

Documentation

NUMERE: FRAMEWORK FÜR NUMERISCHE RECHNUNGEN

1.1.x-Series

Free numerical software released under the GNU GPL v3

Erik Hänel et al.

November 9, 2017



For the friends of science

CONTENTS

Legal Information	9
GNU Free Documentation Licence v1.2	9
GNU General Public Licence v3	9
Team	9
Installation	11
System Preconditions	11
Portable and Stable Version	11
I. Basic Usage	13
1. First Steps	15
1.1. The User Interface	15
1.2. Preferences	18
1.3. Simple Calculations	19
1.4. Important Commands	20
2. Loading and Using Data	21
2.1. File Types	21
2.2. Formatting of Text Files	22
2.3. Loading and Using of Files	23
2.4. Data Analysis	24
3. Creating Graphs	29
3.1. Types of Graphs	29
3.2. Output	30
3.3. Usage	30
3.4. Coordinate Systems	34
4. Tables: the Cache	37
4.1. Concept	37
4.2. Creating and Removing Caches	37
4.3. Usage	38
4.4. Sorting, Smoothing and Resampling	38

Contents

4.5. Statistics Functionalities	39
5. NumeRe Scripts	41
5.1. Concept	41
5.2. Syntax Highlighting	41
5.3. A Simple Script	41
II. Advanced Usage	45
6. Definition of Custom Functions	47
6.1. Definition	47
6.2. Conditioned Definition	48
6.3. Usage	48
7. Zeichenketten	49
7.1. Konzept	49
7.2. VariablenTyp	49
7.3. Konvertierung	50
8. Schleifen und Verzweigungen	53
8.1. Verzweigungen	53
8.2. Bedingte Schleifen	53
8.3. Zählschleifen	54
8.4. Ablaufkontrollen	54
9. Matrix-Operationen	55
9.1. Ausführen einer Matrix-Operation	55
9.2. Spezielle Funktionen	55
10. Spezielle Kommandos	59
10.1. Nullstellen	59
10.2. Extrema	59
10.3. Integration	60
10.4. Ableitung	60
10.5. Taylorentwicklung	61
10.6. Funktionswerte in 1D und 2D	61
10.7. Fouriertransformation	62
10.8. Differentialgleichungen	63
11. NumeRe-Prozeduren	65
11.1. Konzept	65
11.2. Struktur	65
11.3. Lokale und globale Variablen	66

11.4. Rückgabewerte	67
11.5. Namensräume	67
11.6. Fehlerbehandlung	68
11.7. Debugging	68
12. Animierte Graphen	71
13. Zusammengesetzte Graphen	73
14. Plugins	75
14.1. Funktionsweise	75
14.2. Installation	75
14.3. Eigene Plugins	76

LEGAL INFORMATION

GNU Free Documentation Licence v1.2

— Licence of this document —

Copyright © 2017, Erik Hänel.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license may be found at [GNU Free Documentation Licence](#).

GNU General Public Licence v3

— Licence of the application and the displayed code fragments —

Copyright © 2017, Erik Hänel et al.

The described application and the listed code fragments are free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see www.gnu.org/licenses/.

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

Legal Information

- **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. BÄMMERT, J. HÄNEL, R. HUTT, K. KILGUS, E. KLOSTER, K. KURZ, M. LÖCHNER, L. SAHINOVĆ, D. SCHMID, V. SEHRA, G. STADELMANN, R. WANNER, A. WIN-KLER, 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 be complete. All these possible options and their values are not described here. Their description may be found in the integrated documentation.

Note This documentation is supported with marginal texts, which shall point to actions inside 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). With the console one may interact directly with NUMERE and the editor may be used to edit text files, NUMERE scripts and NUMERE procedures. The *entry history* (bottom right) is an additional element, which protocols all entries 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, which are already knowing similar software (such as MATLAB), will get familiar 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 in the opposite direction $->$. You may *only* enter something, if such an arrow (or an explicit claim) appears.

Note In a later chapter we will describe how to use loops and forks. The arrows at the beginning of the line are changing to a more special symbol. However, this is not important in this context.

You may enter mathematical-numerical expressions as well as commands into the NUMERE con-

1. First Steps

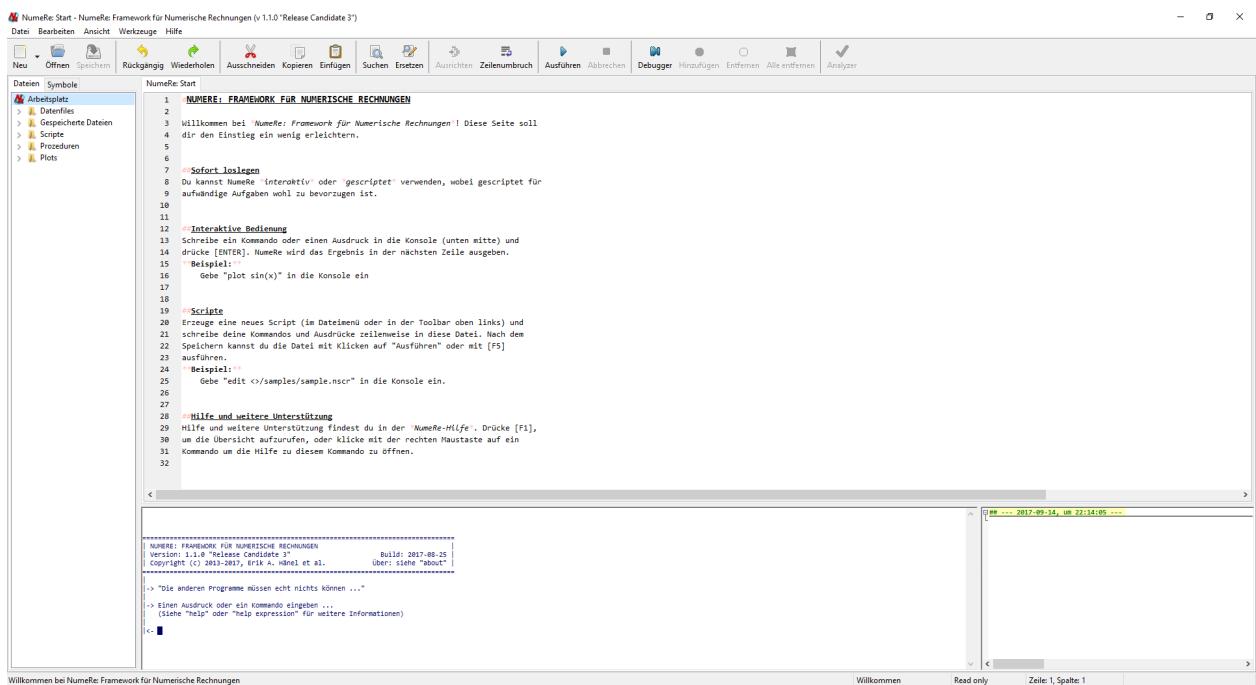


Figure 1.1.: The starting view of NUMERE with the described interface elements. The starting page of NUMERE, which describes some basics, is also visible. Note: All screenshots shown in this document are in german language but if you've installed the english version, the interface language will be English,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

sole. 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 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

```
1 help
  help expression
3 plot sin(x)
  mesh sin(x)+cos(y) -set box
5 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 following parameters.«

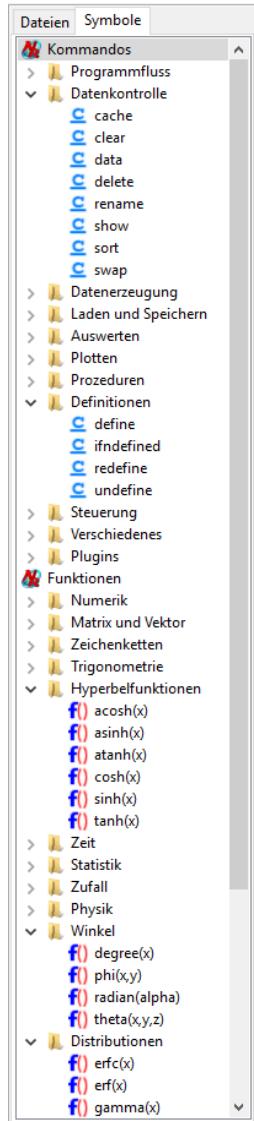
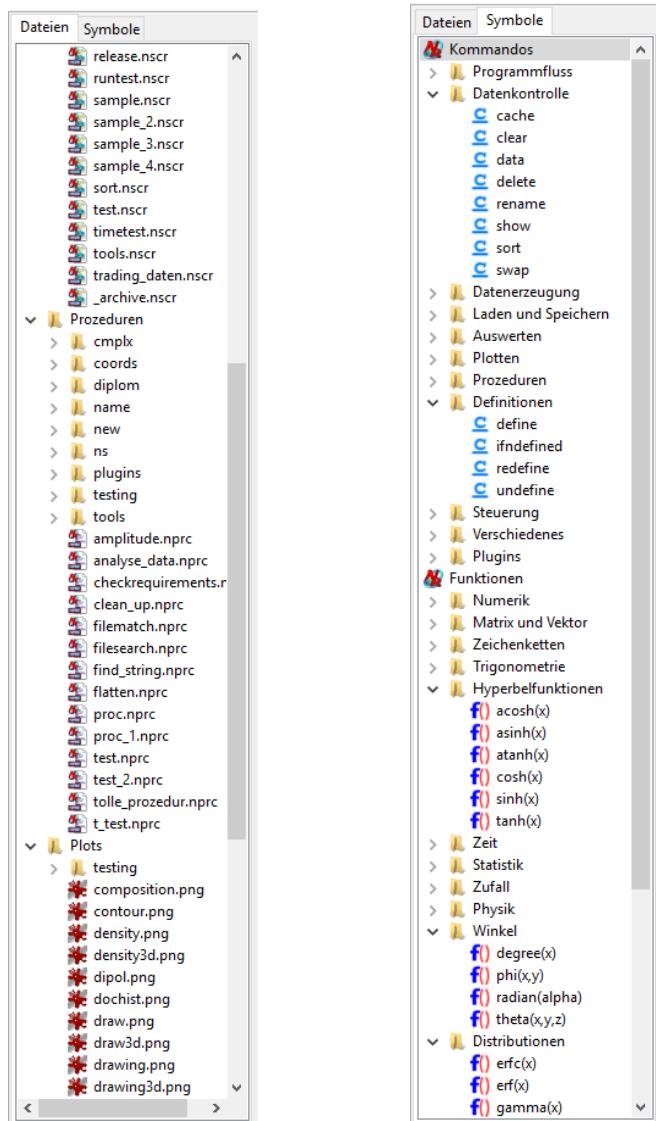


Figure 1.2.: File and symbol tree in the NUMERE interface

1. First Steps

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

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

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 separated in 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 the documentation article concerning the command, e.g. through

help 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 and its new value have to follow this command:

set 1 `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 1 `list -settings`

All default preferences of NUMERE are in principle set reasonable.

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:

```
1 5*cos(_2pi)
```

multiplies $\cos 2\pi$ 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

```
1 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 and 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:

```
1 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

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

assigns 5 to `neue_variable`, calculates $5 \cos 2\pi$ 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. First Steps

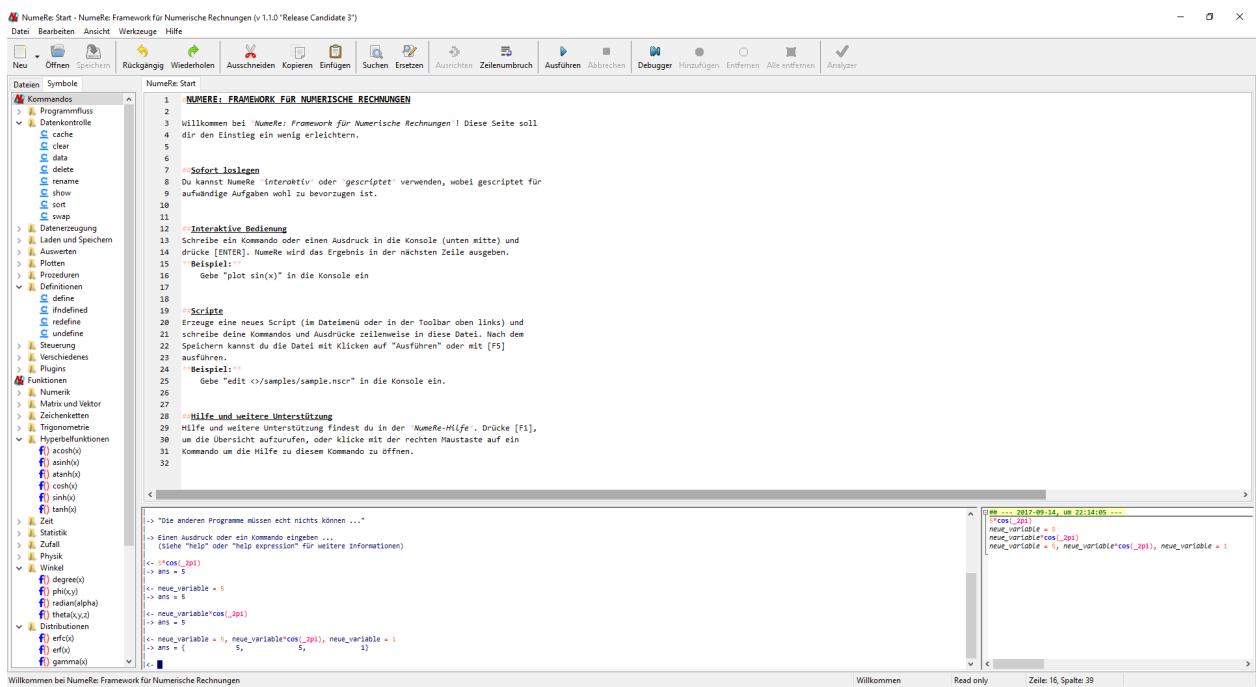


Figure 1.3.: NUMERE after the evalution of the last calculation. You may notice that the input is as well stored in the entry history

1.4. Important Commands

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

```
help
find 1 help
list  help TOPIC
quit 3 find TERMS
list  -OBJECT
```

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 with 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.

2. Loading and Using Data

- **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 verb+.+ 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.
- **T_EX file (*.tex).** NUMERE writes a table using the T_EX standard. The comments in this file contain the preconditions for including the table in a T_EX 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 if 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](#)

Syntax element
[load](#)

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

¹ [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 interpret that as an text file with the foile extension *.dat. 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

¹ [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\(\)](#)).

Syntax element
[show](#)

You may display the data table with the command [show](#):

¹ [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\(\)](#)

¹ [data](#)(3,1)
[data](#)(:,1)
³ [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)

2. Loading and Using Data

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, however, 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 `mat op` 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.

[▶ 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`

¹ `stats` `data(i1:i2, j1:j2)`

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

```

1 std(data(i1:i2,j1))
2 avg(data(i1:i2,j1))
3 [...]

```

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 a oscillation, or the life time of elementary particles or another quantity.

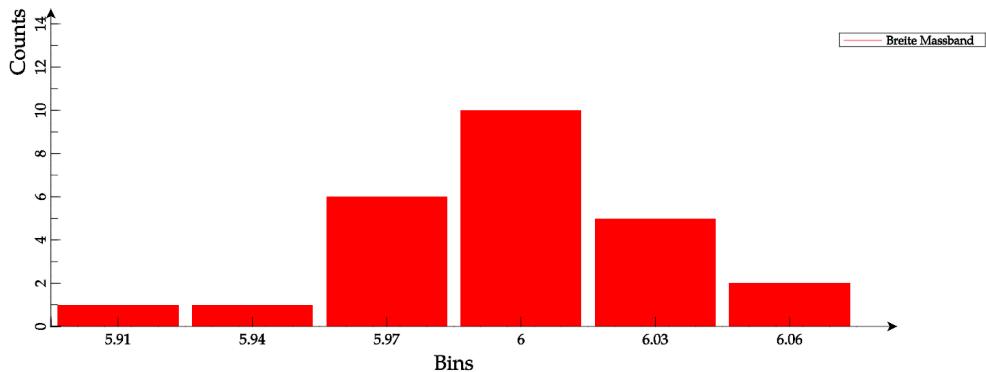


Figure 2.1.: Histogram of a data set, which was created using `hist`

Histograms of the loaded data are created using

Syntax element
`hist`

```

1 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` for a single histogram or the command `hist2d` for histograms in two spatial directions. Futher details are listed in the corresponding documentation article.

Syntax element
`hist2d`

► In the NUMERE documentation at: [help hist](#)

`hist`

2. Loading and Using Data

One additional, very common way of analyzing data next to histograms and statistics is 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 1 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=1,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 deviations 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 be 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`.

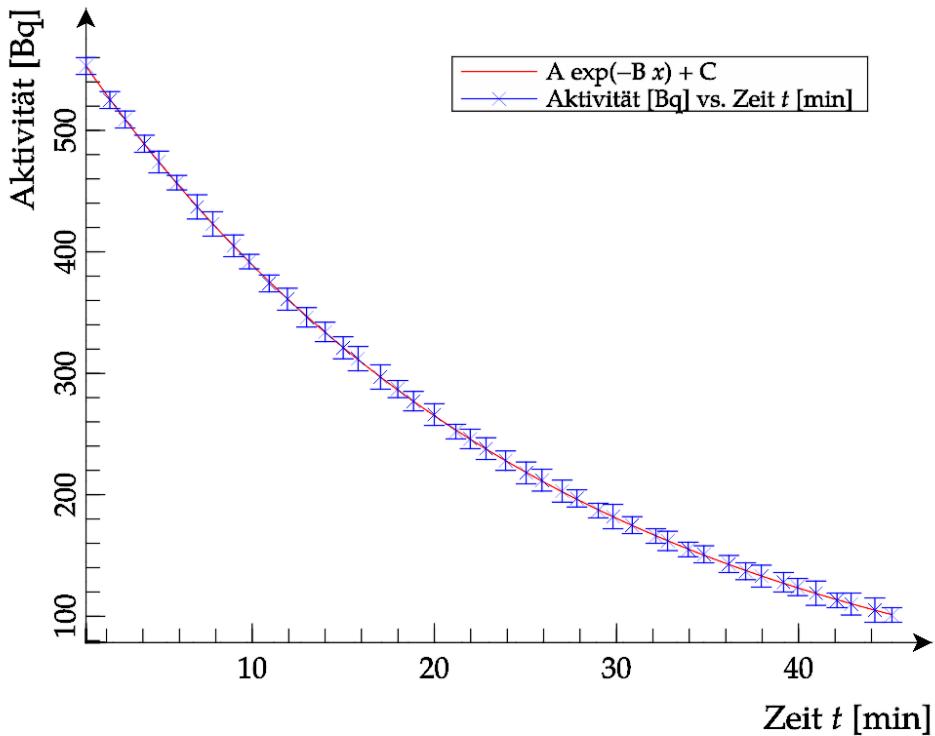


Figure 2.2.: Fitting a model for an exponential decay with `fitw`

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.

2. Loading and Using Data

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).

Syntax element

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

```
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 shall be passed everytime). 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`:

```
1 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](#)

Syntax element

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

```
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 function and data graphs. You may visualize the behaviour of functions and data and analyze them more easily. 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`:

```
1 plot
  plot3d
3 mesh
  dens
5 vect
  vect3d
7 ...
```

- `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

```
1 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 1 [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](#)

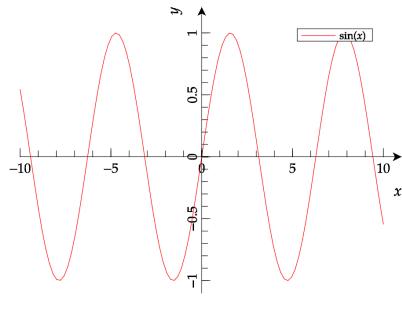
You are getting a surrounding box and a coordinate grid with

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

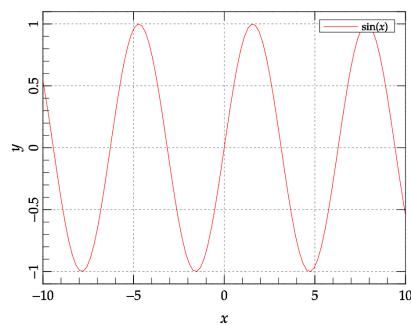
This and all further plots will also get a grid and a box ([Figure 3.1b and c](#)).

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

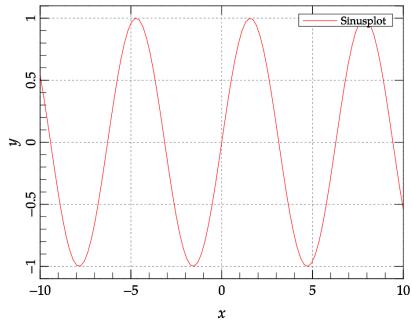
```
1 plot sin\(x\) "Sine function" -set box grid
```



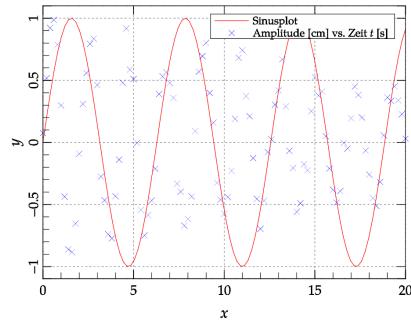
a: without options



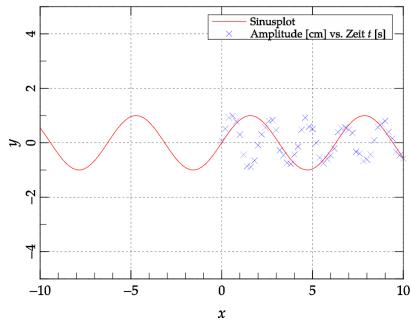
b: with box and grid



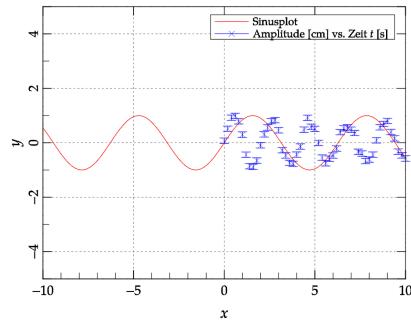
c: with custom legend



d: with data set



e: with custom interval



f: with errorbars

Figure 3.1.: The resulting graphs using the shown options

3. Creating Graphs

(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

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

(Figure 3.1d). Even for `data()` one may pass a customn legend. Otherwise NUMERE will create a combination of the columns titles. Additional it's noticeable 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:

```
1 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:

```
1 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](#)

Syntax element 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):

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

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):

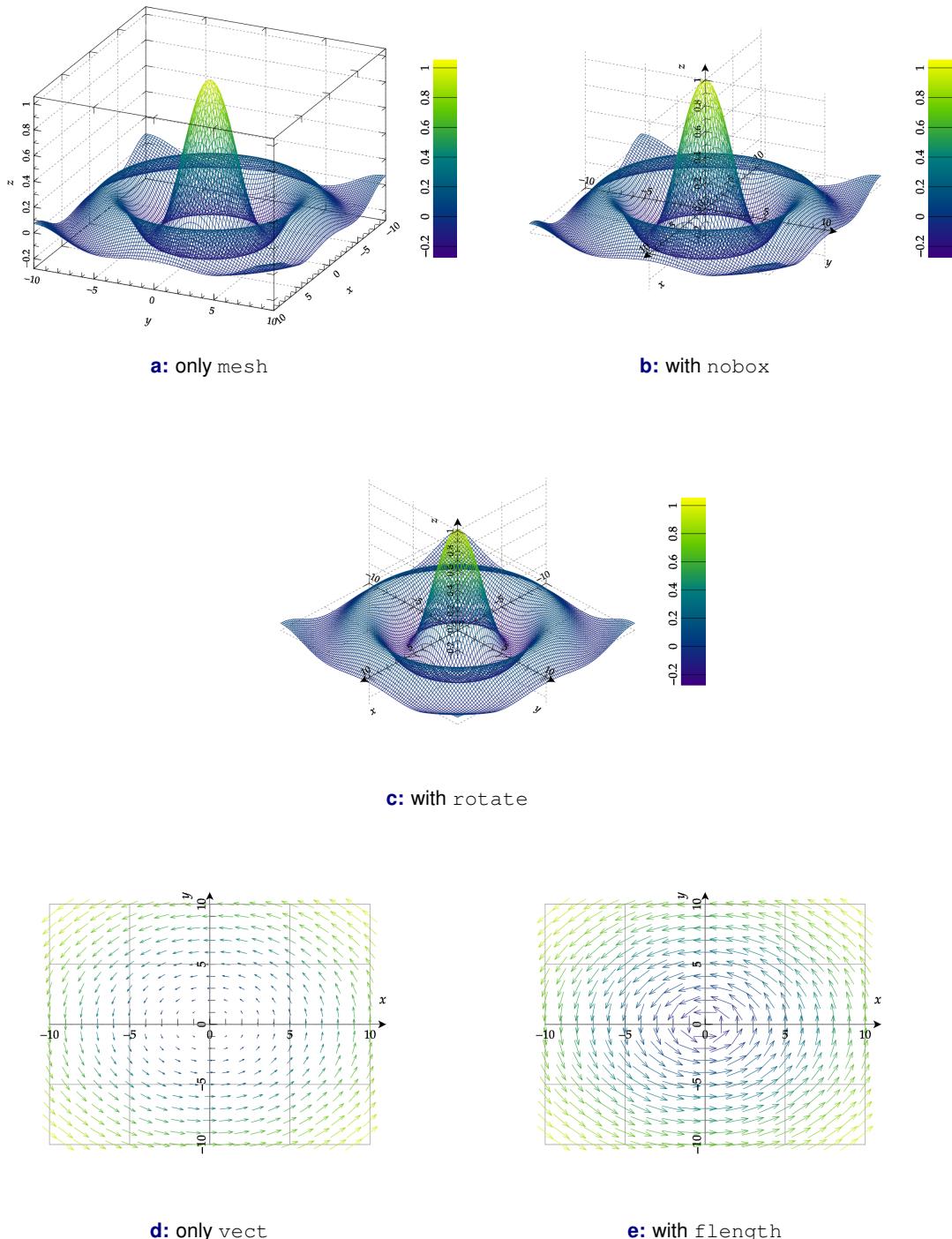


Figure 3.2.: Results of the meshgrid and vector plots

3. Creating Graphs

```
1 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 This is achieved through

```
1 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 only 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):

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

It's noticeable 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 `axiscale=SCALE`, if desired.

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

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

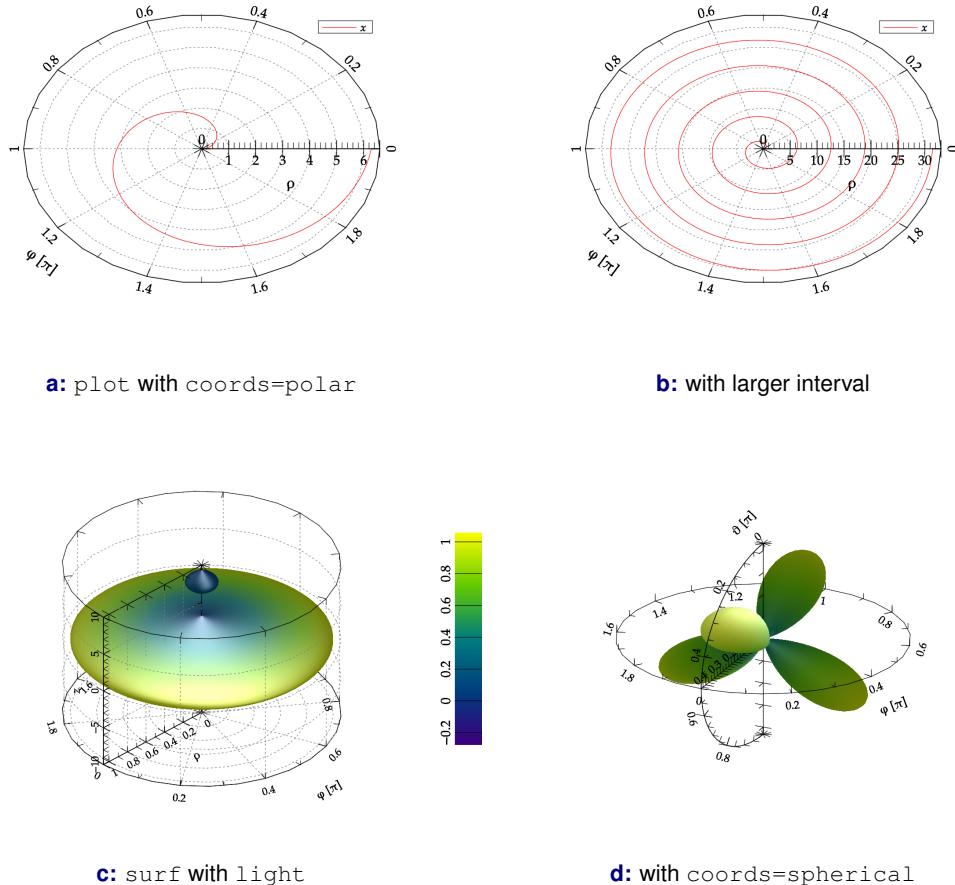


Figure 3.3.: Results of the different coordinate systems

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

```
1 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

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

3. Creating Graphs

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 .

[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):

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

Syntax element
`new`

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:

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

Syntax element
`remove`

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`

4. Tables: the Cache

```
1 rename -CACHE1=NEWNAME  
2 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

```
delete 1 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 similar to variables to their elements:

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

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

At all location, 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:

```
1 hist cache(:,1)  
2 stats cache()  
3 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 than one can reasonable process them. 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. In the following one might find distinct thresholds or points of interest in the measured data more easily.

Syntax element Through entering

```
sort 1 CACHE -sort
```

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

```
1 CACHE -sort=desc
```

Using the option cols to 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 sorted arbitrary, the function values may be sorted according to the x values without losing the connection between the x values and the function values):

```
1 CACHE -sort cols=COLUMNS [COLUMNGROUPS]
cache -sort 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 comparable artifacts. This is achieved through smoothing the data points.

The command

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

Syntax element
smooth

smooths the cache CACHE in the selected range, where the data points are interpolated linearly 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

```
1 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

4. Tables: the Cache

append the desired statistics function as a parameter (there are no parentheses in the expression in this case):

```
1 CACHE -avg  
CACHE -std  
3 ...
```

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 building the basis for NUMERE scripts is quite easy: all entries, which one might enter into the NUMERE console, are typed into a textfile 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 be corrected fastly without having 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 need. This may be done in the options dialogue, which is available in the tools menu, at 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 uses the corresponding item in the file menu or click on the tool in the toolbar or enter

5. NumeRe Scripts

```
1 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

```
edit 1 edit first.nscr
```

opens the NUMERE script in the NUMERE editor so that it may be edited. (The file extension is here necessary, because **edit** would look for the script in the wrong directory.) Between the both text blocks in the script (everything between `#*...*` is a comment and won't be evaluated by NUMERE) one enters the following lines:

```
1 #* Delete the contents of the cache completely */
2 delete cache() -ignore
3
4 #* Create random numbers */
5 random -lines=1e5 cols=2 distrib=uniform mean=0.5 width=1
6
7 #* Are the number inside of the circle of unity? */
8 cache(:,3) = (cache(:,1)^2+cache(:,2)^2) <= 1 ? true : false;
9
10 #* Calculate pi */
11 4*sum(cache(:,3))/1e5
```

where `#` starts a line comment, which is ignored by NUMERE, too. 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 times 100,000 equally distributed random numbers are created in the interval $[0;1]$.

The third expression is the trickiest. Here the result of a condition is written to the third column of `cache()`. If written with words, this means 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

```
1 CONDITION ? TRUE_VALUE : FALSE_VALUE
```

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

```
1 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 really 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 clicking on »Execute« or through

```
start
```

```
1 start first
```

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

```
1 3.1364
2 3.13668
3 3.1454
4 3.13952
5 ...
```

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

If one enlarges the number of random numbers (e.g. $1\text{e}6$ instead of $1\text{e}5$) 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

A custom function is defined using

Syntax element
`define`

```
1 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.:

Syntax element
`"..."`

```
1 define my_function(x,...) := x*sum(...)
```

The name of the function may be chosen freely, while ensuring that it doesn't start with a number nor is identical to a (pre-)defined function.

Syntax element
`redefine`

With

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

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

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 at

```
1 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`

6. Definition of Custom Functions

```
1 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

```
1 define -save
```

and the following line after the restart

```
1 define -load
```

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

```
1 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`, another is using

Syntax element

```
1 ifndefined 1 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

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

and

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

are identical for the definition

```
1 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. ZEICHENKETTEN

Neben numerischen Werten kann NUMERE auch mit Zeichenketten umgehen. Zunächst eingeführt, um die Spaltentitel der Tabellen formatieren zu können, sind die Zeichenkettenfunktionen inzwischen aber zu einem festen und sicher verwendbaren Konstrukt in NUMERE's Architektur geworden.

7.1. Konzept

Zeichenketten sind zunächst aneinandergereihte Zeichen, die NUMERE nicht als Variablen oder numerische Werte interpretiert. Um dies zu erreichen, sind Zeichenketten durch umschließende Anführungszeichen einzugeben:

¹ `"Dies ist eine Zeichenkette."`

Der eigentliche Inhalt der Zeichenkette sind alle Zeichen zwischen den Anführungszeichen. Die Zeichenkette `" "` ist folglich leer und hat die Länge 0.

Zeichenketten können durch spezielle Funktionen modifiziert werden: es gibt Funktionen zur Umwandlung von Groß- in Kleinbuchstaben (oder umgekehrt), zum Suchen von Zeichenketten in Zeichenketten, zum Extrahieren einer Zeichenkette aus einer anderen, zum Ersetzen von Zeichenketten in einer anderen, usw. Der eigentliche Vorteil von Zeichenketten sind die Möglichkeit, Ausgaben zu formatieren und das Verarbeiten mehrerer Dateien zu automatisieren. (Außerdem sind sie Grundvoraussetzung für die Programmierung durch NUMERE-Prozeduren)

► In the NUMERE documentation at: [help string](#)

string

7.2. Variablentyp

Neben den numerischen Variablen und den Tabellen sind die Zeichenkettenvariablen (auch *strings*) der dritte Variablentyp in NUMERE. Numerische Variablen können jedoch nicht in Zeichenketten und Zeichenkettenvariablen nicht in numerische umgewandelt werden. Ihre jeweiligen Inhalte jedoch schon (s.u.).

NUMERE erkennt neue Zeichenkettenvariablen automatisch anhand der Deklaration. Diese muss – im Gegensatz zur Deklaration numerischer Variablen, welche *on-the-fly* vonstatten gehen kann – *immer* mit einem Wert in Form einer Zeichenkette erfolgen (zumindest einer leeren Zeichenkette):

7. Zeichenketten

```
1 numerische_variable = 3.1415926
  auch_numerisch
2 zeichenkette = "Hallo Welt!"
  auch_zeichenkette = ""
```

Eine Deklaration mit dem Rückgabewert einer Zeichenkettenfunktion ist natürlich ebenfalls möglich.

Syntax element

string()

Neben den Zeichenkettenvariablen kennt NUMERE noch das `string()`-Objekt. Dies ist im Prinzip eine einspaltige Tabelle, die beliebig viele Zeichenketten aufnehmen kann. In den Argumentklammern kann die Bereichssyntax verwendet werden, um einen Satz an Zeichenketten zu extrahieren, oder ein einzelner Index für eine einzelne Zeichenkette. Bleiben hier die Argumentklammern leer, wird automatisch die zuletzt zugewiesene Zeichenkette verwendet.

string

► [In the NUMERE documentation at: help string](#)

Syntax element

data(#, :)

cache(#, :)

Mithilfe von Zeichenketten können auch die Tabellenköpfe von `data()` und den Caches geändert werden. Dies erreicht man durch

```
1 data(#, 1) = "Spaltenueberschrift 1"
2 cache(#, :) = "Spalte 1", "Spalte 2", "Spalte 3"
```

Wie man sieht, ist auch hier die Bereichssyntax möglich. Die Raute # referenziert die Tabellenköpfe.

cache

► [In the NUMERE documentation at: help cache](#)

7.3. Konvertierung

Der Wert einer Zeichenkettenvariable kann zu einem numerischen umgewandelt werden und umgekehrt. Dabei wird der Inhalt der Zeichenkette ggf. als neue Variable, als Ausdruck oder ggf. sogar als Kommando interpretiert. Hierbei gibt es zwei Funktionen:

```
1 to_value()
2 to_cmd()
```

Die Funktion `to_value()` wandelt die übergebene Zeichenkette in einen mathematisch-numerischen Ausdruck um und wertet ihn entsprechend aus. `to_cmd()` wandelt die Zeichenkette hingegen in einen Kommandoausdruck um.

Syntax element

#VAR

#(EXPRESSION)

```
1 #VAR
2 #(EXPRESSION)
3 valtostr(EXPR, C, N)
4 to_string()
5 string_cast()
```

Die Umkehrung kann auf verschiedene Arten vonstatten gehen:

Die Syntax `#VAR` oder `#(EXPRESSION)` wertet den darauffolgenden Ausdruck/Variable aus und wandelt den numerischen Wert direkt in eine Zeichenkette um. Zwischen # und dem Ausdruck können noch ein oder mehrere ~ eingefügt werden, welche die Zeichenzahl durch vorangestellte

0 auf eine entsprechende Zahl an Zeichen zzgl. des #'s ergänzen. Dies wird durch die Funktion `valtostr()` verallgemeinert, die das Füllzeichen mittels des Zeichens C übergeben bekommt. Die Funktion `to_string()` wandelt alles, was keine Zeichenkette ist, direkt in eine Zeichenkette um, ohne etwas zuvor auszuwerten, und `string_cast()` wandelt selbst Zeichenkettenvariablenamen in Zeichenketten um.

8. SCHLEIFEN UND VERZWEIGUNGEN

Der Ablauf von NUMERE-Scripten (und den im nächsten Kapitel eingeführten NUMERE-Prozeduren) kann durch die Einführung von Schleifen und Verzweigungen teils drastisch vereinfacht werden (was nicht heißen soll, dass Schleifen und Verzweigungen nicht auch direkt in der NUMERE-Konsole verwendbar wären).

8.1. Verzweigungen

Eine Verzweigung ist eine Stelle im Script, an der der weitere Verlauf des Scriptes von einer gegebenen Bedingung abhängt. Solche Verzweigungen werden durch das folgende Konstrukt repräsentiert:

```
1 if (BEDINGUNG1)
    AUSFUEHREN, FALLS WAHR
3 elseif (BEDINGUNG2)
    AUSFUEHREN, FALLS BEDINGUNG1 FALSCH UND BEDINGUNG2 WAHR
5 else
    AUSFUEHREN, FALLS ALLE BEDINGUNGEN FALSCH
7 endif
```

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

Eine Verzweigung muss mindestens aus einem `if ()` und einem abschließenden `endif` bestehen. Es können dazwischen beliebig viele `elseif ()` und höchstens ein `else` als allgemeiner *Sonst-Fall* verwendet werden, wobei `else` als letzter Fall vor `endif` verwendet werden muss. Eine Verzweigung, die nur aus einem `if ()` und einem `endif` besteht, wird nur dann ausgeführt, wenn die Bedingung erfüllt ist, anderenfalls wird sie komplett übersprungen.

Die Blöcke, die zwischen `if ()`, `endif` und den restlichen Schlüsselwörtern stehen, können beliebig lange sein und sowohl Ausdrücke als auch Kommandos enthalten. Zusätzlich können die Blöcke auch weitere Verzweigungen und Schleifen enthalten.

► In the NUMERE documentation at: [help if](#)

`if`

8.2. Bedingte Schleifen

Eine bedingte Schleife wird nur (so lange) ausgeführt, so lange die Bedingung wahr ist. Die Syntax dazu ist

Syntax element
`while ()
...
endwhile`

8. Schleifen und Verzweigungen

```
1 while (BEDINGUNG)
      AUSFUEHREN, SO LANGE WAHR
3 endwhile
```

Auch hier können Ausdrücke und / oder Kommandos im Ausführungsblock enthalten sein, ebenso wie weitere Verzweigungen und Schleifen.

`while`

► In the NUMERE documentation at: [help while](#)

8.3. Zählschleifen

Eine Zählschleife macht ihre Auswertung vom Wert einer Indexvariable abhängig. Über den Bereichssyntax muss hierbei der Start- und Endwert vorgegeben werden. Sollte der Endwert *kleiner* als der Startwert sein, zählt die Zählschleife automatisch rückwärts. Nach jedem Schleifendurchlauf wird der Indexwert automatisch um eins erhöht (bzw. erniedrigt, wenn die Schleife rückwärts zählt). Der Index selbst kann in der Schleife als Variable verwendet werden.

Syntax element

Die Syntax einer Zählschleife lautet wie folgt:

```
for () ...
1 for (INDEX = START:END)
      AUSFUEHREN, SO LANGE INDEX IN [START;END]
3 endfor
```

Ebenso wie zuvor können in diesem Block weitere Ausdrücke, Kommandos, Schleifen und Verzweigungen verwendet werden. Die Indexvariable wird nach Abschluss der Schleife – falls sie nicht bereits vor der Schleife bekannt war – automatisch gelöscht.

`for`

► In the NUMERE documentation at: [help for](#)

8.4. Ablaufkontrollen

Syntax element

Auf den Ablauf einer Schleife kann mit einem der beiden Kommandos

```
continue
break 1 continue
      break
```

weiter Einfluss genommen werden. `continue` überspringt den restlichen Schleifenablauf und beginnt sofort einen neuen Schleifendurchlauf. Hingegen bricht `break` die gesamte Schleife sofort ab und springt in die nächsthöhere (also umschließende) Schleife, sofern vorhanden. Falls es keine umschließende Schleife gibt, wird NUMERE mit dem restlichen Script-/Programmablauf fortfahren. Eine sinnvolle Anwendung dieser Kommandos kann eigentlich nur aus einer `if`-Verzweigung im inneren der betreffenden Schleife geschehen.

`if`

► In the NUMERE documentation at: [help if](#)

9. MATRIX-OPERATIONEN

NUMERE ist in Form einer Tabellenkalkulation ausgelegt, da es in der Wissenschaft ungemein wahrscheinlicher ist, dass Messreihen verarbeitet werden müssen, als dass eine Matrix-Operation durchgeführt werden muss. Dennoch ist auch NUMERE in der Lage, mit Matrizen umzugehen und wesentliche Auswertungen durchzuführen.

9.1. Ausführen einer Matrix-Operation

Matrix-Operationen verbergen sich hinter dem `matop`- bzw. `mtxop`-Kommando (Synonyme). Dieses Kommando leitet einen Ausdruck ein, der mittels Matrix-Operationen verarbeitet werden soll, wobei die Matrizen durch Ausschnitte des Caches, eines Datenfiles oder durch spezielle Funktionen repräsentiert werden. Hierbei ist jedoch noch zu beachten, dass alle Auswertungen (also auch die Multiplikation zweier Matrizen) standardmäßig elementweise durchgeführt werden.

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

Syntax element
`matop`

Um eine Matrix-Matrix- oder Matrix-Vektor-Multiplikation durchzuführen, muss der `**`-Operator verwendet werden. Dieser Operator hat eine höhere Priorität als alle anderen Operatoren, daher ist es ggf. wichtig, entsprechend zu klammern. Außerdem ist es wichtig, dass die Dimensionen der Matrizen gemäß den Regeln der Matrizenmultiplikation übereinstimmen.

Syntax element
`... ** ...`

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

Wenn `matop` kein Zielcache, in den die Matrix gespeichert werden kann, vorgegeben wird, wird automatisch `matrix()` verwendet. In diesem Fall werden die Inhalte von `matrix()` komplett überschrieben.

► In the NUMERE documentation at: [help matop](#)

`matop`

9.2. Spezielle Funktionen

Spezielle oder temporäre Matrizen bzw. weitergehende Matrix-Operationen können mit den folgenden Funktionen erreicht werden, wenn sie innerhalb des `matop`-Kommandos verwendet werden.

- `cross(MAT)` berechnet das n -dimensionale Kreuzprodukt der $n \times (n - 1)$ -Matrix `MAT`.

9. Matrix-Operationen

- `det (MAT)` berechnet die Determinante der quadratischen Matrix `MAT`.
- `diag (x, y, z, ...)` erzeugt eine Diagonalmatrix mit x, y, z, \dots auf der Hauptdiagonalen.
- `diagonalize (MAT)` diagonalisiert die quadratische Matrix `MAT`. Sollten die berechneten Diagonalelemente komplex sein, wird eine $n \times 2n$ -Matrix zurückgegeben mit den Realteilen auf der unteren und den Imaginärteilen auf der oberen ersten Nebendiagonalen.
- `eigenvals (MAT)` berechnet die Eigenwerte der quadratischen Matrix `MAT` und gibt diese in Form eines Vektors zurück. Sind die Eigenwerte komplex, werden sie als zweispaltige Matrix zurückgegeben, mit dem Realteil in der ersten und dem Imaginärteil in der zweiten Spalte.
- `eigenvects (MAT)` berechnet die Eigenvektoren der quadratischen Matrix `MAT` und gibt diese in Form einer Matrix mit den Eigenvektoren als Spalten zurück. Sind die Eigenvektoren komplex, werden sie als $n \times 2n$ -Matrix zurückgegeben, mit den Realteilen in den ungeraden und den Imaginärteilen der Vektorkomponenten in den geraden Spalten.
- `identity (n)` generiert eine $n \times n$ -Einheitsmatrix
- `invert (MAT)` invertiert die symmetrische Matrix `MAT`, falls eine Inverse existiert.
- `matfc (X, Y, Z, ...)` konstruiert eine Matrix aus den Spalten X, Y, Z, \dots . Fehlende Elemente werden durch 0 ergänzt.
- `matfcf (X, Y, Z, ...)` konstruiert eine Matrix aus den Spalten X, Y, Z, \dots . Fehlende Elemente werden aus den vorhandenen logisch ergänzt.
- `matfl (X, Y, Z, ...)` konstruiert eine Matrix aus den Zeilen X, Y, Z, \dots . Fehlende Elemente werden durch 0 ergänzt.
- `matflf (X, Y, Z, ...)` konstruiert eine Matrix aus den Zeilen X, Y, Z, \dots . Fehlende Elemente werden aus den vorhandenen logisch ergänzt.
- `one (n, m)` erzeugt eine $n \times m$ -Matrix, die vollständig mit 1-en gefüllt ist. Wird nur eine Dimension vorgegeben, erzeugt `NUMERE` eine symmetrische Matrix.
- `solve (MAT)` löst das lineare Gleichungssystem, das durch die Matrix `MAT` beschrieben wird.
- `trace (MAT)` berechnet die Spur der quadratischen Matrix `MAT`.
- `transpose (MAT)` transponiert die Matrix `MAT` (vertauscht Zeilen mit Spalten).
- `zero (n, m)` erzeugt eine $n \times m$ -Matrix, die vollständig mit 0-en gefüllt ist. Wird nur eine Dimension vorgegeben, erzeugt `NUMERE` eine symmetrische Matrix.

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

10. SPEZIELLE KOMMANDOS

An dieser Stelle sollen einige spezielle Kommandos Erwähnung finden, die bisher noch nicht genannt wurden, aber einen großen Teil von NUMERE's Funktionalität ausmachen.

10.1. Nullstellen

NUMERE kann mittels des Kommandos `zeroes` Nullstellen von Funktionen und Datenreihen suchen. Im Falle von Datenreihen gibt dieses die Indices der Nullstellen (bzw. des Punktes, der sich am nächsten befindet) oder, wenn ein x -Datensatz übergeben wurde, den entsprechenden x -Wert zurück:

Syntax element
`zeroes`

```
1 zeroes DATEN()  
2 zeroes DATEN() -set x=XWERTE()
```

Wenn die Nullstellen von Funktionen bzw. der Schnittpunkt zweier Funktionen gesucht werden soll, muss allerdings ein x -Intervall vorgegeben werden, in dem NUMERE suchen soll:

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

Im entsprechenden Hilfeartikel finden sich noch weitere Optionen, die `zeroes` übergeben werden können.

► In the NUMERE documentation at: [help zeroes](#)

`zeroes`

10.2. Extrema

Ähnlich wie bei `zeroes` verhält es sich bei `extrema`. NUMERE kann die Extrema von Funktionen und Datensätzen bestimmen. Bei Funktionen werden die x -Werte zurückgegeben, bei Datensätzen die Indices bzw. die x -Werte der Punkte, die das Extrema beschreiben.

Syntax element
`extrema`

```
1 extrema f(x) -set x=x1:x2  
2 extrema DATEN()  
3 extrema DATEN() -set x=XWERTE()
```

Anmerkung Im Gegensatz zu Nullstellen sind Extrema von Datensätzen nicht *wirklich eindeutig* bestimmbar – zumindest, wenn die Daten mit Rauschen behaftet sind. Dies ist aber in den meisten Fällen erfüllt, daher beschränkt NUMERE sich immer auf die Bestimmung der Indices bzw.

10. Spezielle Kommandos

x -Werte der Maxima bzw. Minima eines Datensatzes anstatt sie, wie bei `zeroes` ggf. zu interpolieren. Ebenso kann NUMERE aufgrund der numerischen Berechnung Sattelpunkte gar nicht oder nur sehr schlecht identifizieren, da diese Stellen keine Vorzeichenwechsel in der Ableitung aufweisen.

`extrema`

► In the NUMERE documentation at: [help extrema](#)

10.3. Integration

Syntax element

`integrate`

NUMERE kann sowohl Funktionen als auch Daten mittels `integrate` numerisch integrieren, wobei nur für Funktionen eine 2D-Integration unterstützt wird. Ob eine 2D-Integration ausgeführt werden soll, entscheidet NUMERE anhand der übergebenen Integrationsintervalle. Wenn nur ein x -Intervall übergeben wird, wird eine 1D-Integration ausgeführt; wenn zusätzlich ein y -Intervall übergeben wird, wird zweidimensional integriert.

Mittels des Parameters `-set` werden das Integrationsintervall und weitere Optionen übergeben, wie z.B. die Präzision der Integration, die Methode, ob die Funktionswerte des Integrals zurückgegeben werden sollen, etc. Details finden sich im entsprechenden Hilfeartikel.

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

Um 1D-Datensätze zu integrieren, sind diese statt der Funktion zu übergeben: wenn nur eine Spalte übergeben wird, wird nur die Summe des Datensatzes berechnet, werden zwei Spalten angegeben, wird die erste als x - und die zweite als die entsprechenden y -Werte interpretiert und das entsprechende Integral bestimmt.

1 `integrate DATEN(:,1)`
2 `integrate DATEN(:,1:3)`

`integrate`

► In the NUMERE documentation at: [help integrate](#)

10.4. Ableitung

Syntax element

`diff`

Neben der Integration ist NUMERE auch zur numerischen Ableitung erster Ordnung mittels des Kommandos `diff` fähig. Abhängig von den übergebenen Parametern wird dabei die Ableitung an der vorgegebenen x -Stelle oder entsprechend der Zahl der gewünschten Stützstellen (`samples`) über ein ganzes Intervall berechnet.

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

Ebenfalls ist NUMERE in der Lage, Datensätze numerisch zu differenzieren. Wenn hierbei eine Spalte übergeben wird, wird der Abstand zwischen den Stützstellen als 1 angenommen, wenn zwei Spalten angegeben werden, wird die erste als x - und die zweite als y -Werte interpretiert.

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

Hierbei werden nur die y -Werte der Ableitung berechnet. Mittels der Option `xvals` werden stattdessen die neuen x -Werte (die mit den vorgegebenen nicht übereinstimmen) bestimmt und zurückgegeben.

► In the NUMERE documentation at: [help diff](#)

`diff`

10.5. Taylorentwicklung

NUMERE kann mittels `taylor` Funktionen in Abhängigkeit von einer Variablen mittels des Taylor-Verfahrens durch ein Polynom der Ordnung $n \geq 0$ approximieren. Dieses Polynom muss allerdings (außer an der Entwicklungsstelle) nirgends mit der approximierten Funktion übereinstimmen (dies ist eine Eigenschaft der Taylorentwicklung und tatsächlich keine numerische Problematik).

Syntax element
`taylor`

Die Approximation geschieht rein numerisch. Dadurch kommt es zu unvermeidlichen Rundungsfehlern, die für die niedrigen Ordnungen der Entwicklung begrenzt sind. Bis etwa zur Ordnung $n = 10$ (anzugeben durch die Option `n=ORDNUNG`; Standard ist 6) kann NUMERE gute Koeffizienten bestimmen, oberhalb davon kommt es jedoch zu starken Abweichungen im Vergleich zu einer analytischen Berechnung.

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

Das entwickelte Polynom wird automatisch als eine Funktion im Funktionenspeicher definiert. Im Allgemeinen wird als Funktionsname `Taylor(x)` gewählt, wenn allerdings die Option `unique` übergeben wurde, ist der Funktionsname deutlich komplexer.

Anmerkung Bereits existente Funktionen, die mit demselben Namen im Funktionsspeicher vorhanden sind, werden bei dieser Definition überschrieben. Die Option `unique` generiert jedoch relativ zuverlässig eindeutige Funktionsnamen, da Ausdruck und Ordnung enthalten sind.

► In the NUMERE documentation at: [help taylor](#)

`taylor`

10.6. Funktionswerte in 1D und 2D

Mittels der Kommandos `eval` und `datagrid` kann NUMERE Funktionswerte von ein- oder zweidimensionalen Funktionen in einem vorgegebenen Intervall und zu einer vorgegebenen Anzahl an Stützstellen auswerten und entsprechend zurückgeben.

Syntax element
`eval`
`datagrid`

Das Kommando `eval` gibt die Funktionswerte eindimensionaler Funktionen im vorgegebenen Intervall zurück. Dieses Ergebnis kann direkt in einen Cache gespeichert werden. Die Stützstellen werden durch `samples` vorgegeben (Standard ist 100) und standardmäßig linear verteilt. Durch die Option `logscale` kann dies auf logarithmische Skalierung geändert werden.

10. Spezielle Kommandos

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

eval

► In the NUMERE documentation at: [help eval](#)

An manchen Stellen erwartet NUMERE sogenannte *Datengitter*. Dies ist ein tabellarischer Datensatz, wobei die x -Werte durch die erste, die y -Werte durch die zweite und die z -Werte durch die darauffolgenden Spalten repräsentiert werden, wobei die Zahl der Zeilen mit der ersten Spalte und die Zahl der Spalten mit der zweiten Spalte übereinstimmen muss.

Diese Datengitter kann NUMERE durch `datagrid` selbstständig erzeugen. Dabei können die Werte für x und y auf verschiedene Weisen gegeben werden: entweder als Intervall in der gemeinsamen Form `[x0:x1, y0:y1]` oder einzeln durch `x=x0:x1` oder als Spalte/Zeile eines Datensatzes wie `y=data(:, 3)`.

Für z stehen ebenfalls mehrere Möglichkeiten zur Verfügung: entweder als Funktion von x und y ($f(x, y) = \cos(x) \exp(-y)$), als Matrix eines Datensatzes (`cache(3 :, 7 : 100)`) oder als einzelne Spalte/Zeile eines Datensatzes (`data(4, 2 :)`). Im letzteren Fall versucht NUMERE, die dadurch definierten (x, y, z) -Punkte durch Triangulation zu verbinden, und ein Gitter durch lineare Interpolation zu erzeugen.

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

Der Parameter `samples=STÜTZSTELLEN` ist optional und beschreibt nur, wie viele Stützstellen berechnet werden sollen, falls eine Komponente des Datengitters berechnet werden muss. Standardmäßig werden (wie bei den 2D-Plots) 100×100 Stützstellen berechnet.

Falls die x - und y -Achse der Datenpunkte vertauscht sind (x = Zeilen, y = Spalten), kann `datagrid` noch der Parameter `transpose` übergeben werden. Dadurch werden Zeilen und Spalten beim Erzeugen des Datengitters vertauscht.

Das erzeugte Datengitter wird dabei automatisch an eine freie Stelle im Cache `grid()` (wird automatisch erzeugt, falls erforderlich) rechts von bereits bestehenden Daten kopiert und kann von dort aus geplottet werden.

datagrid

► In the NUMERE documentation at: [help datagrid](#)

10.7. Fouriertransformation

Zu den häufig verwendeten Auswertealgorithmen in der modernen Wissenschaft gehören auch Fouriertransformationen, die zum Beispiel auch frequenzabhängige Informationen aus einem vollkommen verrauschten Signal extrahieren können.

NUMERE ist hierzu mit einem Algorithmus zur schnellen Fouriertransformation (*fast fourier transform*) ausgestattet, der durch das Kommando `fft` ausgerufen wird und die übergebenen Datensätze zur Analyse in ihre enthaltenen Frequenzen zerlegen kann:

```
fft DATEN()
```

Syntax element

fft

Das übergebene Datenobjekt muss mindestens zwei Spalten besitzen: Achsenwerte in der ersten Spalte (Zeit oder Frequenz) und die zugehörige Amplitude in der zweiten Spalte. Werden drei Spalten angegeben, so werden diese als Achsenwerte, Amplitude und Phase (in dieser Reihenfolge) oder – falls die zusätzliche Option `-complex` übergeben wird – als Achsenwerte, Real- und Imaginärteil der Amplitude interpretiert.

`fft` wird die transformierten Daten als neue Spalten im übergebenen Datenobjekt (bzw. in `cache()`, falls das Datenobjekt `data()` ist) anlegen. Hierbei wird Frequenz, Amplitude (auch *Magnitude*) und Phase zurückgegeben. Mit der Option `-complex` werden Real- und Imaginärteile statt Amplitude und Phase zurückgegeben.

Um Daten invers zu transformieren, kann die Option `-inverse` übergeben werden.

► In the NUMERE documentation at: [help fft](#)

`fft`

10.8. Differentialgleichungen

Die wenigsten Differentialgleichungen lassen sich analytisch oder durch zufriedenstellende Näherungen lösen. Dies ist das Fachgebiet der Numerik, die die Differentialgleichungen mittels numerischen Algorithmen integriert und dadurch die entsprechenden Trajektorien berechnen kann.

NUMERE bietet einen solchen Integrationsalgorithmus mittels des Kommandos `odesolve`. Dieser Algorithmus kann Differentialgleichungen erster Ordnung numerisch integrieren. Da man aber auch Differentialgleichungen n -ter Ordnung in n Differentialgleichungen erster Ordnung zerlegen kann, stellt dies kein allzu großes Hindernis dar.

Syntax element
`odesolve`

Die Differentialgleichung kann dabei aus ein oder mehreren Teilgleichungen bestehen und damit auch ein ganzes System bilden. Die Gleichungen müssen dem folgenden Schema genügen:

```

1 dy1/dx = f1(x,y1,y2,...)
    dy2/dx = f2(x,y1,y2,...)
3 ...,

```

wobei nur die Funktionen $f_1()$ bis $f_n()$ in dieser Reihenfolge durch Kommata getrennt angegeben werden müssen. x ist hierbei die Integrationsvariable und y_1 bis y_n sind vordefinierte Funktionsvariablen, in denen NUMERE die Ergebnisse des letzten Integrationsschritts ablegt. Alle anderen Variablen werden als Parameter betrachtet.

Differentialgleichungen n -ter Ordnung $DGL(x, y, y', y'', \dots, y^{(n)})$ können stets in ein System von n Differentialgleichungen erster Ordnung überführt werden, indem $n - 1$ zusätzliche Funktionen eingeführt werden: $y' = dy1/dx = y_2, y'' = dy2/dx = y_3, \dots$ Ein solches System folgt dann dem Schema:

```

1 dy1/dx = y2
    dy2/dx = y3
3 ...
    dyn/dx = DGL(x,y1,y2,...,y(n-1))

```

10. Spezielle Kommandos

Die Ergebnisse der Integration werden standardmäßig in den Cache `ode()` als Tabelle geschrieben. Die erste Spalte enthält dabei die x -Werte und die folgenden Spalten die dazu integrierten Funktionswerte.

Um Startwerte vorzugeben (NUMERE wird anderen falls stets 0 verwenden), kann die Option `fx0=STARTWERTE` verwendet werden. Außerdem kann die Integrationsmethode, die zu verwendenden Toleranzen, die Zahl der berechneten Funktionswerte und andere Dinge modifiziert werden. Details hierzu finden sich in der integrierten Dokumentation.

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

Anmerkung NUMERE wird die Zahl der Funktionsvariablen entsprechend der Zahl der Gleichungen bereitstellen: bei einer Gleichung ist nur y_1 verfügbar, bei zwei y_1 und y_2 , etc. Werden mehr Variablen benötigt (um z.B. ein vektorielles Problem zu lösen), können 0-Gleichungen als entsprechende Gleichung eingeführt werden, die Variable wird dann jedoch als Konstante betrachtet.

`odesolve`

► In the NUMERE documentation at: [help odesolve](#)

11. NUMERE-PROZEDUREN

Neben den NUMERE-Scripten können Befehlsroutinen auch in NUMERE-Prozeduren ausgelagert werden. Allerdings bieten NUMERE-Prozeduren deutlich umfassendere Möglichkeiten, Aufgabenstellungen zu abstrahieren. Dazu gehört auch die Möglichkeit, NUMERE-Prozeduren rekursiv aufzurufen und ihre Rückgabewerte weiter zu verarbeiten.

11.1. Konzept

NUMERE-Prozeduren sind im Prinzip gestaltet wie eine Mischung aus einer selbst definierten Funktion und einem NUMERE-Script. Zusätzlich dazu kann eine NUMERE-Prozedur sich auch selbst rekursiv aufrufen (`define` würde einen Fehler zurückgeben) und deutlich komplexere Befehle und Ausdrücke abarbeiten (Prozeduren sind nicht auf Ausdrücke begrenzt, die sich in einer Zeile ausdrücken lassen müssen).

Mittels NUMERE-Prozeduren ist es des Weiteren möglich, eigene Unterprogramme in NUMERE zu schreiben oder weitere Funktionalität in Form eines neuen Kommandos hinzuzufügen (dies ist als *Plugin* bekannt).

Der Stellenwert einer NUMERE-Prozedur innerhalb von NUMERE ist als in etwa das Äquivalent einer Funktion in einer gewöhnlichen Programmiersprache.

► In the NUMERE documentation at: [help procedure](#)

procedure

11.2. Struktur

NUMERE-Prozeduren gliedern sich in ihrer Struktur in drei Teile: Prozedurkopf, Prozedurrumpf und Prozedurfuß. Der Kopf enthält den Namen, die Argumentliste und zusätzliche »Flags«, der Prozedurrumpf enthält die kompletten Befehlsroutinen:

```
procedure $PROZEDURNAME (ARGLIST) :: FLAGS    ## Prozedurkopf
2      PROZEDURRUMPF                         ## Befehle und Ausdrücke
  endprocedure                                     ## Prozedurfuss
```

Syntax element
procedure
...
endprocedure

Um eine NUMERE-Prozedur aufzurufen, gibt man `$PROZEDURNAME (VARS)` in NUMERE-Scrips, anderen NUMERE-Prozeduren oder der NUMERE-Konsole an. Die VARS sollten dabei mit der Zahl der Argumente in ARGLIST übereinstimmen. Falls in ARGLIST Defaultwerte für die

11. NumeRe-Prozeduren

Argumente angegeben wurden, können diese in VARS auch weggelassen werden. Es werden dann die Defaultwerte für die entsprechenden Argumente verwendet.

Die hier erwähnten »Flags« haben auf die gesamte NUMERE-Prozedur Einfluss und unterbinden oder erlauben bestimmte Verhaltensweise von NUMERE in dieser Prozedur. Im Grunde setzen diese Flags jedoch das Wissen der folgenden Abschnitte und Kapitel voraus:

Syntax element

explicit
private 1 [explicit](#)
inline [private](#)
3 [inline](#)

Der Flag `explicit` verhindert die Ausführung von Plugins in dieser Prozedur, `private`-geflagte Prozeduren können nur von Prozeduren desselben Namensraumes aufgerufen werden und Prozeduren mit dem `inline`-Flag erlauben schnellere Schleifenausführungen, wenn die Schleifen nur NUMERE-Prozeduren dieses Typs enthalten. Dafür sind `inline`-Prozeduren vergleichsweise eingeschränkt.

Flags können in jeder beliebigen Kombination/Reihenfolge angegeben werden, da sie sich nicht gegenseitig beeinflussen.

Jede NUMERE-Prozedur muss sich in einer eigenen *.nprc-Datei befinden, die den Namen der Prozedur trägt. Was im ersten Moment jedoch aufwändig und fehleranfällig klingt, kann von voll und ganz von NUMERE erledigt werden. Durch den entsprechenden Menüpunkt im Datei-Menü, durch Klicken auf die Schaltfläche der Toolbar oder durch

1 [new -proc=\\$PROZEDURNAME](#)

wird die NUMERE-Prozedur `$PROZEDURNAME` in <procpath>/`PROZEDURNAME.nprc` angelegt. Durch die darauffolgende Eingabe von

1 [edit \\$PROZEDURNAME](#)

kann `$PROZEDURNAME` im verknüpften Texteditor bearbeitet werden.

Der Prozedurrumpf einer NUMERE-Prozedur kann im Großen und Ganzen analog zu einem NUMERE-Script gestaltet sein. Die wesentlichen Besonderheiten werden in den folgenden Abschnitten besprochen.

11.3. Lokale und globale Variablen

NUMERE-Prozeduren unterscheiden zwischen lokalen und globalen Variablen. Globale Variablen sind Variablen, die von jeder Prozedur, jedem Script und der NUMERE-Konsole selbst aufgerufen und verwendet werden können. Demgegenüber sind lokale Variablen, die *nur* in *dieser* Prozedur und auch dann *nur* in *dieser* Rekursion der Prozedur verwendet werden können.

Syntax element

Lokale Variablen werden mittels

var
str 1 [var](#) VARIABLEN
tab [str](#) ZEICHENKETTENVARIABLEN
3 [tab](#) CACHES

deklariert. Diese Kommandos dürfen je Prozedur nur einmal auftreten und deklarieren jeweils einen Satz an lokalen numerischen Variablen, Zeichenkettenvariablen oder lokalen Caches. Diese

Variablen werden am Ende der Prozedur automatisch wieder gelöscht. (Lokale Variablen können genauso wie globale heißen. In einer NUMERE-Prozedur haben die lokalen Variablen dann die höhere Priorität. Man spricht dann von *schattieren*.)

11.4. Rückgabewerte

Standardmäßig geben NUMERE-Prozeduren den Wert `true` zurück, wenn die Auswertung die Zeile `endprocedure` erreicht. NUMERE-Prozeduren können aber auch andere Werte zurückgeben, wenn

¹ `return` WERT

Syntax element
`return`

an der entsprechenden Stelle in der Prozedur auftritt. NUMERE wird die Prozedur auf jeden Fall verlassen, sobald NUMERE an einer Stelle auf dieses Kommando trifft (Es kann auch mehrmals in einer Prozedur verwendet werden). `WERT` kann dabei entweder ein numerischer Wert oder eine Zeichenkette sein. Davon abgesehen kann auch explizit `true` oder `false` als `WERT` verwendet werden.

Dabei muss `WERT` nicht auf einen einzigen Wert beschränkt werden. Es können auch mehrere numerische Werte oder mehrere Zeichenketten zugleich zurückgegeben werden. Sogar die Kombination aus beiden Variablentypen ist möglich, aber zugleich auch fehleranfälliger als Werte eines einzelnen Variablentyps, da die ggf. aufrufende Prozedur auch mit dem Satz an Werten zurecht kommen muss.

Werden Prozeduren mit mehreren Rückgabewerten für `while`-Schleifen- oder `if`-Bedingungen verwendet, so wird dort nur der erste Wert für die Logikoperationen ausgewertet.

Der spezielle Wert `void` dient dazu, NUMERE mitzuteilen, dass diese Prozedur tatsächlich *keinen* Rückgabewert hat.

11.5. Namensräume

Ähnlich wie C++ ist NUMERE mit Namensräumen ausgestattet, die es erlauben, Prozeduren gleichen Namens aber unterschiedlichen Verhaltens in unterschiedlichen Namensräumen zu besitzen. Aufgerufen werden Prozeduren aus einem anderen als dem Standardnamensraum `main` durch

¹ `$NAMENSRÄUM~PROZEDURNAME` (VARS)

Die entsprechende Prozedur `$PROZEDURNAME` liegt dabei in der Datei

¹ `<procpfad>/NAMENSRÄUM/PROZEDURNAME.nscr`

Das Kommando `new` kann auch automatisch Prozeduren in untergeordneten Namensräumen erzeugen, wenn dies entsprechend angegeben wird:

¹ `new -proc=$NAMENSRÄUM~PROZEDURNAME`
`edit $NAMENSRÄUM~PROZEDURNAME`

11. NumeRe-Prozeduren

Gleiches gilt aber auch, wenn man die Funktionen der graphischen Oberfläche verwendet.

Werden in einer Prozedur häufiger NUMERE-Prozeduren aus einem bestimmten Namensraum aufgerufen, kann dieser Namensraum auch als temporärer Standardnamensraum vorgegeben werden, indem

Syntax element

namespace

namespace NAMENSRAUM

in die Prozedur eingebunden wird. (Prozeduren aus anderen Namensräumen können jedoch immer noch aufgerufen werden, indem der Namensraum wie oben gezeigt explizit angegeben wird.)

11.6. Fehlerbehandlung

NUMERE-Prozeduren verfügen über ein simples Verfahren, mit Fehlern umzugehen. Tritt ein Fehler auf, der auf keine sinnvolle Art behandelt werden kann (z.B. in Form einer falschen Eingabe), dann kann mittels des Kommandos

Syntax element

throw

1 throw

ein unmittelbares Abbrechen der Prozedurauswertung erzwungen werden. Dieses Kommando wird bis ganz oben durchgereicht, bis es schließlich auf die Standard-NUMERE-Konsole trifft und dort eine entsprechende Fehlermeldung produziert.

Etwas informativer wird die Fehlermeldung durch Übergeben einer eigenen Meldung. Dies geschieht in Form einer Zeichenkette

1 throw "Das ist die Fehlermeldung."

die in der NUMERE-Konsole schließlich angezeigt wird.

11.7. Debugging

In den wenigsten Fällen wird eine NUMERE-Prozedur beim ersten Versuch bereits lauffähig und fehlerfrei sein. Hierzu sind kleinere Tipp- oder Logikfehler viel zu wahrscheinlich. Beim Ausführen einer solchen fehlerhaften NUMERE-Prozedur wird NUMERE ggf. an einer Stelle abbrechen, wenn es sich um einen Syntaxfehler handelt, aber nicht unbedingt, wenn ein Rechenfehler aufgetreten ist.

NUMERE besitzt zum Aufspüren und Entfernen solcher Fehler einen sogenannten NUMERE-Debugger (Figure 11.1). Dieser kann durch den entsprechenden Menüpunkt im Werkzeuge-Menü oder durch Klicken auf die Schaltfläche der Toolbar aktiviert werden und listet bei Syntaxfehlern automatisch den fehlerhaften Ausdruck, die ungefähre Zeile des Ausdrucks, das fehlerhafte Modul (Datei), eine Stacktrace und alle lokalen Variablen der aktuellen Prozedur inklusive ihrer Werte. Mittels der Stacktrace kann rekonstruiert werden, in welcher Prozedur der Fehler aufgetreten ist und welche Argumente an diese übergeben wurden.

Zusätzlich zum automatischen Listen bei Syntaxfehlern kann der NUMERE-Debugger vergleichbare Informationen an einem *Breakpoint* liefern. Ein temporärer Breakpoint kann durch die

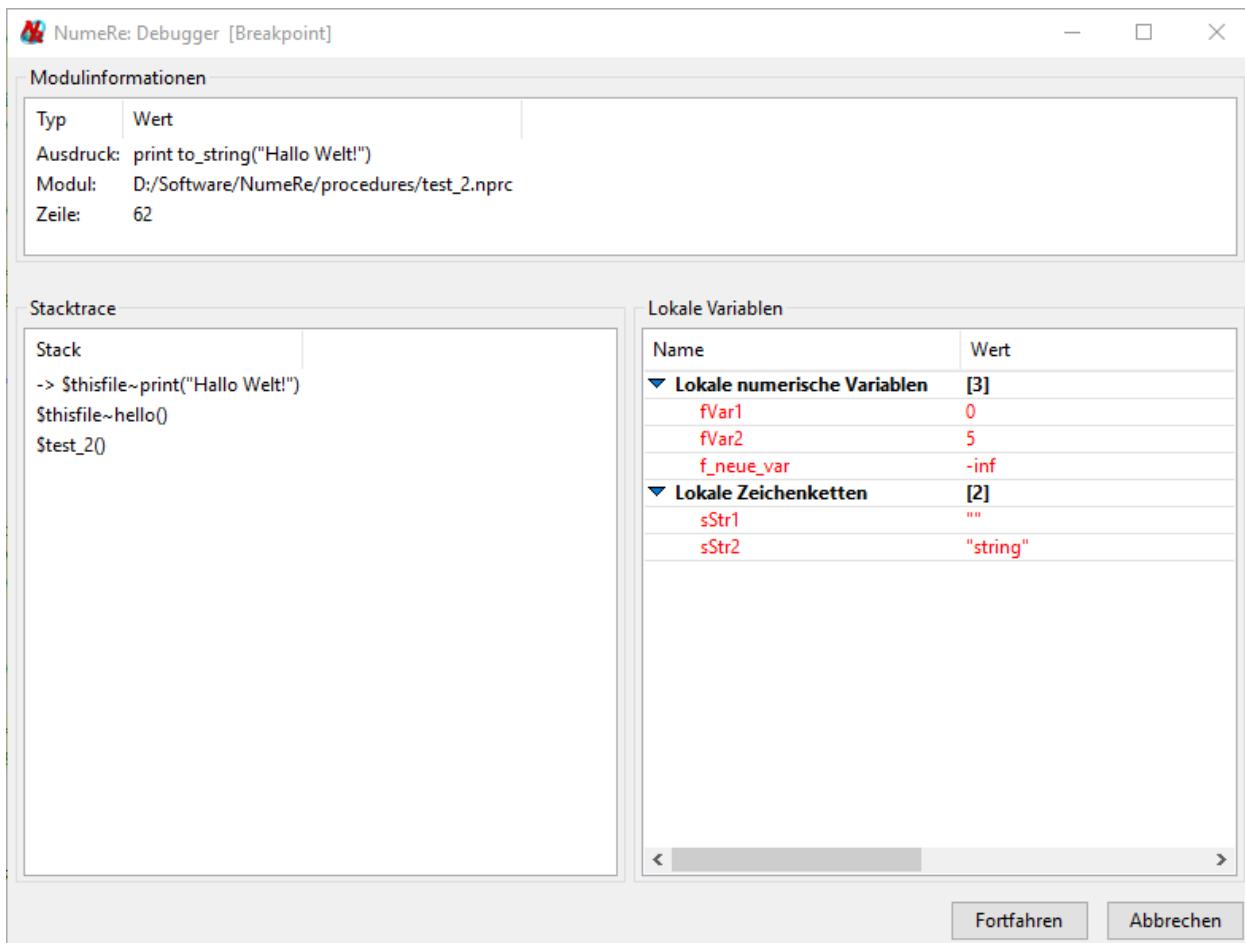


Figure 11.1.: Der NUMERE-Debugger an einem Breakpoint. Mit rot werden die Änderungen zum letzten ausgewerteten Breakpoint hervorgehoben

Funktionen der Toolbar oder durch Klicken auf die betreffende Zeile der Seitenleiste gesetzt werden. Man beachte, dass in Zeilen, die nur aus Kommentaren bestehen, oder Zeilen, die leer sind, *keine* temporären Breakpoints gesetzt werden können.

Permanente Breakpoints werden mittels `|>` am Anfang einer Zeile gesetzt, sind aber selten nötig. NUMERE unterbricht, bevor die *aktuelle Zeile* ausgewertet wird, und listet Stacktrace und Variablenwerte. Durch Klicken auf »Fortsetzen« kann die Auswertung fortgesetzt werden, durch »Abbrechen« wird die Auswertung komplett abgebrochen. (Außerhalb des Debug-Modus werden Breakpoints selbstverständlich ignoriert.)

► In the NUMERE documentation at: [help_debugger](#)

debugger

12. ANIMIERTE GRAPHEN

Manchmal kann es hilfreich sein, wenn man den zeitlichen Verlauf einer Größe mithilfe eines animierten Graphen verdeutlichen kann. NUMERE verfügt über eine fest implementierte Möglichkeit, eine solche Animation zu erzeugen. Eine Animation kann aber natürlich auch mittels einer Schleife erzeugt werden, indem die Frames einzeln gespeichert und später zu einer Animation zusammengesetzt werden.

Um eine Animation direkt durch NUMERE erzeugen zu lassen, muss mindestens an einer Stelle die Variable t in einem zu plottenden Ausdruck verwendet werden. Diese Variable ist in diesem Fall der Zeitparameter, der für die Animation variiert wird. Zusätzlich muss die Option `animate` bei einem Plotvorgang übergeben werden:

¹ PLOTCMD AUSDRUCK(t , VARS) `-set` `animate` OPTIONEN

Syntax element
`animate`

Dies wird eine Animation erzeugen, die aus mehreren, inkludierten Einzelbildern besteht. Im NUMERE-GraphViewer kann diese Animation abgespielt oder die Frames einzeln betrachtet werden.

► In the NUMERE documentation at: [help plotoptions](#)

`plotoptions`

Anmerkung NUMERE kann nativ keine Animationen aus Datensätzen erzeugen. Dies ist u.A. der Tatsache geschuldet, dass der Zeitparameter auch nicht-ganzzahlig Werte annehmen kann, Tabellenobjekte allerdings nur ganzzahlige Indices akzeptieren. Um eine Animation aus Datensätzen zu generieren, muss folglich auf die Einzelbilderzeugung mittels einer Schleife zurückgegriffen werden.

Standardmäßig erzeugt NUMERE 50 Frames je Animation mit einer jeweiligen Dauer von 1/25 s und verwendet als Zeitparameter t das Intervall [0;1]. Dies kann natürlich geändert werden, allerdings kann es bei zu vielen Frames zu Speicherschwierigkeiten kommen, da die Frames einzeln im Speicher abgelegt werden müssen:

¹ PLOTCMD AUSDRUCK(t , VARS) `-set` $t=0:2$ `animate=100` OPTIONEN

Nun wird die Animation mit 100 Frames erzeugt, wobei der Zeitparameter das Intervall [0;2] verwendet.

Anmerkung Manche Plotarten können möglicherweise zu viel Speicherplatz erfordern, so dass eine Animation nicht möglich ist. Es kann hierbei helfen, die `samples` der Abbildung zu reduzieren.

13. ZUSAMMENGESETZTE GRAPHEN

Es mag vorkommen, dass ein gewünschtes Plotlayout nur erreichbar ist, wenn man zwei Plotstile miteinander kombinieren könnte. Dies ist in NUMERE auch möglich, wenn man

```
1 compose
  PLOTSTIL1
3   PLOTSTIL2
...
5 endcompose
```

Syntax element
compose
...
endcompose

verwendet. Der **compose**-Modus nimmt allerdings nur Plotkommandos auf, das heißt, dass die für die jeweiligen Plotstile nötigen Berechnungen bereits zuvor abgeschlossen sein müssen.

Manche Plotoptionen gelten für einen kompletten zusammengesetzten Plot (wie z.B. Plotintervall, Achsenbeschriftung, Gitter, etc.), andere können je nach Plot aus- oder eingeschaltet werden (z.B. Transparenz, Lichteffekt, zusätzliche Konturlinien, Fehlerbalken, etc.). Die Plotfarben und Linienstile können auch für jeden Plot einzeln gewählt werden, jedoch ist zu beachten, dass NUMERE einen internen Zähler für alle Linien hat und dieser für jeden Plot erhöht wird. Demzufolge müssen die Styleinformationen ggf. entsprechend angepasst werden.

Mithilfe des **compose**-Modus können Plots zwei verschiedener Größen gemeinsam dargestellt werden, z.B. ein Vektorfeld zusammen mit seinem Potential. Es spielt hierbei keine Rolle, in welcher Reihenfolge **vect** und **dens** angegeben werden, da NUMERE die Plots intern in eine sinnvolle Reihenfolge bringt:

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

compose

► In the NUMERE documentation at: [help compose](#)

Der **compose**-Modus erlaubt es des Weiteren, die Beschränkung auf eine einzelne Funktion bei den 3D-Plotstilen (außer **plot3d**) zu umgehen, indem die Funktionen innerhalb des **compose**-Modus in sukzessiven 3D-Plots angegeben wird.

14. PLUGINS

NUMERE wird zwar beständig weiterentwickelt, doch es wird immer Funktionalitäten geben, die nicht implementiert werden, oder Funktionen, die nicht dem tatsächlichen Wunsch des NUMERE-Benutzers entsprechen. Hier kommen die NUMERE-Plugins in Spiel, die bestehende Funktionalitäten umschreiben oder neue hinzufügen können.

14.1. Funktionsweise

Plugins werden vollständig in die Kommandoarchitektur integriert, so dass von außen gar nicht bemerkt wird, dass im Augenblick ein Plugin ausgeführt wird. Sie werden durch eigene oder bereits bestehende Kommandos (deren Funktionen sie ersetzen) aufgerufen und verwenden die übliche NUMERE-Syntax.

Da außerdem die Möglichkeit besteht, dass Plugins einen eigenen Artikel zur NUMERE-Hilfe hinzufügen, ist die Beschreibung eines Plugins gleich enthalten und kann auf die übliche Art und Weise aufgerufen werden.

14.2. Installation

Plugins werden in Form von NUMERE-Scripten veröffentlicht, die

Syntax element
<install>
...
<endinstall>

```
<install>
2   DEKLARATIONEN UND PROZEDUREN
<endinstall>
```

enthalten. Um ein solches Plugin zu installieren, muss lediglich

Syntax element
install

```
1 install SCRIPTNAME
```

in die NUMERE-Konsole eingegeben werden (sollte das Script nicht in <scriptpath> vorliegen, muss der Pfad mit angegeben werden). NUMERE wird nun die Installationsroutine ausführen, die entsprechenden Prozeduren erzeugen und die ebenfalls nötigen Verknüpfungen generieren. Es ist *nicht* möglich, ein Plugin mit den Funktionen der graphischen Benutzeroberfläche zu installieren.

► In the NUMERE documentation at: [help install](#)

install

Nach Abschluss des Scripts ist das Plugin installiert. Es sollte nun unter

```
1 list -plugins
```

14. Plugins

gelistet werden und kann folglich mit dem dort angegebenen Kommando ausgeführt werden. Die angegebene Beschreibung sollte eine knappe Information über die Funktionsweise des Plugins enthalten.

Syntax element

Um ein Plugin wieder zu deinstallieren, gibt man einfach

uninstall 1 uninstall PLUGINNAME

in die NUMERE-Konsole ein. Der PLUGINNAME ist unter

1 list -plugins

in eckigen Klammern angegeben. Ggf. ist es nötig, dass der PLUGINNAME in Anführungszeichen angegeben wird.

Anmerkung Es ist nicht nötig, ein Plugin zunächst zu deinstallieren, um eine neue Version desselben zu installieren. Die Installation kann direkt über die bestehende ausgeführt werden. NUMERE wird die entsprechenden Verknüpfungen selbstständig ändern.

14.3. Eigene Plugins

Die Entwicklung eines eigenen Plugins mag zunächst kompliziert klingen, aber tatsächlich ist es das eigentlich nicht. NUMERE gibt bereits eine Hilfestellung, indem das Template (Vorlage) verwendet wird. Mittels der entsprechenden Funktionen der graphischen Benutzeroberfläche oder durch

1 new -plugin=PLUGINNAME

wird ein Plugin des Namens PLUGINNAME in Form eines NUMERE-Scripts unter

1 <scriptpath>/plgn_PLUGINNAME.nscr

mit der Hauptprozedur

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

erzeugt, das bereits alle Elemente für eine vollständige Installation (inklusive der Hilfedatenbankeinträge) mit entsprechenden Platzhaltern umfasst.

Plugins bestehen im Wesentlichen aus ein oder mehreren Prozeduren, welche die Pluginfunktionalität repräsentieren. Bei der Deklaration eines Plugins muss eine Hauptprozedur angegeben werden, die NUMERE aufruft, sobald das Pluginkommando eingegeben wird. Dieser Prozedur übergibt NUMERE den entsprechenden Kommandoausdruck in einer oder mehreren der folgenden Gestalten, wobei NUMERE dies bei der Deklaration mitgeteilt werden muss:

Syntax element

<CMDSTRING>

<EXPRESSION>

<PARAMSTRING>

1 <CMDSTRING>

<EXPRESSION>

3 <PARAMSTRING>

- <CMDSTRING> übergibt die gesamte Komandozeile (inklusive des Plugin-Kommandos)

- <EXPRESSION> übergibt den Ausdruck, der zwischen dem Kommando und den optionalen Parametern gefunden wird
- <PARAMSTRING> übergibt den Parametersatz, der entweder ab -set (bei einer vorhandenen <EXPRESSION>) oder ab dem ersten – nach dem Kommando beginnt

Die Prozeduren, die die Pluginfunktionalität umfassen, kopiert man in ein Script, umschließt sie mit

```

1 <install>
  <info>
3   -author="AUTORNAME"
4   -version="VERSION"
5   -type=TYPE_PLUGIN
6   -flags=ENABLE_DEFAULTS
7   -name="PLUGINNAME"
8   -pluginmain=$PLUGINHAUPTPROZEDUR(<CMDSTRING>)
9   -plugincommand="PLUGINKOMMANDO"
10  -plugindesc="BESCHREIBUNG"
11 <endinfo>
12  PROEDUREN
13 <endinstall>
```

Syntax element
<info>
...
<endinfo>

und installiert das Plugin mit dem Kommando `install` (Die Werte zu `type` und `flags` sind tatsächlich in Großbuchstaben anzugeben).

Wenn nun das `PLUGINKOMMANDO` in die NUMERE-Kommandozeile eingegeben wird, ruft NUMERE die Prozedur `$PLUGINHAUPTPROZEDUR()` auf und übergibt dieser dabei auch gleichzeitig den <CMDSTRING>. Es können auch andere oder weitere Kommandoausdruckabschnitte übergeben werden, wenn diese als eigene Argumente der Prozedur angegeben werden.

Soll das Plugin einen Wert zurückgeben, mit dem man weiterarbeiten können soll, dann muss als `type` der Typ

```
1 -type=TYPE_PLUGIN_WITH_RETURN_VALUE
```

angegeben werden.

Der `version`-Flag kann als

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

angegeben werden. Bei jeder Installation des Plugins wird die Versionsnummer nun automatisch erhöht. Dies ist zum Beispiel während der Entwicklung hilfreich, wenn man anhand der Zahl der Änderungen die Versionsnummer festlegen will.

Um die Ausgabe auf der NUMERE-Konsole zu unterdrücken, kann

```
1 -flags=DISABLE_SCREEN_OUTPUT
```

statt `ENABLE_DEFAULTS` angegeben werden. Weitere Flags sind

```
1 ENABLE_FULL_LOGGING
ENABLE_FORCE_OVERRIDE
```

14. Plugins

die entweder die komplette Installation zeilenweise in <>/install.log protokollieren oder bereits vorhandene Plugins eines anderen Autors mit gleichem Pluginkommando überschreiben.

plugins

► In the NUMERE documentation at: [help plugins](#)

Vor <endinstall> besteht noch die Möglichkeit, dass ein eifriger Programmierer die Information für den NUMERE-Hilfeartikel in Form einer XML-Syntax eingebindet:

Syntax element
<helpindex>
...
</helpindex> 2
<helpfile>
...
</helpfile> 4
 INDEXINFORMATIONEN
 6
 </helpindex>
 8
 <helpfile>
 HILFEARTIKEL
 10
 </helpfile>
 <endinstall>

Die INDEXINFORMATIONEN enthalten die Schlüsselwörter, unter denen NUMERE den betreffenden Artikel zeigen soll, sowie die Informationen zur Darstellung der Artikelübersicht, wenn man help idx eingibt.

Im HILFEARTIKEL finden sich dann die tatsächlichen Erläuterungen und Beschreibungen zum geschriebenen Plugin.

documentation

► In the NUMERE documentation at: [help documentation](#)
