

Dokumentation

NUMERE: FRAMEWORK FÜR NUMERISCHE RECHNUNGEN

1.1.x-Serie
Freie numerische Software unter der GNU GPL v3

Erik Hänel et al.

13. Dezember 2017



Für die Freunde der Wissenschaft

INHALTSVERZEICHNIS

Rechtliche Informationen	9
GNU Free Documentation Licence v1.2	9
GNU General Public Licence v3	9
Mitwirkende	9
Installation	11
Systemvoraussetzungen	11
Portable und Stable Version	11
I. Grundlegende Bedienung	13
1. Erste Schritte	15
1.1. Das User Interface	15
1.2. Einstellungen	18
1.3. Einfache Rechnungen	19
1.4. Wichtige Kommandos	20
2. Daten laden und verwenden	21
2.1. Dateitypen	21
2.2. Formatierung von Textdateien	22
2.3. Laden und Verwenden	23
2.4. Datenanalyse	25
3. Erzeugen von Graphen	29
3.1. Typen von Graphen	29
3.2. Ausgabe	30
3.3. Verwendung	30
3.4. Koordinatensysteme	34
4. Tabellen: der Cache	37
4.1. Konzept	37
4.2. Erzeugen und Entfernen von Caches	37
4.3. Verwendung	38
4.4. Sortieren, Glätten und Resampeln	38

Inhaltsverzeichnis

4.5. Statistik-Funktionalität	40
5. NumeRe-Scripte	41
5.1. Konzept	41
5.2. Syntaxhervorhebung	41
5.3. Ein einfaches Script	41
 II. Fortgeschrittene Bedienung	 45
6. Eigene Funktionen	47
6.1. Definition	47
6.2. Bedingte Definition	48
6.3. Verwendung	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

RECHTLICHE INFORMATIONEN

GNU Free Documentation Licence v1.2

— Lizenz dieser Dokumentation —

Copyright © 2017, Erik Hänel.

Kopieren, Verbreiten und/oder Modifizieren ist unter den Bedingungen der GNU Free Documentation Licence, Version 1.2 oder einer späteren Version, veröffentlicht von der Free Software Foundation, erlaubt. Es gibt keine unveränderlichen Abschnitte, keinen vorderen Umschlagtext und keinen hinteren Umschlagtext. Eine Kopie des Lizenztextes ist unter [GNU Free Documentation Licence](#) zu finden.

GNU General Public Licence v3

— Lizenz des Programms und der gezeigten Codefragmente —

Copyright © 2017, Erik Hänel et al.

Das beschriebene Programm und die hier gezeigten Codefragmente sind freie Software. Sie können es unter den Bedingungen der GNU General Public License, wie von der Free Software Foundation veröffentlicht, weitergeben und/oder modifizieren, entweder gemäß Version 3 der Lizenz, oder (nach Ihrer Option) jeder späteren Version.

Die Veröffentlichung dieses Programms erfolgt in der Hoffnung, dass es Ihnen von Nutzen sein wird, aber OHNE IRGENDEINE GARANTIE, sogar ohne die implizite Garantie der MARKTREIFE oder der VERWENDBARKEIT FÜR EINEN BESTIMMTEN ZWECK. Details stehen in der GNU General Public Licence.

Sie sollten ein Exemplar der GNU GPL zusammen mit diesem Programm erhalten haben. Falls nicht, siehe www.gnu.org/licenses/.

Mitwirkende

- **Projektleitung:** Erik HÄNEL
- **Konzept/UI:** Erik HÄNEL, Chameleon Team
- **Mathe-Parser:** Ingo BERG, Erik HÄNEL (muParser)

Rechtliche Informationen

- **Plotting:** Alexey BALAKIN (MathGL)
- **numerische Algorithmen:** GNU Scientific Library, Erik HÄNEL, Alexey BALAKIN
- **Tokenizer:** Boost-Library
- **Matrix-Algorithmen:** Eigen-Library
- **XML-Parser:** Lee THOMASON (TinyXML-2)
- **Excel-(97-2003)-Funktionalität:** YAP CHUN Wei (BasicExcel)
- **Splines:** Tino KLUGE
- **Testing:** C. ALONSO, D. Bammert, J. HÄNEL, R. HUTT, K. KILGUS, E. KLOSTER, K. KURZ, M. LÖCHNER, L. SAHINOVĆ, D. SCHMID, V. SEHRA, G. STADELmann, R. WANNER, A. WINKLER, F. WUNDER, J. ZINSSER

INSTALLATION

Seit Version v1.0.7 »Bose« wird NUMERE in Form eines Installers auf [SourceForge](#) veröffentlicht. Um NUMERE auf dem eigenen Rechner zu installieren, muss der Installer heruntergeladen und ausgeführt werden. Nachdem man den entsprechenden Schritten gefolgt ist, ist NUMERE auf dem eigenen Rechner installiert und kann ausgeführt werden.

Um eine neuere Version von NUMERE zu installieren, muss (und sollte) die vorherige Version *nicht* deinstalliert werden. Die neuere Version kann direkt über die bereits existierende installiert werden.

Anmerkung Der Installer gibt als Standardpfad »C:/Software/NumeRe« vor. Dieser Pfad kann geändert werden, jedoch sollte auf »C:/Programme/...« bzw. »C:/Programme (x86)/...« verzichtet werden. An diesen Orten kann NUMERE unter Umständen nicht korrekt ausgeführt werden.

Systemvoraussetzungen

NUMERE kann auf allen Desktop-Windowsversionen von XP bis 10 ausgeführt werden. Für Windows 8.1 und 10 muss die zusätzliche Komponenten-Kompatibilität installiert werden, da es anderenfalls zu Abstürzen während des Plotvorgangs kommen kann.

Außerdem erfordert NUMERE eine Tastatur zur korrekten Ausführung. Die Bildschirmtastatur kann ggf. auch verwendet werden, ist aber weniger komfortabel.

Portable und Stable Version

Standardmäßig wird die Installation der Stable Version empfohlen, welche die Dateiverknüpfungen für NUMERE vornimmt. Dabei werden Administratorrechte vorausgesetzt. Die Portable Version kann auch ohne Administratorrechte ausgeführt werden, legt dabei aber keine Dateiverknüpfungen an. NUMERE-Portable kann aber (prinzipiell) von jedem Verzeichnis aus gestartet, beliebig verschoben und zum Deinstallieren einfach gelöscht werden.

Anmerkung Der »Stable Version«-Installer schreibt lediglich die Dateiverküpfungen und den Link zum Uninstaller in die Windows-Registry. Es werden keine Konfigurationen oder andere für die Programmausführung relevante Daten in die Registry geschrieben.

Teil I.

GRUNDLEGENDE BEDIENUNG

1. ERSTE SCHRITTE

Aller Anfang ist schwer. Das ist uns sehr wohl bewusst, daher haben wir hier die wesentlichen ersten Dinge zusammengetragen, die den Einstieg in NUMERE leichter gestalten sollen. Allerdings erhebt diese Dokumentation natürlich keinen Anspruch auf Vollständigkeit. Die Feinheiten aller Optionswerte werden hier ebenfalls nicht beschrieben. Eine Beschreibung derselben findet man in der integrierten Dokumentation.

Anmerkung Diese Dokumentation ist mit Marginalien ausgestattet, die auf Aktionen in NUMERE verweisen sollen. Marginalien in rot sind Syntaxelemente, die in die NUMERE-Konsole eingegeben werden können. Gerahmte Marginalien in dunkelblau bezeichnen Artikel der integrierten NUMERE-Hilfe.

1.1. Das User Interface

Das User Interface von NUMERE ist in fünf wesentliche Elemente unterteilt. Zentral sind hierbei die *Konsole* (Mitte unten) und der *Editor* (oben rechts). Mit der Konsole kann man direkt mit NUMERE interagieren und der Editor kann genutzt werden, um Textdateien, NUMERE-Scripte und NUMERE-Prozeduren zu bearbeiten. Daneben gibt es noch die *Eingabehistorie* (unten rechts), die alle Eingaben in die Konsole protokolliert, so dass man sie durch Drag'n'Drop oder per Doppelklick erneut ausführen kann. Zur Navigation in den Dateien existiert der *Dateibaum* (linke Sidebar, erster Tab) und zur Übersicht über Funktionen und Kommandos noch der *Symbolbaum* (linke Sidebar, zweiter Tab). Der Editor ist zusätzlich auch noch in Tabs organisiert, so dass mehrere Dateien zugleich geöffnet sein können.

Beim ersten Start zeigt der NUMERE-Editor eine Startseite an, die den Einstieg in die Applikation vereinfachen soll. Nutzer, die bereits ähnliche Software, wie z.B. MATLAB, kennen, werden sich mit diesen Informationen vermutlich schnell zurecht finden. Ein paar einleitende Worte werden wir dennoch anfügen.

Die Bedienung über die Konsole soll unser erstes Augenmerk sein. Am Anfang der Zeile erscheint in der Konsole ein Pfeil `<-`, der verdeutlichen soll, dass hier eine Eingabe erwartet wird ([Abbildung 1.1](#)). Ausgaben, die das Programm tätigt, werden alle durch den entgegen gerichteten Pfeil `->` begonnen. Es können *nur dann* Eingaben getätigter werden, wenn ein solcher Eingabepfeil (oder die explizite Aufforderung) erscheint.

In die NUMERE-Konsole können an dieser Stelle sowohl mathematisch-numerische Ausdrücke als auch Kommandos eingegeben werden. Die Syntax der mathematisch-numerischen Ausdrücke

1. Erste Schritte

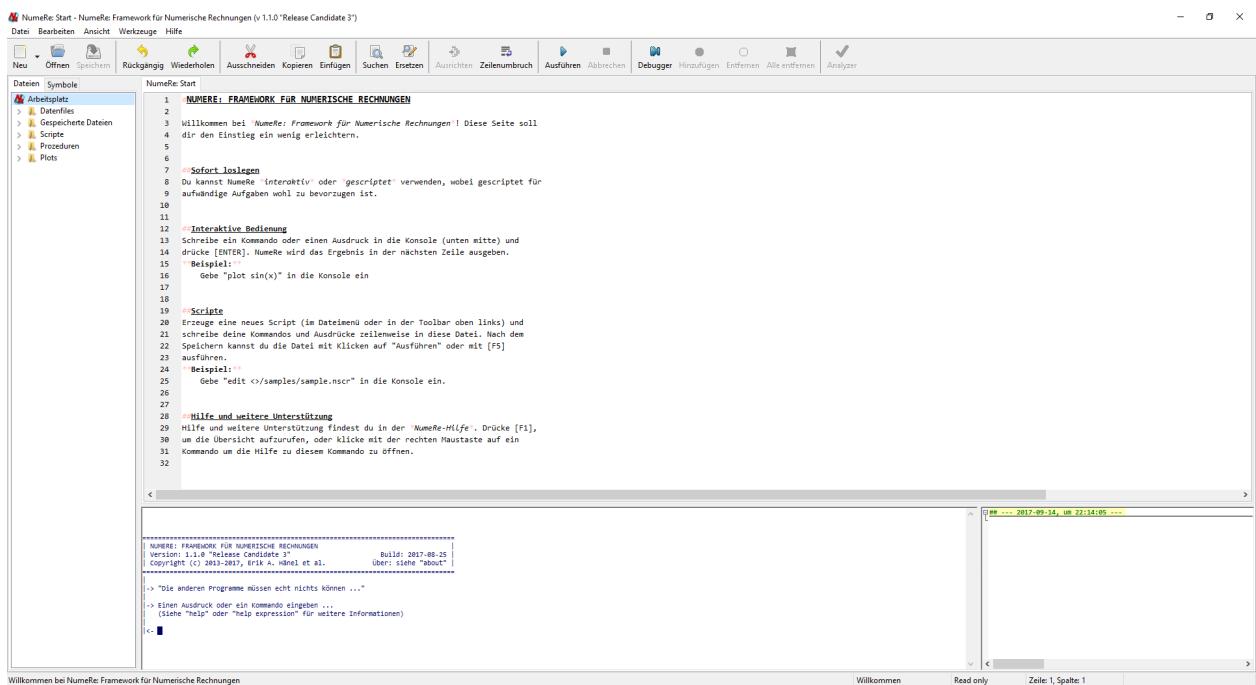


Abbildung 1.1.: Die Startansicht von NUMERE mit den erwähnten Interface-Elementen. Ebenfalls zu sehen ist die Startseite von NUMERE, die ein paar Basics erklärt

ist recht intuitiv und wird im übernächsten Abschnitt nochmals erläutert. Die Syntax der Kommandos bedarf hingegen einer kurzen Erläuterung.

Es wurde versucht, NUMERE so intuitiv wie irgend möglich zu gestalten. Diese Prämisse war auch für die Kommandosyntax vorgegeben, ist aber aufgrund der Tatsache, dass NUMERE ein »gewachsenes« Programm ist, an ein paar Stellen vielleicht nicht wirklich erfüllt. Dennoch halten sich die Kommandos alle an dasselbe Schema:

KOMMANDO [AUSDRUCK] [-PARAMETER[=WERT]] [OPTIONEN[=WERT]]]

Syntaxelemente in eckigen Klammern sind teils optional teils nicht bei jedem Kommando vorhanden. Beispiele für die Kommandosyntax sind

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

Hier ist zu sehen, dass (manche) Kommandos nicht zwangsläufig einen Ausdruck oder einen Parameter benötigen.

Diese Kommandosyntax wird durch den »Merksatz«

»Führe eine Aktion [auf etwas] aus und verwende dabei die folgenden Parameter.«

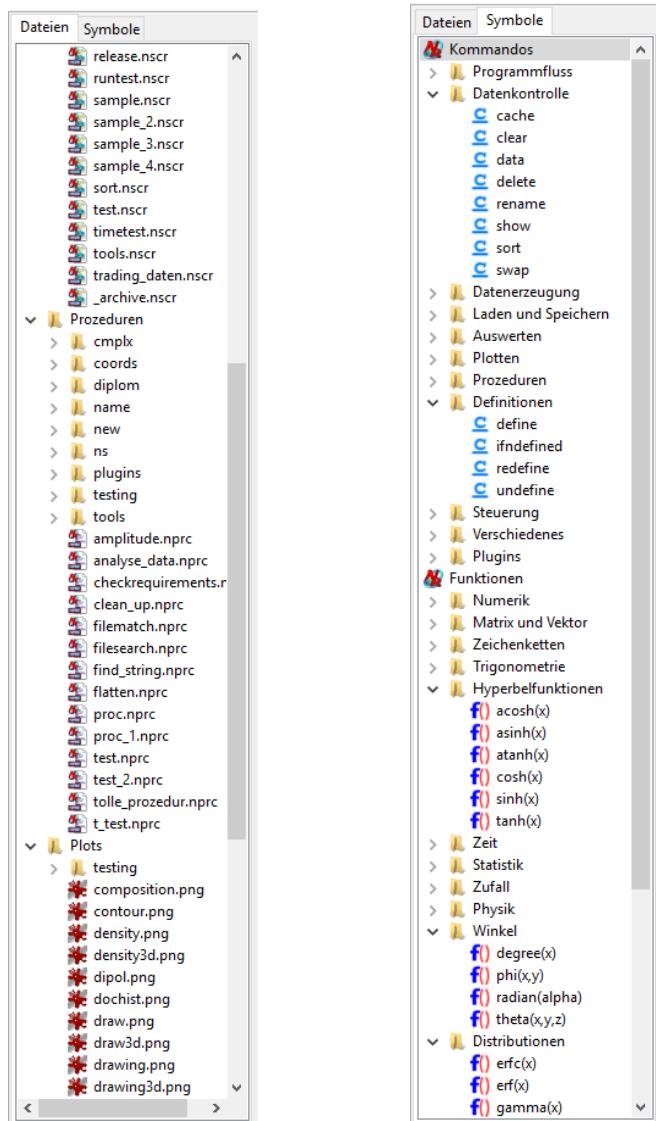


Abbildung 1.2.: Datei- und Symbolbaum im NUMERE-Interface

1. Erste Schritte

vermutlich etwas eingängiger. Auf die obigen Beispiele ausgeführt, könnte dies zum Beispiel

»Öffne die Hilfe, die sich mit »expression« befasst.«

oder

»Generiere einen Gitterplot der Funktion » $\sin(x)+\cos(y)$ « und verwende eine umfassende Box.«

lauten.

syntax

► In der NUMERE-Hilfe unter: [help syntax](#)

Alle Kommandos sind im Symbolbaum aufgelistet und dort in verschiedene Kategorien eingeteilt, so dass das nötige Kommando rasch gefunden werden kann. Wenn man mit der Maus über einem Kommando verweilt, so wird eine kurze Erklärung angezeigt. Sollte auch dies nicht weiterhelfen, so kann der zum Kommando gehörende Artikel der NUMERE-Hilfe aufgerufen werden, z.B. durch

help 1 [help plot](#)

In diesen Artikeln finden sich alle Parameter und Syntaxinformationen, sowie ein Beispiel der Syntax, die für die korrekte Ausführung nötig sind.

numere

► In der NUMERE-Hilfe unter: [help numere](#)

Sollte einmal eine Eingabe fehlerhaft sein, wird NUMERE eine entsprechende Meldung auf die Konsole ausgeben. Wenn eine mathematisch-numerischer Ausdruck fehlerhaft ist, wird die Position des Fehlers – sofern zutreffend – mit ausgegeben. Bei Fehlern in der Verwendung der Kommandosyntax gibt NUMERE neben der Meldung auch eine Referenz auf den entsprechenden Hilfearikel zurück.

Ausdrucks- und Kommandofehler werden sämtliche Auswertungen abbrechen. Demzufolge werden auch NUMERE-Scripte und NUMERE-Prozeduren mit Ausgabe der entsprechenden Meldung abgebrochen.

1.2. Einstellungen

Alle Einstellungen in NUMERE werden im Optionendialog getätigter, der im Werkzeuge-Menü zu finden ist. Zusätzlich kann man auch mit dem Kommando [set](#) Einstellungen setzen. Als Parameter muss dabei die Bezeichnung der Einstellung und der gewünschte Wert folgen:

set [set -EINSTELLUNG=WERT](#)

Falls der Wert der Einstellung Leerzeichen enthält (z.B. in einem Dateipfad), muss dieser mit umschließenden Anführungszeichen angegeben werden:

¹ [set -EINSTELLUNG="WERT MIT LEERZEICHEN"](#)

Syntaxelement Das [list](#)-Kommando kann eine Auflistung aller Einstellungen anzeigen:

list [list -settings](#)

Die geänderten Einstellungen werden zum Programmende hin gespeichert und sind bei einem Neustart wieder verfügbar.

1.3. Einfache Rechnungen

NUMERE kann als Framework für numerische Rechnungen natürlich numerisch rechnen. Die zu berechnenden Gleichungen können ähnlich wie in einen Taschenrechner eingegeben werden. Abstände zwischen Operatoren und Werten spielen natürlich keine Rolle, allerdings dürfen zwischen Funktionsnamen und ihren Argumentklammern (ebenso bei den späteren Tabellenobjekten) *keine Leerzeichen* sein. Daneben wird zwischen Groß- und Kleinschreibung unterschieden und der Malpunkt * muss stets mit eingegeben werden:

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

multipliziert $\cos 2\pi$ mit 5 und gibt das Ergebnis direkt aus. Als Dezimaltrennzeichen wird der Punkt . verwendet: die Zahl 5.2 ist in deutscher Dezimalschreibweise folglich identisch zu 5,2. _2pi ist dabei eine vordefinierte Konstante für 2π und cos() ist natürlich die Kosinus-Funktion.

Die vordefinierten Konstanten und Funktionen sind im Symbolbaum zu finden. Sie können aber auch mittels

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

aufgelistet werden. list -func kann noch weiter eingeschränkt werden.

► In der NUMERE-Hilfe unter: help list

list

NUMERE kann aber nicht nur mit Zahlen umgehen, sondern auch mit Variablen. Die Variablen x, y, z, t und ans sind bereits vordefiniert. Es können aber auch weitere Variablen definiert werden. Das geschieht entweder automatisch (sobald NUMERE auf eine unbekannte Variable stößt) oder durch eine direkte Zuweisung:

```
neue_variable = 5
```

Hier wird die neue Variable neue_variable mit dem Wert 5 definiert. Die obige Gleichung kann damit auch auf folgende Art geschrieben werden:

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

NUMERE kann auch mehrere Ausdrücke zugleich auswerten. Die Ausdrücke müssen dabei durch jeweils ein Komma , getrennt sein. Innerhalb dieser Ausdrücke wird von links nach rechts vorgegangen. Die Zeile

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

setzt neue_variable auf 5, berechnet $5 \cos 2\pi$ und setzt neue_variable anschließend auf 1 (Abbildung 1.3). In Schleifen kann diese Schreibweise erhebliche Geschwindigkeitsvorteile bringen.

► In der NUMERE-Hilfe unter: help expression

expression

1. Erste Schritte

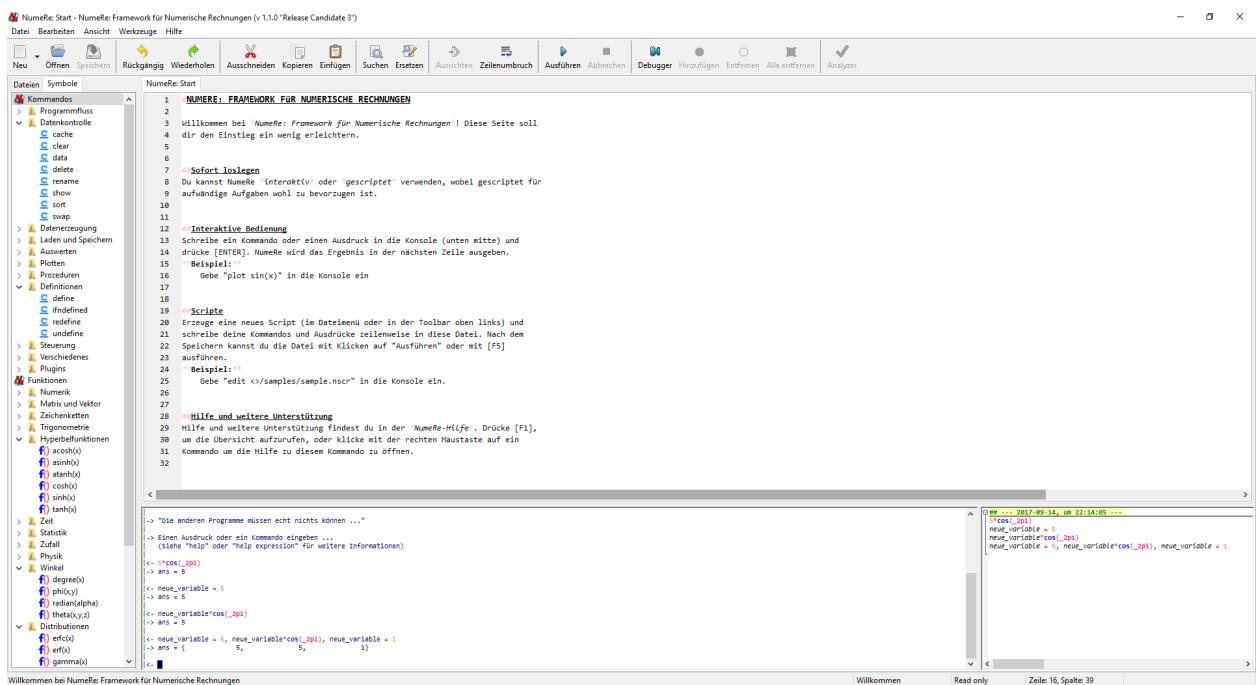


Abbildung 1.3.: NUMERE nach Eingabe der letzten Rechnung. Zu beachten ist, dass die Eingaben auch in der Eingabehistorie protokolliert wurden

1.4. Wichtige Kommandos

Syntaxelement NUMERE wird unter anderem durch Kommandos bedient. Wichtige Kommandos sind hierbei

```
help  
find  
list 2 help THEMA  
quit find BEGRIFFE  
4 list -OBJEKT
```

Diese Kommandos rufen die integrierte Dokumentation ([help](#) bzw. [help](#) THEMA zum THEMA), die Stichwortsuche ([find](#) BEGRIFFE) auf und listen OBJEKTE ([list](#) -OBJEKT).

Alle Kommandos haben einen Eintrag in der NUMERE-Hilfe. Dieser Artikel kann durch die Eingabe von [help](#) KOMMANDO oder die Verwendung des entsprechenden Eintrags im Kontextmenü angezeigt werden und enthält alle Informationen zu einem Kommando und ggf. noch Referenzen auf weitere wichtige Dokumentationsartikel. Im Falle, dass eine Information einmal nicht direkt in der NUMERE-Hilfe gefunden werden kann, kann die Stichwortsuche ([find](#)) große Dienste erweisen, da selbige auch direkt passende Artikel der NUMERE-Hilfe liefert.

Anmerkung Mittels Semikola »;« können auch mehrere Kommandos und mehrere Ausdrücke, die nacheinander auszuwerten sind, gemeinsam in einer Zeile angegeben werden:

```
KOMMANDO1; KOMMANDO2; AUSDRUCK1; KOMMANDO3; AUSDRUCK2; ...
```

2. DATEN LADEN UND VERWENDEN

Da ein großer Teil der wissenschaftlichen Arbeit auf der Datenanalyse beruht, ist es notwendig, Programme zu verwenden, die entsprechende Datenmengen in einem vernünftigen Zeitrahmen laden und verarbeiten können. NUMERE interpretiert alle gelesenen Daten in einer tabellarischen Darstellung und kann Statistiken, Histogramme und fortgeschrittenere Auswertungen durchführen.

2.1. Dateitypen

Neben reinen Textdateien im ANSI-Format (Dateiendungen *.txt und *.dat) kann NUMERE die folgenden Dateiformaten lesen:

- **CassyLab-Datei (*.labx).** Dieser Datentyp wird nur von CassyLab verwendet und enthält ein komplettes CassyLab-Experiment. NUMERE wird jedoch nur die tabellarischen Daten extrahieren.
- **CSV-Datei (*.csv).** Comma-Separated-Values-Dateien sind ein Quasi-Standard zum Austauschen von Daten. Allerdings existiert kein »CSV-Standard«, so dass es trotz umfassenden Tests vorkommen kann, dass eine CSV-Datei nicht korrekt interpretiert werden kann. (Der Programmierer freut sich über eine entsprechende Bugmeldung inkl. der angehängten Datei.)
- **JCAMP-DX-Datei (*.dx, *.jdx und *.jcm).** JCAMP-DX-Dateien (von *Joint Committee on Atomic and Molecular Physical data – Data eXchange*) sind ein Dateistandard für den Austausch von Spektroskopie-Daten. NUMERE wird auch aus diesen Dateien nur die tabellarischen Daten extrahieren.
- **OpenDocument-Spreadsheet (*.ods).** Tabellarische Daten, die z.B. von OpenOffice Calc erzeugt wurden. NUMERE wird hier nur die numerischen Werte und die Zeichenketten, jedoch nicht die Gleichungen lesen. Zu beachten ist, dass OpenOffice Calc geöffnete Dateien *lockt*: Spreadsheets, die in OpenOffice Calc geöffnet sind, können *nicht* von NUMERE gelesen werden.
- **Excel-Workbooks (*.xls und *.xlsx).** Tabellarische Daten, die z.B. von Excel erzeugt wurden. NUMERE wird hier nur die numerischen Werte und die Zeichenketten, jedoch nicht die Gleichungen lesen. Eine Einschränkung des alten Excelformates ist zusätzlich, dass Ergebnisse, die durch Gleichungen berechnet wurden, ebenfalls nicht gelesen werden können. Zu

2. Daten laden und verwenden

beachten ist außerdem, dass Excel wie OpenOffice Calc geöffnete Dateien *lockt*: Workbooks, die in Excel geöffnet sind, können *nicht* von NUMERE gelesen werden.

- **IGOR Binary Waves (*.ibw).** IGOR Binary Waves enthalten die Daten einer (1-, 2- oder 3-dimensionalen) IGOR-Welle.
- **NumeRe-Datenfile (*.ndat).** NumeRe-Datenfiles sind ein NUMERE-eigenes, binäres Dateiformat zum schnellen Speichern und Lesen der Datensätze.

load

► In der NUMERE-Hilfe unter: [help load](#)

NUMERE kann jedoch nicht alle Formate schreiben. Als Ausgabeformate stehen die folgenden zur Verfügung:

- **Textdatei (*.dat oder *.txt).** NUMERE erzeugt eine Textdatei in ANSI-Codierung, die dem Formatierungsstandard des folgenden Abschnitts genügt. Diese Textdateien sollten von den meisten anderen Programmen gelesen werden können
- **NumeRe-Datenfile (*.ndat).** Dieses Dateiformat kann (nur) von NUMERE gelesen werden. Das Dateiformat ist jedoch online dokumentiert und kann in andere Programme implementiert werden.
- **CSV-Datei (*.csv).** NUMERE erstellt eine CSV-Datei mit Kommata , als Spalten- und Punkten . als Dezimaltrennzeichen. (Möglichlicherweise kann dies von gewöhnlichen Tabellenkalkulationen nicht korrekt gelesen werden: das Ersetzen der Kommata durch Semikola ; und der Punkte durch Kommata behebt dieses Problem gegebenenfalls. Der Grund hierfür entzieht sich dem Programmierer.)
- **Excel Workbook (*.xls).** NUMERE kann die Daten im alten Excel-(97-2003)-Format (*.xls) exportieren, so dass sie in Excel und Konsorten weiterverarbeitet werden können.
- **TeX-Datei (*.tex).** NUMERE schreibt eine Tabelle im TeX-Format. In den Kommentaren der erzeugten Datei finden sich die Voraussetzungen, um die erzeugte Tabelle in ein TeX-Dokument einzubinden (*booktabs*- und *longtable*-Packages). Es werden ebenfalls Punkte als Dezimaltrennzeichen verwendet.

save

► In der NUMERE-Hilfe unter: [help save](#)

2.2. Formatierung von Textdateien

Obwohl NUMERE in der Konsole den Punkt . als Dezimaltrennzeichen erfordert, ist dies bei Dateien nicht erforderlich. NUMERE kann sogar Dateien laden, die den Punkt und das Komma gemischt verwenden. Intern werden die Kommata in einen Punkt umgewandelt.

Textdateien müssen allerdings auf jeden Fall tabellarisch formatiert sein. Es spielt hier jedoch keine Rolle, ob die Spalten durch Leerzeichen, Tabulatoren oder einer Mischung derselben getrennt sind.

Text innerhalb von tabellarischen Daten wird ignoriert. Soll eine Spaltenüberschrift angegeben werden, muss dies in der letzten Zeile vor den eigentlichen Daten geschehen. Leerzeichen in einem einzelnen Spaltenkopf müssen durch einen Unterstrich _ ersetzt werden. Text, der als Kommentar dienen soll, sollte durch ein # am Anfang der Zeile auskommentiert werden. (Auch Spaltenüberschriften können so auskommentiert werden. Sie werden trotzdem gefunden.) Zwischen der Kopfzeile und den tabellarischen Werten kann auch noch eine trennende Linie aus Gleichheitszeichen = verwendet werden.

```
# KOMMENTAR
# KOMMENTAR
# KOPF_1  KOPF_2  KOPF_3 [...]
# =====[...]
0,225      12      0 [...]
0,245      12.5    .5 [...]
```

► In der NUMERE-Hilfe unter: [help data](#)

[data](#)

2.3. Laden und Verwenden

Dateien in den oben genannten Dateiformaten können geladen werden durch Drag'n'Drop in die Konsole, den entsprechenden Eintrag im Kontextmenü des Dateibaums und durch

[load DATEIPFAD/DATEI.EXT](#)

Syntaxelement
[load](#)

Dateipfade und -Namen mit Leerzeichen müssen von Anführungzeichen umgeben sein. Der hier gezeigte DATEIPFAD kann dabei auch weggelassen werden, wenn die Datei sich im Standardordner <loadpath> (Standardmäßig der Unterordner »data« im NUMERE-Stammverzeichnis) befindet. Wird die Dateierweiterung .EXT weggelassen, verwendet NUMERE die erste Datei, deren Name mit DATEI übereinstimmt und bestimmt den Dateityp selbst. In der NUMERE-Hilfe finden sich unter [help load](#) noch weitere Informationen.

In einer Standard-NUMERE-Installation befinden sich Beispieldateien im Unterordner »samples«. Diese können durch z.B.

[load <>/samples/data](#)

geladen werden. <> ist der Pfadplatzhalter, der auf das NUMERE-Stammverzeichnis weist. Diese Zeile lädt die Datei »data.dat« in NUMERE's Speicher. Die Daten dieser Datei sind nun in tabellarischer Form im Datenobjekt data() abgelegt und können nun verwendet werden (Es ist reiner Zufall, dass die Datei und das Datenobjekt denselben Namen haben. Geladene Daten befinden sich immer in data()).

Die tabellarische Darstellung der geladenen Daten kann man mittels des Kommandos [show](#) erhalten:

Syntaxeelement
[show](#)

2. Daten laden und verwenden

```
show data()
```

Die Tabelle wird daraufhin in einem externen Fenster angezeigt.

Der Zugriff auf die Daten in `data()` geschieht mittels der *Bereichs- oder Intervallsyntax* `a:b` (für Werte von `a` bis einschl. `b`). Dabei werden die Zeilen- und Spaltenbereiche angegeben, aus denen die Werte entnommen werden sollen und den Argumentklammern von `data()` übergeben:

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

In diesen Beispielen wird ein einzelnes Element (`data(3,1)` für dritte Zeile, erste Spalte), eine ganze Spalte (`data(:,1)` für alle Zeilen, erste Spalte), ein Spaltenausschnitt (`data(4:55,4)` für vierte bis fünfundfünfzigste Zeile, vierte Spalte) oder ein Zeilenausschnitt (`data(3,3:)` für dritte Zeile, dritte bis letzte Spalte) referenziert.

In Rechnungen können die Daten entweder nur zeilen- oder nur spaltenweise extrahiert werden, d.h., die Bereichssyntax darf entweder für in Zeilen oder nur für Spalten verwendet werden. Die Rechnung mit Untertabellen ist hier nicht, jedoch mittels Matrix-Operationen möglich, die aber in einem späteren Kapitel besprochen werden. Aber natürlich kann ein Ausdruck mehrmals Elemente aus `data()` verwenden:

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

Bei dieser Rechnung werden nun so viele Ergebnisse zurückgegeben, wie der längste Bereich Elemente umfasst. Es gibt aber auch ein paar Funktionen, die beliebig viele Elemente aufnehmen können und nur einen einzelnen Wert als Ergebnis ausgeben:

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

Diese Funktionen berechnen Statistiken der Elemente (Mittelwert, Median, Standardabweichung, Minimum und Maximum), Summieren oder Multiplizieren alle Elemente, suchen Elemente, zählen Elemente oder wandeln die Werte in ASCII-Zeichencodes um (die Verwendung von Zeichenketten wird im Abschnitt »Fortgeschrittene Bedienung« erläutert und soll hier nicht weiter von Belang sein).

Anmerkung Nutzer, die von Matlab oder Octave kommen, werden die Unterscheidung zwischen OPERATOR und .OPERATOR kennen. Dies existiert in NUMERE nicht, da NUMERE vollständig als Tabellenkalkulation ausgelegt ist. Elemente aus verschiedenen Spalten/Zeilen werden also immer elementweise verarbeitet und nicht gemäß einer Matrix-/Vektoralgebra (eine Matrix-Matrix- oder Matrix-Vektor-Multiplikation wird in diesem Framework im Rahmen des `matop`-Kommandos durch den Operator `**` dargestellt).

Manche Kommandos, wie `fit`, `fft` und `plot` (sowie Konsorten) können auch mit Untertabellen umgehen. Bei diesen Kommandos darf die Bereichssyntax sowohl in Spalten als auch Zeilen auftreten.

Die Tabelle in `data()` ist ein sogenannter *Read-Only*-Datensatz. Dies bedeutet, dass die Daten in dieser Tabelle nicht beschrieben oder anderweitig geändert werden können. In einem späteren Kapitel werden Caches als manipulierbare Tabellen besprochen.

► In der NUMERE-Hilfe unter: [help data](#)

`data`

2.4. Datenanalyse

Die Datenanalyse ist wohl einer der Hauptgründe, sich nach einem schnell und einfach zu bedienenden Numerikprogramm umzusehen. NUMERE bietet viele vordefinierte Analysefunktionen, die schnell und größtenteils unkompliziert eingesetzt werden können. Die Bedürfnisse des gewöhnlichen Statistikers sollten damit hinreichend erfüllt sein.

Statistiken können entweder kompakt durch das Kommando [stats](#)

Syntaxelement
`stats`

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

oder einzeln durch die im vorherigen Abschnitt genannten Statistikfunktionen bestimmt werden:

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

Das Kommando [stats](#) gibt dabei nahezu alle sinnvollen Statistikwerte der Tabelle aus, u.A. Mittelwert, Standardabweichung und -fehler, RMS, Schiefe, Exzess und Student-Faktor (für einen zweiseitigen, 95 %-Konfidenzbereich). Die meisten darunter können auch als einzelne Funktionen aufgerufen werden.

► In der NUMERE-Hilfe unter: [help stats](#)

`stats`

Eine weitere verbreitete Analysemethode ist die Erzeugung eines Histogramms. Histogramme sind eine Visualisierung der Häufigkeiten einer gemessenen Größe. Dies kann die Periodendauer einer Schwingung, die Lebensdauern von Elementarteilchen oder eine andere Größe sein.

Histogramme der geladenen Daten erhält man durch

Syntaxelement
`hist`

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

Das Kommando [hist](#) unterstützt dabei eine große Zahl an Parametern, die das erzeugte Histogramm noch weiter beeinflussen können, aber bereits ohne Parameter werden gute Ergebnisse erreicht ([Abbildung 2.1](#)). Die Zahl der verwendeten Rubriken (auch *bins*) wird durch die *Sturges-Regel* bestimmt, kann aber auch direkt oder durch die *Scott-* oder die *Freedman-Diaconis-Regel* festgelegt werden.

Es wird für jede Spalte eine Häufigkeitsanalyse durchgeführt und das Ergebnis in einem gemeinsamen Diagramm dargestellt. Spalten, die keine für eine Häufigkeitsanalyse sinnvollen Daten enthalten, sollten durch die Bereichssyntax ausgeschlossen werden.

NUMERE kann auch Histogramme von zweidimensionalen Datensätzen anfertigen. Hierzu sind entweder die Option `grid` für ein einzelnes Histogramm oder das Kommando [hist2d](#) für Histogramme in zwei Raumrichtungen nötig. Details finden sich in dem entsprechenden Hilfearikel.

Syntaxelement
`hist2d`

2. Daten laden und verwenden

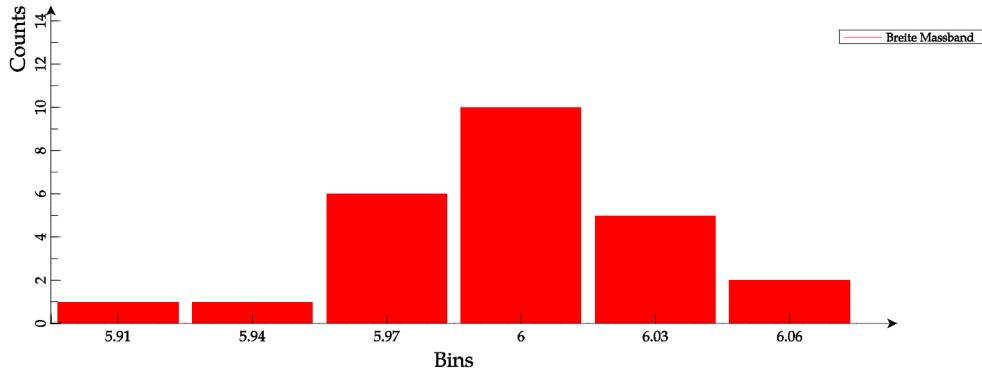


Abbildung 2.1.: Histogramm eines Datensatzes erzeugt durch [hist](#)

[hist](#)

► In der NUMERE-Hilfe unter: [help hist](#)

Neben den Histogrammen und der Berechnung der Statistiken ist eine weitere, übliche Analysemethode durch das Anpassen einer Funktion (auch *Modellkurve* oder *Regressionsfunktion* genannt) an die gemessen (und ggf. zusätzlich aufbereiteten) Daten gegeben.

Syntaxelement

Um eine Funktion FUNCTION() an Daten in data() anzupassen kann [fit](#) verwendet werden:

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

NUMERE wird nun versuchen, die Parameter PARAMS so zu variieren, dass FUNCTION() die Datenpunkte möglichst treffend beschreibt. Die angepassten Werte werden in den Parametern hinterlegt, so dass mit diesen direkt weiter gerechnet werden kann. (In neueren NUMERE-Versionen ist [params](#) nicht mehr zwingend erforderlich, da die Parameter auch automatisch erkannt werden können.)

Sobald NUMERE ein Minimum erreicht hat, bricht es den Algorithmus ab und gibt eine Übersicht aller Parameter aus, die im Folgenden beispielhaft dargestellt ist:

Funktion: $0.646875 \cdot \sin(3.00223 \cdot x + 0.00837336)$
Datenpunkte: 101 ohne Gewichtungsfaktoren
Freiheitsgrade: 98
Algorithmusparameter: TOL=0.0001, MAXITER=500
Iterationen: 7
Gewichtete Summe der Residuen (chi^2): 2.33825
Varianz der Residuen (red. chi^2): 0.0238597
Standardabweichung der Residuen: 0.1544658

Parameter	Initialwert	Anpassung	Asymptotischer Standardfehler
-----------	-------------	-----------	-------------------------------

A	1	0.6468754	\pm 0.02188576	(3.383%)
B	3	3.002228	\pm 0.005649476	(0.1882%)
C	0	0.008373364	\pm 0.06579646	(785.8%)

Korrelationsmatrix der angepassten Parameter:

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

Fitanalyse:

Die angepasste Funktion kann den Verlauf der Datenpunkte beschreiben, jedoch ist noch Raum für Optimierungen.

Diese Übersicht enthält den bedeutenden χ^2 -Wert, der die Summe der quadratischen Abweichungen der Datenpunkten von der angepassten Funktion beschreibt. Außerdem finden sich hier die Ergebniswerte der Parameter und die berechneten Fehlerwerte, die in eine daraus erfolgende Fehlerfortpflanzung eingefügt werden können. Die Korrelationsmatrix der Parameter beschreibt, wie sehr die Parameter voneinander abhängen und die Fitanalyse fasst in Worte, was aus χ^2 bzw. dem reduzierten χ^2 herausgelesen werden kann. Diese Übersicht wird des Weiteren in <savepath>/numerefifit.log protokolliert.

Anmerkung Inwiefern der χ^2 -Wert eine Aussage über die Qualität einer Anpassung ist, ist eine Streitfrage. Im Allgemeinen nimmt man an, dass ein Fit mit einem möglichst kleinen χ^2 -Wert besonders gut ist. Genauer betrachtet ist die aussagekräftigere Größe der reduzierte χ^2 -Wert: für ihn gilt, dass wenn er deutlich kleiner als 1 ist, scheint die Anpassung gut zu treffen. Bei einem Fit, der Fehlerwerte berücksichtigt (s.u.), sollte dieser Wert jedoch möglichst nahe an 1 liegen. NUMERE berücksichtigt dies und liefert eine entsprechende Fitanalyse. Die Fitanalyse kann allerdings den optischen Vergleich von angepasster Funktion und Datenpunkten (siehe nächstes Kapitel) nicht ersetzen.

Im Anschluss an die Anpassung kann die angepasste Funktion zusammen mit den Daten mittels des Kommandos `plot` (siehe nächstes Kapitel) graphisch dargestellt werden (Abbildung 2.2). Dies bietet eine gute Möglichkeit, das Ergebnis der Anpassung zu überprüfen (und sollte eigentlich auch stets ausgeführt werden).

Eine Anpassung an Datenpunkte, die mit Fehlern versehen sind, kann diese berücksichtigen. Dazu verwendet man

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

`fitw` (für *weighted fit*) verwendet die Fehlerwerte in der dritten Spalte um die Datenpunkte entsprechend zu gewichten: Datenpunkte mit hohen Fehlern werden schwächer gewichtet als Datenpunkte mit geringen Fehlern.

Syntaxelement
`fitw`

2. Daten laden und verwenden

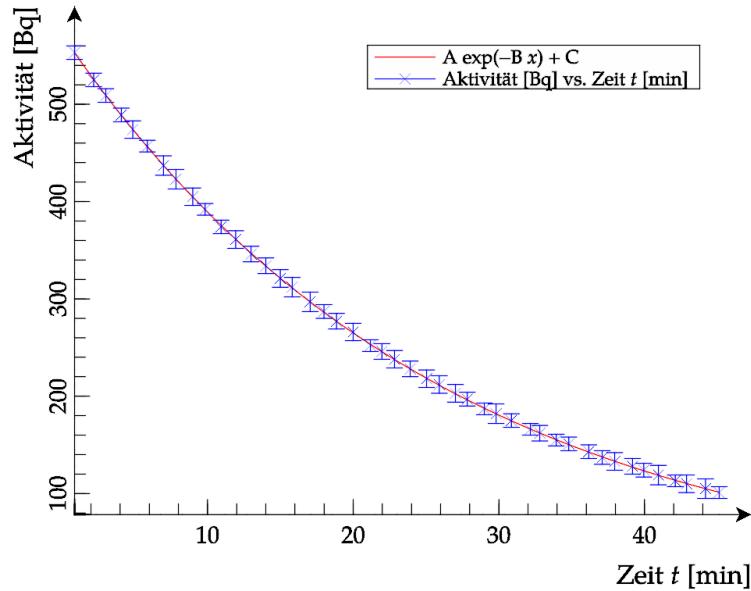


Abbildung 2.2.: Anpassung des Modells eines exponentiellen Zerfalls an Messpunkte mittels [fitw](#)

Sollte ein Fit mal nicht konvergieren, kann es helfen, die Initialwerte der Parameter (die stets angegeben werden sollten) selbst zu variieren. Ebenfalls kann es hilfreich sein, den Fitbereich einzuzgrenzen. Dies kann entweder durch eine Einschränkung der Zeilen in `data()` oder durch Übergabe der Option `x=a:b` erreicht werden:

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

Zusätzlich können die Zahl der maximalen Iterationen, die Genauigkeit und einzuhaltende Bedingungen für die angegebenen Parameter vorgegeben werden. Letztere Option kann die Stabilität des Algorithmus allerdings drastisch beeinflussen.

`fit`

► In der NUMERE-Hilfe unter: [help fit](#)

Syntaxelement Alle erwähnten Kommandos agieren spaltenweise. Falls die Daten zeilenweise vorliegen, muss man die folgende Zeile ausführen

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

und alle `data` in den vorherigen Beispielen durch `cache` ersetzen.

3. ERZEUGEN VON GRAPHEN

Eine besonders wichtige Funktionalität von NUMERE ist die Erzeugung von Funktions- und Datengraphen. Hiermit können die Verläufe von Funktionen und Daten visualisiert und damit einfacher analysiert werden. NUMERE gibt alle generierten Darstellungen automatisch als Bilddatei aus. Wenn NUMERE mit einem Bildbetrachter verknüpft ist, wird dieser im Anschluss an den Plotvorgang automatisch mit der Bilddatei gestartet.

3.1. Typen von Graphen

NUMERE kennt eine große Zahl unterschiedlicher Plotarten, die durch die Angabe weiterer Plotoptionen noch auf unzählige weitere Arten modifiziert werden können. Hier sollen nur die wichtigsten aufgezählt werden, der komplette Satz an Plottypen ist unter [list -cmd](#) zu finden:

```
plot
2 plot3d
mesh
4 dens
vect
6 vect3d
...
```

- [plot](#): dies erzeugt einen Standardgraphen, bei dem y gegen x aufgetragen ist. Dieses Kommando wird verwendet, um z.B. $\sin x$ oder x^2 graphisch darzustellen.
- [plot3d](#): dieses Kommando generiert einen Graphen auf Basis einer 3D-Trajektorie. Aus drei Funktionen mit der Variable t wird eine Bahnkurve berechnet und diese dreidimensional dargestellt.
- [mesh](#): eine Funktion $z = f(x, y)$ kann mit diesem Kommando dargestellt werden. Es wird ein Gitterplot berechnet und dreidimensional dargestellt.
- [dens](#): im Gegensatz zu [mesh](#) stellt dieses Kommando die Funktion $z = f(x, y)$ nur durch reine Farbwerte als Projektion auf die x - y -Ebene dar.
- [vect](#): dieser Plotstil berechnet einen Vektorplot eines 2D-Vektorfeldes: $\vec{A}(x, y) = A_x(x, y) \hat{e}_x + A_y(x, y) \hat{e}_y$.

3. Erzeugen von Graphen

- `vect3d`: ähnlich wie `vect` berechnet dies einen Vektorplot, allerdings den eines dreidimensionalen Vektorfeldes: $\vec{A}(x, y, z) = A_x(x, y, z)\hat{e}_x + A_y(x, y, z)\hat{e}_y + A_z(x, y, z)\hat{e}_z$.

3.2. Ausgabe

Das standardmäßige Ausgabe für Plots ist der NUMERE-GraphViewer, in dem Plots noch zusätzlich modifiziert werden können. Zusätzlich können Plots aber auch direkt in eine Bilddatei im PNG-Format im Verzeichnis `<plotpath>` (Dies ist standardmäßig der Unterordner »plots« im NUMERE-Stammverzeichnis) exportiert werden. Dies kann mit den Plotoptionen `opng`, `oeps`, `ogif`, `osvg` und `otex` erreicht werden.

Um einen einfachen Plot in z.B. »graph_der_messung.png« zu erzeugen, ist die Option

```
opng=graph_der_messung
```

zu übergeben. Sollte der Dateiname Leerzeichen enthalten, muss dieser in Anführungszeichen angegeben werden.

`plotoptions`

► In der NUMERE-Hilfe unter: [help plotoptions](#)

3.3. Verwendung

Alle Plotbefehle folgen in ihrer Syntax dem folgenden Schema:

1 KOMMANDO FUNKTIONEN/DATEN - `set` OPTIONEN

wobei die OPTIONEN optional sind und nicht angegeben werden müssen.

Standardvariablen sind x, y, z und t . Je nach Plotkommando wird eine, zwei, drei oder alle vier dieser Variablen als Variablen verwendet und die restlichen als Parameter betrachtet. `plot` verwendet nur x , `plot3d` nur t , `mesh`, `dens` und `vect` x und y und `vect3d` x, y und z . Alle anderen Variablen werden als Parameter betrachtet.

Syntaxelement

Ein einfacher Sinus-Plot wird durch

```
plot sin(x)
```

erreicht. Hier wird automatisch ein Achsenkreuz und ein Sinus von -10 bis 10 berechnet, wobei die y -Achse passend, jedoch ein kleines Stück größer gewählt wird. Die Achsenbeschriftung und die Legende wird ebenfalls selbstständig bestimmt ([Abbildung 3.1a](#))

`plot`

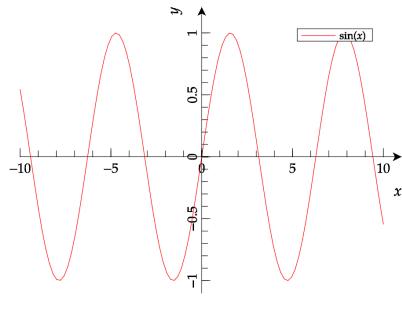
► In der NUMERE-Hilfe unter: [help plot](#)

Eine umfassende Box und ein Koordinatengitter wird erreicht durch

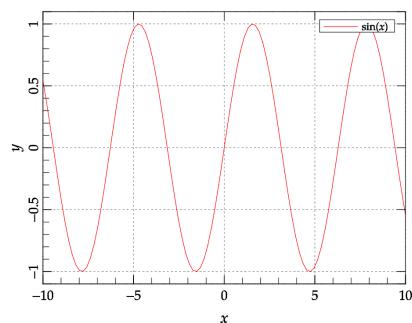
```
plot sin(x) -set box grid
```

Ab diesem Plot werden alle folgenden Plots ebenfalls mit Gitter und umfassender Box ausgestattet ([Abbildung 3.1b](#) und [c](#)).

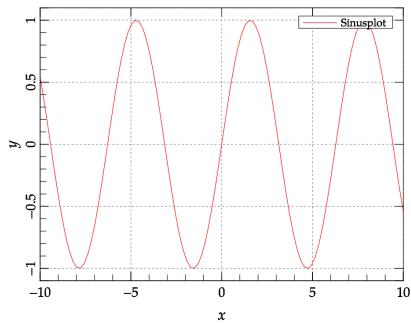
Möchte man außerdem noch »Sinusfunktion« statt $\sin(x)$ als Legende haben, gibt man folgendes an:



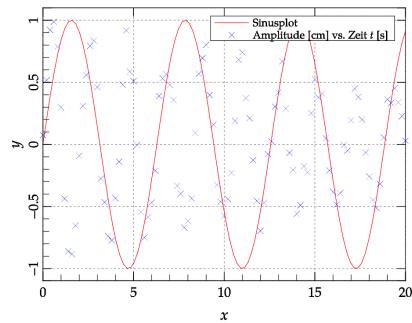
a: ohne Optionen



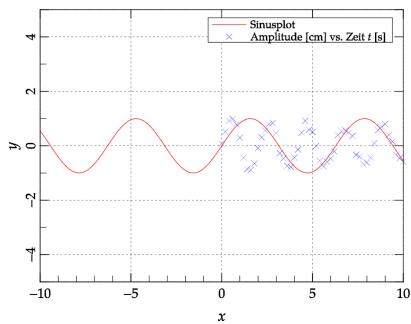
b: mit `box` und `grid`



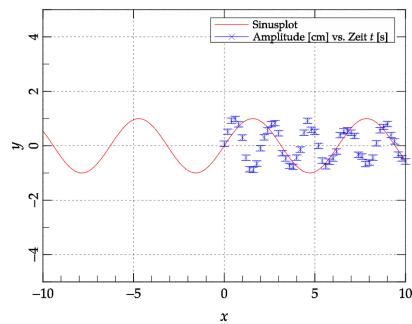
c: mit eigener Legende



d: mit Datensatz



e: mit eigenen Intervallgrenzen



f: mit Fehlerbalken

Abbildung 3.1.: Die Ergebnisgraphen unter Verwendung der gezeigten Optionen

3. Erzeugen von Graphen

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

(Abbildung 3.1c; Eine leere Legende erreicht man durch Angabe einer leeren Zeichenkette „“)

NUMERE kann auch Datensätze graphisch darstellen. Dazu gibt man den Datensatz in `data()` analog zu einer Funktion an. Bleibt die Argumentklammer leer, wird dies automatisch durch `data(:, :)` ersetzt. Es können aber auch direkt die nötigen Spalten angegeben werden, z.B. `data(:, 1:4)`. Dies verwendet entweder Spalte 1 und 4 oder Spalte 1 bis 4, falls der entsprechende Plotstil mehr als zwei Spalten benötigt. Es können hierbei bis zu 6 einzelne Spalten in beliebiger Reihenfolge angegeben werden: `data(:, 4:2:6:1:3:8)`.

In Verbindung mit dem vorherigen Beispiel könnte man Spalten 1 und 2 und den Sinus zusammen durch

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

darstellen (Abbildung 3.1d). Auch für `data()` kann eine eigene Legende angegeben werden. Andernfalls wird eine Kombination aus den Spaltentiteln verwendet. Interessant ist des Weiteren, dass mit Einbringen des Datensatzes die x -Achse passend zum Datensatz gewählt wird und nicht mehr zwangsläufig von -10 bis 10 verläuft. Dazu wird auffallen, dass die NUMERE Datenpunkte als einzelne Punkte darstellt und sie nicht durch eine (unphysikalische) Linie verbindet.

Die Plotintervalle können für einen Plot global überschrieben werden, wenn sie explizit angegeben werden:

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

Dies erzeugt einen Graphen mit dem x -Intervall $[-10; 10]$ und dem y -Intervall $[-5; 5]$ (Abbildung 3.1e). Diese Option wird *nicht* für alle folgenden Plots übernommen.

Messungen von Datenpunkten sind in vielen Fällen mit Fehlern belastet. Sind diese bekannt, so kann NUMERE diese auch darstellen. Sind nur die y -Werte mit Fehlern behaftet, so benötigt NUMERE drei Spalten ($x, y, \Delta y$), bei Fehlern in beide Richtungen sind es vier ($x, y, \Delta x, \Delta y$). Die dazu nötige Plotoption heißt `errorbars` bzw. `yerrorbars`, wenn nur y -Fehler vorhanden sind.

Nehmen wir nun an, dass die Datenpunkte in obigem Beispiel mit y -Fehlern behaftet sind:

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

Es erscheinen Fehlerbalken in y -Richtung, obwohl die Angabe der Spaltenzahl nicht geändert wurde! NUMERE interpretiert die leere Argumentklammer nun automatisch um und verwendet die ersten drei Spalten (Abbildung 3.1f).

plotoptions

► In der NUMERE-Hilfe unter: [help plotoptions](#)

Syntaxelement **mesh** In einem anderen Fall soll eine zweidimensionale Funktion durch ein Gitterplot dargestellt werden: ein Sinus Cardinalis von ϱ ($\text{sinc } \varrho$):

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

`mesh` erzeugt den gewünschten Gitterplot (Abbildung 3.2a). Die Funktion `norm()` ist die n -dimensionale euklidische Vektornorm (diese Funktion kann beliebig viele Argumente aufnehmen):

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

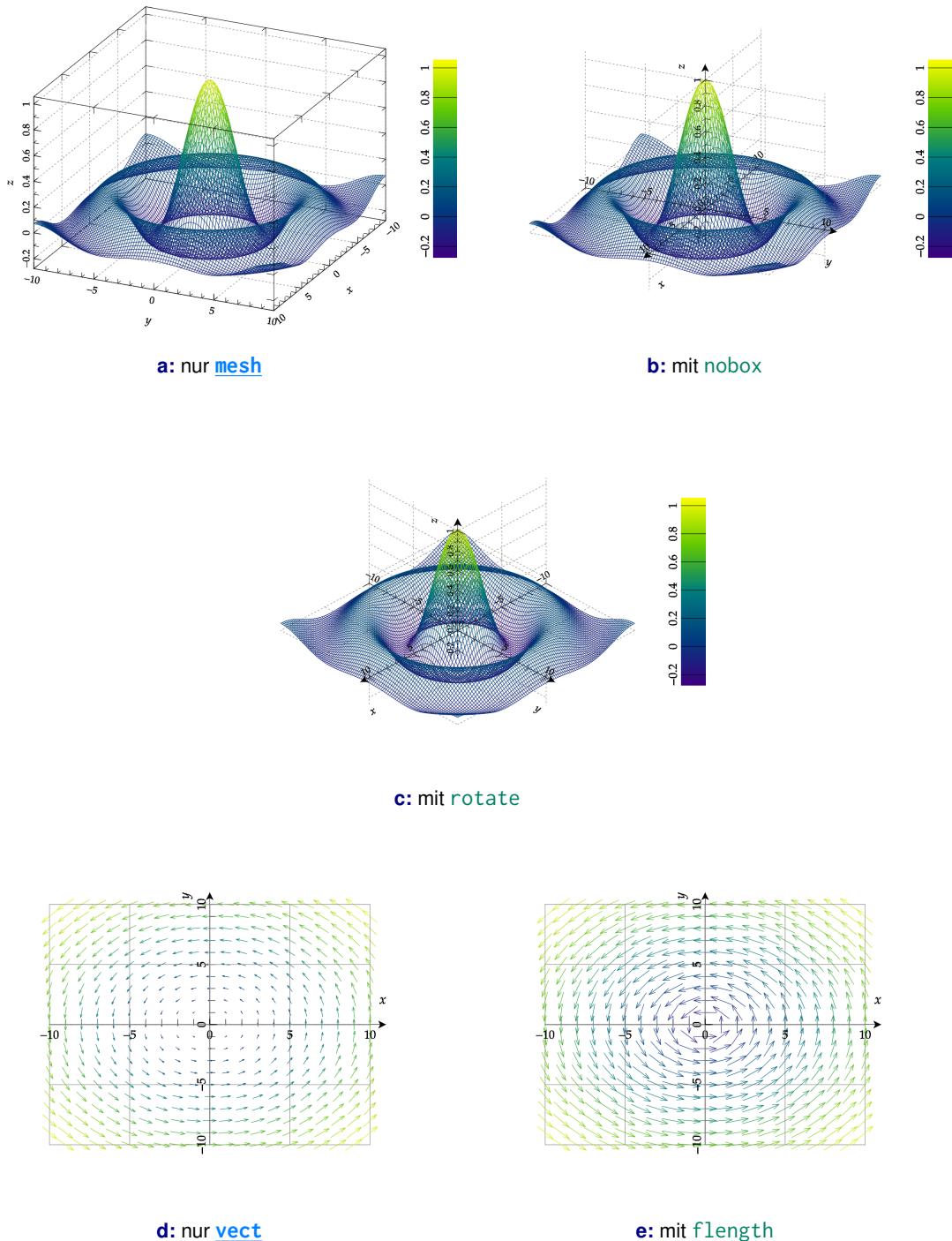


Abbildung 3.2.: Ergebnisse der Gitter- und Vektorplots

3. Erzeugen von Graphen

Für diesen Fall beschreibt `norm(x, y) = q`. Direkt im Anschluss an den vorherigen Plot ausgeführt sollte auffallen, dass auch der Gitterplot von einer umfassenden Box umgeben und ein Gitter im Hintergrund zeigt. Um die Box zu entfernen, gibt man `nobox` an ([Abbildung 3.2b](#)):

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

Der Gitterplot ist automatisch auf eine bestimmte Ausrichtung gedreht. Dies kann auch geändert werden:

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

Der Graph erscheint nun etwas mehr gekippt und auch die x - und y -Achsen erscheinen symmetrisch zu beiden Seiten ([Abbildung 3.2c](#)). Tatsächlich wurde der Graph mit diesen Werten in Richtung der ersten Raumdiagonale gedreht. Die Winkelwerte von `rotate` sind in Grad angegeben und zwar in der Reihenfolge ϑ, φ , wobei ϑ den Graphen kippt und φ ihn rotiert.

`mesh`

► In der NUMERE-Hilfe unter: [help mesh](#)

Ein weiteres Beispiel soll der Vektorplot eines 2D-Rotationsfeldes sein:

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

Syntaxelement Dies erreicht man durch

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

Hier wird die erste Funktion als Amplitude in \hat{e}_x -Richtung und die zweite Funktion in \hat{e}_y -Richtung interpretiert ([Abbildung 3.2d](#)). Zwecks der Übersicht akzeptiert `vect` (und `vect3d`) nur ein Vektorfeld.

Die Länge der Vektorpfeile entspricht der lokalen Amplitude des Vektorfeldes. Um diesen Effekt zu deaktivieren, kann

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

angegeben werden ([Abbildung 3.2e](#)).

`vect`

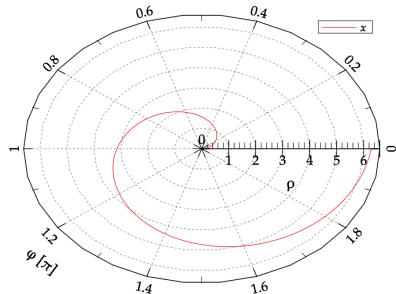
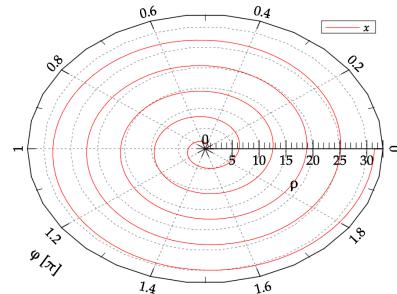
► In der NUMERE-Hilfe unter: [help vect](#)

3.4. Koordinatensysteme

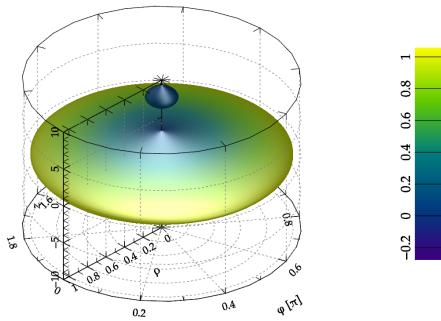
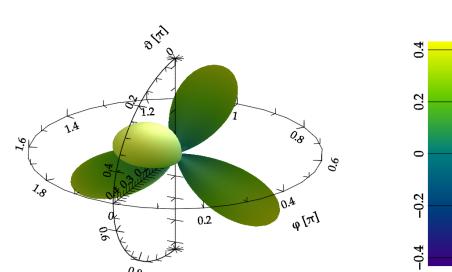
Zuletzt soll an dieser Stelle noch auf die unterschiedlichen Koordinatensysteme eingegangen werden. NUMERE unterstützt drei verschiedene Koordinatensysteme: das *kartesische*, das *polare* oder *zylindrische* und das *sphärische*.

Um das Koordinatensystem zu ändern, verwendet man die Option `coords=KOORDINATEN`. Um beispielsweise zu polaren Koordinaten zu wechseln, gibt man die folgende Zeile ein ([Abbildung 3.3a](#)):

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


 a: `plot` mit `coords=polar`


b: mit größerem Intervall


 c: `surf` mit `light`

 d: mit `coords=spherical`
Abbildung 3.3.: Ergebnisse verschiedener Koordinatensysteme

Man sieht, dass die Variable x hier als φ und der Funktionswert y als radiale Koordinate ρ interpretiert werden. Die Azimut-Achse ist in Einheiten von π dargestellt, kann aber ggf. mittels der Option `xscale=SCALE` geändert werden.

Wenn man des Weiteren das Intervall für x (ergo: φ) ändert, erhält man mit der folgende Zeile das Ergebnis von Abbildung 3.3b:

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

ggf. kann es hierbei von Vorteil sein, die Zahl der Samples mittels `samples=WERT` zu erhöhen.

In Kombination mit dreidimensionalen Plotfunktionen wechselt man nun automatisch ins zylindrische System, z.B. mittels

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

3. Erzeugen von Graphen

erhält man [Abbildung 3.3c](#). Die eingebrachte Option `light` aktiviert den dreidimensionalen Lichteffekt, der das Volumen verdeutlicht.

In diesem Fall wird deutlich, dass die Variable y nun als z und der Funktionswert $\Phi(x, y)$ nun die Funktion der radialen Variable ρ eingenommen hat.

Abschließend soll hier noch das sphärische Koordinatensystem dargestellt werden. In dieses wechselt man durch z.B.

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

und erhält die nette Kugelflächenfunktion (ihr Realteil) $Y_{32}(\vartheta, \varphi)$ in [Abbildung 3.3d](#). In diesem Koordinatensystem hat y die Funktion von ϑ , x ist wieder φ und der Funktionswert $\Phi(x, y)$ wird durch die radiale Koordinate r dargestellt.

Anmerkung Zu beachten ist, dass die gezeigte Achsenzuordnung geändert werden kann. Dazu gibt es weitere Optionen:

¹ `polar_pz`
² `polar_rp`
³ `polar_rz`
spherical_pt
⁵ spherical_rp
spherical_rt

coords

► In der NUMERE-Hilfe unter: [help coords](#)

4. TABELLEN: DER CACHE

Große Datensätze verwaltet NUMERE in Form von Tabellen. Die Standardtabelle für geladene Daten ist `data()`, wie bereits erwähnt wurde. Allerdings ist diese *Read-Only*. Als manipulierbare Tabellen in NUMERE existieren die sogenannten *Caches*.

4.1. Konzept

Caches sind Tabellen in NUMERE, deren Inhalt ähnlich gewöhnlicher Variablen frei manipuliert werden kann. Ihre Größe ist beliebig und passt sich automatisch den Wünschen des Nutzers an. Standardmäßig existiert bereits der Cache `cache()`, doch es können noch weitere erzeugt werden.

Caches besitzen außerdem eine Autosave-Funktion. Sobald der Inhalt eines Caches verändert wird, werden diese Änderungen nach Ablauf des Autosave-Intervalls in eine externe Datei übertragen. Bei einem Neustart von NUMERE werden diese Caches und die in ihnen enthaltenen Daten automatisch wiederhergestellt und können direkt weiterverwendet werden. Eine Tabellenkalkulation kann auf diese Weise bequem unterbrochen werden, ohne dass Datenverlust zu befürchten ist.

4.2. Erzeugen und Entfernen von Caches

Um weitere Caches zu erzeugen, kann das Kommando `new` verwendet werden (dieses Kommando kann auch noch weitere Objekte generieren):

Syntaxelement
new

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

Die dadurch generierten Caches enthalten keine Daten und können wie `cache()` verwendet werden. Sollten ein oder mehrere Caches der Liste bereits existieren, wird NUMERE diese Caches in keiner Weise beeinflussen.

► In der NUMERE-Hilfe unter: `help new`

new

Selbst generierte Caches können auch wieder entfernt werden, falls der Speicher nicht mehr für weitere Analysen benötigt wird:

Syntaxelement
remove

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

4. Tabellen: der Cache

Der von NUMERE tatsächlich benötigte Speicher wird jedoch erst nach einem Neustart NUMERE reduziert, um eine zusätzliche Fragmentierung zu vermeiden.

`remove`

► In der NUMERE-Hilfe unter: `help remove`

Syntaxelement

Allerdings können Caches, anstatt entfernt zu werden, auch umbenannt werden:

`rename`

`rename -CACHE1=NEUERNAME`
2 `rename -amplitude=phase`

Um alle Caches auf einmal zu entfernen und den kompletten Speicher freizugeben, kann

`clear cache()`

Syntaxeelement

verwendet werden. Hierbei wird auch der tatsächlich benötigte Speicher reduziert. Für eine einfache Löschung einzelner Elemente genügt auch

`delete`

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

`cache`

► In der NUMERE-Hilfe unter: `help cache`

4.3. Verwendung

Syntaxelement

Caches können verwendet werden wie die `data()`-Tabelle. Ein wesentlicher Unterschied besteht jedoch in der Tatsache, dass die Inhalte von Caches auch bearbeitet werden können. Demzufolge können ihre Elemente entsprechend Variablen neue Werte zugewiesen werden:

`cache()`

1 `CACHE(:,1) = WERT1, WERT2, WERT3, ...`
2 `CACHE(4,5:) = data(:,1)`
`CACHE(:,1) = CACHE(2,:)`

Die neuen Werte überschreiben dabei die bereits vorhandenen Werte in `CACHE()`.

An allen Stellen, an denen bisher `data()` verwendet wurde, können mit der identischen Syntax auch Caches eingefügt werden. Intern besteht (außer der Möglichkeit, Elemente zu ändern) kein Unterschied zwischen `data()` und den Caches:

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

Daten, die manche Kommandos (z.B. `datagrid`) erzeugen, werden teilweise automatisch in `cache()` geschrieben, teilweise erzeugen sie ihre Zielcaches auch selbst.

4.4. Sortieren, Glätten und Resampeln

Ggf. muss man Datenpunkte weiter modifizieren, um sie vernünftig verarbeiten zu können. NUMERE bietet hier die Möglichkeiten, die Daten zu sortieren, sie zu glätten und die Zahl der Datenpunkte zu verändern.

Die Daten in einem Cache (und im Grunde auch in `data()`) können auf- oder absteigend sortiert werden. Damit können z.B. bestimmte Schwellenwerte oder interessante Stellen in Messpunkten leichter gefunden werden.

Mittels der Eingabe von

```
sort -CACHE
```

Syntaxelement
sort

wird der gesamte Cache CACHE aufsteigend sortiert. Um ihn absteigend zu sortieren, verwendet man den Wert `desc`

```
sort -CACHE=desc
```

Mittels der Option `cols` können die zu sortierenden Spalten ausgewählt werden. Außerdem können damit eine Gruppe an Spalten angegeben werden, die anhand einer Index-Spalte sortiert werden sollen (wenn beispielsweise x -Werte und Funktionswerte zufällig angeordnet sind, können die Funktionswerte anhand der x -Werte sortiert werden und die Beziehung zwischen x -Werten und Funktionswerten bleibt bestehen):

```
sort -CACHE cols=SPALTEN[SPALTENGRUPPEN]
```

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

► In der NUMERE-Hilfe unter: `help cache`

cache

Neben dem Sortieren der Daten können Daten im Cache von Rauschen oder anderen vergleichbaren Artefakten befreit werden. Dies erreicht man durch ein Glätten der Daten.

Das Kommando

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

Syntaxeelement
smooth

glättet den Cache CACHE im ausgewählten Bereich, wobei über ORDNUNG Punkte linear interpoliert wird. Standardmäßig wird hier zweidimensional geglättet, sofern dies aufgrund der Dimension des verwendeten Bereichs Sinn macht. Um dies auf zeilen- oder spaltenweises Glätten umzuschalten, können hier die Optionen `lines` und `cols` verwendet werden.

Glätten funktioniert im Prinzip durch Einbringen eines Tiefpass-Filters. Sollten daher hochfrequente Daten interessant sein, kann Glätten diese teilweise oder auch vollständig zerstören. Als Probe bietet es sich an, die geglätteten Daten von den verrauschten Daten abzuziehen und das Ergebnis zu plotten. Wenn nur Rauschen entfernt wurde, sollte der Plot auch nur ein Rauschen zeigen.

► In der NUMERE-Hilfe unter: `help smooth`

smooth

Um die Zahl der Datenpunkte anzupassen (z.B. ist die Samplerate unpassend oder man möchte zwei Datenreihen miteinander verrechnen) kann NUMERE weitere Stützstellen in die Daten interpolieren oder die Zahl der Datenpunkte ebenso auch verringern.

Dies wird mittels

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

Syntaxelement
resample

erreicht. Im gewählten Bereich wird dabei spaltenweise die Zahl der Stützstellen zu SAMPLES verändert. Sollte SAMPLES größer als der gewählte Bereich sein, geht keine Information verloren, da die

4. Tabellen: der Cache

neuen Datenpunkte aus den bisherigen interpoliert werden. Wird Zahl hingegen kleiner gewählt, muss Information verloren gehen..

resample

► In der NUMER-E-Hilfe unter: [help resample](#)

4.5. Statistik-Funktionalität

Caches (und auch die `data()`-Tabelle) können zusätzlich zu [stats](#) und den Statistik-Funktionen (`std()`, `avg()`, ...) auch globale Statistikoperationen ausführen. Dazu werden die Caches als Kommando verwendet und mit den Statistikfunktionen als Parameter ausgeführt (in diesem Fall tauchen keine Klammern in dem Ausdruck auf):

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

Dies berechnet Mittelwert/Standardabweichung/etc. der gesamten Tabelle als gemeinsamen Wert. Mit zusätzlichen Optionen kann dies noch weiter eingeschränkt werden: die Option `grid` berechnet die entsprechenden Statistikwerte, wobei der Cache als Datengitter interpretiert wird. Die Option `cols` berechnet die Werte spalten- und `lines` zeilenweise. Die Optionen `lines` und `cols` können mit `grid` kombiniert werden.

5. NUMERE-SCRIPTE

NUMERE-Scripte bieten die Möglichkeit, komplexe Rechnungen und häufig wiederkehrende Analysen in eine Datei auszulagern, von der sie einfach und wiederholt ausgeführt werden können.

5.1. Konzept

Das Konzept hinter NUMERE-Scripten ist simpel: alle Eingaben, die man in die NUMERE-Konsole eingeben möchte, werden stattdessen zeilenweise in eine Textdatei geschrieben, die im Anschluss von NUMERE gelesen wird. NUMERE wird dann die Ausdrücke und Kommandos in der entsprechenden Reihenfolge abarbeiten und auswerten.

Der Vorteil liegt auf der Hand: die Befehlskette kann einfach wiederholt werden und Fehler können schnell und simpel korrigiert werden.

Anmerkung In einem späteren Kapitel werden auch NUMERE-Prozeduren besprochen. Diese repräsentieren die Programmierbarkeit von NUMERE. Die meisten Problemstellungen, die keine allzu komplexen Abstraktionen erfordern, lassen sich aber problemlos mithilfe von NUMERE-Scripten lösen.

► In der NUMERE-Hilfe unter: [help script](#)

script

5.2. Syntaxhervorhebung

Die Syntax von NUMERE-Scripten wird automatisch hervorgehoben, wenn die Datei mit der Endung ».nscr« gespeichert wird. Dies ist ein fester Bestandteil des NUMERE-Editors. Sie umfasst neben der Hervorhebung von Syntaxeelementen auch das Hervorheben von zusammengehörigen Klammern und Kontrollflussblöcken (z.B. `if ... else ... endif`).

Es steht jedem frei, die Farbgebung der Syntaxhervorhebung zu bearbeiten. Dies kann man im Optionen-Dialog, der im Werkzeuge-Menü gefunden werden kann, unter dem Tab »Style« durchführen.

5.3. Ein einfaches Script

Als Beispiel soll hier ein einfaches Script gezeigt werden.

5. NumeRe-Scripte

Um ein NUMERE-Script schnell und einfach zu erstellen, verwende man den Menüpunkt im Datei-Menü, klicke man auf die Schaltfläche in der Toolbar oder gebe man

```
new - script=erstes
```

Syntaxeelement
in die NUMERE-Konsole ein. Es wird ein NUMERE-Script »erstes.nscr« in <scriptpath> erzeugt, dass in den ersten beiden Fällen automatisch im Editor geöffnet wird. Im letzteren Fall öffnet die Zeile

```
edit edit erstes.nscr
```

dieses NUMERE-Script im NUMERE-Editor, so dass es bearbeitet werden kann. (Die Dateierweiterung ist hier nötig, da edit sonst nicht im korrekten Verzeichnis nach dem Script sucht) Zwischen die beiden im Script erscheinenden Textblöcke füge man die folgenden Zeilen ein:

```
## Cacheinhalt komplett loeschen
2 delete cache() -ignore

4 ## Zufallszahlen erzeugen
random -lines=1e5 cols=2 distrib=uniform mean=0.5 width=1

6
## Innerhalb des Viertel-Einheitskreises?
8 cache(:,3) = (cache(:,1)^2+cache(:,2)^2) <= 1 ? true : false;

10 ## PI berechnen
4*sum(cache(:,3))/1e5
```

Syntaxeelement
random wobei **##** einen Zeilenkommentar einleitet, der von NUMERE ignoriert wird (Alternativ kann man **/*...*/** als Blockkommentar verwenden). Das ignore in der zweiten Zeile unterdrückt die Sicherheitsabfrage, ob der Cacheinhalt wirklich gelöscht werden soll. random erzeugt Zufallszahlen und schreibt selbige in den Cache. Hier werden zweimal 100 000 gleichverteilte Zufallszahlen im Intervall [0;1] erzeugt.

Die dritte Anweisung ist die tückischste. Hier wird an die dritte Spalte in cache() das Ergebnis einer Bedingung zugewiesen. Sinngemäß steht hier so was wie

»Ist der Vektorbetrag kleiner-gleich als 1, dann schreibe »wahr«, sonst schreibe »falsch« in den Cache.«

Der sogenannte *Ternary*-Operator

```
BEDINGUNG ? DANN_WERT : SONST_WERT
```

Syntaxeelement
()? ... : ... ist eine Abkürzung für die if...else...endif-Verzweigung, die an späterer Stelle besprochen werden soll. Der Vorteil des *Ternarys* ist, dass er tendenziell schneller ausgeführt wird, dafür jedoch keine Kommandos aufnehmen kann. Eine Auflistung aller Logikausdrücke erhält man durch

```
list -logic
```

Am Ende der Zeile findet man auch ein Semikolon ;. Dieses unterdrückt die Ausgabe der Ergebnisse auf der NUMERE-Konsole.

Das gesamte NUMERE-Script ist eine ganz simple *Monte-Carlo*-Simulation. Es werden zufällig Punkte auf einem Quadrat mit der Kantenlänge 1 platziert und anschließend gezählt, wie viele der Punkte auf dem (Viertel-)Einheitskreis liegen, der Teil des Quadrates ist. Unter der Annahme, dass die Wahrscheinlichkeit für jeden Punkt im Quadrat gleich hoch ist, muss das Verhältnis von den im Kreis liegenden zur Gesamtzahl der Punkte $\pi/4$ sein.

Führt man das NUMERE-Script durch Klicken auf »Ausführen« oder mittels

Syntaxelement
start

start erstes

aus ([random](#) benötigt einen Moment), so bekommt man Ergebnisse ähnlich zu

3.1364
2 3.13668
3 .1454
4 3.13952
...

Diese Zahlen liegen schon relativ nahe an π dran, obwohl sie im Grunde nur durch eine clevere Ausnutzung des Zufalls generiert wurden.

Erhöht man die Zahl der erzeugten Zufallszahlen (z.B. 1e6 statt 1e5), so wird die Annäherung zunehmend genau. Allerdings unterstützt der Cache nicht beliebig viele Elemente. Für noch größere Zahlen muss man also auf andere Verfahren zurückgreifen.

Teil II.

FORTGESCHRITTENE BEDIENUNG

6. EIGENE FUNKTIONEN

NUMERE bringt bereits eine große Zahl vordefinierter Funktionen mit (siehe [list -func](#)). Dennoch kann es häufig hilfreich und praktisch sein, wenn man sich eigene Funktionen definieren kann, die z.B. bereits bestehende zusammenfassen. NUMERE kann bis zu 100 selbst definierte Funktionen speichern.

6.1. Definition

Eine eigene Funktion wird mittels

```
define meine_funktion(ARGS) := AUSDRUCK(ARGS)
```

Syntaxelement
define

definiert. Diese Zeile erzeugt die Funktion `meine_funktion()`, die den komplexeren Ausdruck der Argumente zusammenfasst. Die Bezeichnungen und die Zahl der Argumente können beliebig gewählt werden, so lange sie die Maximalzahl von 10 nicht überschreiten. Wenn das letzte Argument der Funktion »...« lautet, so kann für dieses Argument von einem bis zu beliebig vielen Werten angegeben werden. Der definierte Ausdruck muss dann allerdings auch mit beliebig vielen Werten umgehen können, z.B.:

```
define meine_funktion(x,...) := x*sum(...)
```

Syntaxelement
»...«

Der Name der Funktion ist auch beliebig, so lange er nicht mit einer Ziffer beginnt oder einer bereits (vor-)definierten Funktion entspricht.

Mit

Syntaxelement
redefine

```
1 redefine meine_funktion(ARGS) := NEUER_AUSDRUCK(ARGS)
```

kann die Funktion `meine_funktion()` umdefiniert werden. Hierbei müssen die neuen Argumente natürlich nicht mit den bisherigen übereinstimmen.

Selbst definierte Funktionen können mit Kommentaren versehen werden, um zu erläutern, was der Zweck der Funktion ist. Dieser Kommentar wird dann unter

```
list -define
```

zusammen mit der Definition angezeigt.

Der Kommentar kann bei der Definition mittels

```
1 define meine_funktion(ARGS) := AUSDRUCK(ARGS) -set comment="KOMMENTAR"
```

angegeben werden. Eine nachträgliche Kommentierung kann mittels `redefine` durchgeführt werden.

6. Eigene Funktionen

► In der NUMERE-Hilfe unter: help define

define

Syntaxelement Um selbst definierte Funktionen wieder zu entfernen, verwendet man

undefined `undefined meine_funktion()`

am Programmende wird der Funktionsspeicher jedoch auch automatisch geleert. Möchte man dies verhindern, kann man entweder

`1 define -save`

eingeben und beim nächsten Start

`1 define -load`

um die Funktionen zu laden, oder man aktiviert die automatische Definitionsverwaltung durch

`1 set -defcontrol=true`

set

► In der NUMERE-Hilfe unter: help set

set

6.2. Bedingte Definition

In NUMERE-Scripten kann es von Vorteil sein, wenn NUMERE nicht bei jedem Durchlauf den Fehler bringt, dass eine Funktion, die im Script definiert werden soll, bereits definiert ist. Für diesen Fall gibt es die Möglichkeit, entweder `redefine` zu verwenden, oder man gibt stattdessen

undefined

`ifundefined meine_funktion(ARGs) := ...`

an. Nun wird die Funktion nur dann definiert, wenn sie nicht schon bereit im Funktionsspeicher vorhanden ist.

6.3. Verwendung

Eine selbst definierte Funktion kann wie die vordefinierten Funktionen verwendet werden, da sie intern vor der eigentlichen Auswertung in ihre Definition umgewandelt werden. Daher ergeben

`1 meine_funktion(1, 2, 3)`

und

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

für die Definition

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

dasselbe Ergebnis. Es können auch weniger Werte angegeben werden, wie die definierte Funktion Argumente besitzt. NUMERE wird die fehlenden Werte dann durch 0 ergänzen.

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:

`"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 der NUMERE-Hilfe unter: `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

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

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

Syntaxelement

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

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

Syntaxelement

data(#, :)

cache(#, :)

```
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

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:

```
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.

Syntaxelement

#VAR

#{EXPRESSION}

Die Umkehrung kann auf verschiedene Arten vonstatten gehen:

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

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:

```
if (BEDINGUNG1)
2    AUSFUEHREN, FALLS WAHR
elseif (BEDINGUNG2)
4    AUSFUEHREN, FALLS BEDINGUNG1 FALSCH UND BEDINGUNG2 WAHR
else
6    AUSFUEHREN, FALLS ALLE BEDINGUNGEN FALSCH
endif
```

Syntaxelement
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 der NUMERE-Hilfe unter: [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

Syntaxelement
while ()
...
endwhile

8. Schleifen und Verzweigungen

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

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

while ► In der NUMERE-Hilfe unter: 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.

Syntaxelement Die Syntax einer Zählschleife lautet wie folgt:

```
1 for () ...
2   for (INDEX = START:END)
3     AUSFUEHREN, SO LANGE INDEX IN [START;END]
4   endfor
5 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 der NUMERE-Hilfe unter: help for

8.4. Ablaufkontrollen

Syntaxelement Auf den Ablauf einer Schleife kann mit einem der beiden Kommandos

```
1 continue
2 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 der NUMERE-Hilfe unter: 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. `mtrxop`-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(:, :) / ...
```

Syntaxelement
`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.

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

Syntaxelement
`... ** ...`

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 der NUMERE-Hilfe unter: `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.
- `eigenvecs(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.

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

Syntaxeelement
[zeroes](#)

```
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 der NUMERE-Hilfe unter: [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.

Syntaxeelement
[extrema](#)

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

10. Spezielle Kommandos

Fällen erfüllt, daher beschränkt NUMERE sich immer auf die Bestimmung der Indices bzw. 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 der NUMERE-Hilfe unter: [help extrema](#)

10.3. Integration

Syntaxelement

`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.

[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 der NUMERE-Hilfe unter: [help integrate](#)

10.4. Ableitung

Syntaxelement

`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 der NUMERE-Hilfe unter: [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).

Syntaxelement
taylor

Die Approximation geschieht rein numerisch. Dadurch kommt es zu unvermeidlichen Run- dungsfehlern, 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 der NUMERE-Hilfe unter: [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.

Syntaxelement
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

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

eval ► In der NUMERE-Hilfe unter: 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=STUETZSTELLEN` 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 ($x = \text{Zeilen}$, $y = \text{Spalten}$), 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 der NUMERE-Hilfe unter: 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.

Syntaxelement **fft** 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()
```

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 der NUMERE-Hilfe unter: [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.

Syntaxelement
[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:

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

```

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: $y' = dy_1/dx = y_2, y'' = dy_2/dx = y_3, \dots$ Ein solches System folgt dann dem Schema:

```
dy1/dx = y2
2 dy2/dx = y3
...
4 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 der NUMERE-Hilfe unter: [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 der NUMERE-Hilfe unter: `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
2   PROZEDURRUMPF
  endprocedure
```

Syntaxelement
procedure
...
endprocedure

Um eine NUMERE-Prozedur aufzurufen, gibt man `$PROZEDURNAME(VARS)` in NUMERE-Scripten, 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 Argumente angege-

11. NumeRe-Prozeduren

ben 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:

Syntaxeelement

explicit
private
inline² **private**
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

new -proc=\$PROZEDURNAME

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

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.

Syntaxeelement

var
str
tab² **var** VARIABLEN
str ZEICHENKETTENVARIABLEN
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

`return WERT`

Syntaxelement
`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

`$NAMENSRÄUM~PROZEDURNAME (VARS)`

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

`<procpth>/NAMENSRÄUM/PROZEDURNAME.nscr`

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

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

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

11. NumeRe-Prozeduren

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

namespace `namespace NAMENRAUM`

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

Syntaxelement 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

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 ([Abbildung 11.1](#)). Dieser kann durch den entsprechenden Menüpunkt im Werkzeug-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 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.

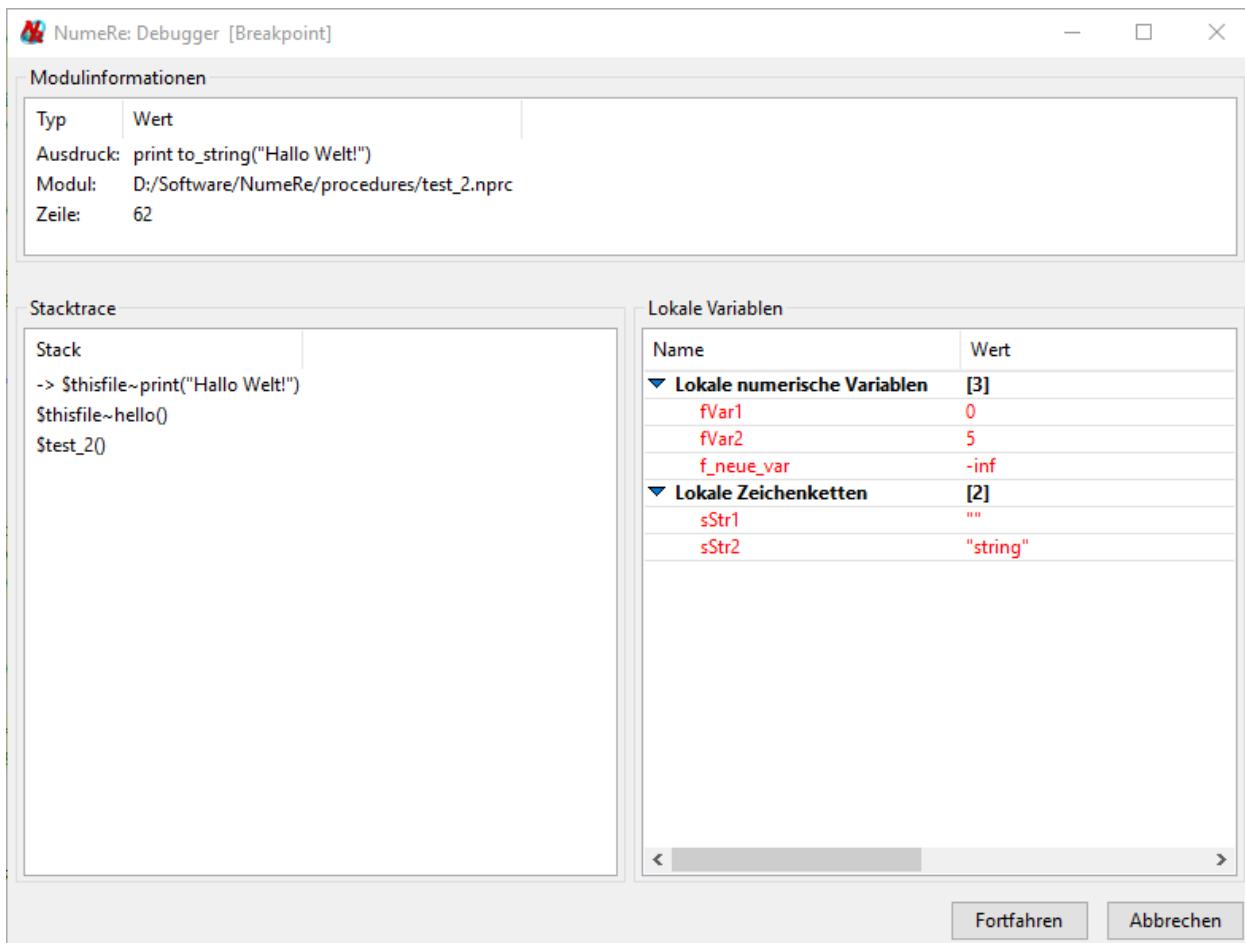


Abbildung 11.1.: Der NUMERE-Debugger an einem Breakpoint. Mit rot werden die Änderungen zum letzten ausgewerteten Breakpoint hervorgehoben

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 der NUMERE-Hilfe unter: [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:

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

Syntaxelement
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 der NUMERE-Hilfe unter: [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:

```
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

```
compose  
2   PLOTSTIL1  
    PLOTSTIL2  
4   ...  
endcompose
```

Syntaxelement
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:

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

► In der NUMERE-Hilfe unter: help compose

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

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

Syntaxelement
`<install>`
...
`<endinstall>`

enthalten. Um ein solches Plugin zu installieren, muss lediglich

```
1 install SCRIPTNAME
```

Syntaxelement
`install`

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 der **NUMERE-Hilfe unter:** `help install`

`install`

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

```
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.

Syntaxelement Um ein Plugin wieder zu deinstallieren, gibt man einfach

uninstall 1 `uninstall PLUGINNAME`

in die NUMERE-Konsole ein. Der `PLUGINNAME` ist unter

`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

`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 Hilfedatenbank-Einträge) 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:

Syntaxelement

`<CMDSTRING>`

`<EXPRESSION>`

`<PARAMSTRING>`

1 `<CMDSTRING>`

`<EXPRESSION>`

3 `<PARAMSTRING>`

- `<CMDSTRING>` übergibt die gesamte Kommandozeile (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

Syntaxelement
<info>
 ...
<endinfo>

```

<install>
  <info>
    -author="AUTORNAME"
    -version="VERSION"
    -type=TYPE_PLUGIN
    -flags=ENABLE_DEFAULTS
    -name="PLUGINNAME"
    -pluginmain=$PLUGINHAUPTPROZEDUR(<CMDSTRING>)
    -plugincommand="PLUGINKOMMANDO"
    -plugindesc="BESCHREIBUNG"
  <endinfo>
  PROZEDUREN
<endinstall>

```

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

```
-type=TYPE_PLUGIN_WITH_RETURN_VALUE
```

angegeben werden.

Der **version**-Flag kann als

```
-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

```
ENABLE_FULL_LOGGING
```

```
2 ENABLE_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 der NUMERE-Hilfe unter: [help plugins](#)

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

Syntaxelement
`<helpindex>`
 `...`
`</helpindex>` 2
 `...`
`<helpfile>` 4
 `...`
`</helpfile>` 4
 `INDEXINFORMATIONEN`
 `</helpindex>`
 `<helpfile>`
 `HILFEARTIKEL`
 `</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 der NUMERE-Hilfe unter: [help documentation](#)
