot types, which may be modified further by passing further options to the options list. The following list shall only mention the most important plot types. You can get the full list by entering the command `list -cmd`:

```
plot
plot3d
mesh
dens
vect
vect3d
...
```

- `plot`: this creates a standard graph, which displays y against x. This command is being used, if you want to visualize $\sin x$ or x^2.
- `plot3d`: this command creates a graph on the basis of a 3-dimensional trajectory. A trajectory is calculated from three functions sharing the variable t and visualized three-dimensionally.
- `mesh`: a function $z = f(x,y)$ may be displayed with this command. A meshgrid is calculated and displayed three-dimensionally.
- `dens`: this command visualizes the function $z = f(x,y)$ in contrast to `mesh` only through colour values projected to the x-y plane.
- `vect`: this plotting style calculates a vector plot of a 2D vector field: $\vec{A}(x,y) = A_x(x,y)\,\hat{e}_x + A_y(x,y)\,\hat{e}_y$.
- `vect3d`: this calculates a vector plot similar to `vect`, but it uses a three-dimensional vector field as data basis: $\vec{A}(x,y,z) = A_x(x,y,z)\,\hat{e}_x + A_y(x,y,z)\,\hat{e}_y + A_z(x,y,z)\,\hat{e}_z$.

3.2. Output

The default output channel for plots is the NUMERE GraphViewer, in which the plots may be modified further. Additionally, the plots may be stored into an image file in the PNG format in the directory <plotpath> (This is per default the subdirectory »plots« in the NUMERE root). You may change the image file format to other formats. This may be done with the plotoptions opng, oeps, ogif, osvg and otex.

To create a simple plot in for example »graph_of_the_measurement.png«, you've to pass the option

```
opng=graph_of_the_measurement
```

If the file name of the target file shall contain whitespaces, you've to pass it in enclosing quotation marks.

plotoptions

▶ **In the NUMERE documentation at:** help plotoptions

3.3. Usage

The syntax of all plotting commands is following this scheme:

```
1 COMMAND FUNCTIONS/DATA -set OPTIONS
```

where OPTIONS are optional and may be omitted.

Default variables are x, y, z and t. Depending on the actual plotting command, one, two, three or all four of these variables are used as plotting variables and the others are parameters. plot uses only x, plot3d only t, mesh, dens and vect x and y and vect3d uses x, y and z. All other variables are parameters.

Syntax element

A simple plot of a sine is created with

plot

```
plot sin(x)
```

This calculates a sine function from -10 to 10 automatically, where the y axis was chosen fitting but a small amount larger than the minimal and maximal value of the function. The axis labels and the legend are also determined automatically (Figure 3.1a)

plot

▶ **In the NUMERE documentation at:** help plot

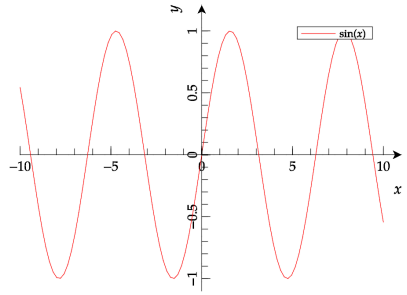
You are getting a surrounding box and a coordinate grid with

```
plot sin(x) -set box grid
```

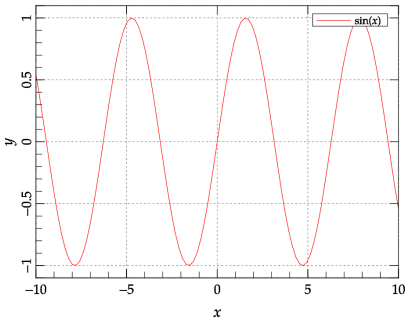
This and all further plots will also get a grid and a box (Figure 3.1b and c).

If you want to have »Sine function« instead of »sin(x)« as legend, you'll have to enter the following:

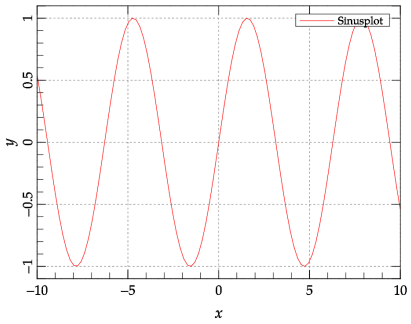
```
1 plot sin(x) "Sine function" -set box grid
```



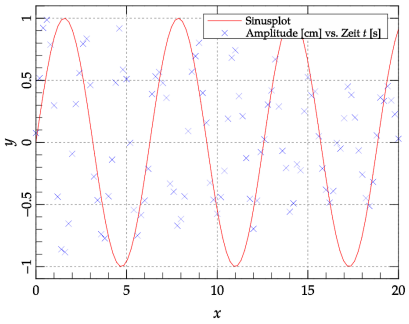
a: without options



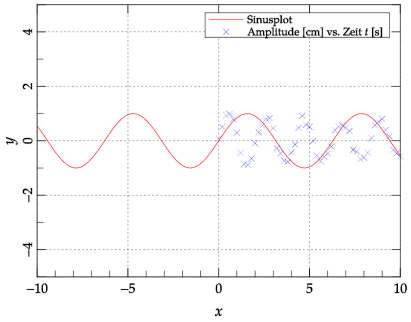
b: with `box` and `grid`



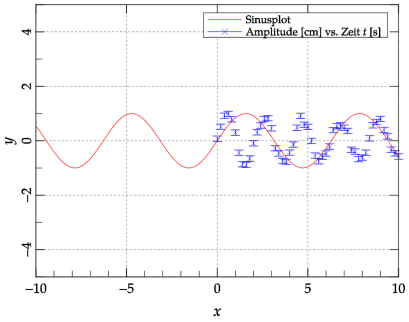
c: with custom legend



d: with data set



e: with custom interval



f: with errorbars

Figure 3.1.: The resulting graphs using the shown options

(Figure 3.1c; An empty legend is achieved by passing an empty string "")

NUMERE may also display data points graphically. To achieve this, one has to pass the data set in `data()` analogous to a function. If you leave the argument's parentheses empty, this will be replaced automatically with `data(:,:)`. You may also specify the desired columns by yourself, e.g. `data(:,1:4)`. NUMERE will use either the columns 1 and 4 or the columns 1 to 4, if the corresponding plotting style needs more than two columns. You may specify up to six columns in an arbitrary order: `data(:,4:2:6:1:3:8)`.

In combination with the previous example, one may visualize the columns 1 and 2 together with the sine function through

```
plot sin(x) "Sine function", data() -set box grid
```

(Figure 3.1d). Even for `data()` one may pass a custom legend. Otherwise NUMERE will create a combination of the columns titles. Additional it's noticable that adding the data set has scaled the x axis corresponding to the data set. Also noticeable is that NUMERE displays the data points as single points and doesn't connect them with a (non-physical) line.

The plotting intervals may be overwritten for a plot, if they are passed explicitly:

```
plot sin(x) "Sine function", data() -set box grid [-10:10,-5:5]
```

This creates a graph with the x interval $[-10;10]$ and the y interval $[-5;5]$ (Figure 3.1e). This option will *not* be used for successive plots.

Measurements of data points are often combined with measurement errors. If these are known, NUMERE may display them correspondingly. If only the y values have errors, NUMERE needs three columns $(x, y, \Delta y)$, if both directions have errors, NUMERE will need four $(x, y, \Delta x, \Delta y)$. The needed plotting option is `errorbars` or `yerrorbars`, if only y errors are available.

If wie assume that the data points in the upper example have errors in y direction, we may enter:

```
plot sin(x) "Sine function", data() -set box grid [-10:10,-5:5] yerrorbars
```

Errorbars in y direction are appearing, although the number of columns was not changed! NUMERE interprets the empty argument's parenthesis now automatically in another way and uses the first three columns (Figure 3.1f).

`plotoptions` ▶ **In the NUMERE documentation at:** `help plotoptions`

Syntax element `mesh`

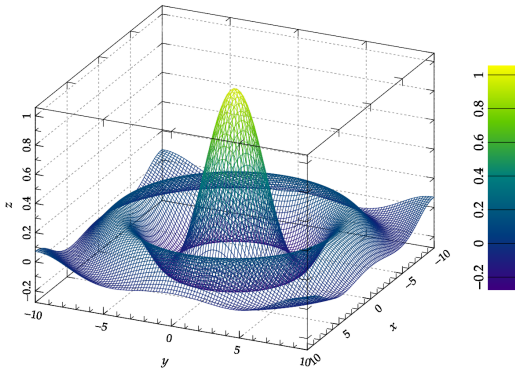
In another case a two-dimensional function shall be visualized using a meshgrid plot: the function is the cardinal sine of ϱ (sinc ϱ):

```
1 mesh sinc(norm(x,y))
```

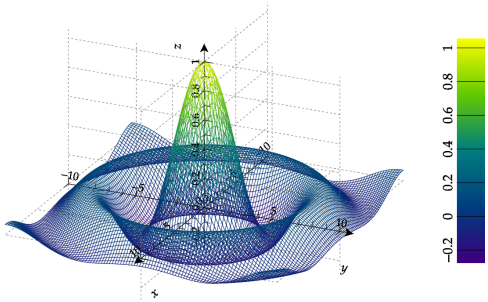
The command `mesh` creates the desired meshgrid plot (Figure 3.2a). The function `norm()` is the n-dimensional, euklidic vector norm (This function will accept an arbitrary number of arguments):

$$\text{norm}(x, y, z, \ldots) := \sqrt{x^2 + y^2 + z^2 + \ldots}$$

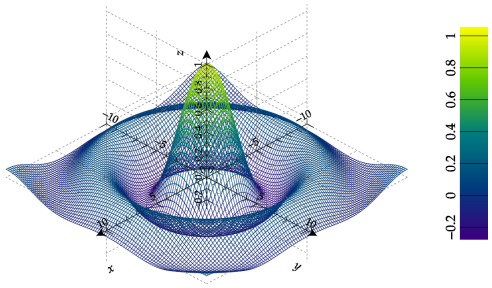
In this case $\text{norm}(x, y) = \varrho$. Because this was visualized directly after the previous plots in this chapter, you should notice that the meshgrid plot is also surrounded by a box and has a grid in the background. To remove the box, you may enter `nobox` (Figure 3.2b):



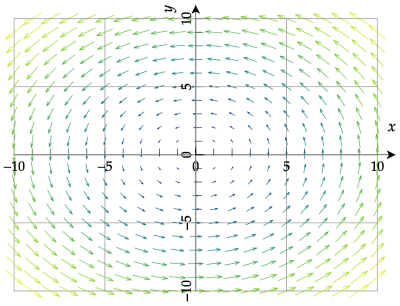
a: only `mesh`



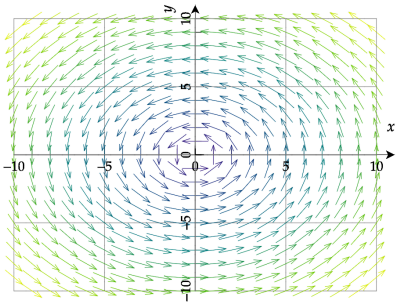
b: with `nobox`



c: with `rotate`



d: only `vect`



e: with `flength`

Figure 3.2.: Results of the meshgrid and vector plots

```
mesh sinc(norm(x,y)) -set nobox
```

The meshgrid plot is oriented automatically into a predefined direction. This may also be changed:

```
1 mesh sinc(norm(x,y)) -set rotate=45,135
```

The graph now appears now more tilted and the x and y axes seem to be symmetric to both sides (Figure 3.2c). You probably notice that the graph was oriented with these angular values into the direction of the first space diagonal. The angular values of `rotate` have to be passed in degrees in the order ϑ, φ, where ϑ tilts the graph and φ rotates it.

`mesh` ▶ **In the NUMERE documentation at:** `help mesh`

A further example is the vectorfield plot of a two-dimensional rotation field:

$$\vec{A}(x,y) = 2\begin{pmatrix} -y \\ x \end{pmatrix}$$

Syntax element

`vect`

This is achieved through

```
vect -2*y,2*x
```

The first function will be interpreted as the amplitude in $\hat{e}_x$ direction and the second in $\hat{e}_y$ direction (Figure 3.2d). This command will accept only one vectorfield per plot.

The length of the vector arrows correspond to the local amplitude of the vectorfield. To deactivate this effect, you may pass

```
1 vect -2*y,2*x -set flength
```

(Figure 3.2e).

`vect` ▶ **In the NUMERE documentation at:** `help vect`

3.4. Coordinate Systems

The last section of this chapter shall focus on the different coordinate systems. NUMERE supports three different coordinate systems: the *cartesian*, the *polar* or *cylindrical* and the *spherical* one.

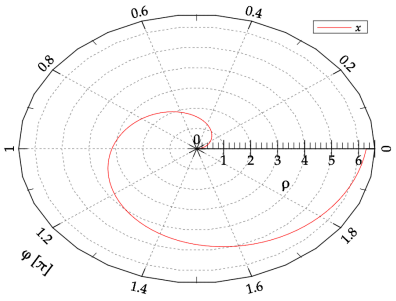
To change the coordinate system, one uses the option `coords=COORDINATES`. For example: to switch to polar coordinates, one passes the following line (Figure 3.3a):

```
plot x -set coords=polar
```

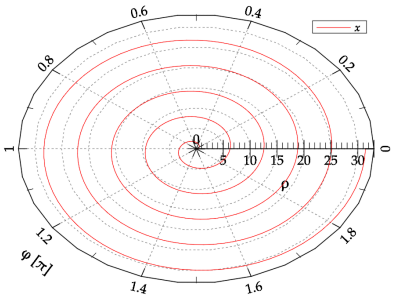
It's noticable that the variable x is used as φ and the functions value y as the radial coordinate ϱ. The azimuthal axis is displayed in units of π, but may be changed with the option `xscale=SCALE`, if desired.

If one changes the interval for x (meaning φ), one gets the result of Figure 3.3b with the following line:

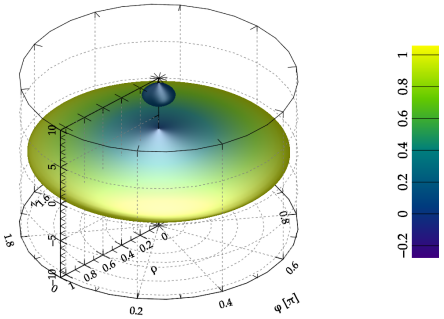
```
plot x -set coords=polar [0:10*_pi]
```



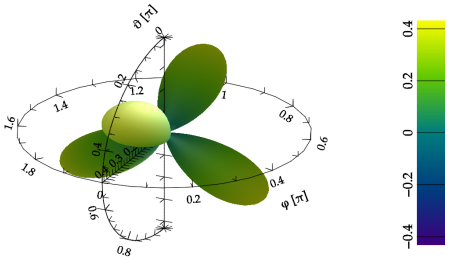
a: `plot` with `coords=polar`



b: with larger interval



c: `surf` with `light`



d: with `coords=spherical`

Figure 3.3.: Results of the different coordinate systems

It's possible an advantage, to enlarge the number of samples with `samples=WERT`.

In combination with three-dimensional plotting functions, one switches automatically into the cylindrical system, e.g. through

```
surf sinc(norm(y)) -set light
```

one gets Figure 3.3c. The added option `light` activates the three-dimensional lighting, which emphasises the volume of the plot.

It's getting obvious in this case that the variable y has been used as z and the function value $\Phi(x,y)$ has been interpreted as the radial variable ϱ.

Last but not least, the spherical coordinate system shall be presented here. One switches with

```
surf Y(3,2,y,x) -set coords=spherical
```

to this system and creates the neat Spherical Harmonics (its real part) $Y_{32}(\vartheta, \varphi)$ in Figure 3.3d. In this coordinate system y is used as ϑ, x is again φ and the function value $\Phi(x, y)$ is presented through the radial coordinate r.

Note Please note that the presented axis assignment may be changed. This can be done with the following options:

```
polar_pz
polar_rp
polar_rz
spherical_pt
spherical_rp
spherical_rt
```

coords ▶ **In the NUMERE documentation at:** `help coords`

4. TABLES: THE CACHE

NUMERE manages large datasets as tables. As previously mentioned, the default table for loaded data is `data()`. However, this table is *read only*. If you need a table, which you may modify to your needs, you've to use the so-called *caches*.

4.1. Concept

Caches are tables in NUMERE, whose content—similar to usual variables—may be modified freely. Their sizes are arbitrary and fit always to the needs of the user. As default, the default cache `cache()` already exists. You may also add additional caches.

Caches are autosaving their contents. If the content of a cache is altered, these changes are saved to an external file after the autosave interval is passed. The caches and their contents are automatically recreated after a restart so that you can continue your work immediately.

4.2. Creating and Removing Caches

You can use the command `new` to create new caches (this command may also create further objects):

Syntax element
new

```
new CACHE1(), CACHE2(), CACHE3(), ...
new amplitude(), phase(), results()
```

You may use the created caches just like the standard `cache()`. If one or more caches of the list are already existing, NUMERE won't change them in any way.

▶ **In the NUMERE documentation at:** `help new`

`new`

Custom created caches may also be removed, if their memory is not needed any more for further calculations:

Syntax element
remove

```
remove CACHE1(), CACHE2(), CACHE3(), ...
remove amplitude(), phase(), results()
```

However, the memory, which is occupied by NUMERE, is only freed after a restart.

▶ **In the NUMERE documentation at:** `help remove`

`remove`

You may also rename caches instead of removing them:

Syntax element
rename

```
rename -CACHE1=NEWNAME
rename -amplitude=phase
```

To remove all caches at once and free the complete memory, you may use

```
clear cache()
```

This will also free all the occupied memory of NUMERE. If you only want to delete the contents of a cache, just use

Syntax element

delete

```
delete CACHE(i1:i2,j1:j2)
```

cache

▶ **In the NUMERE documentation at:** `help cache`

4.3. Usage

You may use caches similar to the `data()` table. The central difference is the fact that the contents of caches may be altered. This means, you can also assign new values to their elements similar to the way, variables are handled:

Syntax element

cache()

```
CACHE(:,1) = VALUE1, VALUE2, VALUE3, ...
CACHE(4,5:) = data(:,1)
CACHE(:,1) = CACHE(2,:)
```

The new values replace the already available values in `CACHE()`.

At every occasion, where we used `data()` previously, you may also use caches with the identical syntax. Internally there's no difference (except of the possibility of changing elements) between `data()` and the caches:

```
hist cache(:,1)
stats cache()
fit cache(:,4:5) -with=A*sin(B*x+C) params=[A=1, B=1, C=0]
```

Data, which is created by some commands (e.g. datagrid), is sometimes written automatically to `cache()`. Also, some of the commands are creating their target caches by themselves.

4.4. Sorting, Smoothing and Resampling

Sometimes one has to modify the data points further so that one can process them reasonably. NUMERE offers the possibility to sort, smooth or resample (change the number of data points) the data.

The data in a cache (and in principle also in `data()`) may be sorted ascending or descending. As a consequence one might find distinct thresholds or points of interest in the measured data more easily.

Syntax element

Through entering

sort

```
sort -CACHE
```

the whole cache CACHE is sorted ascending. To sort it descending, one has to use the value desc

```
sort -CACHE=desc
```

Using the option cols then columns, which shall be sorted, may be selected. In addition one may select a group of columns, which shall be sorted according an index column (if, for example, x values and their function values are in an arbitrary order, the function values may be sorted according to the x values without losing the connection between the x values and the function values):

```
sort -CACHE cols=COLUMNS[COLUMNGROUPS]
sort -cache cols=5[2:4]
```

▶ **In the NUMERE documentation at:** `help cache` — cache

In addition to sorting the data, the data points in a cache may be freed from noise and similar artifacts. This is achieved through smoothing the data points.

The command

Syntax element **smooth**

```
smooth CACHE(i1:i2,j1:j2) -order=ORDER
```

smooths the cache CACHE in the selected range, where the data points are interpolated linearily over ORDER data points. NUMERE will smooth the data in two dimensions automatically, if this is reasonable according to the selected range. To switch this to column- or linewise smoothing, one may pass the options lines and cols.

Smoothing is in principle similar to applying a low-pass filter. If high-frequent data is of interest, smoothing will destroy this information partly or sometimes completely. As a test one might subtract the smoothed from the noisy data and plot the result. If only noise was removed, the plot should only display noise as well.

▶ **In the NUMERE documentation at:** `help smooth` — smooth

To alter the number of data points (e.g. if the sample rate is not fitting or one wants to combine different data rows) NUMERE may interpolare further samples into the data or remove them as well.

You'll achieve this with

Syntax element **resample**

```
resample CACHE(i1:i2,j1:j2) -samples=SAMPLES
```

The number of samples is altered columnwise to SAMPLES. If SAMPLES is larger than the selected range, no information is lost, because the new data points are interpolated from the previous ones. If the number is smaller, of course information is lost.

▶ **In the NUMERE documentation at:** `help resample` — resample

4.5. Statistics Functionalities

Caches (and the `data()` table as well) may execute global statistics operations on the data (similar to **stats** and statistics functions **std**(), **avg**(), ...). One has to use the caches as a command and append the desired statistics function as a parameter (there are no parentheses in the expression in this case):

```
CACHE -avg
CACHE -std
...
```

This calculates the average/standard deviation/etc. of the whole table as a single value. The additional options may be restricted further: the option `grid` calculates the statistics and considers the cache as a datagrid. The option `cols` calculates the values column- and `lines` linewise. The options `lines` and `cols` may be combined with `grid`.

5. NUMERE SCRIPTS

NUMERE scripts provide the possibility to outsource complex calculations and repeated analyses into a file, from where they may be easily and repeatedly executed.

5.1. Concept

The concept, which builds up the basis for NUMERE scripts, is quite easy: all entries, which one might enter into the NUMERE console, are typed into a text file instead, which will be read by NUMERE afterwards. NUMERE will execute the expressions and commands in the corresponding order and evaluate them.

The advantage is obvious: the order of commands may easily be repeated and errors may corrected fast without being forced to retype everything.

Note We will talk about NUMERE procedures in a later chapter. These represent the programmability of NUMERE. However, most problems, which do not need too complex abstractions, may be solved using NUMERE scripts.

▶ **In the NUMERE documentation at:** `help script` `script`

5.2. Syntax Highlighting

The syntax of NUMERE scripts is highlighted automatically, if the file was saved with the extension ».nscr«. It's a central part of the NUMERE editor. It provides highlighting of syntax elements, of matching parentheses and control flow blocks (e.g. `if` ... `else` ... `endif`).

Everybody is free to change the colours of the syntax highlighting to his or her needs. This may be done in the options dialogue, which is available in the tools menu. In the dialogue, go to in the tab »Style«.

5.3. A Simple Script

As example, a simple script is presented here.

To create a NUMERE script fast and easily, one either uses the corresponding item in the file menu, clicks on the tool in the toolbar or enters

```
new -script=first
```

into the NUMERE console. A NUMERE script with the name »first.nscr« is created in <scriptpath>, which is in the first two cases opened automatically in the editor. In the latter case the line

Syntax element
edit

```
edit first.nscr
```

opens the NUMERE script in the NUMERE editor so that it may be edited. (The file extension is here necessary, because otherwise edit would look for the script in the wrong directory.) Between the both text blocks in the script one adds the following lines:

```
## Delete the contents of the cache completely
delete cache() -ignore

## Create random numbers
random -lines=1e5 cols=2 distrib=uniform mean=0.5 width=1

## Are the number inside of the circle of unity?
cache(:,3) = (cache(:,1)^2+cache(:,2)^2) <= 1 ? true : false;

## Calculate pi
4*sum(cache(:,3))/1e5
```

where ## starts a line comment, which is ignored by NUMERE (Alternatively, one may use #*...*# as a block comment). The option ignore in the second line suppresses the confirmation, if one is sure that the contents of the cache shall be deleted. random creates random numbers and writes them into the cache. In this case, two sets of 100,000 equally distributed random numbers are created in the interval $[0;1]$.

Syntax element
random

The third expression is the trickiest. Here, the result of a condition is written to the third column of cache(). If written with words, it would mean something like

> *»If the length of the vector is smaller or equal to 1, then write »true«, otherwise write »false« to the cache.«*

The so-called *ternary* operator

```
CONDITION ? TRUE_VALUE : FALSE_VALUE
```

Syntax element
()? ... :
...

is an abbreviation for the if...else...endif fork, which is explained in a later chapter. The advantage of the *ternary* is that it may be executed faster but it cannot contain commands. A list of all possible logical expressions is obtained through

```
list -logic
```

At the end of this line one finds a semicolon ;. This suppresses the output of its result to the NUMERE console.

The whole NUMERE script is a real simple *Monte-Carlo* simulation. Points are placed arbitrary into a square of the length 1 and afterwards it's counted, how many of them are located in the circle of unity, which is part of this square. Assuming that the probability for each point in the square is equal, the relation of the number of all points to the number of points in the circle of unity has to be equal to $\pi/4$.

If one executes the NUMERE script by either clicking on »Execute« or through entering the command

Syntax element
start

```
start first
```

(the evaluation of random takes a moment), one gets results similar to

```
3.1364
3.13668
3.1454
3.13952
...
```

These numbers are quite near to π, although they are created through smart usage of random numbers.

If one enlarges the number of random numbers (e.g. `1e6` instead of `1e5`) the approximation gets even more precise. However, the cache doesn't support an arbitrary number of elements. For even larger numbers one has to use other ways of calculating.

Part II.

ADVANCED USAGE

6. Definition of Custom Functions

NumeRe already contains a large number of predefined functions (see `list -func`). However, it's sometimes useful and practicable, if one may define his own functions, which are e.g. combining already existing ones. NumeRe may store up to 100 custom defined functions.

6.1. Definition

Syntax element
define

A custom function is defined using

```
define my_function(ARGS) := EXPRESSION(ARGS)
```

This line creates the function `my_function()`, which combines the complex expression of the arguments. The names and the number of arguments may be chosen arbitrary, as long as the number of arguments is not larger than 10. If the last argument of the function is called »...«, one may pass from one to an arbitrary number of values for this argument. You have to ensure that the defined expression is able to handle an arbitrary number of values, e.g.: `my_function(x,...):= x*sum(...)`

Syntax element
"..."

The name of the function may be chosen freely, while ensuring that it neither starts with a number nor is identical to a (pre-)defined function.

Syntax element
redefine

With

```
redefine my_function(ARGS) := NEW_EXPRESSION(ARGS)
```

the function `my_function()` may be redefined. The number of arguments doesn't have to be identical to the previous definition, of course.

Custom defined functions may be supported by comments, which illustrate, what the purpose of the function is. This comment is displayed together with the definition in the function tree or at

```
list -define
```

The comment may be added to the definition with

```
1 define my_function(ARGS) := EXPRESSION(ARGS) -set comment="COMMENT"
```

or passed later with `redefine`.

▶ **In the NumeRe documentation at:** `help define`

define

To remove custom defined functions, you may use the command

Syntax element
undefine

```
undefine my_function()
```

However, the function storage is cleared at the end of the application automatically. If one wants to prevent this behavior, one may either enter

```
define -save
```

and the follwing line after the restart

```
define -load
```

to load the functions, or one activates the automatic definition management through

```
set -defcontrol=true
```

set ▶ **In the NUMERE documentation at:** `help set`

6.2. Conditioned Definition

In NUMERE scripts it may be an advantage, if NUMERE won't always raise an error that a function, which shall be defined in that script, is already existing. One possible solution for this case is using the command `redefine`, the other is using

Syntax element ifndefined

```
ifndefined my_function(ARGS) := ...
```

instead. The function is now only defined, if it isn't already available in the function storage.

6.3. Usage

A custom defined function may be used just like the predefined functions, because it is transformed in its definition internally before its execution. Therefore the calls to

```
my_function(1,2,3)
```

and

```
sin(1)+cos(2)+sinc(3)
```

are identical for the definition

```
define my_function(x,y,z) := sin(x)+cos(y)+sinc(z)
```

You may also pass a lesser number of values as the function has arguments. NUMERE will replace the missing ones automatically with `0`.

7. CHARACTER STRINGS

In addition to numerical values NUMERE may also handle character strings. First introduced to format the column titles of the tables, character strings are now a major and elaborate part or NUMERE's architecture.

7.1. Concept

Character strings (*strings* for short) are successive chains of characters, which are not interpreted as variables or numerical values by NUMERE. To achieve this, strings have to be entered with enclosing quotation marks:

```
"This is a string."
```

The actual content of the string are all characters between the quotation marks. As a consequence, the string `""` is empty and has the length 0.

Strings may be modified by special functions: there are functions for converting upper- to lowercase letters (or the other way around), for searching strings inside of strings, to extract a string from another string, to replace strings inside of another one, etc. The main advantage of character strings is the possibility, to format the output and to automate the processing of many files (and they are also a precondition for the programming with NUMERE procedures).

▶ **In the NUMERE documentation at:** `help string` `string`

7.2. Variable Type

The variable type for strings is the third variable type (next to numericals and tables) in NUMERE. Numerical variables may neither be converted to string variables nor the other way around. However, their values may (see below).

NUMERE recognizes new string variables automatically using the declaration. This declaration has to be—in contrast to numerical variables, which may be done *on-the-fly—always* followed by a string value (at least an empty string):

```
numerical_variable = 3.1415926
also_numerical
string = "Hello World!"
also_string = ""
```

A declaration with the return value of a string function is also possible.

Syntax element
string()

In addition to the string variables, NUMERE knows the string() object. This is in principle a single column table, which may contain an arbitrary number of strings. The interval syntax may be used in the argument parentheses to extract a range of strings or a single value. If the argument parentheses are kept empty, the last written string is used automatically.

string ▶ **In the NUMERE documentation at:** help string

Using strings one may modify the column headings of data() and the caches. This is achieved through entering

Syntax element
data(#,:)
cache(#,:)

```
data(#,1) = "Column heading 1"
cache(#,:) = "Column 1", "Column 2", "Column 3"
```

As you can see, you may use the interval syntax in this context, too. The hash sign # references the column headings.

cache ▶ **In the NUMERE documentation at:** help cache

7.3. Conversion

The value of a string variable may be transformed to a numerical value and the other way around. The content of a string will be interpreted as a new variable, as an expression or even as a command, if applicable. Two functions exist for this purpose:

```
to_value()
to_cmd()
```

The function to_value() converts the passed string to a mathematical-numerical expression and evaluates it correspondingly. to_cmd() transforms the string directly to a command expression.

Syntax element
#VAR
#(EXPRESSION)

The inverted conversion may be done in different ways:

```
#VAR
#(EXPRESSION)
valtostr(EXPR,C,N)
to_string()
string_cast()
```

The syntax #VAR or #(EXPRESSION) evaluates the following expression/variable and transforms the numerical value directly to a string. Between # and the expression one or more ~ may be inserted, which will add zeros in front of the value until the corresponding number of characters plus one for the # is reached. The function valtostr() is doing similar, although it's more versatile, because one may pass the filling character through the character C.The function to_string() transforms everything, which isn't a string, directly to a string without evaluating it and string_cast() transforms even string variable names into strings.

8. Loops and Forks as Control Flow Statements

The creation of NumeRe scripts (and the in one of the following chapters introduced NumeRe procedures) may be simplified drastically by introducing loops and forks. (However, loops and forks are also usable directly from the NumeRe console.)

8.1. Forks

A fork is a location in a script, where the further evaluation of the script depends on the evaluation of a condition. Such forks are represented through the following construct:

Syntax element
```
if ()
...
elseif ()
...
else
...
endif
```

```
if (CONDITION1)
    EXECUTE, IF TRUE
elseif (CONDITION2)
    EXECUTE, IF CONDITION1 IS FALSE AND CONDITION2 IS TRUE
else
    EXECUTE, IF ALL CONDITIONS ARE FALSE
endif
```

A fork has to be composed at least out of a `if ()` and a closing `endif`. In between an arbitrary number of `elseif ()` and at most one `else` as *fallback case* may be used, where the `else` case has to be the last case before the closing `endif`. A fork, which is only composed out of a `if ()` and a `endif`, will only be executed, if the condition evaluates to true, otherwise it is ignored completely.

The blocks between `if ()`, `endif` and the other keywords may contain an arbitrary number of commands and expressions. In addition, these blocks may contain further loops and forks.

▶ **In the NumeRe documentation at:** `help if`

`if`

8.2. Conditioned Loops

A conditioned loop will only be executed, as long as the condition evaluates to true. The syntax is as follows

Syntax element
```
while ()
...
endwhile
```

```
while (CONDITION)
    EXECUTE, AS LONG AS TRUE
endwhile
```

In the contained block commands and/or expressions or even further loops and forks may be used.

while ▶ **In the NUMERE documentation at:** `help while`

8.3. Counting Loops

The execution of a counting loop depends on the value of an index variable. Using the interval syntax, a starting and an ending value have to be passed. If the ending value is *smaller* than the starting value, the counting loop will count backwards automatically. After each single loop pass the index value is increased (or decreased, if the loop counts backwards) by one automatically. The index can be used in the execution block of the loop as a usual variable.

Syntax element
for ()
...
endfor

The syntax of a counting loop is as follows:

```
1 for (INDEX = START:END)
      EXECUTE, AS LONG AS INDEX IS IN [START;END]
3 endfor
```

Similar to the other control flow statements, one may use commands, expressions and even further loops and forks in the execution block. After the termination of the counting loop, the index variable will be deleted automatically, if it didn't already exist before the loop.

for ▶ **In the NUMERE documentation at:** `help for`

8.4. Further Control Flow Statements

Syntax element
continue
break

You may influence the execution of a loop with the both commands

```
1 continue
  break
```

`continue` jumps over the remaining part of the current execution block and starts a new loop pass. The command `break` cancels the loop completely and jumps the surrounding execution block. If the surrounding execution block is outside of any loop or fork, the whole loop or fork is terminated. A reasonable application of these commands is using them in the execution block of a fork.

if ▶ **In the NUMERE documentation at:** `help if`

9. MATRIX OPERATIONS

NUMERE was created as a table calculation, because it's much more probable that one has to process measurement data instead of processing some matrix operations. However, NUMERE may also do so and evaluate matrix expressions.

9.1. Execution of Matrix Operations

Matrix operations can only be executed in the context of the **matop** or **mtrxop** command (those are synonymes). This command starts an expression, which shall be processed using matrix operations, where the actual matrices are realized as excerpts from caches or data files or special functions. But note that as default all evaluations are still executed elementwise (even the multiplication of two matrices).

Syntax element
matop

```
matop CACHE(i1:i2,j1:j2) * DATA(i1:i2,j1:j2) + CACHE(:,:) / ...
```

To process a matrix-matrix or matrix-vector multiplication, one has to use the ** operator. This operator has a higher priority than all other operators, so it might be necessary to use parentheses correspondingly. In addition it is important that the dimensions of the matrices are matching each other in the context of matrix multiplication.

Syntax element
... ** ...

```
matop CACHE() ** (DATA() * CACHE())
```

If one doesn't provide a target cache for **matop**, where the result may be stored, the cache `matrix()` is used automatically. In this case the contents of `matrix()` are overwritten completely.

▶ **In the NUMERE documentation at:** `help matop`

matop

9.2. Special Functions

Special or temporary matrices or advanced matrix operations may be done with the following functions, if they are used inside of the **matop** command.

- **cross**(MAT) calculates the n dimensional cross product (vector product) of the vectors, which form the $n-1$ columns of the matrix MAT.
- **det**(MAT) calculates the determinant of the matrix MAT, if MAT is a square matrix.

- **diag**(x,y,z,...) creates a diagonal matrix with the elements x,y,z,... as main diagonal.
- **diagonalize**(MAT) diagonalizes the square matrix MAT. If the calculated diagonal elements should be complex, then a $n \times 2n$ matrix will be returned with the real parts on the lower and the imaginary parts on the upper first diagonal.
- **eigenvals**(MAT) calculates the eigenvalues of the square matrix MAT and returns them in the shape of a vector. If the eigenvalues are complex, then they will be returned as a matrix with two columns, where the first contains the real and the second contains the imaginary part.
- **eigenvects**(MAT) calculates the eigenvectors of the square matrix MAT and returns them in the shape of a matrix, where each column is one eigenvector. If the eigenvectors are complex, then a $n \times 2n$ matrix will be returned with the real parts in the odd and the imaginary parts in the even columns.
- **identity**(n) creates a n dimensional identity matrix.
- **invert**(MAT) inverts the matrix MAT, if an inverse matrix exists.
- **matfc**(x,y,z,...) creates a matrix out of the columns x,y,z,.... If the number of elements is not sufficient for the maximal dimension, the missing ones will be replaced by 0.
- **matfcf**(x,y,z,...) creates a matrix out of the columns x,y,z,.... If the number of elements is not sufficient for the maximal dimension, the missing ones will be logically generated out of the already present ones.
- **matfl**(x,y,z,...) creates a matrix out of the lines x,y,z,.... If the number of elements is not sufficient for the maximal dimension, the missing ones will be replaced by 0.
- **matflf**(x,y,z,...) creates a matrix out of the lines x,y,z,.... If the number of elements is not sufficient for the maximal dimension, the missing ones will be logically generated out of the already present ones.
- **one**(n,m) creates a $n \times m$ matrix, which is filled with ones. If only one argument was passed, then a square matrix will be created.
- **solve**(MAT) solves the linear system of equations, which is described by the matrix MAT, with the Gaussian algorithm.
- **trace**(MAT) calculates the trace of the square matrix MAT.
- **transpose**(MAT) transposes the matrix MAT (column and line indices will be exchanged).
- **zero**(n,m) creates a $n \times m$ matrix, which is filled with zeroes. If only one argument was passed, then a square matrix will be created.

```
matop matfc({1,2,3},{4,5,6},{7,8,9})
/ 1   4   7 \
| 2   5   8 |
\ 3   6   9 /
matop zero(2,4)
/ 0   0   0   0 \
\ 0   0   0   0 /
```

10. Special Commands

This chapter shall focus on some special commands, which weren't mentioned up to here but building a great part of the functionality of NumeRe.

10.1. Roots

NumeRe may locate roots of functions and data sets using the command `zeroes`. In the case of data sets this command returns the indices of the roots (or of the location, which is nearest) or, if a data set for the x axis was passed, the corresponding x value:

Syntax element
zeroes

```
zeroes DATA()
zeroes DATA() -set x=XVALUES()
```

If the roots of a function or the intersection of two functions shall be searched, a searching interval for the x axis has to be specified:

```
zeroes f(x) -set x=x1:x2
```

Further options, which may be passed to `zeroes`, can be found in the corresponding documentation article.

▶ **In the NumeRe documentation at:** `help zeroes` `zeroes`

10.2. Extrema

The command `extrema` is working similar to `zeroes`. Using this command, NumeRe will locate the extrema of functions and data sets. In the case of functions, their x values are returned. In the case of data sets, their indices or the x values of the locations, which are describing the extrema, are returned.

Syntax element
extrema

```
extrema f(x) -set x=x1:x2
extrema DATA()
extrema DATA() -set x=XVALUES()
```

Note Extrema are—in contrast to roots—not easy to locate—at least, if the data has some noisy parts. However, this is true in most cases, therefore NumeRe only returns the indices or the x values of the minimal or maximal value instead of interpolating them. Saddle points are also not

always found. This is related to the numerical algorithm, which is sensitive to sign changes (which are not available at saddle points).

extrema ▶ **In the NUMERE documentation at:** `help extrema`

10.3. Integration

Syntax element **integrate**

NUMERE may numerically integrate functions and data sets using the command **integrate**, however, only functions may be integrated two-dimensionally. It is determined by the number of passed integration interval, if a 2D integration shall be calculated. If only an interval for x was passed, then NUMERE will calculate a one-dimensional integration. If the command string contains a second interval for y then the integration is performed in two dimensions.

The integration intervals and further options are passed using the parameter **-set**. Further options are for example the precision of the integration, the numerical method, whether the integration shall return the function values of the integral, etc. Further details are noted in the corresponding documentation article.

```
integrate x^2 -set [1:2]
```

If you want to integrate data sets, you've to pass them instead of the function: if the data set contains only one column, the integral is identical to the sum of this column. If two columns are passed, then the first is used as x and the second as the corresponding y values for the calculated integral.

```
1 integrate DATA(:,1)
  integrate DATA(:,1:3)
```

integrate ▶ **In the NUMERE documentation at:** `help integrate`

10.4. Differentiation

Syntax element **diff**

NUMERE does also provide the ability to differentiate functions numerically to the first order using the command **diff**. Depending on the passed parameters the differentiation is calculated at the location of x or for a number of `samples` over a complete interval.

```
  diff sin(x) -set x=1
2 diff sin(x) -set [0:1]
```

In addition, NUMERE may differentiate data sets numerically. If only one column is provided, then the distance between the samples is assumed to be 1. Otherwise, the first column is used as x and the second as the corresponding y values.

```
  diff DATA(:,1)
2 diff DATA(:,1:2)
```

Only the y values of the differentiation are calculated. Using the option `xvals` the corresponding x values (which are not matching to the provided ones) are calculated and returned.

▶ **In the NUMERE documentation at:** `help diff` `diff`

10.5. Taylor Expansion

Syntax element **taylor**

With the command **taylor** NUMERE may approximate functions of one variable numerically with a polynomial of the order $n \geq 0$ using the taylor expansion. However, this polynomial doesn't have to share more than one point (the expansion point) with the original function (this is an issue of the taylor expansion and not a numerical or algorithmic error).

The approximation is done only numerically. As a result, numerical errors are unavoidable. However, they are limited for low orders of the expansion. NUMERE will calculate numerical stable coefficients up to the order of $n = 10$ (passed through the option `n=ORDER`, default is 6). Over this limit the numerical errors are enormous and will lead to large deviations compared to an analytic determination.

```
taylor cos(x)*exp(-x/2) -set x=2
```

The calculated polynomial is automatically defined as a function in function memory. Usually, NUMERE will choose the name `Taylor(x)` for this definition, however, if the option `unique` was passed to the command **taylor**, then the used function name will be much more complex, because it will contain the expression and the order of the polynomial expansion.

Note Already existing functions, which are stored with an identical name in the function memory, are automatically overwritten by the new definition done by this command. Therefore, the option `unique` creates quite reliable function names, which won't interfere with other defined functions in function memory.

▶ **In the NUMERE documentation at:** `help taylor` `taylor`

10.6. Function Values in 1D and 2D

Syntax element **eval** **datagrid**

Using the commands **eval** and **datagrid** NUMERE will calculate and return the function values of one- or two-dimensional functions over a predefined interval and for a predefined number of samples.

The command **eval** returns the function values of one-dimensional functions in the predefined interval. The result may be stored directly into a cache. The number of samples is chosen using the option `samples` (where the default is 100) and they are linearly distributed by default. Through the option `logscale` this distribution is switched to a logarithmic one, which may be an advantage if one plans to display them on a logarithmic x axis.

```
eval FUNCTION(x) -set [x0:x1] OPTIONEN
eval sin(x) -set [0:_2pi] samples=200
```

eval ▶ **In the NUMERE documentation at:** `help eval`

NUMERE needs sometimes so-called *datagrids*. A datagrid is a tabular data set, where the x values are stored in the first, the y values are in the second and the z values are stored in the remaining columns, where the number of lines has to match to the first column and the number of columns has to match to the second column.

These datagrids may be created by `datagrid`. The values for x and y may be passed in different ways: either as an interval in the common form `[x0:x1,y0:y1]` or separate through `x=x0:x1` or as a column/row of a data set such as `y=data(:,3)`.

For the z values there are also multiple possibilities: either as a function of x and y ($f(x,y) = \cos(x)\exp(-y)$), as a matrix of a data set (`cache(3:,7:100)`) or as a single column/row of a dataset (`data(4,2:)`). NUMERE tries in the latter case to connect the defined (x,y,z) points by triangulation and to create a grid using linear interpolation.

```
datagrid z-VALUES -x=x-VALUES y=y-VALUES
datagrid data(:,3) -x=data(:,1) y=data(:,2)
```

The parameter `samples=SAMPLES` is optional und defines, how many samples NUMERE shall calculate if a component of the datagrid has to be calculated. As default (just like in 2D plots) NUMERE calculates 100×100 samples.

If the x and y axis of the data points are swapped (x = lines, y = columns), one can pass the parameter **`transpose`** to `datagrid` This way the datapoint matrix is transposed before the datagrid is constructed.

The created datagrid is automatically saved to a free location in the cache `grid()` (will be created automatically, if necessary) right of already existing data and may be plotted using this cache.

datagrid ▶ **In the NUMERE documentation at:** `help datagrid`

10.7. Fourier Transformation

One of the quite common evaluation algorithms in modern science is the calculation of a Fourier transformation, which may extract frequency-dependent information out of a total noisy signal.

NUMERE provides an algorithm for a fast Fourier transform, which is invoked with the command `fft` and may calculate the amplitudes of the contained frequencies of the passed data:

Syntax element

fft

```
fft DATA()
```

The passed data object has to contain at least two columns: the axis values in the first column (time or frequency) and the corresponding amplitude in the second one. If three columns are passed, then they are interpreted as axis values, amplitude and phase (in this order) or—if the additional option `-complex` was specified—as axis values, real and imaginary part of the amplitude.

`fft` will store the transformed data as new columns in the passed data set (or in `cache()`, if `data()` was passed). The values returned are frequency, amplitude (or *magnitude*) and phase. Using the option `-complex` the real and imaginary parts are returned instead of amplitude and phase.

An inverse transformation is achieved by passing the option `-inverse`.

▶ **In the NUMERE documentation at:** `help fft` `fft`

10.8. Differential Equations

Few of all possible ordinary differential equations may be solved analytical or through a reasonable approximation. This is the central topic of numerics, which solves the differential equations using numerical algorithms and calculate the corresponding trajectories.

NUMERE provides an integration algorithm with the command `odesolve`. This algorithm may numerically integrate differential equations of the first order. Because one may transform a differential equation of the n-th order into n equations of the first order, this is not a problem.

Syntax element
odesolve

The differential equation may be composed out of multiple equations and may form a whole system. The equations have to follow this scheme:

```
dy1/dx = f1(x,y1,y2,...)
dy2/dx = f2(x,y1,y2,...)
...,
```

where only the functions `f1()` to `fn()` (in this order) have to be passed. `x` is the integration variable and `y1` to `yn` are predefined function variables, in which NUMERE stores the results of the previous integration step. All other variables are considered as parameters.

Differential equations of the n-th order $DGL(x, y, y', y'', \ldots, y^{(n)})$ may always be transformed into n equations of the first order by introducing $n-1$ additional functions: $y' = dy_1/dx = y_2, y'' = dy_2/dx = y_3, \ldots$ Such a system follows this scheme:

```
dy1/dx = y2
dy2/dx = y3
...
dyn/dx = DGL(x,y1,y2,...,y(n-1))
```

The results of the integration are stored per default as a table in the cache `ode()`. The first column contains the x values and the following columns the corresponding integrated function values.

To pass initial values (NUMERE uses otherwise 0 as initial value), one may use the option `fx0=INITIALVALUES`. Additionally one may select the integration method, the tolerances, the number of samples and other things. Details may be found in the integrated documentation.

```
odesolve DGL(x,y1,y2,...) -set [x0:x1] OPTIONS
odesolve y2,-sin(y1) -set [0:20] fx0=[0,1]
```

Note NUMERE will provide the number of function variables corresponding to the number of equations: for one equation only `y1` is available, for two it's `y1` and `y2`, etc. If for some reason more

function variables are necessary (e.g. for a vector problem), one may pass additional 0-equations. However, the function variables are considered as constants in this case.

odesolve

► **In the NUMERE documentation at:** `help odesolve`

11. NumeRe Procedures

In addition to be written in NumeRe scripts, command sequences may also be outsources to NumeRe procedures. NumeRe procedures provide additional functionalities to solve problems more abstract. One example is the possibility to call NumeRe procedures recursively and further process their return values.

11.1. Concept

NumeRe procedures are in principle similar to a mixture of a custom defined function and a NumeRe script. In addition, NumeRe procedures may call themselves recursively (`define` would raise an error if one would try that) and evaluate more complex commands and expressions (Procedures are not restricted to expressions, which have to fit into one single line).

Using NumeRe procedures it's possible to write own subprograms in NumeRe or add further functionalities as new commands (this is known as *plugin*).

One may compare a NumeRe procedure in NumeRe with a function of a usual programming language.

▶ **In the NumeRe documentation at:** `help procedure`

`procedure`

11.2. Structure

The structure of NumeRe procedures has three main sections: the procedure head, the procedure body and the procdure tail. The head contains the name, the argument list and additional »flags«, the body contains the complete commands and expressions:

Syntax element
procedure
. . .
endprocedure

```
procedure $PROCEDURE_NAME(ARGLIST) :: FLAGS   ## Procedure head
    PROCEDURE BODY                            ## Commands and expressions
endprocedure                                  ## Procedure tail
```

To call a NumeRe procedure, one enters `$PROCEDURE_NAME(VARS)` in NumeRe scripts, other NumeRe procedures or the NumeRe console. The `VARS` should correspond to the number of arguments in `ARGLIST`. If `ARGLIST` contains default values for arguments, then the values for these arguments may be omitted in `VARS`. The default values for the corresponding arguiments are used in this case.

The mentioned »flags« influence the processing of a NUMERE procedure as a whole and suppress or allow a distinct behaviour of NUMERE in this procedure. However, a correct usage of flags requires the knowledge of the following sections and chapters:

Syntax element
explicit
private
inline
mask

```
explicit
private
inline
mask
```

The flag `explicit` suppresses the execution of plugins in this procedure, procedures with the flag `private` may only be called from the same namespace and procedures with the flag `inline` allow faster execution of surrounding loops, if only procedures of this type are used in the loops. However, `inline` procedures are quite restricted in their usage. The flag `mask` suppresses all system messages during the execution of this and all from here called procedures.

Flags may be used in an arbitrary combination/order, because they do not interfere with each other.

Each NUMERE procedure has to be written to its own *.nprc file, which carries the name of the procedure. Although this sounds difficult and may be a additional error source, it might be done completely by NUMERE through the corresponding menu item in the file menu, through clicking on the button of the toolbar or through

```
new -proc=$PROCEDURE_NAME
```

a NUMERE procedure with the name `$PROCEDURE_NAME` is created at the default procedure storing location `<procpath>/PROCEDURE_NAME.nprc`. By entering

```
edit $PROCEDURE_NAME
```

the procedure `$PROCEDURE_NAME` may be edited in the NUMERE editor.

The procedure body of a NUMERE procedure may be written just like a NUMERE script. The few differences are explained in the following sections.

11.3. Local and Global Variables

NUMERE procedures distinguish between local and global variables. Global variables are variables, which may be used from every procedure, every script and the NUMERE console. Local variables, in contrast, may *only* be used in *this* procedure and *only* in *this* recursion of this procedure.

Syntax element
var
str
tab

Local variable are declared with

```
var VARIABLES
str STRINGVARIABLES
tab CACHES
```

These commands may only be used once per procedure and declare a set of local numerical variables, string variables and local caches. These variables will be deleted automatically at the end of the procedure. (Local variables may be have an identical name to a global one. In the body of a NUMERE procedure the local variables have the higher priority. This behaviour is called *shadowing*.)

11.4. Return Values

NUMERE procedures return as default the value `true`, if the evaluation reaches the final line of a procedure: `endprocedure`. Additionally, NUMERE procedures may also return other values if

Syntax element
return

```
return VALUE
```

occurs at the corresponding line in the procedure. NUMERE will leave the procedure as soon as NUMERE reaches this command (It may be used multiple times in a procedure). `VALUE` may be either a numerical value or a string. Additionally one may use `true` and `false` explicitly as `VALUE`.

`VALUE` doesn't have to be one single value. You may also return multiple numerical values or multiple strings together. Even a combination of both variable types is possible, although another error source, because the calling procedure must be able to handle the returned set an values.

If procedures with multiple return values are used for `while` loops or `if` conditions, then only the first returned value is used for the logical operations.

The special value `void` is used to notify NUMERE that this procedure explicitly *doesn't* have a return value.

11.5. Namespaces

NUMERE can handle namespaces similar to C++, which allow to have multiple procedures with the same name but different behaviour in different namespaces. Procedures of other namespaces than the `main` namespaces are called through

```
$NAMESPACE~PROCEDURNAME(VARS)
```

The corresponding procedure `$PROCEDURENAME` is stored in the file

```
<procpath>/NAMESPACE/PROCEDURENAME.nscr
```

The command `new` may create procedures in namespaces automatically, if that is mentioned explicitly:

```
new -proc=$NAMESPACE~PROCEDURENAME
edit $NAMESPACE~PROCEDURENAME
```

If one uses the functionalities of the graphical user interface, this is also true.

If NUMERE procedures out of a specific namespace are called more often in one single procedure, one can predefine this namespace as a temporary default namespace by adding

Syntax element
namespace

```
namespace NAMESPACE
```

to the procedure's body. (Procedures out of other namespaces may be still be called, by naming their namespace explicitly.)

11.6. Exception Handling

NUMERE procedures provide a somehow rudimentary but effective way of handling exceptions. If an exception occurs, which can not be handled in any reasonable way in a NUMERE procedure

(e.g. a wrong input), then the evaluation of the procedure can be aborted immediately through the command

Syntax element

throw

```
throw
```

This command is passed through the whole stack until it reaches the default NUMERE console, where it produces a corresponding error message.

It's somehow more informative, if one passes an explicit error message. This is done as a string

```
throw "This is the error message."
```

which is finally displayed in the NUMERE console.

11.7. Debugging

A new NUMERE procedure will only in very few cases run without problems and be free of errors for the first time. Typos and logical errors are much too probable to avoid such kind of mistakes. During the execution of such an erroneous NUMERE procedure NUMERE will, if it's a syntax error, abort the evaluation at the position of the error. However, it won't always abort if there's an error in the calculations.

To locate and solve such errors, NUMERE provides the so-called NUMERE debugger (Figure 11.1). This functionality may be activated using the corresponding item in the Tools menu or through clicking on the button of the toolbar. If NUMERE faces a syntax error, it lists automatically the erroneous expression, the approximated line number, the erroneous module (the file), a stack trace and all local variables including their values of the current procedure. Using the stack trace one may reconstruct, in which procedure the error occured and which arguments were passed to this procedure.

In addition to the automatic listing at syntax errors the NUMERE debugger may provide similar information at a *breakpoint*. A temporary breakpoint may be set using the function of the toolbar or through clicking on the sidebar at the corresponding line. Please note that you *cannot* place temporary breakpoints at empty or comment-only lines.

Permanent breakpoints are set with |> at the beginning of a line, but they are seldom necessary. NUMERE stops, before the *current line* is evaluated and lists stack trace and variable values. Through clicking on »Continue« the evaluation is continued, through clicking on »Cancel« the whole evaluation is aborted. (Of course, breakpoints are ignored, if the debugger is not active.)

debugger

▶ **In the NUMERE documentation at:** `help debugger`

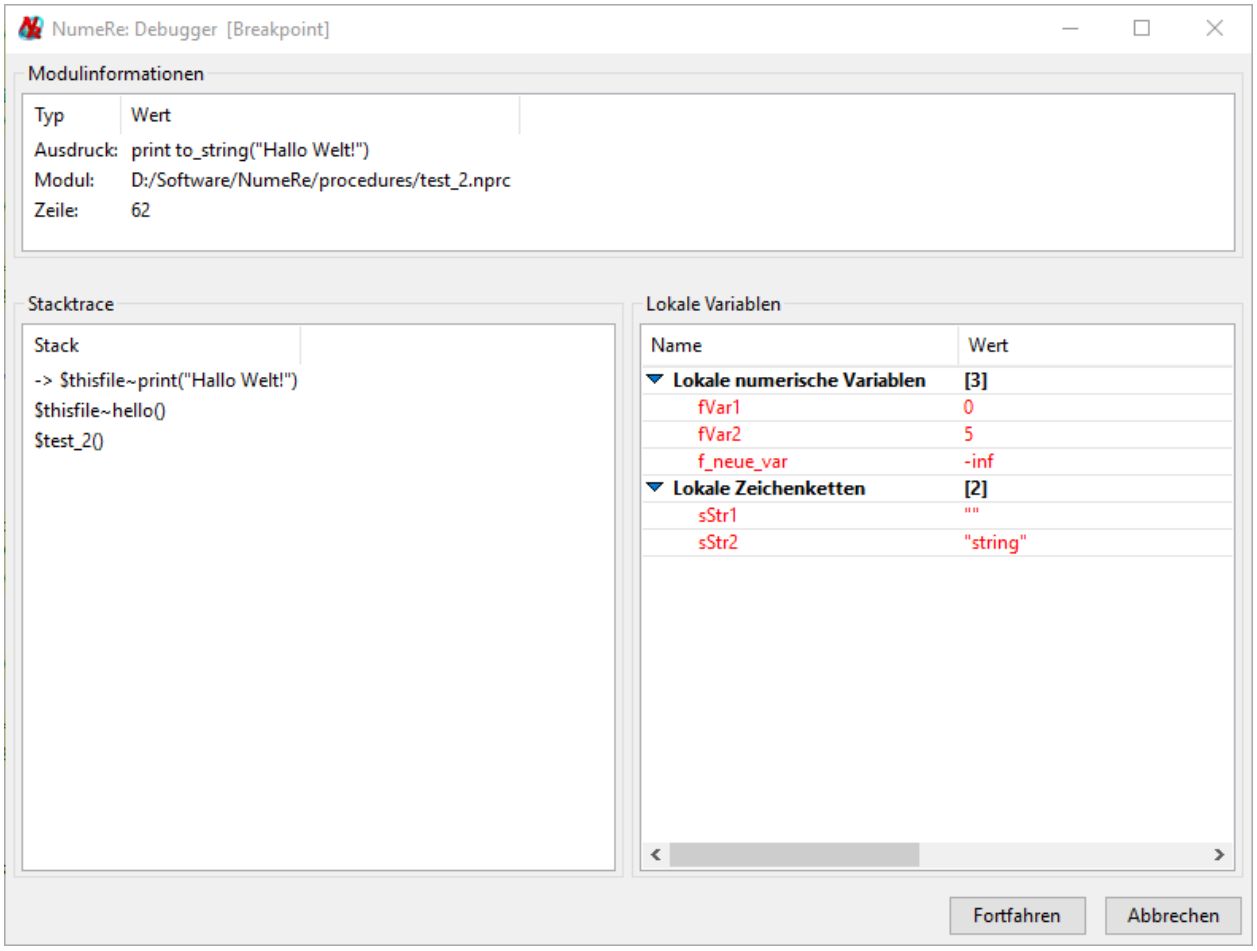


Figure 11.1.: The NUMERE debugger at a breakpoint. Changes to the previously evaluated breakpoint are highlighted in red

12. Animated Graphs

In some cases it's suitable that one may visualize the time dependency of a quantity using an animated graph. NumeRe has a built-in possibility to create such an animation. However, an animation may also be created using a loop, saving every frame independent and combine all frames to an animation afterwards.

To create an animation by NumeRe one has to use at least once the variable `t` in the expression, which shall be plotted. This variable is the time parameter, which will be varied by the animation. Additionally, one has to pass the option `animate` to the plotting command:

Syntax element
animate

```
PLOTCMD EXPRESSION(t,VARS) -set animate OPTIONS
```

This will create an animation, which is composed out of multiple frames. One can use the NumeRe GraphViewer to play this animation or examine each frame independently.

▶ **In the NumeRe documentation at:** `help plotoptions`

`plotoptions`

Note NumeRe cannot create animations from data sets. The reason is that the time parameter is a floating number, whereas indices for tables are integral values. To create an animation from a dataset, one has to create the frames independent using a loop.

NumeRe creates per default 50 frames per animation with a duration of $1/25$ s each and uses the interval $[0;1]$ for the time parameter `t`. This may be altered of course, however, it's possible that too many frames will result in memory problems, because every single frame is stored in memory first:

```
PLOTCMD EXPRESSION(t,VARS) -set t=0:2 animate=100 OPTIONS
```

Now an animation with 100 frames is created and the time parameter uses the interval $[0;2]$.

Note Some plot types may need too much memory so that an animation is not possible. It may help to reduce the number of `samples` for this image.

13. Composed Graphs

It's possible that a desired plotting layout is only possible, if one could combine two plotting types. Using the command block

Syntax element
compose
...
endcompose

```
compose
    PLOTTYPE1
    PLOTTYPE2
    ...
endcompose
```

it's possible to do that in NumeRe. The compose mode may only contain plotting commands. This means that all calculations for the desired plot types have to be finished in advance.

Some plot options are valid for the whole composed plot (e.g. the plotting interval, the axis labels, the grid, etc.), others may be switched on or off for each plot independent (e.g. transparency, lighting, additional contour lines, errorbars, etc.). The plotting colours and the line styles may also be selected for each plot, however, one has to keep in mind that NumeRe has an internal counter, which will be incremented for each line in all plots of the composed plot. The styling information has to be adapted to deal with this behaviour.

Using the compose mode one may display plots of different quantities together, e.g. a vector field together with its potential. It's not necessary to use a distinct order, NumeRe will order the plots by itself so that they are in a reasonable order:

```
compose
    dens 1/norm(x,y)
    vect x/norm(x,y)^3,y/norm(x,y)^3
endcompose
```

▶ **In the NumeRe documentation at:** `help compose` — compose

The compose mode also enables to work around the restriction of a single function per 3D plot type (except of plot3d), if the functions are passed as successive 3D plots in a compose environment.

This mode allows additionally, if the parameter `-multiplot` and the command subplot are used, to create multiple graphs next to each other but in the same frame. Further details are found in the corresponding documentation article.

▶ **In the NumeRe documentation at:** `help subplot` — subplot

14. PLUGINS

Although NUMERE is constantly extended, some functionalities might not get implemented or they do not correspond to the desires of the NUMERE user. This is the regime of the NUMERE plugins, which may overwrite existing functionalities or add new ones.

14.1. Functionality

Plugins are completely integrated into the command architecture so that one might not recognize from the outside that the routines currently executed are part of a plugin. They are called through own or already existing commands (replacing their functionalities) and use the usual NUMERE syntax.

Additionally, one might add a documentation article to the documentation index. So the description of the plugin is included and may be called in the usual way.

14.2. Installation

Plugins are published as installation routines in NUMERE scripts, which contain

Syntax element
<install>
. . .
<endinstall>

```
<install>
    DECLARATIONS AND PROCEDURES
<endinstall>
```

To install such a plugin one has to enter

Syntax element
install

```
install SCRIPTNAME
```

into the NUMERE console (if the script is not located in <scriptpath>, one has to add the path, too). NUMERE will execute the installation routine, create the corresponding procedures and generate the needed connections. It's *not* possible, to install a plugin with the functionalities of the graphical user interface.

▶ **In the NUMERE documentation at:** `help install` install

After the script was finished, the plugin is installed and may be used. Now it should be listed in the symbols tree or at

```
list -plugins
```

and may be executed with the displayed command. The shown description should contain a brief information of the functionalities of the plugin.

Syntax element

To uninstall a plugin afterwards, one enters

uninstall

```
1 uninstall PLUGINNAME
```

into the NUMERE console. The PLUGINNAME is displayed at

```
list -plugins
```

in brackets next to the description. In some cases it might be necessary to enclose PLUGINNAME in quotation marks.

Note It's not necessary to uninstall a plugin first before a newer version of itself might be installed. The installation may be executed directly over the already existing one. NUMERE will change the corresponding connections by itself.

14.3. Custom Plugins

The development of custom plugins may sound complicated first, but actually it's not so difficult. NUMERE aides by creating a template for the plugin installation routine. This template is created by using the functionalities of the graphical user interface or through entering

```
new -plugin=PLUGINNAME
```

This creates a plugin with the name PLUGINNAME as a NUMERE script at

```
1 <scriptpath>/plgn_PLUGINNAME.nscr
```

with the main procedure

```
1 $plugins~PLUGINNAME~main(<CMDSTRING>)
```

This plugin template contains already all necessary installation elements (including the documentation article) with the corresponding placeholders.

Plugins are composed out of one or more procedures, which represent the plugin's functionality. Together with the declaration of a plugin, a main procedure has to be named, which is called by NUMERE as soon as the plugin's command is entered. NUMERE then passes the command string in one or more of the following shapes to this procedure. The shapes (or »tags«) have to be named during the declaration of the plugin:

Syntax element

<CMDSTRING>
<EXPRESSION>
<PARAMSTRING>

```
1 <CMDSTRING>
  <EXPRESSION>
3 <PARAMSTRING>
```

- <CMDSTRING> passes the whole command line (including the plugin's command)
- <EXPRESSION> passes the expression, which may be found between the command and optional parameters

- <PARAMSTRING> passes the parameter set, which begins either at -set (if an <EXPRESSION> is available) or at the first - after the command

The procedures, which represent the plugin's functionality have to be copied to that script. Afterwards one has to update the installation information block:

Syntax element
<info>
...
<endinfo>

```
<install>
    <info>
        -author="AUTHORNAME"
        -version="VERSION"
        -type=TYPE_PLUGIN
        -flags=ENABLE_DEFAULTS
        -name="PLUGINNAME"
        -pluginmain=$PLUGINMAINPROCEDURE(<CMDSTRING>)
        -plugincommand="PLUGINCOMMAND"
        -plugindesc="DESCRIPTION"
    <endinfo>
    PROCEDURES
<endinstall>
```

The plugin may now installed with the command install (The values for type and flags are actually uppercase letters).

If the PLUGINCOMMAND is now entered into the NUMERE console, then NUMERE calls the procedure $PLUGINMAINPROCEDURE() and passes it the tag <CMDSTRING>. One may also pass other or further sub expressions of the command line, if they are named as arguments of the main procedure. It's not possible to pass other arguments to the main procedure.

If the plugin shall return a value, which one may process further, then the type of the plugin has to be changed to

```
-type=TYPE_PLUGIN_WITH_RETURN_VALUE
```

now it may be used as follows:

```
result = PLUGINCOMMAND
```

The version flag may be passed as

```
-version=<AUTO>
```

For each installation of the plugin the version number is incremented automatically. This is for example of great use during the development, if one wants to fix the version number to the number of changes.

Normally, the installed procedures are logged in the console, so that the user may see, what's happening. One may also suppress this output by setting

```
-flags=DISABLE_SCREEN_OUTPUT
```

instead ENABLE_DEFAULTS. Further flags are

```
ENABLE_FULL_LOGGING
ENABLE_FORCE_OVERRIDE
```

They are either logging the complete installation linewise to <>/install.log or they enable the overwriting of an already available plugin with the same command but a different author, which is prevented by default.

plugins

▶ **In the NUMERE documentation at:** help plugins

Before <endinstall> one has the possibility to add information to a documentation article for the NUMERE documentation using a XML-like syntax:

Syntax element
<helpindex>
...
</helpindex>
<helpfile>
...
</helpfile>

```
<install>
    ...
    PROCEDURES
    <helpindex>
        INDEX INFORMATION
    </helpindex>
    <helpfile>
        DOCUMENTATION ARTICLE
    </helpfile>
<endinstall>
```

The INDEX INFORMATION contain the keywords, for which NUMERE shall display the corresponding article, as well as information for the index itself, if one enters help idx.

The section DOCUMENTATION ARTICLE contains the actual documentation and description of the installed plugin.

documentation

▶ **In the NUMERE documentation at:** help documentation

```
#*
    ******************************************************************************

 * NUMERE-PROZEDUR: $analyseLogs()
 *
    ==============================================================================

 * Hinzugef\"ugt: 2017-12-21, um 21:37:30 *#

procedure $analyseLogs(_sBasePath = "D:/CPP/bosch_ebike/pc_app/app/",
    _sFileScheme = "T01")
    ## Reads the file list of the selected file scheme and hands it over to
        the logfile parser
    str sLogFilePath = _sBasePath + "/" + _sFileScheme + "*_EBIKE-PC-*_app.log
        ";
    print "Analyzing \"" + sLogFilePath + "\" ..."
    ## Clear memory and create a new empty table
    delete string(3) -ignore
    new logfiletable() -free
    ## Get the file list, which corresponds to the desired file scheme
    string(,3) = getfilelist(sLogFilePath, true);
```

```
    if (!num(string(:,3)))
        return void;
    endif

    ## walk through all files in the list
    for (i = 1:num(string(:, 3)))
        ## Hand the files over to the logfile parser
        if (!$thisfile~parseLogFile(string(i, 3)))
            print "WARNING: " + string(i, 3) + " could not be parsed"
        endif
    endfor -mask

    ## Return nothing
    return void;
endprocedure

#* Ende der Prozedur
 * NumeRe: Framework f r Numerische Rechnungen | Freie numerische Software
    unter der GNU GPL v3
 * https://sites.google.com/site/numereframework/
 ********************************************************************************************
    *#


procedure $parseLogFile(_sLogFilePath)
    ## Delete the contents of "string(2)"
    delete string(4) -ignore
    ## Create local variables
    var nLine, nCol, nLastTimeStamp = 0
    ## Read the current size of the log file table
    ## (lines should be incremented per file)
    matop {nLine, nCol} = size(logfiletable());
    ## Read the contents of the log file to memory (string(2))
    if (!findfile(_sLogFilePath))
        return false;
    endif
    string(:, 4) = read _sLogFilePath;

    ## Walk through every line and hand them over to the single lines parser
    for (i = 1:num(string(:, 4)))
        nLastTimeStamp = $thisfile~parseSingleEntry(string(i, 4), nLine+1,
           nLastTimeStamp);
    endfor -mask

    ## Return true (Error handling not implemented)
    return true;
endprocedure

```

```
procedure $parseSingleEntry(_sLogEntry, _nLine, _nLastTimeStamp)
    ## Create local variables
    str s1, s2, s3, s4, s5
    var nColumn1, nColumn2, nLastTimeStamp = _nLastTimeStamp

    ## Get the numbers of entities in the current line
    ## this are either 3 (TIME MESSAGE CLOCK)
    ## or 4 (TIME !! MESSAGE CLOCK)
    ## or 5 (TIME MESSAGE MESSAGE MESSAGE CLOCK)
    if (num(split(_sLogEntry, " ")) == 3)
        ## Case for three entities
        {s1, s2, s3} = split(_sLogEntry, " ");
        ## If the second entity is "!!" we have an error message
        if (s2 == "!!")
            ## Get fitting column
            {nColumn1, nColumn2} = $this~findColumn(s3);
            ## write to table
            $thisfile~saveToTable(s3, "true", _nLine, nColumn1, nColumn2, 0);
        else
            ## Get fitting column
            {nColumn1, nColumn2} = $this~findColumn(s2);
            ## write to table
            nLastTimeStamp = $thisfile~saveToTable(s2, s3, _nLine, nColumn1,
                nColumn2, nLastTimeStamp);
            ## Override the "CONNECION_INIT" 0 clock with the parsed absolute
                timestamp
            if (s2 == "CONNECTION_INIT")
                {s1, s2, s3} = split(s1.sub(s1.rfnd("_")+1), ":");
                logfiletable(_nLine, 1) = to_value(s1)*3600 + to_value(s2)*60
                    + to_value(s3);
            endif
        endif
    elseif (num(split(_sLogEntry, " ")) == 4)
        ## Case for four entities
        {s1, s2, s3, s4} = split(_sLogEntry, " ");
        ## Get fitting column
        {nColumn1, nColumn2} = $this~findColumn(s3);
        ## Write to table
        nLastTimeStamp = $thisfile~saveToTable(s3, s4, _nLine, nColumn1,
            nColumn2, nLastTimeStamp);
    elseif (num(split(_sLogEntry, " ")) == 5)
        ## Case for five entities
        {s1, s2, s3, s4, s5} = split(_sLogEntry, " ");
        ## Get fitting column
        {nColumn1, nColumn2} = $this~findColumn(s2);
        ## Write to table
        nLastTimeStamp = $thisfile~saveToTable(s2, s5, _nLine, nColumn1,
            nColumn2, nLastTimeStamp);
```

```
104     endif

106     ## Return the time stamp (i.e. the clock count) from this entry
        return nLastTimeStamp;
108 endprocedure

110
procedure $saveToTable(_sHeadLine, _sValue, _nLine, _nColumn1, _nColumn2,
    _nLastTimeStamp)
112     ## Writes the value at the correct location to the table
        ##
114     ## Create local variables
        var nLine, nCol, n1 = _nColumn1, n2 = _nColumn2, nLastTimeStamp = 0##
            _nLastTimeStamp
116     str sHead = _sHeadLine
        ## Read the current size of the log file table
118     matop {nLine, nCol} = size(logfiletable());

120     ## Exclude the case that the values is equal to "true"
        if (_sValue == "true" || _sValue == "false")
122         nLastTimeStamp = 0;
        endif
124
        if (is_nan(logfiletable(_nLine, _nColumn1)))
126         ## If the field for the first column is empty, we simply write the
                value here
            logfiletable(_nLine, _nColumn1) = to_value(_sValue) - nLastTimeStamp;
128     else
            ## If the field for the first one os alreay occupied by a value, we
                write it in
130         ## the second column.
            ## The index for the second one might probably also be empty, so we
                create a new
132         ## column with the corresponding heading
            if (is_nan(_nColumn2))
134             ## Create a new column with the passed headline
                logfiletable(#, nCol+1) = _sHeadLine;
136             logfiletable(_nLine, nCol+1) = to_value(_sValue) - nLastTimeStamp;
            else
138             logfiletable(_nLine, _nColumn2) = to_value(_sValue) -
                    nLastTimeStamp;
            endif
140     endif

142     ## Exclude the case that value is equal to "true"
        if (_sValue == "true" || _sValue == "false")
144         ## If the value is "true", then the timestamp didn't change for this
                message
```

```
        return _nLastTimeStamp;
146     else
        return to_value(_sValue);
148     endif

150 endprocedure

152 #*
TC = "T02_BCM", "T03_BCM_BHU", "T04_BCM_med", "T05_BCM_BHU", "T06_BCM_BHU", "
    T07_BCM_BHU_30_R1", "T08_BCM_BHU_30", "T09_BCMJapan", "T10_BCMJapan", "
    TL01_BCMJapan", "TL02_BCMJapan", "TL03_BCM_BHU", "TL04_BCM_BHU", "TL06_BCM
    "
154 1   "T02_BCM"
    2   "T03_BCM_BHU"
156 3   "T04_BCM_med"
    4   "T05_BCM_BHU"
158 5   "T06_BCM_BHU"
    6   "T07_BCM_BHU_30_R1"
160 7   "T08_BCM_BHU_30"
    8   "T09_BCMJapan"
162 9   "T10_BCMJapan"
    10  "TL01_BCMJapan"
164 11  "TL02_BCMJapan"
    12  "TL03_BCM_BHU"
166 13  "TL04_BCM_BHU"
    14  "TL06_BCM"
168
*
```