

Entwickeln einer
transparenten Datenschnittstelle
von MATLAB zu einem
objektorientierten Datenanalysetool

Diplomarbeit

für die Prüfung zum
Diplom-Informatiker (BA)

im Studiengang Angewandte Informatik
an der Berufsakademie Karlsruhe

von
Johannes Kissel

August 2008

Bearbeitungszeitraum	3 Monate
Kurs	TAI05
Ausbildungsfirma	Forschungszentrum Karlsruhe
Betreuer der Ausbildungsfirma	Dipl.-Inf. (BA) Michael Zapf
Betreuerin der Berufsakademie	Dipl.-Inf. Gertrud Nieder

Ehrenwörtliche Erklärung

Karlsruhe, 11. August 2008

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbstständig verfasst habe. Wörtlich oder dem Sinn nach aus Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

(Johannes Kissel)

Dieses Werk ist unter einem Creative Commons by-nd 3.0 Deutschland Lizenzvertrag lizenziert.
Um die Lizenz anzusehen, gehen Sie bitte zu <http://creativecommons.org/licenses/by-nd/3.0/de/>.



Inhaltsangabe / Abstract

Deutsch

Viele wissenschaftliche Projekte setzen wegen der Fähigkeit zum schnellen Prototyping auf MATLAB als Programmiersprache. Auch das Projekt Ultraschall Computertomographie (USCT), das diese Arbeit motiviert, nutzt MATLAB. Es ist ein kommerzielles Produkt, das sich durch Spezialisierung auf multidimensionale Matrizen, umfassende Visualisierung und Erweiterbarkeit mit sogenannten Toolboxen auszeichnet. Das objektorientierte Datenanalysetool ROOT wird als Open-Source-Projekt entwickelt. Es ist ein Standard für das Ablegen und den Austausch großer Datenmengen. Diese Diplomarbeit ergänzt MATLAB um die effiziente Datenhaltung von ROOT, indem eine transparente Datenschnittstelle von MATLAB zu ROOT geschaffen wird. Mit Hilfe dieser neuen Toolbox können aus MATLAB heraus ROOT-Dateien gelesen und geschrieben werden, so wie es für MATLAB-Dateien üblich ist. Für USCT, das Datenaufkommen im mehrstelligen Gigabyte-Bereich hat, ermöglicht dies den Austausch mit anderen Projekten in einem standardisierten Format.

English

Many research projects prefer MATLAB as programming language because of its capability for fast prototyping. The project Ultrasound Computer Tomography (USCT), which motivates this work, also uses MATLAB. This commercial product offers fast and simple access to multidimensional matrices, comprehensive visualisation and extensibility with so-called Toolboxes. The object-oriented tool ROOT for data analysis is an open source project. It is considered to be a standard for storing and exchanging large data sets. This diploma thesis adds ROOT's efficient data management to MATLAB by providing a transparent data interface from MATLAB to ROOT. With this new Toolbox it is possible to read and write ROOT files from within MATLAB just as you do with MATLAB files. For the USCT project, which produces gigabytes of data, this enables comfortable data interchange with other projects in a standardised format.

Inhaltsverzeichnis

1	Einleitung	5
1.1	Motivation und Aufgabenstellung	5
1.2	MATLAB und MEX	6
1.3	ROOT	7
1.3.1	Abstraktes Speicherformat	8
1.3.2	Motivation für die Nutzung von ROOT	9
1.4	Arbeitsumgebung	10
1.4.1	MATLAB und ROOT	11
1.4.2	Versionierung	12
1.4.3	Unit Testing	13
1.4.4	Dokumentation	15
1.4.5	Sonstiges	16
2	Umsetzung	17
2.1	Evaluation führt zum Konzept	17
2.1.1	Funktionen	19
2.1.2	Dateitypen	24
2.1.3	Datentypen	26
2.1.4	Programmierung	29
2.2	Implementierung	30
2.2.1	Stabilisierung des Prototyps	31
2.2.2	Transformation der Variablen	32
2.2.3	Dateiname parsen	36
2.2.4	MATLAB-Wrapper für MEX-Dateien	38
2.2.5	MEX-Dateien	40
2.2.6	MATLAB-Hilfsfunktionen	42

3	Ergebnisse	47
3.1	Benutzerschnittstelle	47
3.2	Tests	49
3.2.1	Standard-Testfall	49
3.2.2	Unit Tests	50
3.2.3	Skalierung der Performanz	51
3.2.4	Test unter realen Bedingungen	52
3.3	Publikation	54
4	Diskussion und Ausblick	55
4.1	Weitere Schritte	56
4.2	Visionäre Ausblicke	57
4.2.1	mroot-Dienst	57
	Literaturverzeichnis	60
	Anlagenverzeichnis	61
	Abbildungsverzeichnis	62
	Listings	63
	Tabellenverzeichnis	64
	Abkürzungsverzeichnis	65

Kapitel 1

Einleitung

1.1 Motivation und Aufgabenstellung

Am Institut für Prozessdatenverarbeitung und Elektronik (IPE) des Forschungszentrum Karlsruhe (FZK) wird ein auf Ultraschall basierendes, bildgebendes Verfahren zur Brustkrebsfrüherkennung entwickelt. Diese Ultraschall Computertomographie (USCT) [1] erzeugt dreidimensionale Volumebilder einer Brust in wesentlich höherer Bildqualität als herkömmliche Verfahren, die Ultraschall nutzen.

Eine Messung mit allen Ultraschallwandlern erzeugt ca. 20 GByte Rohdaten, die zur Zeit im MATLAB [2] eigenen Speicherformat mit der Dateierweiterung `.mat` abgelegt werden. Dabei hat sich im Laufe der Zeit eine projektinterne Verzeichnisstruktur entwickelt.

Die gesamte Datenhaltung ist – was die Nutzung verschiedener Datentypen anbelangt – sehr heterogen.

Im Rahmen einer Projektarbeit [3] wurden die Möglichkeiten des objektorientierten Datenanalysetools ROOT [4] als Option für die Datenhaltung bei USCT evaluiert. Dabei entstand ein einfacher Proof-of-Concept-Prototyp, der die für USCT wichtigsten Datentypen in einfachen ROOT-Dateien ablegen und wieder daraus laden kann.

Basierend auf den Erkenntnissen dieser Arbeit wurde nun eine aus Sicht von MATLAB möglichst transparente Datenschnittstelle zu ROOT entwickelt. Diese unterstützt alle MATLAB-Datentypen. Als Datensenke können nun außer binären Dateien auch XML-Dateien und SQL-Datenbanken dienen.

Die Schnittstelle hat den kompakten Namen „mroot“ mit dem Untertitel „ROOTkit for MATLAB“.

1.2 MATLAB und MEX

MATLAB ist ein kommerzielles Produkt der Firma Mathworks, Inc. Der proprietäre Code ist auf verschiedenen Plattformen lauffähig. Die einfache Syntax und der mathematisch orientierte Befehlssatz ermöglichen ein schnelles Prototyping von Lösungsansätzen im wissenschaftlichen Bereich. MATLAB kann mit einer Vielzahl von sogenannten „Toolboxen“ für bestimmte Einsatzfelder erweitert werden.

In der MATLAB-Entwicklungsumgebung können Quelltext-Dateien (Datei-erweiterung `.m`) komfortabel erstellt, ausgeführt und einem Debugging unterzogen werden. Sie bietet zudem einen Kommandozeileninterpreter und umfangreiche Visualisierungsmöglichkeiten für 2D und 3D.

MATLAB-Funktionen können sowohl funktional, als auch imperativ benutzt werden (s. Lst. 1.1). Beim imperativen Aufruf werden alle Argumente als Strings interpretiert. Der Gültigkeitsbereich einer Funktion heißt bei MATLAB „Workspace“.

```
1 % functional
2 save('matlab.mat', arg1, arg2, ...)
3
4 % imperative
5 save matlab.mat arg1 arg2 ...
```

Listing 1.1: Funktionaler vs. imperativer MATLAB-Funktionsaufruf

Mit den MATLAB External Interfaces (MEX) [5] können eigene, in C/C++ geschriebene Funktionen aus MATLAB heraus aufgerufen werden. MEX stellt hierfür Bibliotheken und Header-Dateien zur Verfügung sowie die MATLAB-Funktion `mex` zur Übersetzung solcher Funktionen. `mex` ist eine Art Kompatibilitätsebene, die es erlaubt, verschiedene Übersetzer auf unterschiedlichen Plattformen über eine einheitliche Schnittstelle zu nutzen. Es werden nicht beliebige Compiler unterstützt, sondern nur eine bestimmte Auswahl: Unter Linux wird z.B. die GCC unterstützt, unter Windows VC++. Im MATLAB-Lieferumfang ist ein LCC enthalten.

1.3. ROOT

Eine MEX-Funktion muss mindestens Folgendes enthalten:

```
1 #include "mex.h"
2
3 void mexFunction(int nlhs, mxArray *plhs[],
4                 int nrhs, const mxArray *prhs[]) {
5     // work
6 }
```

Listing 1.2: Grundgerüst einer MEX-Datei

Für Matrix-Operationen wird zusätzlich die Datei `matrix.h` eingebunden. `plhs` und `prhs` sind Arrays von MATLAB-Variablen für Aus- bzw. Eingabe der Einsprungsfunktion `mexFunction` (left/right hand side). `nlhs` und `nrhs` geben jeweils die Anzahl der Elemente an.

1.3 ROOT

ROOT ist ein am CERN entwickeltes C++-Framework zur Datenhaltung und -Analyse. Es ist auf den Umgang mit großen Datenmengen (Petabyte-Bereich) optimiert und verspricht deshalb eine gute Performanz auch bei den großen USCT-Datenaufkommen. Die Daten werden effizient abgespeichert: Sie erlauben selektiven Zugriff und können on-the-fly mit GNU ZIP komprimiert werden. ROOT ist vollständig objektorientiert und thread-fähig.



Abbildung 1.1: Logos von CERN und ROOT

ROOT ist in C++ programmiert und ist auf vielen verschiedenen Plattformen lauffähig. Es gibt fertige Binärpakete für zahlreiche Umgebungen sowie einen komfortablen Windows-Installer. Das System wird unter einer Open Source-Lizenz vertrieben (GNU LGPL [6]) – mit allen bekannten Vorteilen. Es gibt eine große, heterogene Nutzergemeinde; die Entwicklung geht schnell voran.

Eine große Stärke ROOTs ist die einfache Erstellung von Histogrammen, wie sie in der Teilchenphysik am CERN genutzt werden. Diese und weitere

1.3. ROOT

graphische Elemente und Benutzeroberflächen werden durch externe Bibliotheken wie z.B. OpenGL oder Qt dargestellt.

Dank des mitgelieferten C/C++-Interpreters CINT kann ROOT-Code sowohl kompiliert als auch interpretiert werden. In Verbindung mit dem Kommandozeileninterpreter (s. Abb. 1.2) kann dies das Testen mitunter sehr erleichtern.

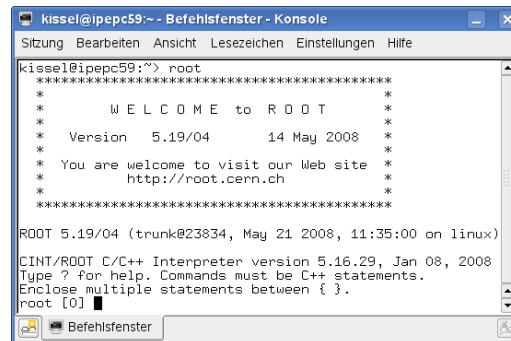


Abbildung 1.2: ROOT-Konsole

Im Laufe der Evaluation hat sich gezeigt, dass ROOT bzgl. Funktionsumfang und Bedienung mit MATLAB vergleichbar ist.

1.3.1 Abstraktes Speicherformat

In einer ROOT-Datei wird ein UNIX-artiges Dateisystem mit Verzeichnissen und persistierten Objekten als „Dateien“ abgebildet. Dieses Dateisystem kann mit dem ROOT Object Browser durchsucht werden (s. Abb. 1.3).

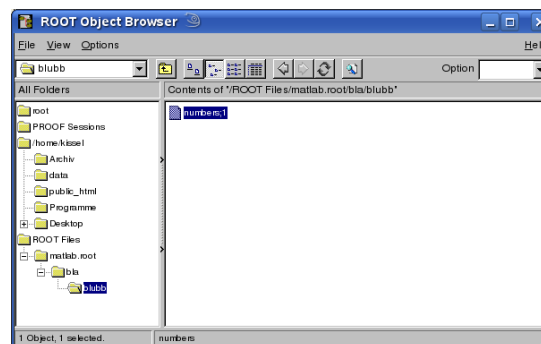


Abbildung 1.3: ROOT Object Browser

ROOT-Dateien werden auf Objektebene von der Klasse `TFile` repräsentiert, Verzeichnisse von der Klasse `TDirectory`. `TFile` erbt von `TDirectory` und

1.3. ROOT

ist somit ein spezielles Verzeichnis. Jeder Eintrag in einem Verzeichnis hat den Typ `TKey` und verweist auf ein Unterverzeichnis oder eine Datei. Diese Organisation erlaubt den wahlfreien Zugriff auf Objekte einer ROOT-Datei, ohne dass diese komplett in den Hauptspeicher geladen werden muss. Der Zugriff ist dadurch schneller und speichereffizienter.

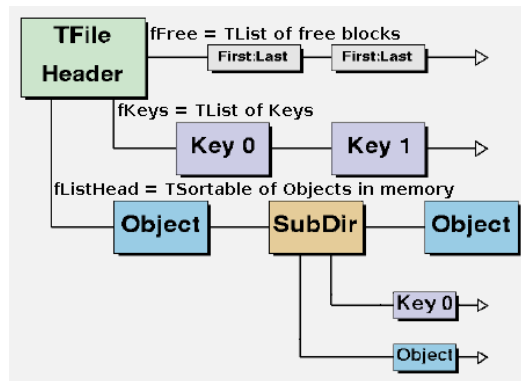


Abbildung 1.4: Beschreibung des ROOT-Dateiformats

Es gibt noch weitere, spezialisierte I/O-Klassen, die von `TFile` erben und deren Fähigkeiten ergänzen. Da diese Klassen von `TFile` erben, haben sie das gleiche Verhalten.

Eine ROOT-Programm wird immer in einem ROOT-Verzeichnis ausgeführt, auch wenn noch keine ROOT-Datei geladen ist. In diesem Fall ist das Arbeitsverzeichnis das virtuelle Verzeichnis `TR00T`. Der Pfad des aktuellen Verzeichnisses wird von der globalen Variablen `gDirectory` repräsentiert.

1.3.2 Motivation für die Nutzung von ROOT

Grid ist eine Infrastruktur zur gemeinschaftlichen Nutzung autonomer Ressourcen und wird ebenfalls am FZK mitentwickelt. ROOT wurde bereits erfolgreich im Grid getestet und eignet sich wegen seiner eingebauten Thread-Fähigkeiten für die parallele Datenverarbeitung.

MATLAB hat eine kostspielige Einzelplatz-Lizenzierung, während das plattformunabhängige ROOT-Framework unter einer freien Lizenz steht. Weltweit sind zahlreiche ROOT-Installationen in Nutzung, was den Datenaustausch mit anderen Projekten vereinfachen könnte.

Auf lange Sicht möchte das USCT-Projekt die Vorteile einer datenbankbasierten Datenhaltung (Geschwindigkeit, Einheitlichkeit, etc.) nutzen und sich damit von der dateibasierten Datenhaltung abwenden. ROOT könnte

hier das entscheidende Interface sein, obwohl auch MATLAB eigene, kommerzielle Datenbank-Schnittstellen besitzt.

Im IPE gibt es bereits andere Projekte, die ROOT für die Datenhaltung einsetzen. mroot könnte den Trend zu einem im Institut einheitlichen Datenformat „ROOT“ fördern.

1.4 Arbeitsumgebung

Für die vorliegende Arbeit wurden zwei Computer mit unterschiedlichen Betriebssystemen ausgewählt:

- openSUSE Linux 10.2 mit der Desktop-Oberfläche KDE
- Microsoft Windows XP SP3 Professional

mroot wurde auf dem Linux-Rechner entwickelt. Es muss aber möglichst plattformunabhängig sein: Im USCT-Projekt wird hauptsächlich unter Windows gearbeitet, während bspw. im Grid-Umfeld Linux verbreitet ist.

Da die für Windows vorübersetzten ROOT-Versionen mit Microsofts Visual C++ (VC++) [7] kompiliert wurden, ist VC++ unter Windows als MEX-Compiler zu bevorzugen. Die verwendete MATLAB-Version unterstützt das veraltete VC++ 2005, anstatt des aktuellen VC++ 2008. Eine Distribution von VC++ 2005 Express zusammen mit der Entwicklungsumgebung Visual Studio kann kostenlos heruntergeladen werden. Die ROOT-Header-Dateien, die in mroot verwendet werden, benötigen zusätzliche Header-Dateien und Bibliotheken aus dem Microsoft Platform SDK. Dieses ist aber seit über einem Jahr nicht mehr verfügbar. Nach längerer Suche stellte sich heraus, dass es im Platform SDK 2003 Server aufgegangen ist. Auch dieses steht nach Registrierung einer Windows Live ID kostenlos zur Verfügung.

Auf dem Linux-Rechner waren G++, der C++-Compiler der GNU Compiler Collection (GCC) [8], und alle benötigten Dateien bereits vorinstalliert. Um adäquate Antwortzeiten zu erreichen, musste der Hauptspeicher dieses Computers auf 1,5 GB aufgerüstet werden. Das beschleunigte nicht nur den Entwicklungsprozess, sondern auch die Übersetzung von ROOT, für das es keine vorkompilierten openSUSE-Pakete gibt. Es war vorgesehen, zu Debugging-Zwecken Eclipse in Verbindung mit dem GNU Debugger (GDB) zu verwenden. Dem war aber kein Erfolg beschieden, da es nicht gelang, den GDB an MATLAB anzuheften.

1.4.1 MATLAB und ROOT

Auf beiden Plattformen wurden der Vergleichbarkeit wegen jeweils die gleichen Versionen von MATLAB und ROOT benutzt:

- MATLAB 7.4 (R2007a)
- ROOT 5.19/04

MATLAB kann auf allen unterstützten Plattformen problemlos mit Hilfe eines grafischen Dialogs installiert werden. Seit geraumer Zeit werden jährlich zwei Versionen a und b veröffentlicht. Dabei wird jeweils die Stelle des Minor Release in der Versionsnummer hochgezählt. Version 7.4 ist also im ersten Halbjahr 2007 erschienen.

ROOT 5.19/04 war zu Beginn der Arbeiten an mroot die zur Nutzung empfohlene Version. Das Windows-Installationsprogramm setzt selbstständig die erforderlichen Umgebungsvariablen. Für openSUSE gibt es keine vorkompilierten Pakete, sodass ROOT aus den Quellen übersetzt werden musste. Dies geschieht – wie unter Linux üblich – durch die folgende Befehlsfolge:

```
1 ./configure
2 make
3 make install
```

Listing 1.3: Build- und Installationsprozess unter Linux

`./configure` untersucht, um was für ein System es sich handelt, welche Bibliotheken vorhanden sind, ob die zur Kompilierung benötigten Werkzeuge installiert sind und erzeugt aufgrund dieser Informationen ein Makefile. Um das Installationsverzeichnis festzulegen, wird zuvor die Umgebungsvariable `$ROOTSYS` gesetzt. Dem Befehl `./configure` können Parameter übergeben werden [9]. Das Installationsverzeichnis kann bspw. auch mit Hilfe des Parameters `--prefix` festgelegt werden. Es sollte jedoch entweder `$ROOTSYS` oder `--prefix` verwendet werden, da die Kombination zu Problemen führen kann.

Für die Übersetzung von ROOT waren nur diese Parameter nötig:

```
./configure --with-mysql-libdir=<LDIR>
            --with-mysql-incdir=<IDIR>
```

Die Fähigkeit auf MySQL-Datenbanken zuzugreifen wird zwar standardmäßig installiert, um sie zu nutzen, muss ROOT aber mitgeteilt werden, wo die entsprechenden Bibliotheken und Header-Dateien liegen.

make übersetzt die Quellen mit Hilfe des von **./configure** erzeugten Makefiles. Abschließend werden die erzeugten Dateien mit **make install** installiert.

Falls ROOT nicht in einem Standard-Verzeichnis installiert wurde, muss \$ROOTSYS den Suchpfaden für ausführbare Dateien und Bibliotheken hinzugefügt werden:

```
1 export ROOTSYS=~ /Programme /root_5.19
2 export PATH=$ROOTSYS/bin:$PATH
3 export LD_LIBRARY_PATH=$ROOTSYS/lib:$LD_LIBRARY_PATH
```

Listing 1.4: Umgebungsvariablen für die ROOT-Nutzung

1.4.2 Versionierung

Als Versionierungssystem kommt im USCT-Projekt Subversion (SVN) [10] zum Einsatz. Die Versionierung erfolgt hier in einem zentralen Repository. Änderungen durch einzelne Entwickler werden inkrementell mit Hilfe einer Client-Software in dieses Archiv übertragen. Kollisionen beim Zusammenführen der Änderungen müssen manuell gelöst werden. Subversion wurde als moderne Weiterentwicklung des mit Schwächen behafteten, aber weit verbreiteten Concurrent Version Systems (CVS) konzipiert.

Ein guter SVN-Client ist das Open-Source-Programm TortoiseSVN, das allerdings nur für Windows verfügbar ist. Es integriert sich in die Explorer-Shell (s. Abb. 1.5) und ist damit unabhängig von Entwicklungsumgebungen und anderen Entwicklungswerkzeugen. Die meisten Fähigkeiten des SVN-Systems werden unterstützt, die Bedienung ist sehr komfortabel. Der Status der versionierten Dateien wird durch Icon-Overlays (1) dargestellt. Über das Kontextmenü können Aktionen auf ihnen ausgeführt werden (2).

Unter Linux kam kdesvn als SVN-Client zum Einsatz. Er bringt u.a. einen zusätzlichen KIO-Slave mit. KIO ist ein virtuelles Dateisystem unter KDE. Slaves erweitern dieses Dateisystem um Fähigkeiten wie z.B. FTP, SSH oder eben SVN. Dem Benutzer zeigt sich das bei kdesvn durch einen neuen Ansichtsmodus im Dateiverwalter Konqueror (s. Abb. 1.5). Dieser öffnet sich durch Klick auf einen zusätzlichen Button (3). Daraufhin erscheint ein weiteres Menü (4) und eine detaillierte Darstellung des Repositories (5).

1.4. ARBEITSUMGEBUNG

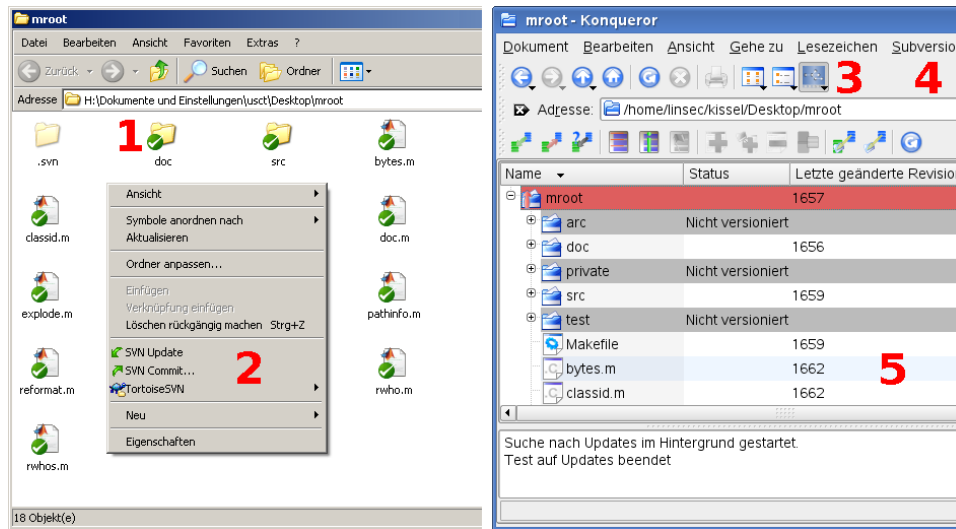


Abbildung 1.5: TortoiseSVN und kdesvn im Vergleich

1.4.3 Unit Testing

Die in MATLAB implementierten Funktionen der Schnittstelle wurden mit Hilfe von Unit Testing auf Fehler geprüft.

mroot wurde nicht testgetrieben entwickelt. Dennoch ist Unit Testing sehr vorteilhaft, um Funktionen gegen ihre Spezifikation zu testen. Unit Tests sind sehr formalisiert, sodass die Implementierung eines einzelnen Tests i.d.R. nicht sehr aufwändig ist. Die Tests lassen sich jederzeit mit nur einem Klick wiederholen, sodass sie fester Bestandteil der Weiterentwicklung werden können. Schlägt dabei ein Test in einer bisher „fehlerlosen“ Software fehl, so ist sofort klar, wo der Fehler zu suchen ist.

MUnit [11] ist eines von mehreren Unit Testing Frameworks für MATLAB [12]. Es ist selbst in MATLAB programmiert und ist ein gutes Beispiel für den Umgang mit Function Handles in MATLAB. Das Grundgerüst einer Test Suite ist sehr einfach:

```
1 function test = test_mroot
2     % create testcase object
3     test = munit_testcase;
4
5     % create structure of constraints for assertions
6     c = munit_constraint;
7
8     % set up fixture
9     function setup; end
```

```

10
11     % tear down fixture
12     function teardown; end
13
14     % test functions
15
16     function test_0
17         x = 1; y = 2;
18         test.assert(c.eq(x, y));
19     end
20
21     ...
22 end

```

Listing 1.5: Grundgerüst einer MUnit Test Suite

Eine solche Suite besteht aus mehreren Test Cases, die durch geschachtelte Funktionen mit dem Präfix `test_` realisiert werden. Der Name der Suite selbst ist frei wählbar. Die einzelnen Tests werden nacheinander ausgeführt. Optional kann eine `setup`- und eine `teardown`-Funktion definiert werden, die jeweils vor und nach einem Test ausgeführt werden. Entscheidender Bestandteil jeder Test-Funktion sind die sogenannten Assertions – Aussagen, die als wahr angenommen werden. `c` (s. Z. 6) ist ein Struct von Function Handles, die bestimmte Nebenbedingungen prüfen und mit denen beliebig komplexe solcher Aussagen formuliert werden können.

The screenshot displays the MUnit web interface, which is divided into two main sections: the MUnit Control Panel and the MUnit Test Report.

MUnit Control Panel: This section includes a header with "MUnit from XTargets" and "Copyright Brad Phelan 2005". It features a "bookmarks" link. The main content area has a "Add Test Suites" button and a text input field for entering a test case file name (e.g., `my_testcase`). Below the input field is an "add" button. There is also a "Test Suites" section listing three test suites: `test_munit1`, `test_munit2`, and `test_munit3`, each with links for "edit", "run", and "remove".

MUnit Test Report 11-Oct-2005: This section shows the results of a test run. It includes a "bookmarks" link and a summary of the test results: "0 Passes", "0 Errors", and "1 Failures". A detailed view of the failure is shown for "example/test_1", indicating an "Assert Failed" at line 40 of `C:\matlab\MATLAB71\work\example.m`. The failure message is `~ (1 == 1)`.

Abbildung 1.6: MUnit: Control Panel und Test Report

Die Test Suites können im „Control Panel“ (s. Abb. 1.6) als Lesezeichen-Sammlung organisiert werden. Das erleichtert den Umgang mit mehreren Suites. Nach Ablauf einer Suite oder auch nur eines einzelnen Tests wird ein Report ausgegeben, der eine Übersicht über die Anzahl der erfolgreichen Tests gibt sowie über die Tests, die durch einen unvorhergesehenen Fehler oder eine falsche Annahme gescheitert sind. Bei den gescheiterten Tests können bei Bedarf Details angezeigt werden (s. Abb. 1.6).

1.4.4 Dokumentation

Da die Ergebnisse dieser Arbeit in der Art veröffentlicht werden sollen, dass andere Entwickler daran weiterarbeiten können, müssen die Quelltexte entsprechend dokumentiert werden. Neben „normalen“ In-Code-Kommentaren wurden dazu spezielle Kommentare verwendet, aus denen bestimmte Programme Webseiten erzeugen, die eine gute Übersicht über die API und die Vernetzung der einzelnen Funktionen und Module geben.

JavaDoc ist Teil des Java Development Kits (JDK) und dient allen Programmen dieser Art als Vorbild. Für C++ ist Doxygen [13] weit verbreitet. Die Syntax der Kommentare ist an die von JavaDoc angelehnt. Außer C++ werden noch einige weitere Sprachen unterstützt. Die Ausgabe der Dokumentation kann detailliert über eine Konfigurationsdatei gesteuert werden. Das Programm ist Open Source.

Bei MATLAB sind JavaDoc-ähnliche Kommentare unüblich. I.d.R. wird hier nur der sogenannte Doc-Kommentar ausgewertet, also der Kommentar, der direkt auf die Signatur einer Funktion folgt. Für Inhalt und Gestaltung dieses Kommentars gibt es weder eine offizielle, noch eine projektinterne Konvention. Doc-Kommentare in mroot folgen deshalb diesem Muster:

```
1 function out = foo(in)
2 % This is the description of foo.
3 %
4 % Arguments:
5 %     in - An input argument
6 %
7 % Return values:
8 %     out - An output argument
9 %
10 % Copyright (C) 2008 Johannes Kissel<johannes.kissel@ipe.fzk.de>
11 % For the licensing terms see ./COPYING.
```

Listing 1.6: Struktur der Doc-Kommentare

Das Programm M2HTML [14] – selbst in MATLAB geschrieben und damit Open Source – erzeugt daraus eine HTML-Dokumentation. Dabei liegt ein starker Fokus auf der Vernetzung der Funktionen, die durch Verlinkung der einzelnen HTML-Seiten und einen Abhängigkeitsgraph gezeigt wird. Die Parametrisierung der HTML-Ausgabe ist nicht so detailliert wie bei Doxygen und erfolgt beim Aufruf der Funktion `m2html`.

Diese Diplomarbeit wurde in der Markup-Sprache \LaTeX [15] geschrieben. Sie vereinfacht die Benutzung des Textsatzsystems \TeX mit Hilfe von Makros. Wie bei HTML wird hier nicht nach dem WYSIWYG-Prinzip gearbeitet, der Text wird lediglich logisch ausgezeichnet und dann in einem weiteren Schritt für den Druck aufbereitet. Im Gegensatz zu HTML ist \LaTeX aber nicht auf die Beschreibung von Webinhalten ausgerichtet, sondern auf die Gestaltung von Druckerzeugnissen.

1.4.5 Sonstiges

Der MEX-Compiler unterstützt nicht alle Funktionen, die mancher native Compiler bietet, sodass Bibliotheken evtl. nicht übersetzt werden können. Außerdem werden Ausgaben auf Standard Out nirgends angezeigt. In eigenem MEX-Code wird für Ausgaben in das MATLAB Command Window stattdessen die MEX-Funktion `mexPrintf` verwendet.

Aufgrund dieser Problematiken wurde der Gebrauch externer Bibliotheken auf solche beschränkt, die ihre Ausgaben in Log-Dateien schreiben können.

Kapitel 2

Umsetzung

2.1 Evaluation führt zum Konzept

Verwendete Schnittstelle

Bereits der Prototyp der Projektarbeit wurde in MEX-Dateien realisiert. Es gab zwar diverse andere Ansätze, die MEX-C++-Schnittstelle war aber naheliegend, da sie sehr gut in die MATLAB-Umgebung integriert ist: MEX-Dateien verhalten sich wie gewöhnliche MATLAB-Funktionen. Daten können so ohne großen Aufwand übergeben und in der MEX-Datei mit Hilfe von MEX-Funktionen weiter verwendet werden. Da ROOT ebenfalls in C++ implementiert ist, kann außerdem direkt auf dessen API zugegriffen werden. Nicht zuletzt verspricht eine C/C++-Implementierung eine hohe Performanz.

Wrapper um MEX-Dateien

Die MEX-Dateien der mroot-Schnittstelle sollen nicht direkt von benutzerdefiniertem Programmcode aufgerufen werden, da sie nicht in der geschützten Umgebung des MATLAB-Interpreters ausgeführt werden. Dies wird verhindert, indem sie in einem Verzeichnis names „private“ abgelegt werden. Somit können sie nur noch von Funktionen im Verzeichnis direkt darüber aufgerufen werden. Der Zugriff erfolgt stattdessen über eine vorgeschaltete Funktion, die die Parameterprüfung übernimmt. So werden nur noch validierte Eingaben an die MEX-Dateien weitergereicht. MEX-Dateien heißen wie die jeweils vorgeschaltete Funktion mit dem Präfix `mex_` (s. Abb. 2.1).

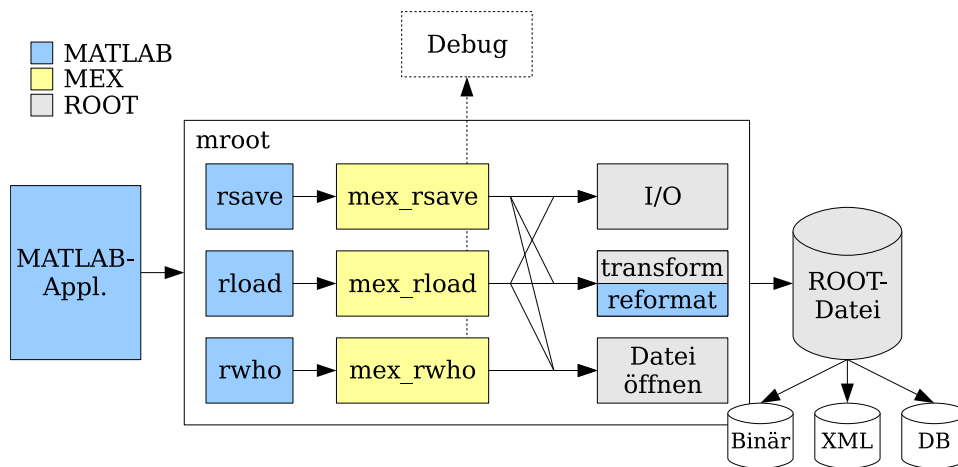


Abbildung 2.1: mroots Schnittstellen-Struktur

Die mroot-Funktionen stehen wie ihre Vorbilder jeweils für sich, da eine ROOT-Datei nicht ohne Weiteres über eine Folge von Funktionsaufrufen offen gehalten werden kann.

Nutzung von ROOT-Basisklassen

MATLABs Datentypen wurden auf native ROOT-Klassen abgebildet. Das hat den Vorteil, dass abgespeicherte Daten mit einer ROOT-Standard-Installation direkt weiter verarbeitet werden können. Bei der Wahl der Klassen wurde darauf geachtet, dass diese von `TObject` abgeleitet sind. Klassen solcher Art haben einige nützliche Eigenschaften für mroot, sie können bspw. in Collections abgelegt werden.

Um die Datensinke zu abstrahieren wurde die I/O-Basisklasse `TFile` verwendet. Von dieser Klasse erben eine Vielzahl weiterer I/O-Klassen, z.B. für XML-Dateien oder Datenbanken (s. Abb. 2.1). Sie alle haben dasselbe Verhalten, sodass mroot leicht um die Unterstützung für weitere Datensinken erweitert werden kann.

Portabilität

mroot schlägt als Datenschnittstelle Brücken zwischen verschiedenen Systemen. Die Entscheidung für C++ als Programmiersprache kommt hier der Plattformunabhängigkeit zugute. Für zahlreiche Plattformen gibt es entsprechende Compiler. Durch den weitgehende Verzicht auf externe Bibliotheken

ist die Code-Basis schlank, was Komplikationen bei der Portierung vermeidet. mroot verwendet beim Speichern und Laden ROOTs portable Datentypen anstatt der C++-Standard-Typen, sodass die Schnittstelle unabhängig von der Busbreite des Laufzeitsystems ist. mroot wurde unter Windows und Linux getestet.

Transparenz

Um dem Ziel einer möglichst transparenten Datenschnittstelle von MATLAB zu ROOT zu entsprechen, wurde zunächst evaluiert, was diese Transparenz im Einzelnen bedeutet. Im Wesentlichen läuft es darauf hinaus, dass der Umgang mit ROOT-Dateien keine Umgewöhnung von der Interaktion mit MATLAB-Dateien erfordert:

- Funktionen werden auf gleiche Art und Weise parametrisiert.
- Es gibt keine Einschränkungen bei der Nutzung von Datentypen.

2.1.1 Funktionen

Zuerst galt es also herauszufinden, welche MATLAB-Funktionen [16] überhaupt mit Dateien interagieren. Bei der Einschätzung der Relevanz einer Funktion für mroot ist zu beachten, dass ROOT-Dateien im Gegensatz zu MATLAB-Dateien keine flache Struktur haben, sondern im Innern ein Dateisystem abbilden. Tab. 2.1 zeigt eine Liste der evtl. relevanten Funktionen, die mit Dateien interagieren.

Die Liste ist nach Prioritäten geordnet. Die Funktionen `save`, `load` und `who/whos` wurden tatsächlich für ROOT implementiert, um die vorhandene MATLAB-Funktionalität bereitzustellen und damit dem Konzept der Transparenz zu entsprechen.

Die weiteren in Tab. 2.1 aufgelisteten Datei-Funktionen könnten zukünftig implementiert werden, um MATLAB weitergehende ROOT-Funktionalität verfügbar zu machen. Mit `move`, `copy` und `delete` könnte z.B. auf das virtuelle ROOT-Dateisystem zugegriffen werden.

Die Befehle `open`, `what` und `info` sind dagegen eher Werkzeuge für Programmierer als Funktionen, die in Programmen benutzt werden. Sie stehen in der Priorität für mroot daher weit unten.

Funktion	Beschreibung
Tatsächlich implementierte Funktionen	
<code>save, load</code>	Variablen speichern und laden
<code>who, whos</code>	Variablen im Workspace oder in einer Datei auflisten, ggf. mit Details
Weitere Datei-Funktionen	
<code>mkdir</code>	Verzeichnis anlegen
<code>move, copy, del</code>	Datei verschieben, kopieren und löschen
<code>open</code>	MATLAB-spezifische Datei in GUI öffnen
<code>what</code>	MATLAB-spezifische Dateien in Verzeichnis listen
<code>info</code>	Info zu MATLAB-spezifischer Datei anzeigen

Tabelle 2.1: Liste der Datei-Funktionen in MATLAB

Funktionen überschreiben

Maximal transparent wäre die Nutzung dieser Funktionen mit ROOT-Dateien, wenn sie in irgendeiner Form überschrieben werden könnten, sodass sie neben der Parametrisierung auch den Namen mit ihren MATLAB-Pendants gemein hätten. Zu diesem Zweck wurden verschiedene Möglichkeiten [17] untersucht:

- Eine benutzerdefinierte MATLAB-Funktion hat zwar i.d.R. eine höhere Priorität als eine in MATLAB enthaltene, produziert aber bei jedem Aufruf eine Fehlermeldung. Das ist evtl. nicht nur verwirrend, sondern kann auch in der Ausführungsgeschwindigkeit negativ zu Buche schlagen.
- Der „offizielle“ Weg sieht sogenannte „Private Functions“ vor. Diese werden in einem Verzeichnis mit dem Namen `private` abgelegt. Eine Funktion, die eine solche private Funktion aufruft, muss direkt im Verzeichnis darüber definiert sein. Die mroot Schnittstelle müsste also in jedes Projekt kopiert werden.
- MATLAB-Funktionen können für bestimmte Datentypen überladen werden. Die Funktionen werden in einem Unterverzeichnis `@datentyp` definiert, z.B. `@int16`. Sie wären dann zwar bezüglich der Funktionsnamen transparent, aber nicht bezüglich der Parametrisierung.

Es ist also nicht möglich bzw. für diese Anwendung nicht praktikabel, MATLAB-Funktionen zu überschreiben. Deshalb haben die Funktionen der mroot-Schnittstelle alle das Präfix **r**. Das Pendant zu **save** heißt bspw. **rsave**.

Dennoch wird gewährleistet, dass in einer MATLAB-Anwendung jederzeit die mroot-Funktion anstatt des MATLAB-Originals verwendet werden kann: Bei allen implementierten Funktionen wird zunächst anhand des Dateinamen-Parameters geprüft, ob der Aufruf überhaupt die mroot-Variante betrifft. Andernfalls wird an das MATLAB-Original delegiert. Mit **eval** ist dies leicht zu bewerkstelligen, da alle Argumente einfache Zeichenketten sind. So kann die **mroot**-Funktion überall verwendet werden.

Verhalten der Funktionen eruieren

Da die Quelltexte der MATLAB-Funktionen nicht vorliegen, muss das Verhalten der einzelnen Funktionen durch Ausprobieren eruiert werden. Dies gestaltet sich bisweilen sehr komplex.

Ein Beispiel (s. Abb. 2.2): Der Funktion **save** kann genau ein Struct mitgegeben werden, dessen Felder als einzelne Variablen abgespeichert werden sollen. Werden mehrere Structs aufgeführt, so werden alle außer dem letzten ignoriert. Alle anderen übergebenen Parameter außer dem Dateinamen (der erste Parameter) und weiteren Options-Flags (Präfix **-**) werden als Muster für Feldnamen interpretiert.

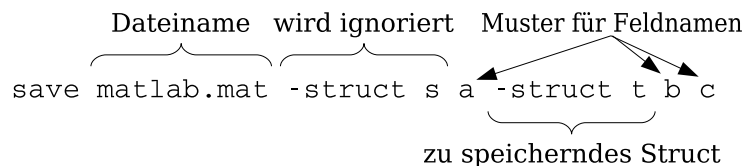


Abbildung 2.2: Verhalten der Funktion **save**

Jede der implementierten Funktionen hat solch schwer durchschaubare Eigenschaften. Sie verleihen ihnen eine größere Toleranz gegenüber fehlerhaften Eingaben, aber auch eine hohe Komplexität. Zudem sind sie nur eine Annäherung an das, was der Benutzer wahrscheinlich tun wollte. Für die mroot-Funktionen wurde deshalb nur die „übliche“ Parametrisierung und Funktionsweise implementiert. Alle Options-Flags, die für ROOT-Dateien sinnvoll sind, sind vorhanden, jedoch ist die Reihenfolge vorgegeben.

save

Die MATLAB-Referenz dokumentiert folgende Syntax für die Funktion **save**:

```
1 save
2 save filename
3 save filename content
4 save filename options
5 save filename content options
6 save('filename', 'var1', 'var2', ...)
```

Listing 2.1: Syntax der Funktion **save**

Wird kein **content** angegeben, werden alle Variablen im Workspace in die Datei mit dem Namen **filename** gespeichert. Wird auch **filename** nicht angegeben, so wird in die Standard-Datei **matlab.mat** im aktuellen Verzeichnis gespeichert. **content** ist eine Liste von Mustern, die die zu speichernden Variablen spezifizieren. Erlaubt sind Variablennamen auch mit Wildcards, das bereits erwähnte Options-Flag **-struct** und das Flag **-regexp**, das dazu führt, dass alle Muster als reguläre Ausdrücke interpretiert werden. Die Option **-append** sorgt dafür, dass an eine bestehende Datei angehängt wird, anstatt sie zu überschreiben. Mit **-mat** und **-ascii** kann angegeben werden, ob die Variablen binär (Standard) oder textuell gespeichert werden sollen.

Für ältere, inkompatible MATLAB-Versionen kann mit **-v<X>** eine Versionsnummer **<X>** übergeben werden. Dies wird dann im Kopf der Datei vermerkt und die Variablen werden im Format dieser Version gespeichert. Es gilt die Version, mit der die Datei angelegt wurde, in Kombination mit **-append** ist diese Option also wirkungslos. Standardmäßig wird die Nummer der tatsächlich verwendeten MATLAB-Version im Dateikopf vermerkt. Der Versuch, die Datei mit einer älteren, inkompatiblen MATLAB-Version zu öffnen (gleichgültig ob lesend oder schreibend), scheitert mit einer Fehlermeldung.

Die **-append**-Option gilt bei **rsave** für ein virtuelles Verzeichnis innerhalb der **ROOT**-Datei und nicht für die Datei selbst. Um **Tfile** und die erbdenden Dateitypen zu unterstützen, sind Format-Flags wie **-mat** unzureichend. Diese Information wurde deshalb in den **filename**-Parameter kodiert (s. 2.2). Das Versionsflag bezieht sich nicht auf verschiedene MATLAB-Versionen, sondern auf verschiedene Versionen von **mroot**. Die Syntax von **rsave** sieht also so aus:

```
rsave [[-v<X>] <filename> [-struct <s>] [-regexp]
      [<patterns>] [-append]]
```

2.1. EVALUATION FÜHRT ZUM KONZEPT

z.B. `rsave matlab.root -struct s a*`

Eckige Klammern stehen für „optional“, Ausdrücke in spitzen Klammern sind Platzhalter.

load

Die Syntax der Funktion `load` ist sehr ähnlich. Variablen können nicht nur einzeln geladen, sondern auch in einem Struct zusammengefasst werden. Dies wird so notiert:

```
s = load('arg1', 'arg2', 'arg3', ...)
```

Eine Versionsnummer muss nicht angegeben werden, da sie aus der Datei selbst ersichtlich ist. `-append` entfällt beim Laden ebenfalls. Die Syntax von `rload` in der Zusammenfassung:

```
[<s> =] rload [<filename> [-regexp] [<patterns>]]
```

z.B. `s = rload matlab.root a*`

who/whos

`who` listet Variablen im Workspace auf, ähnlich wie die Befehle `ls` unter Linux bzw. `dir` unter Windows Dateien auflisten. Auch der Funktion `who` können Muster und reguläre Ausdrücke übergeben werden. Mit dem Schlüsselwort `global` werden Variablen im globalen Workspace gelistet. Wird über die Option `-file` ein Dateiname angegeben, so werden die Variablen dieser Datei angezeigt. Statt der Anzeige können die Variablennamen auch in ein Struct zurückgegeben werden.

`whos` gibt zusätzlich zu den Namen der Variablen noch weitere Informationen aus - ähnlich `ls -l` unter Linux. Dazu gehören die Dimensionalität, die Größe im Speicher, der Typ und diverse andere Eigenschaften der Variablen.

Daraus ergibt sich die folgende Syntax für `rwho/rwhos`:

```
[<s> =] rwho [[-file <filename>] [-regexp] [<patterns>]]
```

z.B. `s = rwho -file matlab.root a*`

2.1.2 Dateitypen

Neben ROOTs I/O-Basisklasse **TFile** gibt es weitere Klassen, die von **TFile** erben (s. Abb. 2.3). Sie haben dasselbe Verhalten wie **TFile**, bringen aber Eigenschaften wie Netzwerkfähigkeit oder die Möglichkeit auf Datenbanken zuzugreifen mit.

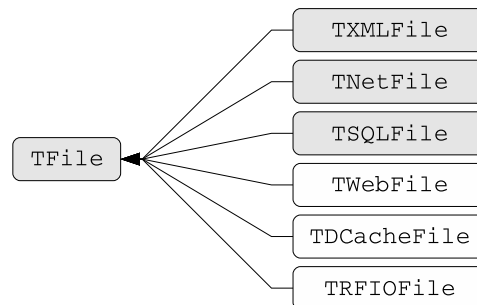


Abbildung 2.3: TFile und Familie

Nicht alle von ROOT unterstützten Dateitypen sind relevant. Auf manche, wie z.B. **TWebFile** und **TArchiveFile**, kann nur lesend zugegriffen werden. Andere sind spezialisierte Schnittstellen zu Produkten wie bspw. zu dCache des DESY-Projekts.

mroot beschränkt sich auf die les- und schreibbaren, produktneutralen Dateitypen **TFile**, **TXMLFile**, **TNetFile** und **TSQLFile**. Aufgrund der Vererbungs-Beziehung von **TFile** ist es jedoch zukünftig kein Problem, Unterstützung für weitere Dateitypen zu implementieren.

TFile ist die Basisklasse der meisten ROOT-Dateitypen. Sie repräsentiert eine einfache, binäre ROOT-Datei mit der Dateierweiterung **.root**.

Um einfacher mit anderen Anwendungen kommunizieren zu können, bringt ROOT die Klasse **TXMLFile** mit. Damit können Daten in XML-Dateien (Dateierweiterung **.xml**) abgelegt werden. Eine von den ROOT-Bibliotheken unabhängige Lösung, um diese Daten zu lesen, existiert aber noch nicht. Da der XML-Baum vor dem Abspeichern zunächst im Hauptspeicher erstellt wird, sollte diese Klasse nur für kleinere Datenmengen genutzt werden.

TNetFile erweitert **TFile** um Netzwerkfähigkeit. Daten werden über den **rootd** Daemon gelesen und geschrieben. Der Dateiname wird dazu im Standard-URL-Format (Protokoll **root**) angegeben. Der Zugriff wird mit Benutzername und Passwort authentisiert. Optional kann die Authentisierung auch mit SRP (Secure Remote Password) oder Kerberos gesichert werden (Protokoll **roots** bzw. **rootk**).

Zugriff auf SQL-Datenbanken ermöglicht die Klasse `TSQLFile`. Eine Datenbank kann dabei eine `ROOT`-Datei aufnehmen, die erforderlichen Tabellen werden automatisch angelegt. Der Zugriff wird per Benutzername und Passwort authentisiert. Unterstützt werden u.a. MySQL, PostgreSQL und Oracle.

Der `ROOT`-Pfad

Eine `ROOT`-Datei beinhaltet ein virtuelles Dateisystem und liegt selbst in einem Dateisystem. `mroot` unterstützt die oben beschriebenen Dateitypen `TFile`, `TXMLFile`, `TNetFile` und `TSQLFile`. Von welchem Typ eine `ROOT`-Datei ist, wird in dem sog. „`ROOT`-Pfad“ ausgedrückt, der den Funktionen als `Dateiname`-Parameter mitgegeben wird.

Ein Rautezeichen (`#`) trennt physikalischen und virtuellen Pfad voneinander:

```
/home/kissel/matlab.root#foo/bar
```

Lokale `TFile`- und `TXMLFiles` werden anhand der Dateinamenserweiterung unterschieden. Dabei wird nicht auf Groß- und Kleinschreibung geachtet:

```
matlab.root#foo/bar
matlab.xml#foo/bar
```

Bei `TNetFiles` und `TSQLFiles` wird der Pfad im URL-Format kodiert, wie es die jeweiligen Konstruktoren verlangen – mit einem Unterschied: Die nötigen Login-Daten werden ebenfalls in den Pfad kodiert:

```
root://kissel:pwd@localhost/~kissel/matlab.root#foo/bar
mysql://kissel:pwd@localhost/matlab
```

Ein `ROOT`-Pfad ist also nach der Syntax in Listing 2.2 aufgebaut. `fsep` ist bei URLs und Linux-Pfaden `'/'` und bei Windows-Pfaden `'\'`.

```
1 rpath ::=  opath [ipath];
2
3 opath ::=  ['/' | (drive ':' '\' ) | url] [path] file ;
4 ipath ::=  '##' [path];
5
```

```

6 url ::= proto login host '/';
7
8 proto ::= (dbms | rootd) '://';
9 login ::= user [':' password] '@';
10 host ::= hostname [':' port];
11
12 dbms ::= 'mysql' | 'pgsql' | 'oracle';
13 rootd ::= 'root' | 'roots' | 'rootk';
14
15 path ::= directory [fsep path];
16 file ::= name ['. ' ext];
17
18 fsep ::= '/' | '\';
19 ext ::= 'root' | 'xml' | other;

```

Listing 2.2: Syntax des ROOT-Pfads in EBNF

2.1.3 Datentypen

MATLABs Basis-Datentypen (Classes) sind ähnlich den von Sprachen wie C++ bekannten (s. Tab. 2.2). Es gibt Ganzzahl-Typen in verschiedenen Größen (**Int8** bis **Int64**), sowohl mit als auch ohne Vorzeichen (signed und unsigned). Die mit der Länge 64 Bit sind zur Zeit aber nur Platzhalter für die Zukunft. Variablen dieser Typen können zwar angelegt werden, es können aber keinerlei Operationen darauf ausgeführt werden. Gleitkommazahlen sind in den zwei Präzisionsstufen **Single** und **Double** möglich (32 bzw. 64 Bit). Für Wahrheitswerte steht die Klasse **Logical** zur Verfügung, Buchstaben werden von **Char** repräsentiert. Auch **Struct** ist bekannt (Menge von Schlüssel-Wert-Paaren). Eine Besonderheit ist die Klasse **Cell**, in der alle Typen beliebig geschachtelt werden können.

Für von anderen Sprachen bekannte Typen wie Enumerations oder Konstanten gibt es keine Entsprechung. Function-Handles wurden als MATLAB-Spezifikum bei der Konzeption von mroot außen vor gelassen, da es sich nicht um Nutzdaten im eigentlichen Sinne handelt. Und MATLAB User Classes oder Java-Klassen wurden nicht betrachtet, weil sie von MEX C++ ohnehin nicht unterstützt werden.

Standardmäßig sind bei MATLAB alle numerischen Werte Fließkommazahlen mit doppelter Präzision. Die Zahltypen können auch komplex sein. *Alle* Typen sind Matrizen, deren Dimensionalität nahezu unbegrenzt ist. Eine Struct-Matrix bspw. ist eine Matrix von Structs mit denselben Feldnamen.

mxClassID	Typ	Beschreibung
1	cell	Zellen beliebiger Art und Dimensionalität
2	struct	Schlüssel-Wert-Paare; Schlüssel: String, Wert: beliebig
3	logical	Wahrheitswerte
4	char	Buchstaben; String = Char-Vektor
5	<i>function</i>	Function Handle; <i>von mroot nicht unterstützt</i>
6	double	Gleitkommazahlen doppelter Präzision; 8 Byte; MATLAB-Standardtyp
7	single	Gleitkommazahlen einfacher Präzision; 4 Byte
8, 10, 12, 14	int	Ganzzahlen mit 8, 16, 32 und 64 Bit Länge; vorzeichenbehaftet; zur Zeit keine 64Bit-Ops.
9, 11, 13, 15	uint	Ganzzahlen mit 8, 16, 32 und 64 Bit Länge; <i>nicht</i> vorzeichenbehaftet

Tabelle 2.2: Übersicht über die MATLAB-Datentypen
(Alle Typen sind Matrizen)

Die MATLAB-Typen können nicht direkt in ROOT Klassen abgebildet werden. Insbesondere die beliebige Dimensionalität der MATLAB-Matrizen ist ein Problem, da es in ROOT nur zweidimensionale Matrizen gibt.

Die Möglichkeit, eigene ROOT-Klassen zu implementieren, um die MATLAB-Datentypen abzubilden, wurde verworfen, da dies den Mehrwert von mroot empfindlich eingeschränkt hätte. Die Daten könnten nur gespeichert und geladen, nicht aber direkt in ROOT weiterverarbeitet werden. Außerdem müssten die neuen Klassen in ROOT einkompiliert werden, was sehr aufwändig gewesen wäre. Deshalb mussten geeignete ROOT-Klassen gefunden werden. Dabei war darauf zu achten, dass diese von der ROOT-Basisklasse TObject erben. Dies hat einige Vorteile, denn deren Instanzen können:

- in ROOT-Collections abgelegt werden (wichtig bei Cell-Arrays).
- sich selbst serialisieren und persistieren.
- sich selbst reflektieren, um ihren Typ zu bestimmen (RTTI).

Der Prototyp aus der Projektarbeit unterstützt nur die für USCT wichtigsten Datentypen – und das unvollständig. Im Einzelnen sind dies reellwertige

2.1. EVALUATION FÜHRT ZUM KONZEPT

Double-Matrizen, Character- und Cell-Arrays und einfache Structs. Das erwähnte Problem mit der Dimensionalität wurde hier durch Reformatierung der Matrizen zu einfachen Arrays und Definition eines Kopfformats gelöst (s. Abb. 2.4). Reformatierung bringt Daten in eine nach ROOT umwandelbare Darstellung.

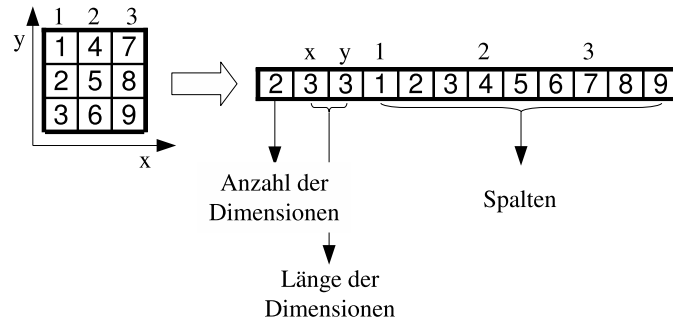


Abbildung 2.4: Umwandlung einer Matrix in ein Array mit Kopfformat

Es gibt in ROOT keine Klasse für Integer-Matrizen oder komplexe Zahlen, zumindest keine, die von `TObject` erbt. Soll all dies – verschiedene Integer-Typen mit und ohne Vorzeichen, komplexe Zahlen, etc. – in die Reformatierung einbezogen werden, so wird das benötigte Kopfformat sehr umfangreich und unflexibel.

Aus diesem Grund werden alle Datentypen bei der Reformatierung in Structs gekapselt. Diese beinhalten jeweils ein `type`- und ein `size`-Feld sowie Felder für die eigentlichen Daten – die weiterhin in ein Array gewandelt werden. Auf diese Weise entfällt das Kopfformat und bei Bedarf können einfach weitere Felder hinzugefügt werden, z.B. ein `complex`-Flag bei numerischen Matrizen. Die Reformatierung erfolgt somit für alle Typen einheitlich. Das `type`-Feld nimmt Werte der für MEX definierten `mxClassID`-Enumeration an. Abb. 2.5 zeigt das Beispiel oben mit der neuen Reformatierung.

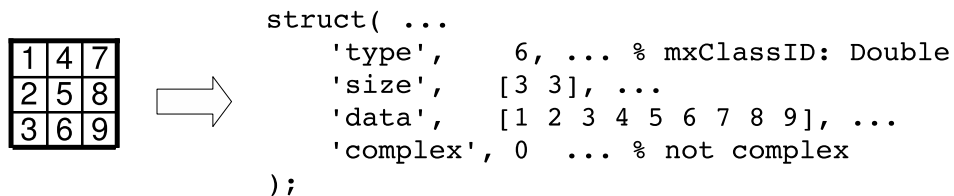


Abbildung 2.5: Umwandlung einer Matrix in ein Struct

Dieses Vorgehen hat noch weitere Vorteile: Daten und Metadaten sind so sauber voneinander getrennt. Außerdem ist das Vorhandensein der beschrie-

benen Struktur ein Indikator dafür, dass die Daten mit Hilfe von mroot gespeichert wurden.

Die Reformatierung bereitet die MATLAB-Variablen beim Speichern auf die eigentliche Umwandlung in ROOT-Objekte vor. Beim Laden gibt sie ihnen nach der Rückwandlung ihre originale Form zurück. Der gesamte Transformations-Vorgang besteht also aus zwei Schritten: Formatieren und Umwandeln.

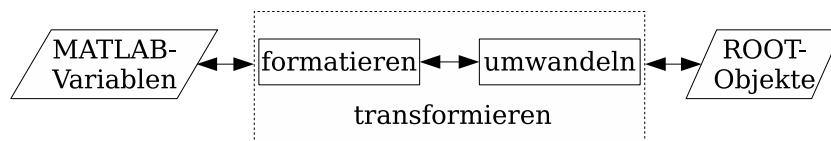


Abbildung 2.6: Transformation der Variablen

2.1.4 Programmierung

Die Art und Weise, in der mroot programmiert wurde, ist von der Erkenntnis geprägt, dass der Aufwand der Software-Entwicklung meist teurer ist als die Laufzeit der Programme. Aus diesem Grund sind Les- und Wartbarkeit sowie Verständlichkeit der Quelltexte sehr wichtig. Ein paar Details:

Bei Programmen ohne automatische Speicherbereinigung sollte ein kritischer Fehler das Programm möglichst weit oben in der Aufrufhierarchie terminieren, um manuelles Freigeben von Speicher zu ermöglichen. Bei mroot transportiert Exception Handling den Fehlerkontext bis in die Fehlermeldungen. Exception Handling hat darüber hinaus den Vorteil, dass die Fehlerbehandlung vom Produktivcode getrennt werden kann.

Man kann nicht auf alle möglichen Fehler prüfen, da ein Programm immer von fremder Software abhängt (Bibliotheken, Betriebssystem, etc.). Tests lohnen deshalb meist nur an kritischen Stellen. Bei mroot sind dies z.B. Dateisystem-Zugriffen, Speicher-Allokation oder MATLAB-Aufrufe aus MEX-Dateien heraus.

Programme wie mroot müssen portabel sein, da sie auf verschiedenen Plattformen benutzt werden sollen. Dies wird u.a. durch standardkonformen Code und Verzicht auf zu spezialisierte Bibliotheken erreicht. Das ist kein Dogma aber eine Regel, die nicht ohne Not gebrochen werden sollte.

Redundanz sollte durch Einsatz von Funktionen und Ausnutzung von Polymorphismus in der Objektorientierung vermieden werden. Insbesondere

Code-Duplizierung macht ein Programm oft unübersichtlich und nachträgliche Änderungen sind fehleranfällig.

Quelltexte sollten unter dem Aspekt geschrieben werden, dass sie später verändert werden. Deshalb sind Literale zu vermeiden, weil sie nicht an zentraler Stelle geändert werden können. Bei `mroot` wurden stattdessen Enumerationen und Konstanten verwendet.

2.2 Implementierung

Kernstück der Implementierung ist die Transformation der MATLAB-Variablen. Sie wird von den wichtigsten Funktionen der `mroot`-Schnittstelle benötigt: von `rsave`, `rload` und `rwhos`. Weitere Funktionen strukturieren den Quellcode und erbringen Hilfsdienste. Manche der in MATLAB implementierten Hilfsfunktionen sind auch außerhalb des `mroot`-Kontexts sehr nützlich, wie bspw. `readdir`.

Die Schnittstelle kann benutzt werden, nachdem sie dem MATLAB-Suchpfad hinzugefügt wurde. Abb. 2.7 zeigt eine Übersicht über das Verzeichnis. Die MATLAB-Funktionen liegen zusammen mit den Software-Lizenzen direkt im `mroot`-Verzeichnis. Im `private`-Verzeichnis finden sich nicht nur die MEX-Dateien, sondern auch Funktionen, die nicht außerhalb des `mroot`-Kontexts benutzt werden sollen. Die Quelltexte der MEX-Dateien liegen in `src`, `test` beinhaltet MUnit-Tests und andere Test-Funktionen. API-Dokumentation in HTML-Form befindet sich getrennt nach MATLAB und C++ (`mat` und `cpp`) im Ordner `doc`.

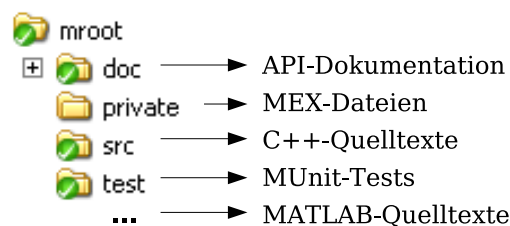


Abbildung 2.7: `mroot`-Verzeichnisstruktur

Der Prototyp aus der Projektarbeit im vierten Semester hatte mit Stabilitätsproblemen zu kämpfen. Er funktionierte zwar, aber immer wieder stürzte MATLAB ohne Hinweis auf den Grund ab – ohne Fehlermeldung und relativ sporadisch. Bevor mit der Entwicklung von `mroot` fortgefahren werden konnte, musste dieser Grund jedoch gefunden werden.

2.2.1 Stabilisierung des Prototyps

Um den Grund für die Stabilitätsprobleme des Prototyps herauszufinden wurde zunächst eine Art „Massentest“ durchgeführt. Dabei wurde eine komplexe MATLAB-Variable (verschachtelte Cell-Arrays) einige hunderttausendmal abgespeichert. Das Absturzverhalten blieb jedoch weiter sporadisch und undurchsichtig. Allerdings war auffällig, dass Abstürze vor allem auftraten, wenn etwas an der MATLAB-Oberfläche verändert wurde, z.B. wenn ein Fenster geschlossen wurde. Da graphische Benutzeroberflächen in MATLAB in einem eigenen Thread laufen, lag die Vermutung nahe, dass es sich evtl. um ein Thread-Problem handeln könnte. Ein Test mit MATLAB im Konsolenmodus widerlegte die Vermutung, da MATLAB weiter abstürzte. In den Konsolenmodus gelangt man, indem man MATLAB mit der Option `-nojvm` startet. Diese bewirkt, dass keine Java Virtual Machine geladen wird und somit auch MATLABs Java-Oberfläche geschlossen bleibt.

Zuletzt sollte die Speichernutzung des Prototyps untersucht werden. Die `mpatrol`-Bibliothek [18] ist dazu ein nützliches Werkzeug. Sie kann detailliert konfiguriert werden und schreibt ihre Ausgaben in eine Log-Datei. Vor allem eine ihrer Funktionen war zielführend, die Funktion, sogenannte Leak Tables ausgeben zu lassen. Diese geben Aufschluss über allokierten, wieder freigegebenen und vor allem nicht freigegebenen Speicher inklusive Zeilennummern (s. Lst. 2.3).

```

1 bottom 7 allocated memory entries in leak table:
2
3     bytes    count    location
4     -----
5         28         4    src/mroot.cpp line 47
6         88         2    src/mex_rsave.cpp line 241
7         ...         ...    ...
8        2672        50    total
9 bottom 7 freed memory entries in leak table:
10
11     bytes    count    location
12     -----
13         28         4    src/mroot.cpp line 47
14         88         2    src/mex_rsave.cpp line 241
15         ...         ...    ...
16        2672        50    total
17 no unfreed memory entries in leak table

```

Listing 2.3: `mpatrol` Leak Tables

Damit mpatrol alle Speicheranforderungen und -freigaben protokollieren kann, wurden die MEX-Funktionen `mxMalloc` und `mxFree` der MATLAB-Speicherverwaltung durch die Standard-Funktionen `malloc` und `free` ersetzt. Es stellte sich heraus, dass der bei der Handhabung von Cell- und Struct-Arrays belegte Speicher nicht mehr freigegeben wurde. Damit war klar, dass das Problem mit ROOTs Collection-Klassen zusammenhing. Aus der ROOT-Referenz ist ersichtlich, dass Collection-Objekte standardmäßig nicht „Besitzer“ der von ihnen gekapselten Objekte sind, wie es erwartet werden könnte. Das bedeutet, dass diese nicht freigegeben werden, sobald das Collection-Objekt selbst freigegeben wird. Die Methode `SetUser` der Collection-Objekte soll zu dem gewünschten Ergebnis führen, funktionierte aber laut mpatrol ebenfalls nicht. Abhilfe schafft nun die Funktion `rdelete`, die Collection-Objekte der Typen `TSeqCollection` (Objekt-Arrays) und `TMap` (Hash Maps) rekursiv löscht.

Es scheint ein MATLAB-Fehler zu sein, dass MATLAB bereits bei kleinsten Speicherlecks, die normalerweise früher oder später vom Betriebssystem geschlossen werden, instabil wird und mit der Zeit abstürzt. Wird auf Bibliotheken verzichtet, die nicht die MATLAB-Speicherverwaltung nutzen, so wird das Problem umgangen: Über Speicher, der mit `mxMalloc` allokiert wurde, führt MATLAB Buch und gibt ihn ggf. wieder frei, falls es der Entwickler versäumt hat. Da mroot aber auf die ROOT-Bibliotheken angewiesen ist, wird darauf verzichtet. So führen Speicherfehler unweigerlich zu Abstürzen und machen so auf sich aufmerksam.

Ein abschließender Massentest mit zufällig zusammengestellten Cell-Arrays führte zu keinen weiteren Auffälligkeiten.

2.2.2 Transformation der Variablen

Beim Speichern in ROOT-Dateien werden die MATLAB-Variablen reformatiert, damit sie auf native ROOT-Klassen abgebildet werden können. Beim Laden gehen sie den umgekehrten Weg, erst die Umwandlung zu MATLAB-Variablen und dann die Reformatierung in die Originaldarstellung.

Dieser Transformationsvorgang ist jeweils in der Funktion `mx2root` bzw. `root2mx` gekapselt. Da in Cell-Arrays Datentypen geschachtelt werden können, sind diese Funktionen rekursiv. Zunächst wird jeweils der Typ des zu transformierenden Objekts bzw. der Variablen geprüft. Ein unbekannter Typ führt zum Programmende mit Fehlermeldung.

Formatierung mit `reformat`

Die Anpassung der Variablen für die Umwandlung übernimmt die Funktion `reformat`. Als kritischer Punkt der Implementierung ist sie in MATLAB implementiert und wird so in der geschützten Umgebung des MATLAB-Interpreters ausgeführt. Außerdem können auf diese Weise hochperformante MATLAB-Funktionen wie `reshape` einfach benutzt werden. `reformat` wandelt für Speichern *und* Laden, die Richtung wird durch einen Wert der Enumeration `Dir_t` in `mroot.h` angegeben.

Beim Speichern wird aus der MATLAB-Variablen ein Struct erzeugt, das im Feld `data` die eigentlichen Daten enthält und weitere Felder für Metadaten beinhaltet (s. Tab. 2.3). Der Typ wird konsequenterweise als `mxClassID` gespeichert und mit der `mroot`-Funktion `classid` bestimmt. Diese bedient sich der MATLAB-Funktion `isa`, um die ID zu ermitteln. `mxClassIDs` sind sonst nur im MEX-Umfeld bekannt, aber nicht direkt in MATLAB.

Für Struct-Arrays kommt ein Feld mit Namen „fields“ hinzu, in dem die Feldnamen als Cell-Array abgelegt werden. Auch das `data`-Feld beinhaltet hier ein Cell-Array: Ein $m \times n$ -Struct-Array mit p Feldern wird von der MATLAB-Funktion `struct2cell` zu einem Cell-Array der Dimensionalität $p \times m \times n$ transformiert.

Bei allen Typen wird das `data`-Feld mit der MATLAB-Funktion `reshape` von einer beliebig dimensional Matrix in ein einfaches Array gewandelt, weil die Dimensionalität der ROOT-Klassen begrenzt ist. Dabei geht die Information über die ursprünglichen Abmessungen verloren. Sie wird zuvor im Feld `size` gespeichert.

Bei numerischen Typen wird das Struct um ein `complex`-Flag ergänzt. Hat dieses den Wert 1 (komplexe Matrix), so wird das `data`-Feld durch die beiden Felder `real` und `imag` (Real- und Imaginärteil) ersetzt. Der Wert 0 steht für reellwertige Matrizen. Diese Zuordnung entspricht der MEX-Enumeration `mxComplexity`.

Bei allen Variablen außer komplexen Matrizen und Struct-Arrays behalten die Daten auch nach der Reformatierung ihren Typ. Metadaten wie das `type`-Feld haben den MATLAB-Standardtyp `double`.

Die Transformation beim Laden funktioniert nach dem umgekehrten Prinzip. Der Typ ist aus dem `type`-Feld direkt ersichtlich. Bei komplexen, numerischen Matrizen werden Real- und Imaginärteil wieder zusammengeführt. Bei allen Typen wird die Dimensionalität anhand des `size`-Feldes mit `reshape` wiederhergestellt. Aus dem Datenfeld und den Feldnamen wird ggf. mit `cell2struct` wieder ein Struct-Array erzeugt.

MATLAB-Typ	type	size	data	fields	complex	real	imag
cell	x	x	x				
struct	x	x	x	x			
logical	x	x	x				
char	x	x	x				
num (real)	x	x	x		x		
num (complex)	x	x			x	x	x

Tabelle 2.3: Struct-Felder nach der Reformatierung beim Speichern

Umwandlung mit `mx2root` und `root2mx`

Die eigentliche Umwandlung von MATLAB-Variablen in ROOT-Objekte und zurück (s. Tab. 2.4) erfolgt direkt in den Funktionen `mx2root` bzw. `root2mx`.

mx2root In `mx2root` iteriert eine Schleife über die Felder des von `reformat` erzeugten Structs und erzeugt daraus eine `TMap`. In dieser Schleife wird anhand des Typs der Felder verzweigt und die Felder werden umgewandelt. Der Feldname wird jeweils von `mxGetObjStr` in einen `TObjString` kopiert. Zusammen mit dem transformierten Feld wird er der `TMap` als neues Schlüssel-Wert-Paar hinzugefügt.

Bei Cell-Arrays werden die einzelnen Zellen durch einen rekursiven Aufruf transformiert und in einem `TObjArray` abgelegt. Structs als Felder können nach der Reformatierung nicht mehr vorkommen. Logical-Arrays werden bitweise in einen `TBits`-Container überführt. Character-Arrays werden wiederum mit `mxGetObjStr` in einen `TObjString` kopiert.

Der Speicher von Gleitkomma-Variablen wird bei der Erzeugung eines `TVectorT`-Objekts kopiert. Diese Klasse gibt es in zwei als generische Typen (Generics) realisierten Ausprägungen für einfache und doppelte Gleitkomma-Präzision (`float` und `double`).

Integer-Matrizen werden wie Logical-Arrays ebenfalls in einem `TBits`-Container gespeichert. Die `TBits`-Klasse bietet dazu geeignete Methoden. Diese verlangen jeweils einen Zeiger des gewünschten Typs. Es werden aber nicht die Standard-C++-Typen verwendet, sondern die prinzipiell gleichwertigen aber plattformunabhängigen ROOT-Typen. Bspw. wird statt `long` der ROOT-Typ `Long64_t` benutzt, der immer 64 Bit lang ist, auch wenn die Laufzeitumgebung eine 32-Bit-Architektur ist.

MATLAB-Typ	ROOT-Klasse
<code>cell</code>	<code>TObjArray</code>
<code>struct</code>	<code>TMap</code>
<code>logical</code>	<code>TBits</code>
<code>char</code>	<code>TObjString</code>
<code>float</code>	<code>TVectorT</code>
<code>int</code>	<code>TBits</code>

Tabelle 2.4: Zuordnung von MATLAB-Typen zu ROOT-Klassen

root2mx `root2mx` wird für das übergebene ROOT-Objekt aufgerufen und rekursiv für jedes darin enthaltene (komplexe oder elementare) ROOT-Objekt. Bei der Rekursion wird Polymorphismus ausgenutzt, die übergebenen Objekte haben alle den Basistyp `TObject`. Da Objekte dieses Typs über Runtime Type Information (RTTI) verfügen, kann die eigentliche Klasse bestimmt werden. Hierfür wurde die Funktion `rootGetClassID` entwickelt, die in Analogie zu der MEX-Funktion `mxGetClassID` für die detektierte ROOT-Klasse die entsprechende `mxClassID` zurückgibt.

`root2mx` ist im Wesentlichen eine große Fallunterscheidung über den Typ des übergebenen ROOT-Objekts. Der im `type`-Feld gespeicherte Wert wird lediglich zur weiteren Unterscheidung bestimmter Fälle herangezogen. Durch diese Vorgehensweise müssen weniger Fälle unterschieden werden und es gibt weniger Code-Redundanz.

In der Fallunterscheidung wird für jeden Typ zunächst mittels Cast die Original-Klasse des Objekts wiederhergestellt, um Zugriff auf die typspezifischen Methoden zu erhalten. Dann wird auf typspezifische Weise eine MATLAB-Variable erzeugt und mit den Daten aus dem ROOT-Objekt gefüllt. `TObjArrays` und `TMaps` werden rekursiv zu Cell- und Struct-Arrays gewandelt.

MATLAB-Variablen werden mit spezialisierten MEX-Funktionen erzeugt:

- `mxCreateCellMatrix` für Cell-Arrays
- `mxCreateStructMatrix` für Struct-Arrays
- `mxCreateLogicalMatrix` für Logical-Arrays
- `mxCreateCharMatrix` für Character-Arrays
- `mxCreateNumericMatrix` für numerische Matrizen

2.2. IMPLEMENTIERUNG

Allen diesen Funktionen muss die Länge der beiden Matrix-Dimensionen mitgegeben werden. `mx2root` bildet alle Matrizen auf $1 \times n$ -Matrizen ab.

Bei `mxCreateNumericMatrix` muss darüber hinaus die Komplexität (komplex oder nicht komplex) sowie der genaue Typ (z.B. Double) angegeben werden.

Weil für die Erzeugung eines Struct Arrays mittels `mxCreateStructMatrix` alle Feldnamen bekannt sein müssen, muss zuvor die Iteration über die Felder der TMap stattfinden, wobei die Feldnamen und auch die Feldinhalte gepuffert werden.

Da die Funktion `rootGetClassID` bei einem TBits-Container nicht unterscheiden kann, ob er ein Logical-Array oder eine Ganzzahl-Matrix enthält, stuft sie ihn als Logical-Array ein. Einen Ganzzahl-Inhalt sowie die Länge des konkreten Typs (z.B. 32 Bit) erkennt `root2mx` anhand des `type`-Feldes.

Die Reformatierung in die endgültige MATLAB-Darstellung geschieht mit der `mroot`-Funktion `reformat`. Der Aufruf dieser Funktion befindet sich am Ende des Struct-Abschnitts der Fallunterscheidung. Er wird nur dort benötigt, da bei der Erzeugung des ROOT-Objekts mittels `mx2root` die MATLAB-Variable unabhängig vom Typ in Structs gekapselt wurde.

2.2.3 Dateiname parsen

Die Funktionen der Datenschnittstelle `mroot` haben alle einen Dateinamen-Parameter, der einen ROOT-Pfad beinhaltet. Dieser wird von der Funktion `pathinfo` jeweils auf Validität geprüft und in seine Bestandteile zerlegt. Die Funktion gibt zwei Structs (`ofile` und `ifile`) mit Informationen zum Umgang mit der physikalischen Datei und dem darin enthaltenen virtuellen Pfad zurück (s. Tab. 2.5).

Feld	Beschreibung
<code>ofile</code>	physikalische Datei (outer)
<code>type</code>	Dateityp; Werte aus der Enumeration <code>File_t</code> in <code>mroot.h</code>
<code>path</code>	Der eigentliche Dateipfad
<code>usr, pwd</code>	Benutzername und Passwort bei TNet- und TSQLFile
<code>ifile</code>	virtueller Pfad (inner)
<code>path</code>	Array mit virtuellen Unterverzeichnissen in der ROOT-Datei

Tabelle 2.5: Übersicht über die Rückgaben von `pathinfo`

2.2. IMPLEMENTIERUNG

Für Dateien, die `mroot` nicht betreffen, z.B. MATLAB-Dateien oder Dateien unbekannten Typs, hat das `type`-Feld den Wert -1.

Da für `TNet`- und `TSQLFile` Login-Informationen benötigt werden, wird zunächst untersucht, ob der Pfad ein `@`-Zeichen enthält. In diesem Fall wird davon ausgegangen, dass es sich um einen der beiden Dateitypen handelt und Protokoll und Login-Informationen werden extrahiert (s. Abb. 2.8, 1a). Das Protokoll wird mit zwei Whitelists für `ROOT` Daemons und DBMS überprüft. Es entscheidet über den Dateityp. Ein unbekanntes Protokoll sorgt für eine Fehlermeldung. Durch die beschriebene Zerlegung dürfen Benutzername und Passwort keinen Doppelpunkt und kein `@`-Zeichen enthalten. Der Pfad wird ohne die Login-Informationen wieder zusammengefügt.

Ist die Datei weder ein `TNet`- noch ein `TSQLFile`, so handelt es sich um eine lokale Datei. In diesem Fall entscheidet bei noch nicht existierenden Dateien die Dateierweiterung über den Typ (s. Abb. 2.8, 1b). Ist sie `.root`, ist es `TFile`, die Endung `.xml` steht für `TXMLFile`. Bei bereits existierenden Dateien wird der Dateinhalt geprüft, um den Dateityp zu bestimmen. Die ersten vier Bytes einer `ROOT`-Datei sind die vier Buchstaben `root`. Bei einer XML-Datei heißt der Root-Tag `root`. Dieser letzte Test erzeugt aus ungeklärten Gründen trotz des ihn umgebenden `try-catch`-Blocks eine Fehlermeldung, die aber keine weiteren Folgen nach sich zieht.

Unabhängig vom Dateityp wird anschließend der Pfad am Rautezeichen (`#`) in physikalischen und virtuellen Dateipfad getrennt. Der virtuelle Pfad wird darüber hinaus am jeweiligen Verzeichnis-Trenner (`'/'` bzw. `'\'`) in die einzelnen Unterverzeichnisse innerhalb der Datei zerlegt (s. Abb. 2.8, 2). Das ist notwendig, damit diese ggf. später sequentiell angelegt werden können.

```
1a  root: //usr:pwd@localhost/matlab.root#foo/bar
      proto  usr  pwd
      root://localhost/matlab.root#foo/bar

      b  matlab.root#foo/bar
          ext

2  matlab.root#foo/bar
```

Abbildung 2.8: Zerlegung des `ROOT`-Pfades
(Syntaktische Zeichen sind rot markiert.)

2.2.4 MATLAB-Wrapper für MEX-Dateien

rsave, **rload** und **rwhos** stellen die Hauptfunktionalität der mroot-Schnittstelle (s. Tab. 2.6). Als Funktionen für den Umgang mit Workspace-Variablen sind sie miteinander verwandt und dementsprechend ähnlich aufgebaut.

rsave	Speichert Variablen in einer ROOT-Datei. <code>rsave [[-v<X>] <filename> [-struct <s>] [-regexp] [<patterns>] [-append]]</code>
rload	Läd Variablen aus einer ROOT-Datei. <code>[<s> =] rload [<filename> [-regexp] [<patterns>]]</code>
rload	Listet Variablen des Workspace oder einer ROOT-Datei auf. <code>[<s> =] rwho [[-file <filename>] [-regexp] [<patterns>]]</code>

Tabelle 2.6: Übersicht über die mroot-Funktionen

Bei jeder der Funktionen wird zunächst geprüft, ob alle Argumente Strings sind. Schließlich werden sie im Wesentlichen mit Variablennamen und einem Dateinamen parametrisiert. Dieser wird als nächstes von **pathinfo** geparkt. U.a. wird dabei der Typ der Datei bestimmt. Handelt es sich nicht um eine ROOT-Datei, also bspw. um eine MATLAB-Datei oder einen anderen, unbekannten Typ, so wird der Aufruf mit Hilfe der Funktion **eval** an die MATLAB-Originalfunktion delegiert. Im Fall von **rsave** wäre das z.B. **save**.

Bei ROOT-Dateien werden stattdessen Standardwerte für nicht gesetzte Parameter gesetzt bzw. es wird auf Options-Flags geprüft, z.B. auf **-regexp**. Ein unbekanntes Options-Flag führt zur Beendigung des Programms mit Fehlermeldung. Daraufhin werden die Parameter für die jeweilige MEX-Datei zusammengestellt. Reguläre Ausdrücke und Wildcards werden durch echte Variablen- oder Feldnamen ersetzt, wobei Wildcards zunächst in reguläre Ausdrücke übersetzt werden. Dann folgt der Aufruf der MEX-Datei. Zuletzt werden evtl. Ausgaben erzeugt.

Das Prüfen und Zerlegen des Dateinamens ist die einzige Funktionalität, die ausgelagert werden kann. Trotz deutlicher Parallelen in der Signatur der Funktionen können ihre sonstigen Parameter nicht alle durch eine gemeinsame Funktion überprüft werden. Das liegt daran, dass diese Überprüfung teilweise im Kontext der aufrufenden Funktion stattfinden muss, um z.B. beim Speichern die Existenz der zu speichernden Variablen festzustellen. Das wäre bei einer gemeinsamen Funktion nur wieder der Kontext von **rsave**.

rsave

rsave ergänzt das **ofile**-Struct aus **pathinfo** um die übergebene und validierte Dateiversion und **ifile** ggf. um das **-append**-Flag. Zusammen mit einer Liste der zu speichernden Variablen (**vlist**) werden sie in dem Cell-Array **rootinfo** zusammengefasst und mit **assignin** in den Workspace der aufrufenden Funktion kopiert. Da nämlich die zu speichernden Variablen in ebendiesem Workspace liegen, muss der Aufruf der MEX-Datei **mex_rsave**, die das eigentliche Speichern übernimmt, in dessen Kontext erfolgen. Dies geschieht mit **evalin**, womit auch **rootinfo** wieder gelöscht wird. Der Variablenname „**rootinfo**“ sollte in Programmen, die **mroot** nutzen, vermieden werden.

rload

rload prüft zunächst mit Hilfe von **rwho**, welche der angeforderten Variablen in der angegebenen Datei vorhanden sind. Diese werden dann von **mex_rload** in einem Struct zusammengefasst geladen. Falls die Variablen nicht en bloc zurückgegeben werden sollen, werden sie einzeln mit **assignin** in den Workspace der aufrufenden Funktion kopiert.

rwho/rwhos

rwho filtert die von **mex_rwho** in der angegebenen Datei gefundenen Variablenamen anhand der übergebenen Muster. Die Ausgabe erfolgt wahlweise als Cell-Array oder als Liste im MATLAB Command Window.

rwhos bedient sich ihrer Schwesterfunktion **rwho**, um die Existenz der angeforderten Variablen zu prüfen. Diese werden anschließend einzeln mit **mex_rload** geladen. Sie werden nicht alle auf einmal geladen, um den Speicherverbrauch zu minimieren. Da der Dateiname **pathinfo** aber bereits durchlaufen hat, wird auf die Redundanz in **rsave** verzichtet (s. Abb. 2.9). Die geladenen Variablen werden dann von der eingebauten Funktion **whos** untersucht und die Ergebnisse werden in einem Struct-Array abgelegt. **rwhos** hat keine eigene, korrespondierende MEX-Datei. Das resultierende Struct-Array wird entweder zurückgegeben oder als Tabelle im MATLAB Command Window ausgegeben.

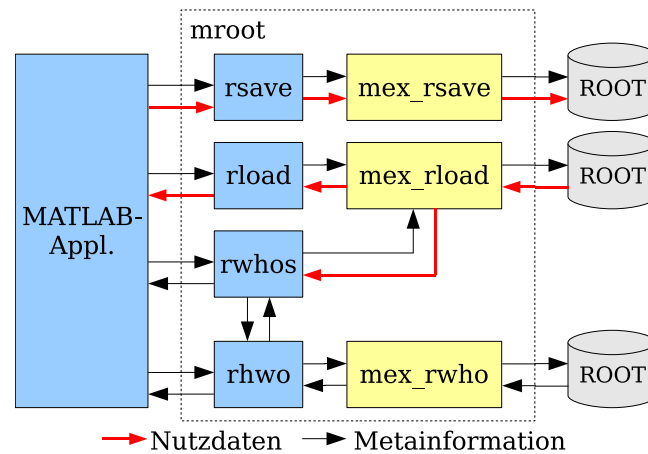


Abbildung 2.9: Datenfluss in mroot

2.2.5 MEX-Dateien

Auch mroots MEX-Dateien sind sich in der Struktur ähnlich. Deshalb teilen sie sich eine gemeinsame Header-Datei, `mroot.h`. Diese bindet weitere Header-Dateien für MEX-Funktionen und ROOT-Klassen ein. Außerdem definiert sie interne mroot-Enumerations:

- `File_t`: Die unterstützten ROOT-Dateitypen
- `Msg_t`: Nachrichten-Typen; Fehler, Warnungen und Nachrichten
- `Dir_t`: Richtung bei der Reformatierung

Dank dieser Enumerations muss bspw. nicht extra geprüft werden, ob der gewünschte Dateityp existiert. Für die Typen und die Komplexität von MATLAB-Variablen bringt MEX bereits eigene Enumerations mit: `mxClassID` und `mxComplexity`. Die Header-Datei `mroot.h` stellt den MEX-Dateien desweiteren einige, allgemeine Funktionen zur Verfügung:

- `mexMsgTxt` fasst die MEX-Funktionen `mexPrintf`, `mexWarnMsgTxt` und `mexErrMsgTxt` zusammen und bietet für alle den Gebrauch von Format-Tags. `mexPrintf` ist die einzige der genannten MEX-Funktionen, die dies unterstützt. Der Typ der Nachricht wird durch einen Wert der `Msg_t`-Enumeration unterschieden.

- `mxGetStr` und `mxGetObjStr` extrahieren einen String aus einem MATLAB-Char-Array. Im Gegensatz zu der MEX-Funktion `mxGetString` werden sie imperativ und nicht funktional aufgerufen. `mxGetStr` gibt ein null-terminiertes C++-Char-Array zurück. `mxGetObjStr` nutzt dies, um selbst einen `ROOT-TObjString` zurückzugeben.
- `openFile` öffnet eine ROOT-Datei des gewünschten Typs. Der Funktion wird `ofile` aus `pathinfo` übergeben. Sie prüft auch die Dateiversion, bzw. speichert sie ab, falls dies noch nicht geschehen ist. Dabei bedient sie sich der Funktion `getFVer`, die nach dem Öffnen der Datei die Version ausliest.
- `cd_seq` wechselt in ein bestimmtes Verzeichnis innerhalb einer ROOT-Datei. Die Funktion wird mit einem Array von Unterverzeichnissen parametrisiert. Durch den schrittweisen Verzeichniswechsel können beim Speichern ggf. nicht existierende Unterverzeichnisse angelegt werden.
- `rdelete` löscht ROOT-Collections rekursiv. Laut `mpatrol` funktioniert nämlich die `SetUser`-Methode der ROOT-Collections nicht, die sie zu „Besitzern“ der von ihnen gehaltenen Objekte macht. Somit werden diese Objekte nicht gelöscht, wenn die jeweilige Collection gelöscht wird.
- `reformat` kapselt für `mex_rsave` und `mex_rload` den Aufruf der MATLAB-Funktion `reformat`. Für diesen Aufruf mit Hilfe der MEX-Funktion `mexCallMATLAB` werden die Ein- und Ausgabe-Arrays zusammengestellt.

In den MEX-Dateien `mex_rsave`, `mex_rload` und `mex_rwho` werden zunächst die einzelnen Argumente aus dem Eingabe-Array extrahiert. Dann wird mit `openFile` die gewünschte ROOT-Datei geöffnet. Anschließend wird das gewünschte Unterverzeichnis innerhalb der ROOT-Datei angesteuert, indem `cd_seq` mit dem in Unterverzeichnisse zerlegten, internen Pfad aufgerufen. Nach der eigentlichen Funktionalität wird die ROOT-Datei wieder geschlossen.

Es gibt in ROOT für ein `TDirectory` keine Methode, mit der überprüft werden kann, ob eine „Datei“ in dem Verzeichnis existiert. Stattdessen wird auf einer Liste der Verzeichnis-Schlüssel die Methode `Contains` mit dem Dateinamen aufgerufen.

Die MEX-Dateien verfügen über Exception-Handling. Für jede Exception ist eine eindeutige Nummer vergeben. Exceptions werden bis in die Einsprungsfunktion `mexFunction` der MEX-Datei durchgereicht, damit evtl. allozierter Speicher freigegeben werden kann. Danach wird das Programm mit einer Fehlermeldung terminiert, die der Exception-Nummer zugeordnet ist.

mex_rsave

Falls beim Speichern das `-append`-Flag nicht gesetzt ist, wird zunächst das Zielverzeichnis geleert. Eine Schleife iteriert über die übergebene Variablenliste `vlist`. Es wird jeweils ein Variablenname extrahiert und die zugehörige Variable mit `mx2root` in ein ROOT-Objekt transformiert. Dieses wird als Kopie in der ROOT-Datei persistiert und anschließend von `rdelete` rekursiv gelöscht. Der Schreib-Funktion werden mit binärem Oder verknüpfte, statische Konstanten der `TObject`-Klasse als Option mitgegeben:

- `kOverwrite`: Beim Schreiben in die Datei werden evtl. existierende Objekte gleichen Namens überschrieben.
- `kSingleKey`: Collections werden beim Schreiben entgegen den ROOT-Standard-Einstellungen nicht in ihre Bestandteile zerlegt.

mex_rload

Auch beim Laden iteriert eine Schleife über die übergebene Variablenliste `vlist`. Das zu einem Variablenname gehörige ROOT-Objekt wird aus der Datei geladen, mit `root2mx` in eine MATLAB-Variable transformiert und wie auch der Name in einem Array gepuffert. Aus diesen beiden Arrays – dem Namens- und dem Variablen-Array – wird anschließend ein Struct erzeugt, das an `rload` zurückgegeben wird.

mex_rwho

In `mex_rwho` iteriert eine Schleife über die Liste der Verzeichnis-Schlüssel. Bei jedem Schlüssel wird für das zugehörige ROOT-Objekt rudimentär geprüft, ob es mit `mroot` in der Datei abgelegt wurde. Dazu muss es sich um eine `TMap` handeln, die den Schlüssel `type` enthält. In diesem Fall wird der Objektname einem Cell-Array hinzugefügt, das am Ende an `rwho` zurückgegeben wird.

2.2.6 MATLAB-Hilfsfunktionen

Übersetzung mit `compile`

Mit wachsendem Funktionsumfang der `mroot`-Schnittstelle stieg auch die Komplexität beim Aufruf des MEX-Compilers. Immer mehr Bibliotheken

und Flags wurden hinzugefügt, bis die Übersicht im MATLAB Command Window verloren ging. Da die meisten dieser Parameter von der Header-Datei `mroot.h` benötigt werden, die in alle anderen C++-Dateien `mroots` eingebunden wird, unterscheiden sie sich nicht von Quelltext-Datei zu Quelltext-Datei. So ist es einfach, sie in einer Funktion zu kapseln, die nur noch mit den Namen der jeweiligen Dateien parametrisiert wird.

Der Funktion `compile` wird als erster Parameter die Quelltext-Datei übergeben, die der resultierenden MEX-Datei den Namen gibt. Darauf folgen weitere Quelltext-Dateien, die dazu gebunden werden, z.B.:

```
compile mex_rsave.cpp mroot.cpp
```

Diesen Dateinamen wird dann intern das Quellen-Verzeichnis vorangestellt. Mit Compile- und Link-Flags werden sie zu einem Aufruf-String zusammengefügt, der dann evaluiert wird. Durch die Realisierung dieses Build-Mechanismus in MATLAB ist er plattformunabhängig und zudem in die MEX-Umgebung integriert.

Flag	Beschreibung
Compile-Flags	
-g	Debug-Symbole erzeugen
-I	Include-Pfad hinzufügen
-D	Symbol definieren
Link-Flags	
-L	Bibliotheks-Pfad hinzufügen
-l	Bibliothek einbinden
-outdir	Ausgaberverzeichnis angeben

Tabelle 2.7: Liste der Compile- und Link-Flags

Tab. 2.7 zeigt einer Übersicht über die verwendeten Compile- und Link-Flags. Zwei -D-Flags definieren die Symbole `DEBUG` und `PATROL`, die Debug-Zwecken dienen. -DDEBUG schaltet Debug-Ausgaben mittels der `mroot`-Funktion `mexMsgTxt` frei, das Flag -DPATROL aktiviert `mpatrol`, das Leak Tables in eine Log-Datei schreibt.

Mit -l werden Bibliotheken eingebunden, für `mpatrol` und für `ROOT`. Die Bibliotheken `libCore` und `libCint` werden von *allen* verwendeten `ROOT`-Klassen benötigt. Die `TVectorT`-Klassen brauchen außerdem die Bibliotheken `libMatrix` und `libMathCore`. `TFile` nutzt die Klasse `libRIO`. Die anderen I/O-Klassen benötigen zusätzlich jeweils eigene Bibliotheken:

- `TXMLFile`: `libXMLIO`
- `TNetFile`: `libNet` und `libRootAuth`
- `TSQLFile`: `libSQL` und `libMySQL`

Dateiversion vergleichen mit `vercmp`

Die Versions-Flags für die Kompatibilität beim Speichern und Laden stehen für die jeweils minimal kompatible mroot-Version, die unterstützt werden soll, d.h. unter Umständen nicht für die tatsächlich Version. Hätte sich bspw. bei einer fiktiven, zukünftigen mroot-Version 1.5 seit zwei Minor Releases nichts an der Art und Weise Daten abzulegen geändert, so wäre das entsprechende Versions-Flag `-v1.3`. Diese Versionsnummer wird dann an die MEX-Datei übergeben.

Da mroot nicht auf die physikalische Struktur einer ROOT-Datei zugreift, sondern lediglich API-Funktionen nutzt, um Objekte darin zu speichern, kann die Versionsnummer nicht im Dateikopf vermerkt werden. Stattdessen wird ein Struct mit einem Versionsfeld im Wurzelverzeichnis der Datei abgelegt. Ein Struct hat den Vorteil, dass es in Zukunft noch weitere Steuerinformationen aufnehmen könnte.

Die Version wird beim Speichern und Laden jeweils direkt nach dem Öffnen der ROOT-Datei geprüft, bzw. – falls sie nicht gefunden wurde – gespeichert. Zur Prüfung wird sie zunächst mit der Funktion `getFVer` (get file version) geladen. Die Funktion geht dabei davon aus, dass das Wurzelverzeichnis der Datei gerade das aktuelle Verzeichnis ist. Ist die in der Datei gespeicherte Versionsnummer größer als die der MEX-Datei übergebene, so wird der Vorgang mit einer Fehlermeldung abgebrochen.

Der Versionsvergleich erfolgt mit Hilfe der mroot-Funktion `vercmp` (version comparison), die auch separat im MATLAB File Exchange zur Verfügung gestellt wurde. MATLAB bietet zwar mit `verLessThan` eine Funktion für Versionsvergleiche, allerdings kann hier nur die Version der installierten MATLAB-Produkte (Toolboxen, etc.) mit einer übergebenen Versionsnummer verglichen werden. Auch einfache String-Vergleiche sind aufgrund der Struktur einer Versionsnummer nicht zielführend. Eine Versionsnummer hat oft diese oder eine ähnliche Struktur:

```
<major release>.<minor release>.<patch level>.<build number>
```

Alles außer dem Major Release ist dabei optional. `vercmp` vergleicht zwei beliebige Nummern miteinander und gibt detailliert Aufschluss darüber, an

2.2. IMPLEMENTIERUNG

welcher Stelle sie sich wie unterscheiden. Für gleiche Versionsnummern liefert die Funktion den Wert 0. Ist ein Parameter keine Versionsnummer, ist das Ergebnis `NaN` (Not a Number). Wäre bspw. Version 1 im Minor Release kleiner als Version 2, so würde -2 zurückgegeben, was sich als „kleiner an zweiter Stelle“ liest.

```
1 vercmp('1.0', '1.0') % result: 0
2 vercmp('1.0', '1.3') % result: -2
3 vercmp('1.0', 'foo') % result: NaN
```

Listing 2.4: Beispiele zu `vercmp`

Strings zerlegen und zusammenfügen mit `explode` und `implode`

Vor seiner Nutzung wird der Dateinamen-Parameter in seine Bestandteile zerlegt. Bei der Übersetzung der MEX-Dateien setzt `compile` einen Aufruf-String zusammen. Das sind nur zwei Beispiele, bei denen ein String an einem Teilstring auseinandergenommen, bzw. zusammengefügt wird.

MATLAB bietet für das Zerlegen eines Strings an einem Teilstring die Funktion `strtok` (string tokenizer). Strings werden in MATLAB als Buchstaben-Matrizen gehandhabt und können so einfach per Vektor-Konkatenation aneinandergehängt werden. Für die oben beschriebenen Anwendungen ist damit jedoch noch Einiges an Handarbeit erforderlich.

PHP kennt dafür die beiden Funktionen `explode` und `implode`. `explode` zerlegt einen String anhand eines gegebenen Trenner-Strings in ein String-Array, aus dem `implode` mit dem selben Trenner-String den Ausgangsstring wiederherstellen kann. Diese beiden Funktionen wurden in MATLAB nachprogrammiert.

```
1 a = 'This is a test.';
2 b = {'This' 'is' 'a' 'test.'};
3
4 explode(a, ' '); % result: b
5 implode(b, ' '); % result: a
```

Listing 2.5: Beispiele zu `explode` und `implode`

Verzeichnisse durchsuchen mit `readdir`

Für den Test unter Realbedingungen sollten die Messdaten eines USCT-Experiments vollständig in eine ROOT-Datei überführt werden. Ein solches Experiment besteht aus zahlreichen MATLAB-Dateien mit der Dateierweiterung `.mat`, die in einer projektspezifischen Verzeichnishierarchie abgelegt sind. Für den Test mussten diese Dateien also zunächst gefunden werden.

Dazu dient die Funktion `readdir`. Sie durchsucht rekursiv ein Verzeichnis. Die Rekursionstiefe kann angegeben werden: Mit 0 wird nur das angegebene Verzeichnis durchsucht. Positive Werte stehen für die maximale Zahl an Rekursionsebenen, die außerdem durchlaufen werden. Ein negativer Wert bewirkt, dass alle Rekursionsebenen bearbeitet werden.

Bei Bedarf können die Suchergebnisse gefiltert werden, indem ein function handle übergeben wird. Dieser entscheidet für eine Datei, ob sie relevant ist oder nicht, z.B. `isdir`:

```
readdir('~', 2, @isdir);
```

Das Beispiel durchsucht das Home-Verzeichnis inklusive zwei Unterverzeichnis-Ebenen und gibt die Namen der durchlaufenen Verzeichnisse zurück. Standardmäßig wird nur das aktuelle Verzeichnis ohne Filter durchsucht.

Kapitel 3

Ergebnisse

3.1 Benutzerschnittstelle

Setup

Um mroot zu benutzen, muss der Pfad der Schnittstelle dem MATLAB-Suchpfad hinzugefügt werden:

```
>> addpath /path/to/mroot
```

Übersetzte Versionen der MEX-Dateien sind für Linux und Windows (Dateiendung `.mexglx` bzw. `.mexw32`) bereits mitgeliefert. Bei Bedarf können sie mit `compile` neu übersetzt werden, z.B. `mex_rsave`:

```
>> compile mex_rsave.cpp mroot.cpp
```

`mroot.cpp` wird in allen MEX-Dateien `mroots` verwendet. Ggf. müssen in `compile` das Quell- und Zielverzeichnis angepasst werden (`SRC_DIR` bzw. `BIN_DIR`). Außerdem können dort die Debug-Funktionen deaktiviert werden, indem die `-D`-Flags auskommentiert werden.

Nutzung

Zwei Variablen `a` und `b` können wie folgt im Unterverzeichnis `foo` der ROOT-Datei `matlab.root` gespeichert werden:

3.1. BENUTZERSCHNITTSTELLE

```
>> a = 7;
>> b = {'a', a};
>> rsave matlab.root#foo a b
```

Der Typ der Variablen ist dabei unerheblich. Die Variablen werden implizit in ROOT-Objekte transformiert und dabei zu Structs normalisiert. Die Normalisierung von `a` sieht bspw. so aus:

```
>> reformat(a, 1)
```

```
ans =
```

```
      type: 6
      size: [1 1]
      data: 7
  complex: 0
```

Die „1“ beim Aufruf von `reformat` steht für die Richtung der Transformation, in diesem Fall von MATLAB zu ROOT. Dieser Aufruf geschieht bei `rsave` und `rload` implizit.

Das Ergebnis des Speichervorgangs kann mit den Funktionen `rwho` und `rwhos` bzw. direkt im ROOT-Object-Browser überprüft werden:

```
>> rwho -file matlab.root#foo
```

```
Your variables are:
```

```
      a
      b
```

Die folgende Zeile lädt die Variablen als Felder des Structs `s` aus der Datei:

```
>> s = rload('matlab.root#foo', 'a', 'b')
```

```
s =
```

```
a: 7
b: {'a' [7]}
```

3.2 Tests

Während der Entwicklung wurde immer wieder ein Standardtest mit mroot durchgeführt, um zu gewährleisten, dass bestehende Funktionalität auch bei Änderungen und Neuerungen erhalten bleibt. Die Funktionalität der wichtigen, zentralen Funktionen `reformat` und `pathinfo` wurde anschließend mit Unit Tests dokumentiert. Um etwas über die Performanz von mroot zu erfahren wurde getestet, wie sich der Datendurchsatz bei wachsender Datenmenge verhält. Da mroot hauptsächlich durch USCT motiviert ist, wurde mroot außerdem in einem realen USCT-TestszENARIO getestet.

3.2.1 Standard-Testfall

Im Standard-Testfall wurde ein komplexes Cell-Array in eine ROOT-Datei gespeichert und wieder daraus geladen. In den Zellen des Arrays waren alle MATLAB-Typen vertreten. Die Kopie des Arrays ist nach einem Speichern-Laden-Zyklus jeweils identisch mit dem Original. Damit ist die prinzipielle Funktionsfähigkeit von mroot erwiesen.

Plattformunabhängigkeit und Portabilität

Getestet wurde unter Linux und Windows. Bei gleicher Konfiguration der Testumgebung gab es keine plattformspezifischen Auffälligkeiten, abgesehen davon, dass es nur unter Linux gelang, mpatrol zu kompilieren. Dies wurde entsprechend in `compile` berücksichtigt.

Probleme kann es z.B. geben, wenn der Compiler, mit der die MEX-Dateien übersetzt wurden, inkompatibel zu der benutzten MATLAB-Version ist. In diesem Fall muss ein anderer Compiler oder eine andere Compiler-Version benutzt werden.

Desweiteren wurde die Lauffähigkeit mroots in verschiedenen MATLAB-Versionen geprüft. mroot ist kompatibel zu den letzten fünf MATLAB-Versionen (R2006a bis R2008a). In früheren Versionen waren manche der verwendeten Funktion nicht enthalten oder hatten ein anderes Verhalten.

Die Funktion `cellfun`, mit der hochperformant eine Funktion auf den Zellen eines Cell-Array ausgeführt werden kann, war bspw. nur für bestimmte Funktionen definiert. Sie wurde für `mroot` in MATLAB nachprogrammiert, wegen der Überschreibe-Problematik jedoch nicht integriert.

Abstraktion der Datensenke

Die Abstraktion der Datensenke zeigt sich im Quellcode von `mroot` nur in der Wahl des jeweiligen Konstruktors, z.B. von `TFile` oder `TSQLFile`. Dennoch funktioniert die Abstraktion der Datensenke nur bei `TFile` und `TNetFile` reibungslos.

Bei `TXMLFile` und `TSQLFile` bricht der Speichervorgang wiederum mit „Failed to write variable <v> to file“ ab, wobei der angezeigte Variablenname nicht dem tatsächlichen entspricht. Dies in Zusammenhang mit der Tatsache, dass der Zugriff auf XML-Dateien und Datenbanken in der ROOT-Konsole funktioniert, deutet auf ein Problem mit der Zeichenkodierung zwischen MATLAB und ROOT hin.

Offensichtlich werden die Daten trotzdem in die XML-Datei geschrieben, denn sie können daraus geladen werden.

Frei von Speicherlecks

An dem Standard-Testfall sind alle Datentypen beteiligt, d.h. es werden alle typspezifischen Codezeilen durchlaufen. `mpatrol` zeigt nach dem Testlauf keine Speicherlecks an. Somit gilt `mroot` als frei von Speicherlecks.

3.2.2 Unit Tests

`mroot` wurde nicht testgetrieben entwickelt. Eine annähernd vollständige Abdeckung mit Unit Tests ist somit kaum zu erreichen. Dennoch wurden für die beiden zentralen `mroot`-Funktionen `reformat` und `pathinfo` Unit Tests geschrieben. Sie dienen weniger dazu, im Nachhinein die Einhaltung der Spezifikation zu prüfen, als vielmehr dazu, selbige zu dokumentieren. Sie sind Beispiele für die Nutzung der Funktionen und zeigen deren Verhalten in üblichen Anwendungsfällen.

Ein zukünftiger Entwickler kann sie damit testgetrieben weiterentwickeln. Schlägt ein Test Fehler, ist sofort klar, dass eine der Neuerungen gegen die ursprüngliche Spezifikation verstößt.

Die Test Cases decken übliche Anwendungsfälle ab und sind so gestaltet, dass möglichst jede Code-Zeile des getesteten Moduls durchlaufen wird. In der Test Suite für **reformat** gibt es bspw. Test Cases für jeden Datentyp.

3.2.3 Skalierung der Performanz

Die Skalierung der Performanz von mroot wurde getestet, indem mit Nullen gefüllte Double-Arrays abgespeichert wurden. Die Arrays wurden so oft gespeichert, dass insgesamt ein GByte Daten belegt wurde oder maximal 256 mal. Bei jedem Speichervorgang wurde die Zeit genommen. Durch die Wiederholungen konnte jeweils der Mittelwert gebildet werden, um Rauschen in den Messwerten zu minimieren. Die Anzahl der Wiederholungen ist ein Kompromiss zwischen adäquater Mittelwertbildung und kurzer Ausführungszeit.

Nach einem solchen Durchlauf wurde die Länge der Arrays um eine Zweierpotenz erhöht. Der Test zeigt damit auch, welche maximale Variablengröße mit **rsave** in dieser Testumgebung abgespeichert werden kann. Abb. 3.1 zeigt den Speicherdurchsatz abhängig vom gespeicherten Datenvolumen. Für große Datenmengen nähert sich der Durchsatz einem Optimum von ca. 30 MB/s an.

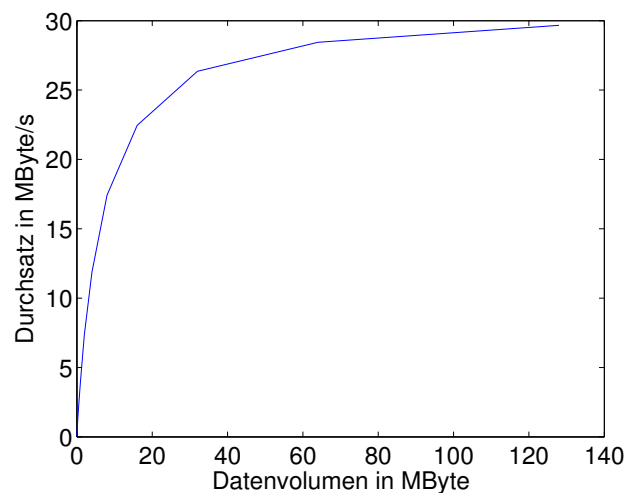


Abbildung 3.1: Skalierung der Performanz

3.2. TESTS

Die maximale Variablengröße liegt bei 2^{24} Doubles bzw. 128 MByte. Die Gründe für diese Begrenzung werden zur Zeit noch untersucht. Durch sie wurde die optimale Performanz in den Tests nicht erreicht. Die Performanz ist ab einer Datenmenge von ca. 60 MByte gut. Wie ROOT selbst ist auch mroot für große Datenmengen optimiert.

3.2.4 Test unter realen Bedingungen

Die Messdaten eines kompletten USCT-Experiments (ca. 20 GByte Daten) werden in hunderten von MATLAB-Dateien in einer projektspezifischen Verzeichnisstruktur gespeichert (s. Abb. 3.2). Für einen Test unter Realbedingungen wurden diese Daten so in eine ROOT-Datei überführt, dass die ursprüngliche Verzeichnisstruktur erhalten blieb. Zum Vergleich wurde die MATLAB-Funktion `save` im gleichen Aufbau gemessen.

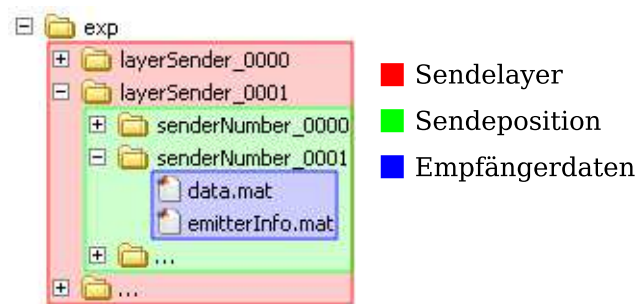


Abbildung 3.2: Verzeichnishierarchie der USCT-Daten

Das Experiment-Verzeichnis wurde mit `readdir` nach MATLAB-Dateien durchsucht. Diese wurden geladen und mit `rsave` bzw. `save` gespeichert. Der Speichervorgang wurde mit dem MATLAB-Profiler aufgenommen (s. Lst. 3.1). Damit `save` als MATLAB-Builtin-Funktion von dem Profiler beachtet wird, muss die Detailschärfe auf „builtin“ eingestellt werden.

```
1 profile -detail builtin
2 profile on
3
4 % work
5
6 profile off
7 profsave(profile('info'), 'myprofile_results')
```

Listing 3.1: MATLAB-Profiler

3.2. TESTS

Die maximale Größe einer ROOT-Datei liegt bei ca. 12 GByte. Es können somit nicht alle Messdaten des Experiments (20 GByte) in eine einzelne ROOT-Datei überführt werden. Der Versuch brach mit der mroot-Fehlermeldung „Failed to write variable <v> to file“ ab. Auch die Gründe hierfür werden untersucht. Einstweilen könnten jeweils große Teilmengen der Messdaten in ROOT-Dateien abgelegt werden. Die 20 GByte Daten könnten z.B. in zwei Dateien zu je 10 GByte überführt werden.

Um vergleichbare Messergebnisse zu erhalten, wurde der Test mit einer Teilmenge der Messdaten wiederholt.

Die 845 MByte Originaldaten wurde im MATLAB-v6-Format unkomprimiert in 23,3s abgespeichert. Dazu wurde beim Speichern die `-v6`-Option angegeben. MATLAB v7.3 komprimiert standardmäßig mit gzip-1. Dadurch sinkt der Speicherbedarf um ca. ein Viertel auf 572 MByte, während der Vorgang ca. dreimal so lange dauert, nämlich 78s.

Das ist ungefähr die Zeit, die `rsave` benötigt, um die Daten unkomprimiert in eine ROOT-Datei zu speichern. Der Overhead von `rsave` entspricht also dem von gzip-1.

Ein Teil der Geschwindigkeit geht bei `rsave` durch die Transformation der MATLAB-Variablen in ROOT-Objekte verloren, die bei `save` nicht gemacht wird. Außerdem bremst die Zahl der beteiligten Schnittstellen in mroot, z.B. MEX oder die abstrahierte Datensenke. Für einen Gutteil des Overheads ist auch der Wrapper um `mex_rsave` verantwortlich, vor allem die teuren String-Operationen in `pathinfo`.

Bei ROOT kann die gzip-Kompressionsstufe direkt angegeben werden. Alle zehn Stufen von 0 = unkomprimiert bis 9 = hochkomprimiert sind möglich. Die Kompression mit gzip-1 dauert bei `rsave` 129,9s, also weniger als doppelt so lange als mit gzip-0. Bei Kompressionsstufe 9 dauert der Speichervorgang fast dreimal so lange, während sich der Speicherbedarf kaum verringert. Für USCT-Daten lohnt sich eine höhere Kompressionsstufe folglich nicht.

	Dauer	Speicher
MATLAB v6	23,3s	845 MByte
MATLAB v7.3	78,0s	572 MByte
ROOT gzip-0	76,6s	845 MByte
ROOT gzip-1	129,1s	572 MByte
ROOT gzip-9	181,1s	568 MByte

Tabelle 3.1: Messwerte bei der Konvertierung eines USCT-Experiments

3.3 Publikation

Für eine Veröffentlichung gibt es zahlreiche Möglichkeiten: Die Firma Mathworks, Inc. bietet mit dem File Exchange [19] eine Plattform für den Austausch MATLAB-spezifischer Dateien, insbesondere Software. Auf der ROOT-Homepage gibt es eine Liste mit Anwendungen, die ROOT nutzen. Eine der größten Anlaufstellen für die Entwicklung und Veröffentlichung von Open Source-Projekten ist die Webseite SourceForge.net. Dies sind nur einige der Möglichkeiten.

Da sich mroot hauptsächlich an MATLAB-Nutzer richtet – es handelt sich schließlich um eine Schnittstelle von MATLAB zu ROOT – war der MATLAB File Exchange die erste Wahl. mroot wurde dort veröffentlicht. Links auf anderen Webseiten erhöhen die Aufmerksamkeit für diese Publikation, weshalb mroot auch in die Liste auf der ROOT-Webseite aufgenommen werden soll. Darüber hinaus wäre eine eigene kleine Homepage mit weiteren Informationen z.B. auf einem IPE-Server denkbar.

Evtl. könnte das Projekt auch auf einer ROOT-Konferenz vorgestellt oder in einem wissenschaftlichen Paper beschrieben werden.

Lizenzierung

mroot steht unter einer freien Lizenz. Dies war im IPE ein kleines Novum, da Forschungsergebnisse in der angewandten Forschung oft auf Vermarktung ausgerichtet sind. Doch mroot ist eine Schnittstelle, die u.a. dadurch motiviert ist, dass mit Hilfe des kostenlosen ROOT der Datenaustausch zwischen verschiedenen wissenschaftlichen Projekten erleichtert wird. Lizenzierung mit dem Ziel der Geldeinnahme kam also nicht in Frage. Vielmehr soll die Schnittstelle der „Allgemeinheit“ zugute kommen und auch Allgemeingut bleiben – auch bei Weiterentwicklungen. Jeder soll sie nutzen können, auch in einem kommerziellen Umfeld. Aus diesen Gründen sind die Quelltexte wie auch ROOT unter der GNU Lesser General Public License (GNU LGPL) lizenziert, allerdings im Ggs. zu ROOT in Version 3 (ROOT: Version 2.1).

In Konsequenz steht auch diese Diplomarbeit unter einer freien Lizenz. Für mediale Erzeugnisse bietet sich eine Creative Commons License an. Diese Lizenzen unterscheiden sich voneinander durch Abstufung der Freiheitsgrade. Diese Arbeit darf mit Namensnennung des Autors beliebig verbreitet, darf aber nicht bearbeitet werden.

Kapitel 4

Diskussion und Ausblick

Mit mroot wurde eine neuartige MATLAB-Toolbox geschaffen, die den Zugriff auf ROOT-Dateien ermöglicht. Mit mroot können Variablen gespeichert, geladen und aufgelistet werden. Dabei werden alle MATLAB-Datentypen unterstützt.

Daten können in ROOT-Dateien strukturiert werden, wie es sonst nur in Verzeichnishierarchien möglich ist. Die ROOT-Dateien bieten somit einen effizienten Container für Ablage und Übertragung von Daten. Bei der Übertragung der Messdaten eines USCT-Experiments ist bspw. hilfreich, dass der Übertragungs-Overhead einer ROOT-Datei deutlich geringer ist, als der vieler kleiner MATLAB-Dateien.

Als Datensenke stehen nun nicht mehr nur binäre Dateien zur Verfügung, sondern auch XML-Dateien und SQL-Datenbanken. Auf binäre ROOT-Dateien kann über einen Dienst auch auf anderen Rechnern zugegriffen werden.

Durch mroot können Projekte, die MATLAB für die Datenanalyse nutzen, Daten in einem standardisierten Format mit anderen Projekten austauschen. Diese können die Daten mit dem kostenlos verfügbaren ROOT weiter verarbeiten.

mroot könnte zur Verbreitung von ROOT beitragen, da es dessen effiziente Datenhaltung mit MATLABs Komfort verbindet.

mroot verhält sich aus Sicht von MATLAB völlig transparent. Die mroot-Funktionen zeigen bei gleicher Eingabe das gleiche Verhalten wie ihre MATLAB-Originale. Auch bei den Datentypen gibt es keine Einschränkungen.

4.1 Weitere Schritte

Einschränkungen gibt es zur Zeit noch bei der Variablengröße: Sie ist auf 128 MByte begrenzt. Auch die maximale Dateigröße hat eine Beschränkung von ca. 12 GByte. Dies sind wohl die größten Limitierungen mroots und es gilt herauszufinden, worin sie begründet liegen.

Für den praktischen Einsatz müssen bei XML-Dateien und SQL-Datenbanken die beschriebenen Probleme gelöst werden. Für die Nutzung im Grid wäre es nützlich, zusätzlich die ROOT-I/O-Klassen `TDCacheFile` und `TRFIOFile` zu integrieren, da diese Produkte (dCache und RFIO) bereits im Grid verwendet werden.

Bei dem Test unter Realbedingungen wurden die Messdaten eines USCT-Experiments in eine ROOT-Datei überführt. Dabei wurde auch mit den verschiedenen Kompressionsstufen von gzip experimentiert, indem diese fest in den Quelltext enkodiert wurden. Es stellte sich heraus, dass sich eine höhere Kompressionsstufe als Stufe 1 für USCT-Daten nicht lohnt, da die Kompression damit deutlich länger dauert, während die Speicherersparnis minimal ist. Für andere Daten sieht die Situation evtl. anders aus. Es weiteres Options-Flag für `rsave` wäre also hilfreich, mit dem die gewünschte Kompressionsstufe angegeben werden kann.

Bei rechenaufwändigen Verfahren wie den USCT-Algorithmen ist die Geschwindigkeit beim Speichern und Laden meist nicht ausschlaggebend für die Gesamt-Performance. Dennoch besteht bei mroot Optimierungs-Potenzial bezüglich der Geschwindigkeit. Bspw. könnten die generischen String-Funktionen in `pathinfo` durch mehr auf die konkrete Anwendung zugeschnittene und damit schnellere Lösungen ersetzt werden.

Weitere Möglichkeiten für die Optimierung können durch weitergehende Tests eruiert werden. Z.B. könnte das Laufzeitverhalten der MEX-Dateien aufgenommen und analysiert werden. Bisher wurde nur ein Versuch in diese Richtung mit dem Shiny-Profiler gemacht.

4.2 Visionäre Ausblicke

mroot ist eine Datenschnittstelle von MATLAB zu ROOT. Es gibt jedoch ein großes Potenzial für Erweiterungen.

Octave ist wie MATLAB eine Scriptsprache zur numerischen Problemlösung, steht allerdings unter einer freien Lizenz. Octave ist weitgehend mit MATLAB kompatibel und verfügt auch über MEX. Falls mroot kompatibel zu Octave ist, würde dies evtl. die Zielgruppe erweitern.

MATLAB-Variablen werden beim Speichern in ROOT-Dateien mit mroot in Structs gekapselt. Sie liegen dann nicht mehr in ihrer originalen Form vor. Dies ist notwendig, um die MATLAB-Typen überhaupt auf native ROOT-Klassen abbilden zu können. Spezielle Funktionen könnten die Arbeit mit den solcherart normalisierten Daten vereinfachen, bspw. eine C++-Funktion, die wie die MATLAB-Funktion `sub2ind` einen mehrdimensionalen Index in einen linearen umrechnet. Das wäre hilfreich für den Zugriff auf die Datenfelder der Structs, die der Information über ihre Dimensionalität beraubt sind.

ROOT kann viel mehr, als Daten zu speichern und zu übertragen. Es ist auch ein umfassendes Framework zur Analyse und Visualisierung von Daten. In einem weiteren Schritt könnte mroot so erweitert werden, dass MATLAB Zugriff auf diese Fähigkeiten gewährt wird.

4.2.1 mroot-Dienst

Wie ihre MATLAB-Originale stehen die mroot-Funktionen jeweils für sich. Beim Speichern wird z.B. die Datei geöffnet, dann werden die Daten darin abgelegt und zuletzt wird die Datei wieder geschlossen. Das alles geschieht in direkter Folge. Der Dateizugriff bei ROOT ist hingegen mehr sitzungsbezogen. Wird eine Datei geöffnet, so wird sie automatisch zum aktuellen Arbeitsverzeichnis. Dann können darin Daten beliebig gespeichert, manipuliert oder gelöscht werden. Am Ende der Sitzung wird die Datei wieder geschlossen.

Ein Weg, diese beiden Nutzungsparadigmen zu vereinen, könnte ein spezieller Dienst sein. Bei der ersten Dateioperation wird der Dienst gestartet und die Datei geöffnet. Der Dienst hält die Datei während der MATLAB-Sitzung offen und führt die gewünschten Operationen darauf aus. Soll auf eine andere Datei zugegriffen werden, dann wird die vorherige Datei geschlossen und die gewünschte geöffnet. Aus Sicht von MATLAB stehen die einzelnen Ope-

4.2. VISIONÄRE AUSBLICKE

rationen weiterhin jeweils für sich. Am Ende der MATLAB-Sitzung wird der Dienst beendet und die jeweils aktuelle Datei geschlossen.

Dieses Vorgehen spart zudem den Overhead, der beim jetzigen mroot bei jedem Dateizugriff anfällt, das Parsen des vollen Dateinamens, etc.

Der Dienst kann von MEX unabhängig implementiert werden. Die Daten für den Zugriff auf ROOT-Dateien bzw. die Nutzdaten selbst werden wie beim aktuellen mroot in ROOT-Objekte gewandelt. Dem Dienst werden somit keine MATLAB-Variablen, sondern reine ROOT-Objekte übergeben. So könnte auch andere Software diese Datenschnittstelle nutzen. Es müsste jeweils nur die Transformation der Daten neu implementiert werden. Für das Implementieren des Dienstes selbst ist ein enormer Aufwand nötig. Vielleicht ließe sich der bereits in ROOT enthaltene Dienst „rootd“ entsprechend anpassen.

Die Idee eines solchen mroot-Dienstes liefert genügend Stoff für eine weitere Diplomarbeit.

Literaturverzeichnis

- [1] H. Gemmeke and N.V. Ruiter. 3D Ultrasound Computer Tomography for Medical Imaging. *Nucl. Instr. Meth.*, 550:1057–65, 2007.
- [2] The MathWorks - MATLAB and Simulink for Technical Computing, August 2008. <http://www.mathworks.com>.
- [3] Johannes Kissel. Evaluation von ROOT als Option für die Datenhaltung bei USCT, 2007.
- [4] The ROOT System Home Page, August 2008. <http://root.cern.ch>.
- [5] MEX-files Guide, August 2008.
<http://www.mathworks.com/support/tech-notes/1600/1605.html>.
- [6] GNU Licenses, August 2008. <http://gnu.org/licenses>.
- [7] Visual C++ 2008 Express Edition, August 2008.
<http://www.microsoft.com/express/vc/>.
- [8] GNU Compiler Collection, August 2008. <http://gcc.gnu.org>.
- [9] Installing ROOT from Source, August 2008.
<http://root.cern.ch/root/Install.html>.
- [10] Subversion, Version Control System, August 2008.
<http://subversion.tigris.org>.
- [11] XTargets Unit Testing, August 2008.
<http://xtargets.com/products/munit/munit.html>.
- [12] Unit Testing With MATLAB, August 2008.
http://mlunit.dohmke.de/Unit_Testing_With_MATLAB.
- [13] Doxygen, Source Code Documentation Generator, August 2008.
<http://www.doxygen.org>.
- [14] M2HTML, Documentation System for MATLAB, August 2008.
<http://www.artefact.tk/software/matlab/m2html/>.

- [15] L^AT_EX – A document preparation system, August 2008.
<http://www.latex-project.org>.
- [16] MATLAB Function Reference (by category), August 2008.
<http://www.mathworks.com/access/helpdesk/help/techdoc/ref/f16-6011.html>.
- [17] MATLAB Function Types, August 2008.
http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_prog/f4-62416.html.
- [18] mpatrol, Debugging Tool for use with dynamically allocated memory, August 2008. <http://www.cbmamiga.demon.co.uk/mpatrol/>.
- [19] The MATLAB Central File Exchange, August 2008.
<http://www.mathworks.com/matlabcentral/fileexchange/>.

Anlagenverzeichnis

MATLAB-Quelltexte (Auswahl)

- Anlage 1: `rsave.m`
- Anlage 2: `rload.m`
- Anlage 3: `pathinfo.m`
- Anlage 4: `reformat.m`

C++-Quelltexte (Auswahl)

- Anlage 5: `mex_rsave.cpp`
- Anlage 6: `mex_rload.cpp`
- Anlage 7: `mroot.h` und `mroot.cpp`

Begleit-CD

- Diese Diplomarbeit
- Alle Quelltexte
- API-Dokumentation
- Vortrag zu der Diplomarbeit
- ROOT: Quelltexte und Windows-Setup
- Linksammlung

Für das Inhaltsverzeichnis der CD die Datei `index.html` im Stammverzeichnis öffnen!

Abbildungsverzeichnis

1.1	Logos von CERN und ROOT	7
1.2	ROOT-Konsole	8
1.3	ROOT Object Browser	8
1.4	Beschreibung des ROOT-Dateiformats	9
1.5	TortoiseSVN und kdesvn im Vergleich	13
1.6	MUnit: Control Panel und Test Report	14
2.1	mroots Schnittstellen-Struktur	18
2.2	Verhalten der Funktion save	21
2.3	TFile und Familie	24
2.4	Umwandlung einer Matrix in ein Array mit Kopfformat	28
2.5	Umwandlung einer Matrix in ein Struct	28
2.6	Transformation der Variablen	29
2.7	mroot-Verzeichnisstruktur	30
2.8	Zerlegung des ROOT-Pfades	37
2.9	Datenfluss in mroot	40
3.1	Skalierung der Performanz	51
3.2	Verzeichnishierarchie der USCT-Daten	52

Listings

1.1	Funktionaler vs. imperativer MATLAB-Funktionsaufruf . . .	6
1.2	Grundgerüst einer MEX-Datei	7
1.3	Build- und Installationsprozess unter Linux	11
1.4	Umgebungsvariablen für die ROOT-Nutzung	12
1.5	Grundgerüst einer MUnit Test Suite	13
1.6	Struktur der Doc-Kommentare	15
2.1	Syntax der Funktion save	22
2.2	Syntax des ROOT-Pfads in EBNF	25
2.3	mpatrol Leak Tables	31
2.4	Beispiele zu vercmp	45
2.5	Beispiele zu explode und implode	45
3.1	MATLAB-Profiler	52

Tabellenverzeichnis

2.1	Liste der Datei-Funktionen in MATLAB	20
2.2	Übersicht über die MATLAB-Datentypen	27
2.3	Struct-Felder nach der Reformatierung beim Speichern	34
2.4	Zuordnung von MATLAB-Typen zu ROOT-Klassen	35
2.5	Übersicht über die Rückgaben von pathinfo	36
2.6	Übersicht über die mroot-Funktionen	38
2.7	Liste der Compile- und Link-Flags	43
3.1	Messwerte bei der Konvertierung eines USCT-Experiments .	53

Abkürzungsverzeichnis

API	Advanced Programming Interface
CVS	Concurrent Version System
CERN	Conseil Européen pour la Recherche Nucléaire
CINT	C++ Interpreter
DESY	Deutsches Elektronen-Synchrotron
EBNF	Erweiterte Backus-Naur-Form
FZK	Forschungszentrum Karlsruhe
GCC	GNU Compiler Collection
GDB	GNU Debugger
GNU	GNU's not Unix (rekursives Akronym)
GPL	General Public License
HTML	Hypertext Markup Language
IPE	Institut für Prozessdatenverarbeitung und Elektronik
I/O	Input/Output
KDE	K Desktop Environment (rekursives Backronym)
KIO	KDE I/O
LaTeX	Lamport TeX
LCC	Little C Compiler
LGPL	Lesser General Public License
MATLAB	Matrix Laboratory
MEX	MATLAB External Interfaces
OpenGL	Open Graphics Library
PHP	PHP Hypertext Processor (rekursives Backronym)
Qt	Quasar Technologies
RTTI	Runtime Type Information
SP	Service Pack
SQL	Structured Query Language
SUSE	Software- und System-Entwicklungsgesellschaft
SVN	Subversion
USCT	Ultraschall-Computertomographie
VC++	Visual C++
WYSIWYG	What you see is what you get