**Bachelorarbeit im Rahmen des Studiengangs**
**Scientific Programming**

Fachhochschule Aachen, Campus Jülich

Fachbereich 9 – Medizintechnik und Technomathematik

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

# Entwicklung einer interaktiven JavaScript-Umgebung für 2D-Grafiken

25. November 2016

**Florian Macherey**

# Eigenhändigkeitserklärung

Diese Arbeit ist von mir selbstständig angefertigt und verfasst. Es sind keine anderen als die angegebenen Quellen und Hilfsmittel benutzt worden.

| | |
|---|---|
| (Ort, Datum) | (Unterschrift) |

Diese Arbeit wurde betreut von:

| | |
|---|---|
| 1. Prüfer: | Prof. Ulrich Stegelmann |
| 2. Prüfer: | Ingo Heimbach, M. Sc. |
| Betreuer: | Josef Heinen |

Sie wurde angefertigt in der Forschungszentrum Jülich GmbH im Peter Grünberg Institut / Jülich Centre for Neutron Science.

Die Beschäftigten des Peter Grünberg Instituts / Jülich Centre for Neutron Science untersuchen in Experimenten und Simulationen die Form und Dynamik von Materialien wie Polymeren, Zusammenlagerungen großer Moleküle und biologischer Zellen sowie die elektronischen Eigenschaften von Festkörpern. Dabei gewonnene Daten werden für eine anschließende Analyse typischerweise visualisiert. Institutsintern wird für Visualisierungsanwendungen häufig das GR-Framework verwendet. Dieses wurde in einer vorherigen Abschlussarbeit bereits so erweitert, dass dessen API (engl. Application Programming Interface) in einem Browserkontext zur Verfügung steht.

Ein Ansatz zur Auswertung besteht in der Nutzung von Jupyter-Notebooks, welche die Möglichkeiten moderner Interpretersprachen mit Elementen aktueller Webtechnologie verbinden. Im Rahmen der vorliegenden Bachelorarbeit eine Endanwenderkomponente JSTerm (JavaScript Terminal, in Anlehnung an bestehende Implementierungen bspw. GKSTerm) entwickelt, die sowohl als Ausgabe für Jupyter-Notebooks genutzt als auch in beliebige Webseiten eingebettet werden kann.

Neben einer einfach einzubettenden Ausgabe stellt JSTerm außerdem Interaktionsmechanismen bereit, die es beispielsweise erlauben, nachträglich den Fokus der Darstellung zu ändern, gezielt Punkte aus Graphen mit der Maus auszulesen oder Attribute wie Sichtbarkeit zu manipulieren. Im Vordergrund steht hier die komfortable Bedienung für den Endanwender.

Um die zuvor beschriebenen Manipulationsmöglichkeiten realisieren zu können, wurde das GR-Framework um ein neues Protokoll erweitert, welches die Eingabedaten ohne vorherige Reduktion an JSTerm weiterleitet. Als Kommunikationsmedium wird die vorhandene Jupyter-Infrastruktur verwendet. Alternativ zu der Jupyter Variante des JSTerms gibt es die sogenannte standalone Variante. Bei dieser werden die Daten über einen Proxy-Server verschickt, welcher mit Tornado realisiert ist.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Listingverzeichnis

# 1. Einleitung

Zur Untersuchung der elektronischen Eigenschaften von Festkörpern sowie ihrer Form und Dynamik führen die Beschäftigten des Peter Grünberg Instituts / Jülich Centre for Neutron Science (PGI/JCNS) physikalische Experimente und Simulationen durch. Dabei fallen Daten von Materialien wie Polymeren, Zusammenlagerungen großer Moleküle und biologischer Zellen an. Diese sollen anschaulich visualisiert werden. Hierzu wird im PGI/JCNS meistens das eigens entwickelte GR-Framework verwendet, welches unter einer MIT-Lizenz auf Github[1] und anderen Softwareverteilungsplattformen zur Verfügung steht.

Die vorliegende Arbeit beschreibt die Realisierung einer Anwendung für Endanwender mit dem Namen JSTerm (JavaScript Terminal, in Anlehnung an bestehende Implementierungen bspw. GKSTerm) zur einfachen Darstellung solcher Daten. Diese nutzt Jupyter-Notebooks [Gra+16] zum Auswerten der Daten mittels Interpretersprachen wie bspw. Python. Alternativ zur Ausführung innerhalb des Jupyter-Kontextes, kann das JSTerm auch alleinstehend im Browser zur Verfügung gestellt werden.

Die Grundlagen dieser Arbeit werden zunächst in Kapitel 2 erläutert. Zur Kommunikation zwischen Anwendungsschichten und dem JSTerm wird ein im Rahmen dieser Arbeit entwickeltes, auf JSON basierendes Protokoll verwendet, auf welches in der Konzeption in Kapitel 3 eingegangen wird. Die Daten werden dabei von den Anwendungsschichten unmittelbar – ohne Reduktion oder Komprimierung – versendet. In der Konzeption befindet sich außerdem die Beschreibung der Schichten der gesamten Anwendung.

Zusätzlich beinhaltet das entwickelte JSTerm auch Interaktionsmöglichkeiten, auf die im Kapitel 4 (Implementierung) eingegangen wird. Diese Interaktionen sind clientseitig im Browser mit JavaScript realisiert. Ein Hauptfokus bei der Realisierung des JSTerms liegt auf der einfachen Benutzbarkeit durch die Endanwender. Zum Schluss werden die Ergebnisse dieser Arbeit zusammengefasst und ein Ausblick gegeben.

[1] `https://github.com/jheinen/gr/`

# 2. Grundlagen

Dieses Kapitel beschreibt die Grundlagen der entwickelten Software. Es werden JavaScript und HTML5 – mit seinen Bestandteilen WebSockets und dem Canvas – erläutert. Außerdem werden IPython, Jupyter sowie Tornado vorgestellt. Weiterhin werden das GKS (Graphisches Kernsystem) und das GR beschrieben, welche das eigentliche Zeichnen der Grafiken übernehmen.

## 2.1. Allgemeines zu JavaScript und HTML5

JavaScript ist eine interpretierte Sprache, die u. a. im Webbrowser clientseitig ausgeführt wird. Dabei findet die Ausführung in einer Sandbox statt, um zu verhindern, dass von der Webanwendung auf Daten außerhalb des Browsers zugegriffen werden kann. Mit JavaScript ist es bspw. möglich, das DOM (engl. Document-Object-Model) zu modifizieren, ohne dabei die Webseite neu zu laden. Das DOM ist eine baumartige Struktur eines HTML-Dokumentes und es gibt verschiedene Methoden zum Ändern der Struktur, des Inhaltes und des Aussehens [Net16].

Ein weiterer klassischer Einsatzbereich von JavaScript ist es außerdem, auf Events des Nutzers zu reagieren. Diese Events können z. B. durch Mausklicks oder Zeigerbewegungen ausgelöst werden.

JavaScript besitzt keine Klassen, aber ermöglicht objektorientierte Programmierung über Prototypen. Die Typisierung ist dabei schwach dynamisch. Dynamisch bedeutet, dass der Typ eines JavaScript-Objektes erst zur Laufzeit der Anwendung bestimmt wird. Schwach bedeutet im Zusammenhang mit der Typisierung, da sich der Typ eines Objektes automatisch ändern kann, bspw. beim Addieren von Objekten ungleichen Typs. Die Typsicherheit ist bei schwach typisierten Sprachen weniger gegeben [Int16; Dro15].

AJAX steht für Asynchronous JavaScript and XML und beschreibt das Zusammenspiel von verschiedenen Technologien u. a. HTML und JavaScript. Mit diesen und mit dem `XMLHttpRequest`-Objekt kann eine Anfrage an den Webserver gestellt werden, um Inhalte zu einer Webseite hinzu zu laden, ohne die gesamte Seite zu aktualisieren [Sch+16]. Dadurch können geringere Antwortzeiten von Webanwendungen realisiert werden.

Die Grafiken werden innerhalb der JavaScript-Implementierung des GR-Frameworks auf ein HTML5-Canvas gezeichnet. Dieses dient zum Zeichnen von Raster-Grafiken [Hic]. Ein Canvas-Element hat dabei eine Höhe und Breite, angegeben in Pixel. Der Ursprungspunkt `(0, 0)` ist in der oberen linken Ecke des Canvas. Das Canvas-Element ist seit HTML5 verfügbar und wird in der Grundfunkionalität von mehr als 97% aller Webbrowser unterstützt [DS16].

## 2.2. Tornado als asynchroner Webserver bei der standalone Kommunikation

Tornado ist ein Python-Webframework und eine Python-Bibliothek, um asynchrone Webserver zu entwickeln. Bei klassischen, synchronen Webanwendungen schickt der Client eine Anfrage an den Server, welcher diese bearbeitet und eine Antwort zurückgibt. Bei einer asynchronen Webanwendung hingegen kann die Routine, welche die Anfrage bearbeitet, die Kontrolle abgeben, sodass andere Anfragen quasi-parallel bearbeitet werden können. Wenn bspw. auf die Daten aus einer Datenbank gewartet wird, kann in dieser Zeit eine andere Anfrage vom Server bearbeitet werden. Ein synchroner Webserver würde an dieser Stelle blockieren und die Anfragen seriell verarbeiten oder mehrere Threads verwenden. Damit ist es mithilfe von Tornado möglich Anfragen nahezu parallel zu bearbeiten, obwohl diese nur in einem Thread ausgeführt werden. Tornado ist daher ideal für Webanwendungen, die Verbindungen lange geöffnet halten sollen [Dar16]. Außerdem ist das WebSocket-Protokoll bereits als Teil des Tornado-Webservers realisiert. Der Tornado-Server wird in der standalone Variante des JSTerms eingesetzt. Die Funktionalität wird im Konzept-Kapitel 3.2.2 erläutert.

## 2.3. WebSockets zur Kommunikation zwischen Tornado und JSTerm

WebSockets sind ein „living Standard“ der WHATWG (Web Hypertext Application Technology Working Group) [Hic+14] sowie ein durch die IETF (Internet Engineering Task Force) standardisiertes Protokoll [FM11]. Sie werden von mehr als 91% aller Webbrowser [Dev16] unterstützt. Das WebSocket-Protokoll ist ein Anwendungsprotokoll, welches auf TCP aufbaut. Der Vorteil von WebSockets im Vergleich zu HTTP ist, dass nach dem Öffnen der Verbindung durch den Client Daten bidirektional gesendet werden können und dass die Verbindung geöffnet bleibt. Außerdem gibt es keinen Overhead durch Metadaten, da die Headerfelder, im Gegensatz zu HTTP, nicht jedes Mal mit ausgetauscht werden müssen [Hic+14].

Das Listing A.1 (s. Anhang) zeigt beispielhaft einen minimalen WebSocket-Client. Bei der Initialisierung wird eine URL angegeben, unter welcher der WebSocket-Server erreichbar ist. Als URL-Schemata kann hier `ws` für unverschlüsselte oder `wss` für SSL verschlüsselte WebSocket-Kommunikation angeben werden. In der im Rahmen dieser Arbeit entwickelten Webanwendung reichen unverschlüsselte Verbindungen aus, da die Daten nur auf demselben Host ausgetauscht werden. Danach sollten die drei Event-Listener zum Öffnen (`onopen`), Schließen (`onclose`) und Empfangen (`onmessage`) einer Nachricht implementiert werden. Im Beispiel-Client (s. Anhang A.1) werden nur Ausgaben auf die Entwicklerkonsole geschrieben. Es gibt zusätzlich noch einen weiteren Event-Listener, welcher bei einem Fehler (`onerror`) ausgelöst wird [Net15].

In dieser Arbeit werden WebSockets verwendet, da die Realisierung damit einfach und das Protokoll sehr verbreitet ist. Innerhalb der Jupyter-Umgebung steht der so-

genannte `CommManager` zur Verfügung, der intern auch Websockets verwendet. In der Jupyter-Variante des JSTerms wird der `CommManager` verwendet, da er sehr komfortabel zu benutzen ist und keinen weiteren Prozess wie den Tornado Server in der standalone Variante benötigt. Auf die genaue Funktionsweise des `CommManagers` wird in Kapitel 3.2.3 der Konzeption eingegangen. Ein weiterer Vorteil von WebSockets in der standalone Variante ist das asynchrone Empfangen von Daten, sowie der Umstand, dass beide Versionen (standalone und Jupyter) damit für Anwender ähnlich zu benutzen sind.

Wie bereits erwähnt, dienen WebSockets zur bidirektionalen Kommunikation. In der entwickelten Anwendung werden allerdings nur Daten zwischen dem Tornado-Server und dem JSTerm ausgetauscht, da alle Interaktionen des Nutzers im Browser nicht persistent sind, sondern nur zur Laufzeit des JSTerms erfolgen sollen.

## 2.4. Jupyter-Notebooks und IPython

Jupyter ist eine Webanwendung, mit welcher sogenannte Jupyter-Notebooks entwickelt werden können [Gra+16]. Mit diesen ist es möglich, Code maschinenunabhängig auszutauschen, indem die Notebook-Datei in einer anderen Jupyter-Instanz gestartet wird. Jupyter unterstützt über 40 verschiedene Programmiersprachen und ist seit 2014 ein eigenständiges Projekt, welches aus IPython hervorgegangen ist. Ein häufiger Anwendungsfall ist die Durchführung von Berechnungen und die anschließender Darstellung innerhalb der Webanwendung. Jupyter ist im Grunde die Präsentationsschicht für verschiedene sogenannte Kernel. Diese führen als Backend den Code aus den Notebooks auf dem Server aus. Man kann Jupyter prinzipiell ohne einen Kernel betreiben, allerdings ist damit keine Ausführung von Jupyter-Notebooks möglich.

IPython ist ein Kernel für die Ausführung von Python-Skripten, welcher häufig mit Jupyter verwendet wird [PG07]. IPython ist dabei eine interaktive Shell, ähnlich wie auch der Standard-Python-Interpreter (CPython). Ein wesentlicher Unterschied von IPython im Vergleich zum CPython ist allerdings, dass es auch über das Python-Modul `IPython` importiert werden kann und so der IPython-Interpreter zur Verwendung in einem eigenen Projekt gestartet werden kann. Bis zur Version `3.x` waren die Notebooks in IPython integriert. Seit der Version `4.x` sind diese, wie oben erwähnt, ein eigenständiges Projekt mit dem Namen Project Jupyter.

## 2.5. Grundlagen zum GKS und GR

Dieser Abschnitt behandelt die Grundlagen des GR und GKS. Es wird auf die Schicht-Struktur der Anwendung eingegangen. Weiterhin wird die JavaScript-Realisierung des GR vorgestellt. Außerdem wird der Zusammenhang der verschiedenen Koordinatensysteme erläutert.

### 2.5.1. Beschreibung des GKS und des GR

GKS steht für Graphisches Kernsystem (engl. Graphical Kernel System), beschreibt eine API für 2D-Grafiken [Enc94] die in ISO/IEC 7942-1:1994 [Enc94] standardisiert ist. Dabei ist diese API unabhängig von der verwendeten Plattform und der Programmiersprache. Jedoch gibt es Sprachbindings für bspw. Fortran, C und Pascal [BB86, S. 6]. Es gibt im GKS fünf verschiedene Funktionen, die als Ausgabeprimitiven bezeichnet werden, welche zum Darstellen von Grafiken verwendet werden. In der Tabelle 2.1 befindet sich eine Beschreibung dieser Funktionen mit Beispielausgabe:

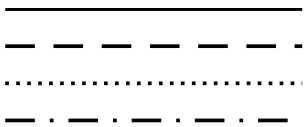
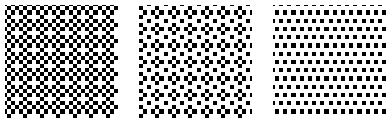
| Funktion | Beispiel |
|---|---|
| `Polyline`: Verkettung von Geraden. |  |
| `Polymarker`: Verschiedene Symbole. |  |
| `Text`: Darstellung von Text. |   |
| `Fill Area`: Bereich, der mit bestimmten Mustern gefüllt ist. |  |
| `Cell Array`: Bereich mit verschiedenfarbigen Farbzellen. Damit können rasterähnliche Grafiken erzeugt werden. |  |

Tabelle 2.1.: Ausgabeprimitiven des GKS. Grafiken wurden mittels eines GR-Beispielprogramm erzeugt [Hei16a].

Das GKS bildet eine abstrakte Schicht zwischen Anwendungsprogrammen und den Treibern, die die eigentliche Darstellung übernehmen. Somit sind alle mit dem GKS erzeugten Grafiken identisch und unabhängig von der verwendeten Programmiersprache und Art der Ausgabe.

Aufbauend auf dem GKS ermöglichet das im PGI/JCNS entwickelte GR-Framework[2] wissenschaftliche Visualisierungen basierend auf dem GKS. Innerhalb der GR-Routinen werden die GKS-Routinen zum Darstellen der Primitiven verwendet. GR kann direkt aus der Programmiersprache C verwendet werden, es gibt zudem auch Im-

[2] `http://gr-framework.org`

plementierungen für die Programmiersprachen Julia, Swift und Python. Die Schichten der Technologien (HTML, Qt, etc.), des GKS und des GR sind in der Grafik 2.1 noch einmal übersichtlich dargestellt.

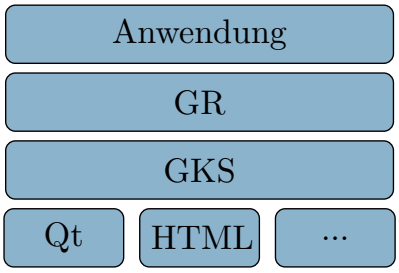



Abbildung 2.1.: Schichten der Anwendung, des GR, des GKS und der Technologien (HTML, Qt, etc.)

### 2.5.2. Vorhandene JavaScript-API des GR

Als Grundlage der JavaScript-Seite des JSTerms wird die vorhandene JavaScript-Implementierung des GR (GR.js) zum Zeichnen der Grafiken verwendet. Dieses wurde bereits in einer vorhergehenden Bachelorarbeit im PGI/JCNS entwickelt [Dro15].

Das GR.js ist Teil des offiziellen Repository des GR-Frameworks [Hei16a]. Um dieses aus den Quelldateien zu erzeugen, muss der Emscripten-Transpiler verfügbar sein. Mit Emscripten ist es möglich u. a. C / C++-Programme in JavaScript zu übersetzten [Mil+15]. Für diesen Vorgang der Transpilierung des GR.js steht ein `Makefile` zur Verfügung, sodass einfach `make gr.js` aufgerufen werden kann. Danach wird das GR.js, sofern es global verfügbar sein soll, in den Unterordner `lib/` (auf Linux- und UNIX-Systemen) der GR-Framework-Installation verschoben. Es kann dann z. B. ein Softlink in den `static/js`-Ordner (auf unixoiden-Systemen) des JSTerms erstellt werden.

### 2.5.3. Anpassung der JavaScript-API des GR

Für die Realisierung des JSTerms musste das bereits erwähnte GR.js angepasst werden. So kann jetzt optional bei der Initialisierung des GR-Objekts eine HTML-ID für ein Canvas-Objekt angegeben werden. Wenn keine ID angegeben wurde, wird standardmäßig `"canvas"` verwendet. Außerdem wird jetzt im GR.js ein Canvas-Element im DOM (Document Object Model) erzeugt, wenn dieses noch nicht existiert. Das Listing 2.1 zeigt die beispielhafte Verwendung mit der Übergabe der HTML-ID.

Weiterhin mussten das GR und das GR.js angepasst werden, da der aktuelle Zustand, wie beispielsweise die Window-Koordinaten, zuvor global gespeichert waren. Es gibt in der neuen Version verschiedene Kontexte, die diesen Zustand jeweils Kontext basiert speichern. Es kann jetzt `gr.selectcontext(index)` – wobei `index` eine Nummer eines Kontextes ist – aufgerufen werden, um einen anderen Kontext zu aktivieren. Zum Löschen eines Kontextes gibt es die Methode `gr.destroycontext(index)`, die den Speicherplatz eines Kontextes freigibt. Beim Selektieren des Kontextes wird dieser mit den Standardwerten initialisiert, wenn der Speicherplatz neu angelegt wurde.

```
var gr;

GR.ready(function() {
    var canvas_id = "testcanvas";
    gr = new GR(canvas_id);
});
```

Listing 2.1.: Initialisieren des GR-Objektes mit Canvas-ID.

### 2.5.4. Transformation der Koordinaten

Innerhalb des GKS muss zwischen drei verschiedenen Koordinatensystemen unterschieden werden, zwischen denen die Koordinaten transformiert werden müssen:

- den Weltkoordinaten
- den normalisierten Gerätekoordinaten
- den Gerätekoordinaten

Die Weltkoordinaten (engl. world coordinates, WC) sind die Koordinaten, in welchem die Daten liegen, so wie sie vom Nutzer erfasst wurden. Die normalisierten Koordinaten (engl. normalized device coordinates, NDC) sind ein normalisierter Koordinatenraum für den gilt: $x, y \in [0;1] \times [0;1]$. Die Gerätekoordinaten (engl. device coordinates, DC) sind die Koordinaten auf dem Ausgabemedium, im Falle des GR.js die Koordinaten des HTML5-Canvas. Beim Transformieren der NDC in DC muss das Seitenverhältnis beibehalten werden, damit es in der Grafik nicht zu Verzerrungen kommt [BB86, S. 50-52]. Dadurch das der Nutzer mit WC arbeitet, muss dieser seine Daten nicht in ein geeignetes Koordinatensystem umrechnen.

Der Ursprung von Grafiken, die das GR zeichnet, ist standardmäßig in der unteren linken Ecke. Dies kann mithilfe von entsprechenden Transformationen auch geändert werden. Im normalisierten Koordinatensystem (s. u.) ist der Punkt `(0; 0)` immer unten links. Im Gegensatz dazu liegt der Ursprung der Canvas-Koordinaten in der oberen linken Ecke des HTML5-Canvas.

Der Nutzer wählt aus den Daten den (Teil-)Bereich aus, den er betrachten möchte. Dieser wird als *Window* bezeichnet. Werte außerhalb werden automatisch abgeschnitten (sog. *Clipping*). Als zweites wählt der Nutzer einen Bereich (den sogenannten *Viewport*) aus dem normalisierten Koordinatenbereich aus, auf den die Daten ausgegeben werden, für welchen typischerweise $\forall x, y : x, y \in [0.1; 0.95] \times [0.1; 0.95]$ verwendet wird, damit bspw. noch Achsenbeschriftungen angezeigt werden können. Das *Window* wird dann vollständig in den Viewport transformiert [BB86, S. 12]. Die Grafik 2.2 verdeutlicht diesen Zusammenhang und die Transformation noch einmal anschaulich.

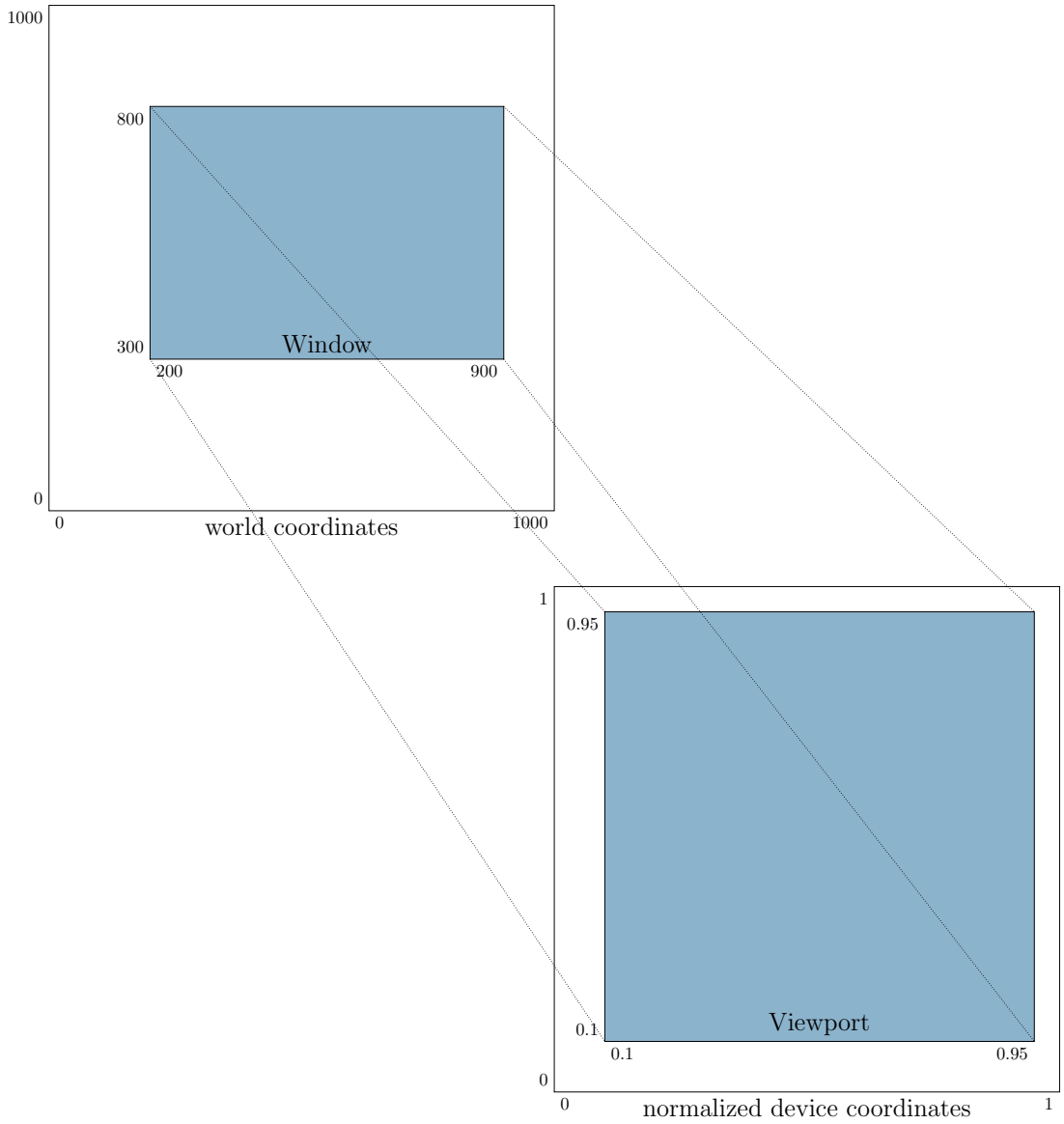



Abbildung 2.2.: Transformation der Koordinaten innerhalb des GKS. Die gezeigten Werte in world coordinates (WC) bzw. normalized device coordinates (NDC) sind hier nur beispielhaft. Die WC sind der gesamte Datenbereich für den in diesem Beispiel gilt: $x, y \in [0; 1000] \times [0; 1000]$. Das Window ist ein vom Benutzer gewählter Teilbereich der Daten in WC, also hier: $x, y \in [200; 900] \times [300; 800]$, welcher vollständig in den Viewport transformiert wird.

## 2.6. Zusammenspiel zwischen zwei Canvas-Objekten pro Grafik

Innerhalb des JSTerms werden pro Grafik zwei Canvas-Objekte genutzt. Das Problem bei HTML5-Canvas-Objekten ist, dass nach dem Zeichnen keine Informationen mehr darüber verfügbar sind, welche Pixel semantisch zusammenhängen. Diese Information ist jedoch relevant, um verschiedene Grafiken voneinander zu unterscheiden, aber auch, um bspw. eine Box für den Box-Zoom (s. Interaktionsmöglichkeiten des JSTerms in Abschnitt 4.1) zu visualisieren. Deshalb werden zwei Canvas-Objekte verwendet, die über CSS (Cascading Style Sheets) genau übereinander gelegt werden. Dies wird über die CSS-Positionierung wie im Listing 2.2 erreicht. Die Zahl in den IDs entspricht dabei der ID einer Grafik. Das „untere“ Canvas enthält die Zeichnungen des GR. Hierauf werden die Zeichnungen aus den empfangenen Daten visualisiert. Die Daten dieses Canvas werden auch über die Download-Funktion verfügbar gemacht. Das „obere“ Canvas hingegen enthält dynamische Visualisierungen des JSTerms. Hier werden z. B. die Boxen für den Box-Zoom oder für die Datenverfolgung (s. Abschnitt 4.1) gezeichnet. Bei jedem Neuzeichnen einer Box in den entsprechenden Events muss zuvor dieses Canvas gelöscht werden. Weiterhin werden auf dem oberen Canvas alle Events gebunden, wie bspw. auch das Zoomen mit dem Mausrad.

```css
#canvas-container {
    position: relative;
}

#canvas-container > canvas:nth-child(2) {
    position: absolute;
    top: 0;
    left: 0;
}
/* ... */
```

```html
<!-- ... -->
<div id="canvas-container">
    <canvas id="canvas_1" width="500" height="500"></canvas>
    <canvas id="canvas_1_overlay" width="500" height="500"></canvas>
</div>
<!-- ... -->
```

Listing 2.2.: Übereinanderlegen von zwei Canvas-Objekten mittels HTML und CSS.

# 3. Konzeption

Dieses Kapitel beschreibt das Konzept der JavaScript-Umgebung, welche implementiert wurde. Zuerst wird auf die Schichten der gesamten Anwendung eingegangen. Danach wird der Ablauf der Kommunikation zwischen dem Kernel und JSTerm erläutert. Im Anschluss wird das Protokoll zur Kommunikation zwischen der MATLAB ähnlichen API des GRs und JSTerm beschrieben.

## 3.1. Schichten der Anwendung

In den Grundlagen (vgl. Abbildung 2.1 aus den Grundlagen) wurde bereits auf die Schichten-Struktur des GR, des GKS und der Technologien (HTML, Qt, etc.) eingegangen. Die Grafik 3.1 verdeutlicht nun die gesamte Schichtenstruktur. Die unteren Schichten des GR, des GKS und der Technologien bleiben wie in den Grundlagen beschrieben. Über der GR-Schicht befindet sich das GR.js. In diesem stehen alle GR-Funktionen in JavaScript zur Verfügung. Über dem GR.js befindet sich das JSTerm, das die Grundlage der vorliegenden Bachelorarbeit darstellt. Die zweite Schicht ist dann bspw. die MATLAB ähnliche API des GRs (mlab.py). Darüber als oberste Schicht findet sich – wie auch in der einfachen Schichtenarchitektur in den Grundlagen (s. Abbildung 2.1) – die vom Nutzer entwickelte Anwendung.

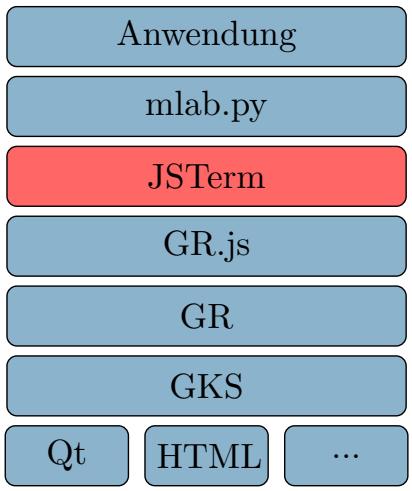



Abbildung 3.1.: Gesamte Übersicht der Schichten von der Anwendung bis zu den Technologien (HTML, Qt, etc.) inklusive der neuen JSTerm Schicht.

Die Kommunikation zwischen dem zweitobersten Layer (mlab.py) und drittobersten (JSTerm) Layer erfolgt über das neu entwickelte Protokoll (s. Abschnitt 3.3) im JSON-Format (*JavaScript Object Notation*). Ein wesentlicher Vorteil ist, dass die empfangenen Daten so direkt als JavaScript-Objekte im JSTerm interpretiert werden können. Außerdem ist das JSON-Format gut menschen- und maschinenlesbar. Bis zu dieser dritten Schicht, dem JSTerm, ist eine Semantik der Daten erkennbar. D. h. das JSTerm kennt zusammenhängende *plot*-Objekte in den Grafiken und kann mit diesen entsprechend arbeiten, bspw. beim Selektieren von einzelnen *plot*-Objekten über die Legende (s. Interaktionsmöglichkeiten des JSTerms in Abschnitt 4.1). Das JSTerm muss die Daten also so speichern, wie sie semantisch zusammengehören. In den Schichten unterhalb des JSTerms ist diese Semantik der Daten nicht mehr rekonstruierbar, da hier mit den GKS-Primitiven gearbeitet wird.

## 3.2. Kommunikation zwischen Kernel und JSTerm

Dieser Abschnitt beschreibt die Kommunikation zwischen der JSTerm-Schicht und der mlab.py-Schicht. Im Falle der Jupyter-Variante des JSTerm wird die Anwendung der Endnutzer in einem Jupyter-Notebook programmiert und mittels des Kernels, also hier IPython, ausgeführt. Zur Kommunikation zwischen Jupyter und dem JSTerm wird hier der `CommManager` von Jupyter verwendet. Dieser nutzt intern WebSockets als Protokoll. Auf die genaue Funktionsweise dieses Kommunikationsweges wird in Unterabschnitt 3.2.3 eingegangen.

Im Falle der standalone Implementierung wird die entwickelte Anwendung über den vom Benutzer ausgewählten Python-Interpreter ausgeführt. Danach werden die Daten im mlab.py über TCP-Sockets an den Tornado-Server geschickt. Von diesem werden die Daten über WebSockets weiter an das JSTerm gesendet. Diese Funktionsweise wird im Unterabschnitt 3.2.2 behandelt. In beiden Fällen werden die Daten dann im JSTerm-Client weiter verarbeitet.

### 3.2.1. Vergleich zwischen standalone Realisierung und Realisierung als Jupyter-Plugin

Die beiden Implementierungen, einerseits die standalone Variante und andererseits das Jupyter-Plugin, nutzen größtenteils die gleiche Codebasis. Der Eintrittspunkt in das JavaScript des JSTerms ist allerdings etwas anders, da in der Jupyter-Variante zuvor die empfangene Nachricht anders verarbeitet wird. In der standalone Variante wird das JSTerm-JavaScript durch den Tornado-Server ausgeliefert. Bei der Jupyter-Variante wird das JSTerm als sogenannte Notebook-Extension ausgeliefert. Das Listing 3.1 zeigt, welche Schritte nötig sind, um das JSTerm als Notebook-Extension für den aktuellen Nutzer einzurichten.

Innerhalb der Jupyter-Umgebung wird u. a. RequireJS zur Verfügung gestellt. Dieses Framework bietet die Möglichkeit aus JavaScript heraus weitere JavaScript-Dateien und JavaScript-Module zu laden [Chu11]. Durch RequireJS wird auch die

sogenannte `define`-Methode zur Verfügung gestellt. Mit diesen `define`-Methoden können Module definiert werden, deren Abhängigkeiten als Argumente übergeben werden und vor der Ausführung erfüllt werden müssen. Ein Vorteil ist, dass durch diese Kapselung nicht alle Abhängigkeiten in den globalen Namensraum geladen werden müssen. Das Listing 3.2 zeigt ein solches Modul beispielhaft, welches dann als Notebook-Extension verfügbar ist.

```
cd ~/.ipython/nbextensions/
ln -s ~/pfad/zum/jsterm.js .
jupyter nbextension enable jsterm
```

Listing 3.1.: JavaScript-Datei, am Beispiel des JSTerms, als Notebook Extension verfügbar machen (auf unixoiden-Systemen).

```
define(function(){
    function _on_load(){
          console.info('Hello World');
    }

    return {load_ipython_extension: _on_load };
})
```

Listing 3.2.: Minimale Jupyter-Notebook Extension.

Die Grafik 3.2 verdeutlich die unterschiedlichen Startpunkte innerhalb des JSTerms. In der Jupyter-Variante wird im RequireJS-Modul das JSTerm gestartet. Nach dem Bearbeiten der empfangenen Daten wird eine Methode des JSTerms aufgerufen, die die weitere Behandlung der Daten vornimmt. In der standalone Variante kann diese Methode unmittelbar aufgerufen werden, da hier RequireJS nicht benötigt wird. In der standalone Variante bringt RequireJS keine Vorteile mit sich, da es nur ein Modul gibt. Damit sind die einzigen Abhängigkeiten jQuery[3], Bootstrap[4], Font-Awesome[5] und das GR.js. Nach der Behandlung der Nachricht im JSTerm werden die weiteren Methoden entsprechend der Grafik 3.2 durchlaufen.

[3] https://jquery.com/
[4] https://getbootstrap.com/
[5] http://fontawesome.io/

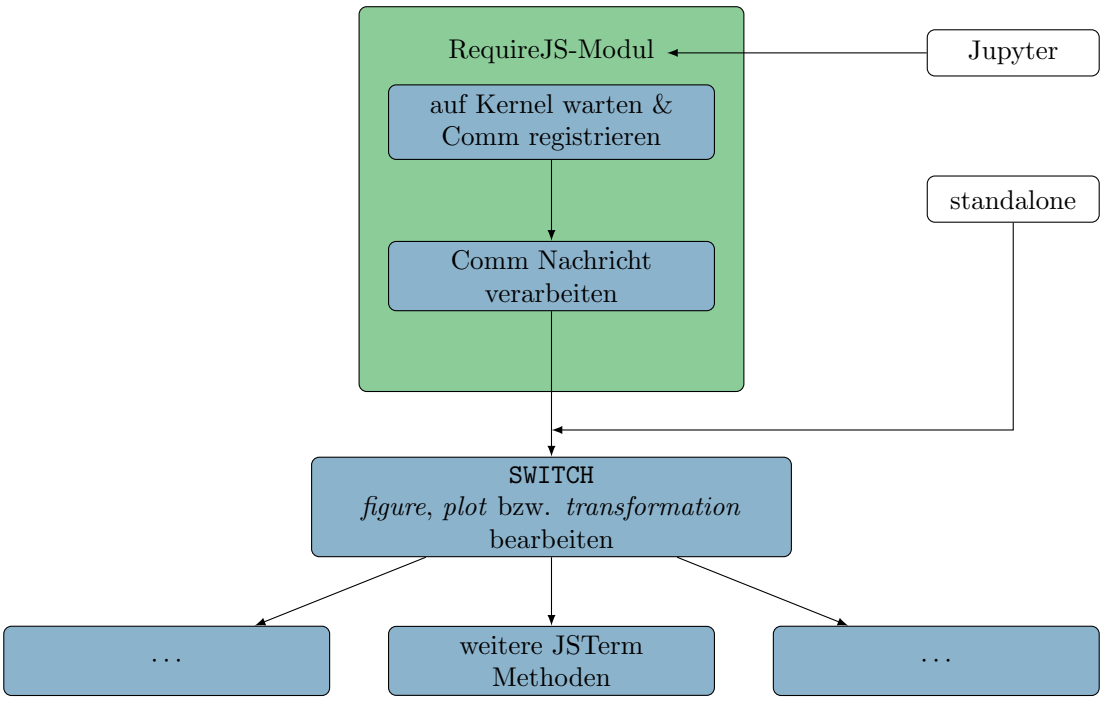



Abbildung 3.2.: Eintrittspunkte in das JSTerm in der Jupyter-Variante und in der standalone Variante.

### 3.2.2. Kommunikation mithilfe von WebSockets in der standalone Realisierung

Im Folgenden wird die Kommunikation in der standalone Variante des JSTerms erläutert. Die Grafik 3.3 zeigt schematisch, wie diese erfolgt. Es werden WebSockets verwendet, da diese optimal zur Kommunikation zwischen Webanwendungen und Servern geeignet sind [Hic+14]. Außerdem benutzt der Jupyter `CommManager` (s.u. in Kapitel 3.2.3) intern auch WebSockets. In der standalone Variante wird das JSTerm durch einen Webserver, welcher mit dem Tornado-Web-Modul (`tornado.web`) implementiert ist, ausgeliefert. Der Tornado-Server arbeitet als Proxy-Server, um zwischen dem JSTerm und dem mlab.py einen Kommunikationsfluss zu ermöglichen. Die Firewall muss entsprechend so konfiguriert werden, dass die Ports des Tornado-Servers auch von außen erreichbar sind. Die rechte und die linke Seite (siehe Grafik 3.3) sind dabei die jeweiligen Clients. Im Falle des JSTerms werden die Daten momentan nur lokal ausgetauscht, sodass die Firewall hier kein Problem darstellt. Trotzdem wird Tornado verwendet, damit die Möglichkeit des Sendens über das Netzwerk für die Zukunft bereits (bis auf das Verschlüsseln der Daten) realisiert ist. Neben der Firewall muss auch die NAT (engl. Network Address Translation) beachtet bzw. richtig konfiguriert werden, sofern die internen IP-Adressen der Netzwerke privat sind. Mithilfe von NAT werden die privaten IP-Adressen in einem Router durch die global gültige IP-Adresse des Routers übersetzt. Der Router speichert die Zuordnung von privaten IP-Adressen, seiner globalen und den Ports in einer sogenannten NAT-Tabelle [Sch97].

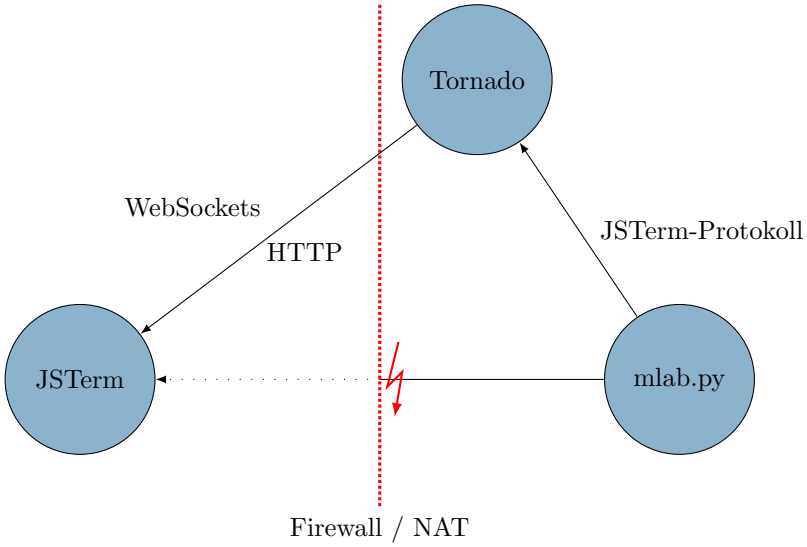



Abbildung 3.3.: Kommunikation mithilfe des Tornado-Servers.

Auf der rechten Seite (s. Abbildung 3.3) werden die Daten aus dem mlab.py Modul mithilfe von TCP Sockets an den Tornado-Server geschickt. Dieser empfängt solange ein Datenpaket, bis das ASCII-Zeichen 23, sprich `ETB (End-Of-Transmission-Block)` gelesen wird. Danach wird dieses Datenpaket an alle verfügbaren WebSocket-Clients gesendet. Die verfügbaren WebSocket-Clients werden innerhalb des Servers verwaltet.

Die linke Seite (s. Abbildung 3.3) besteht aus dem WebSocket-Client, von dem es prinzipiell mehrere unterscheidbare Instanzen geben kann. Diese sind mithilfe des in Tornado verfügbaren WebSocket-Moduls (`tornado.websocket`) realisiert. Weiterhin besteht die linke Seite aus dem Webserver, der die statischen Dateien (CSS und JavaScript) und die `index.html`-Seite ausliefert. Diese kann nach eigenen Bedürfnissen entsprechend angepasst werden.

Die linke und die rechte Seite (s. Abbildung 3.3) sind auf zwei unterschiedlichen Ports verfügbar, da links WebSockets und HTTP sowie rechts TCP als Protokolle verwendet werden. Prinzipiell ist es durch WebSockets möglich, Daten in beide Richtungen zu versenden. Momentan ist es aber nur vorgesehen und implementiert, dass die Daten vom mlab.py-Modul zum JSTerm, versendet werden.

### 3.2.3. Kommunikation mithilfe des `CommManagers` von Jupyter

Bei der Jupyter-Variante des JSTerms kann die Kommunikation mithilfe des `CommManagers` von IPython erfolgen. Dieser nutzt intern auch WebSockets. In der Grafik 3.4 wird gezeigt, dass die Kommunikation zwischen dem Kernel und dem Notebook-Server (Jupyter) über ØMQ (ZeroMQ) erfolgt. ZeroMQ ist eine Programmbibliothek zum asynchronen Versenden von Nachrichten zwischen verschiedenen Prozessen [Cor14]. Die Kommunikation zwischen Jupyter und dem Browser – und damit JSTerm – erfolgt über WebSockets.

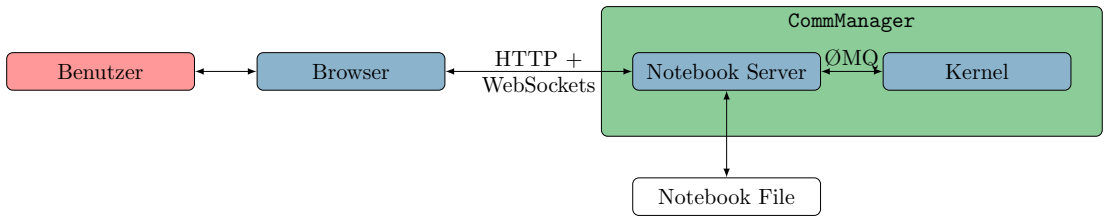



Abbildung 3.4.: Kommunikaiton mithilfe des `CommManagers`.

Der Vorteil beim Senden der Daten über die Jupyter-Instanz ist, dass das Senden durch die eingebauten Module sehr komfortabel realisiert ist. Der Nachteil könnte allerdings sein, dass beim Versenden großer Datenmengen die Performance des Jupyter-Servers – und damit des Wirtssystems – beeinträchtigt werden kann.

Das Senden der Daten durch das mlab.py, im Fall der Jupyter-Variante über den `CommManager` oder im Fall der standalone Variante über TCP-Sockets, ist parallel zur Entwicklung des JSTerms am PGI/JCNS realisiert worden. Dabei ist das Senden der Daten aus dem mlab.py nicht Teil der vorliegenden Bachelorarbeit.

## 3.3. Protokoll der Kommunikation

Dieses Kapitel beschreibt das Protokoll zwischen dem Kernel und JSTerm. Dabei besteht zwischen diesen beiden eine unidirektionale Verbindung. Das heißt, der Kernel sendet nur Daten in Richtung des JSTerms. Alle clientseitigen Änderungen und Manipulationen erfolgen damit nur zur Laufzeit und werden nicht persistent zurückgespeichert. Das entwickelte Protokoll orientiert sich an dem Visualisierungspaket Plots.jl der Programmiersprache Julia [Bre16]. Eine Grafik kann wie ein Baum abstrahiert werden. Dabei besteht diese zwingend aus einer *figure* als Wurzel des Baumes und beliebig vielen *plot*-Objekten sowie beliebig vielen *transformation*-Objekten als Blätter. Die Tiefe des Baumes ist zwischen eins und zwei. Damit ist es nicht möglich *plots* und *transformations* zu schachteln. Diese Baumstruktur ist in der Grafik 3.5 dargestellt.

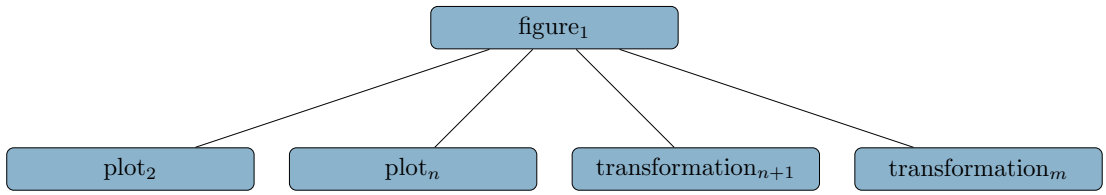



Abbildung 3.5.: Baumstruktur des Protokolls. Die Indizes $1, 2, n, n+1, m \in \mathbb{N}_{>0}$ sind nur beispielhaft und sollen die Eindeutigkeit aller Indizes verdeutlichen.

Bei der Jupyter-Variante werden die empfangenen Daten pro Comm-Nachricht verarbeitet. Bei der standalone Variante wird ein Datenpaket, wie in Abschnitt 3.2.2 erwähnt, mit einem `ETB` terminiert. Dieses Datenpaket ist ein Objekt vom

Typ *figure*, *plot* oder *transformation*. Damit ist ein Datenpaket (ohne Terminierungszeichen) analog zu einer Nachricht über den `CommManager`. Das `ETB`-Zeichen (`End-Of-Transmission-Block`) wird bereits im Tornado-Server entfernt, bevor die Daten über WebSockets weiter versendet werden.

Die Bezeichnungen *figure* und *plot* sind an MATLAB angelehnt [Mat16]. Die Daten werden im JSON-Format übertragen, da diese im JSTerm direkt als JavaScript-Objekte interpretiert werden können. Die Benennung der Werte, die gesendet werden können (bspw. im *transformation*-Objekt), wurden an das GR angelehnt [Hei16b]. So ist es z. B. möglich, den Viewport zu setzen, indem ein Array von vier Werten (`xmin, xmax, ymin, ymax`) als Wert zu dem Schlüssel `content.setviewport` des *transformation*-Objekts gesendet wird. Intern wird dann die GR-Methode `gr.setviewport` aufgerufen.

Das *figure*-Objekt dient zur Gruppierung von *plots* und *transformations* sowie zur Abstraktion einer gesamten Grafik. Ein *plot* beschreibt beispielsweise einen oder mehrere Graphen, der dargestellt werden soll. Verschiedene *plot*-Objekte sind semantisch unterschiedliche Teile der gesamten Grafik (*figure*). Die *transformation*-Objekte dienen zur Modifikation der Grafiken. Sie können einem *plot*-Objekt oder der ganzen *figure* zugeordnet werden. Pro Grafik muss es mindestens eine *figure* vorhanden sein. Von den *plot*- und *transformation*-Objekten kann es beliebig viele (auch keine) pro Grafik geben. Wenn eine neue Grafik dargestellt werden soll, muss damit mindestens ein *figure*-Objekt gesendet werden.

Die Struktur der einzelnen Objekte soll in den folgenden drei Unterkapiteln erläutert werden. Allen gemeinsam ist, dass sie aus einem `content`-Objekt mit Nutzdaten und einem `meta`-Objekt für Metainformationen bestehen. Da die Metainformationen bei allen Objekttypen analog sind, werden diese in der nächsten Auflistung nur einmal für alle Objekttypen dargestellt:

**`meta.id:`** Ist eine eindeutige Identifizierung durch eine Ganzzahl. Die Nummerierung startet bei 1 und ist für alle Objekte eindeutig, unabhängig ob *figure*, *plot* oder *transformation*.

**`meta.protocol_version:`** Ist die aktuelle Version des Protokolls in der Notation `x.y`.

**`meta.type:`** Damit wird der Typ eines Objektes gekennzeichnet, also ob es eine *figure*, ein *plot* oder eine *transformation* ist.

**`meta.figure_id:`** Dies ist die ID der *figure*, die zu einem Objekt gehört. Für alle *figure*-Objekten gilt $id = figure\_id$.

Zunächst werden alle möglichen Werte der Objekte aufgeführt und erläutert, im Anschluss wird ein Beispiel zu dem jeweiligen Objekttyp beschrieben. Als letztes folgen jeweils die ggf. optionale Werte.

### 3.3.1. Beschreibung des *figure*-Objektes

Die Pflichtattribute, die mitgesendet werden müssen, sind im Folgenden aufgelistet. Ein beispielhaftes Objekt ist im Listing 3.3 dargestellt.

**`content.background_color:`** Ist die Hintergrundfarbe der Grafik. Als Wert wird ein Array mit RGB-Werten bestehend aus drei Zahlen zwischen 0 und 1 erwartet. Beispielsweise `[0.5, 0.3, 0.7]`.

**`content.size:`** Dies ist ein Array mit zwei ganzzahligen Werten für die Größe des HTML-Canvas, auf das gezeichnet wird, bspw. `[400, 600]` für ein Canvas der Breite 400 und Höhe von 600 Pixeln.

```
{
    "content": {
        "background_color": [0.5, 0.3, 0.7],
        "size": [400, 600],
        "transformation": 4,
        "keep_org": true
    },
    "meta": {
        "id": 1,
        "protocol_version": "1.0",
        "type": "figure",
        "figure_id": 1
    }
}
```

Listing 3.3.: Beispielhaftes *figure*-Objekt.

Die optionalen Argumente sind:

**`content.transformation:`** Dies ist die ID eines *transformation*-Objektes. Dieses gilt somit für die gesamte Grafik. Da das *figure*-Objekt zuerst gesendet werden muss, existiert das *transformation*-Objekt mit der angegebenen ID erst zu einem späteren Zeitpunkt.

**`content.title:`** Der Titel des Plots als Zeichenkette.

**`content.keep_org:`** Damit wird bestimmt, ob der Koordinatenursprung – und damit auch Achsen und Gitter – (siehe `content.x_org` und `content.y_org`) beim Empfang eines *plot*-Objektes angepasst werden soll oder nicht. Der Standardwert ist `true`. Das heißt, der Ursprung wird auf den kleinsten x- und y-Wert der Daten des ersten *plot*-Objektes gesetzt. Wenn `false` übergeben wird, so wird für jeden empfangenen *plot* der Ursprung neu gesetzt.

**content.x_org:** Die übergebene x-Koordinate des Ursprungs der Grafik.

**content.y_org:** Die übergebene y-Koordinate des Ursprungs der Grafik.

### 3.3.2. Beschreibung des *plot*-Objektes

Die erste Liste führt die Pflichtattribute auf. Die zweite Liste beschreibt die optionalen Attribute. Das Listing 3.4 zeigt ein Beispiel.

**content.kind:** Dies ist die Darstellungsart, wie die Daten dargestellt werden sollen, bzw. welche Funktion gesendet wird. So wird beim Aufruf der *contour()*-Methode im Kernel die Darstellungsart auf `"contour"` gesetzt und bei bspw. dem Aufruf der `plot()`-Methode im Kernel wird das `kind` auf `"line"` gesetzt. Die Benennung der Werte von `content.kind` ist dabei analog zu der im mlab.py.

**content.data:** Hier werden die Daten für einen *plot* abgelegt. Diese sind abhängig vom jeweiligen `content.kind`.

```
{
    "content": {
        "kind": "line",
        "data": {
            "x": [0, 0.33, 0.66, 1],
            "y": [0, 0.2, 0.75, 1]
        }
    },
    "meta": {
        "id": 3,
        "protocol_version": "1.0",
        "type": "plot",
        "figure_id": 1
    }
}
```

Listing 3.4.: Beispielhaftes *plot*-Objekt.

Folgende Argumente des *plot*-Objektes sind optional:

**content.transformation:** Dies ist die ID eines *transformation*-Objektes, welches nur für ein *plot*-Objekt gilt.

**content.overwrite_plots:** Dieses Argument legt fest, dass der aktuelle *plot* alle vorherigen überschreibt.

### 3.3.3. Beschreibung des *transformation*-Objektes

Bei diesem Objekt sind alle Werte optional. Im Listing 3.5 ist ein Beispiel zum Setzen des *Viewports* dargestellt.

```
{
    "content": {
        "setclip": 1,
        "setviewport": [0.1, 0.95, 0.1, 0.95]
    },
    "meta": {
        "id": 4,
        "protocol_version": "1.0",
        "type": "transformation",
        "figure_id": 1
    }
}
```

Listing 3.5.: Beispielhaftes *transformation*-Objekt.

**`content.setscale:`** Hiermit kann in einem Array von Zeichenketten angegeben werden, wie die Achsen der Grafik dargestellt werden. Die verfügbaren Angaben sind in der folgenden Auflistung dargestellt. Wird dieser Wert nicht gesetzt, so wird der Standard (lineare Achsen), verwendet.

- **`OPTION_X_LOG:`** logarithmische x-Achse
- **`OPTION_Y_LOG:`** logarithmische y-Achse
- **`OPTION_Z_LOG:`** logarithmische z-Achse
- **`OPTION_X_FLIP:`** gespiegelte x-Achse
- **`OPTION_Y_FLIP:`** gespiegelte y-Achse
- **`OPTION_Z_FLIP:`** gespiegelte z-Achse

**`content.setwindow:`** Damit werden die Window-Koordinaten angepasst. Es wird ein Array von vier Werten erwartet, welche in ihrer vorliegenden Reihenfolge die minimalen und maximalen x-Werte sowie die minimalen und maximalen y-Werte angeben. Eine korrekte Angabe wäre bspw. `[0.25, 0.75, 0.25, 0.75]`.

**`content.setwswindow:`** Damit werden die Window-Koordinaten der Workstation, im Falle des JSTerms die des Canvas-Objektes, gesetzt. Die Reihenfolge der Werte entspricht der von `content.setwindow`.

**`content.setwsviewport:`** Mit diesem Wert werden die Viewport-Koordinaten der Workstation gesetzt. Die Reihenfolge der Werte entspricht `content.setwindow`.

**`content.setviewport:`** Setzen der Viewport-Koordinaten. Die Reihenfolge der Werte entspricht `content.setwindow`. Die Werte liegen zwischen 0 und 1, da der Viewport Teil der normalized device coordinates ist.

**`content.setspace:`** Hiermit kann eine Projektion von 3D zu 2D (nur für 3D-Routinen wie bspw. *trisurf*) durchgeführt werden. Die Werte sind der minimale und maximale z-Wert der Achse, ein Azimutwinkel und ein Polarwinkel, jeweils in Grad. Der Azimutwinkel ist der Winkel auf der $xy$-Ebene. Der Polarwinkel ist der Winkel zu der $z$-Achse.

**`content.setclip:`** Einstellung ob Clipping ausgeschaltet (Wert = 0) oder eingeschaltet (Wert = 1) werden soll. Der Standard ist, dass Clipping eingeschaltet ist. Wenn Clipping eingeschaltet ist, werden keine Daten außerhalb des Viewports durch das GR gezeichnet. Koordinatenachsen stellen hierbei eine Ausnahme dar. Es wird hier eine *Number* verwendet, um konsistent zu dem Python-Implementierung des GR-Frameworks zu sein.

# 4. Implementierung

In diesem Kapitel soll die Implementierung bzw. Realisierung des JSTerms beschrieben werden. Es wird zuerst auf die Interaktionsmöglichkeiten eingegangen, die dem Endanwender durch das JSTerm zur Verfügung stehen. Danach folgt die Implementierung mit Beschreibungen, wie die relevanten Teile im Quellcode umgesetzt sind. Im letzten Abschnitt wird die Einrichtung des JSTerms in der Jupyter- und standalone Variante beschrieben.

## 4.1. Interaktionsmöglichkeiten des JSTerms

In Grafik 4.1 ist ein Screenshot des JSTerms abgebildet. Dieser zeigt das Canvas-Objekt mit einer *figure* und beispielhaft einem dazugehörigen *plot*-Objekt. Am oberen Rand sind die Buttons für die Interaktionsmöglichkeiten zu sehen.

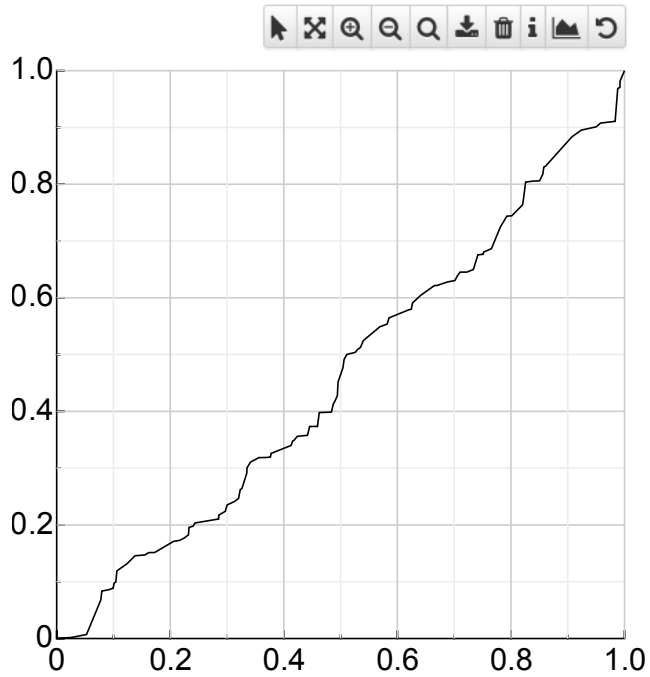



Abbildung 4.1.: Screenshot des JSTerms. Auf diesem sieht man eine *figure* mit einem *plot* als Linienplot. Weiterhin sieht man das Gitter und die x- und y-Achse. Oben rechts befinden sich die verschiedenen Buttons für die Interaktionsmöglichkeiten, auf die in 4.1 eingegangen wird.

Die Interaktionsmöglichkeiten des JSTerms werden im Folgenden kurz aufgelistet. Im Anschluss werden diese im Detail beschrieben.

- Selektieren einzelner *plot*-Objekte über die Legende
-  Verschieben der gesamten *figure*
-  Zoom mit vorgegebenem Faktor
-  Zoom mittels einer Auswahlbox
-  Download als PNG
-  Löschen der gesamten Ausgabe
-  Informationen über JSTerm
-  den von der Mausposition nächstgelegenen Punkt eines *plot*-Objektes anzeigen
-  Koordinatensystem auf minimalen Wertebereich der Daten skalieren, der alle Datenpunkte einschließt

**Selektieren einzelner *plot*-Objekte über die Legende:** Damit ist gemeint, dass über die Legende einzelne *plot*-Objekte ein- und ausgeschaltet werden können. Standardmäßig werden alle empfangenen *plots* in der *figure* angezeigt. Durch das Ein- und Ausschalten kann zu Analysezwecken nur eine Teilmenge der *plots* sichtbar gemacht werden. Für diese Funktion ist die Datenstruktur (und das Protokoll) von erheblicher Bedeutung, da in den unteren Anwendungsschichten ab dem GR.js keine semantischen Informationen mehr verfügbar sind und einzelne *plot*-Objekte somit nicht unterschieden werden können. Diese Informationen müssen im JSTerm gesichert werden. Die Label in der Legende bestehen aktuell aus der Zeichenkette „Plot“ und der ID (als Zahl) des jeweiligen Objektes.

**Verschieben der gesamten *figure*:** Hiermit kann die gesamte *figure* verschoben werden. Wenn die linke Maustaste gedrückt wird, werden die Startkoordinaten erfasst. Beim Loslassen der Taste werden entsprechend die Endkoordinaten gesichert. Daraus wird ein Verschiebungsvektor gebildet und die Window-Koordinaten mittels der `gr.setwindow`-Methode angepasst.

**Zoom mit vorgegebenem Faktor:** Diese Funktion ermöglicht das Zoomen mit einem konstanten Faktor. Wenn auf die Lupe mit dem Plus- bzw. Minus-Symbol gedrückt wird, wird um den Faktor zwei (jeweils pro Achse) in die *figure* bzw. aus ihr heraus gezoomt. Dies wird durch das Anpassen der Window-Grenzen realisiert und ist durch die Verarbeitung im GR.js performant. Weiterhin besteht auch die Möglichkeit mit dem Mausrad zu zoomen. Dabei wird durch Herunterscrollen in die *figure* gezoomt und durch Hochscrollen aus dieser heraus. Da beim Zoomen mit der Maus sehr viele Events ausgelöst werden, wird hier nur der Zoom-Faktor 1.01 verwendet.

**Zoom mittels einer Auswahlbox:** Hierbei kann mit einer Auswahlbox eine bestimmte Stelle der *figure* ausgewählt werden. Der Box-Zoom wird häufig verwendet um eine *Region-of-Interest* (ROI) festzulegen. Wie in Abschnitt 2.6 angeführt, werden für jede *figure* zwei Canvas-Objekte verwendet, damit auf das „obere“ Canvas die ROI-Box gezeichnet werden kann. Das „untere“ Canvas enthält die *figure* selbst, das „obere“ Canvas enthält die Box, welche die ROI beschreibt, in die hineingezoomt werden soll. Bei jedem Neuzeichnen der Box für die ROI wird zuvor der Inhalt des oberen Canvas gelöscht. Beide Canvas-Objekt-Elemente sind zur gesamten Laufzeit im DOM (Document Object Model) verfügbar.

**Download als PNG:** Mit dieser Option kann die aktuelle *figure* als PNG-Grafik heruntergeladen werden. Zu beachten ist, dass die Auflösung hier nur die des Canvas ist. Mit dem GR-Framework ist es prinzipiell möglich, Grafiken in einer höheren Auflösung (`gr.beginprint` und `gr.endprint` Methoden) und in anderen Formaten zu erzeugen, jedoch ist dies bei JSTerm nicht möglich. Es wäre prinzipiell möglich, das GR.js so anzupassen, dass die Daten für einen Download zur Verfügung stehen, bspw. mithilfe der *File System API* des *emscripten*.

**Löschen der gesamten Ausgabe:** Mit dem Mülleimer-Symbol wird die ganze Ausgabe zurückgesetzt. Außerdem wird der Speicherplatz mithilfe der `gr.destroycontext(index)`-Methode (s. Unterabschnitt 4.2.5) freigegeben. Damit kann für diesen Kontext bei einem erneutem Empfang einer Nachricht Speicherplatz neu angelegt werden. Es werden dabei folgende Schritte durchgeführt:

- Speicherplatz mithilfe der `gr.destroycontext(index)`-Methode freigeben
- *figure* aus der Liste aller momentan verfügbaren Grafiken löschen
- DOM-Element der *figure* mit gesamtem Inhalt löschen

**Informationen über JSTerm:** Hiermit wird ein neuer Tab im Webbrowser mit der Internetseite des GR-Frameworks geöffnet.

**Den von der Mausposition nächstgelegenen Punkt eines *plot*-Objektes anzeigen:** Mit dieser Funktion kann der Datenpunkt, welcher sich am nächsten an der aktuellen Mausposition befindet, angezeigt werden. Dabei wird zwischen den Datenpunkten linear interpoliert, sodass sich stetig auf dem Graph bewegt werden kann. Die Datenpunkte selbst sind auch linear verbunden. Es wird dabei von der aktuellen Mausposition der Punkt auf dem *plot*-Objekt genommen, welcher den geringsten euklidischen Abstand zur Mausposition hat. Die Box, die den nächsten Datenpunkt angibt, ist in der Grafik 4.2 beispielhaft dargestellt.

**Koordinatensystem auf minimalen Wertebereich der Daten skalieren, der alle Datenpunkte einschließt:** Mit dieser Funktion wird das Koordinatensystem und das Gitter so angepasst, dass die kleinsten x- und y-Werte die untere Windowgrenze und die größten x- und y-Werte die obere Windowgrenze sind. Diese Funktion ist somit ein *Scale-to-Fit*.

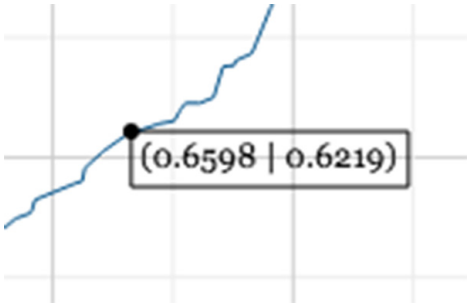



Abbildung 4.2.: Box mit Koordinatenpaar, welches den am nächsten gelegenen Punkt auf einem *plot* von der Mausposition aus angibt.

## 4.2. Implementierung des JSTerms

Dieser Abschnitt geht auf die Realisierung des JSTerms ein. Es werden zuerst die verwendeten Event-Handler aus JavaScript erläutert. Weiterhin wird darauf eingegangen, wie die Grafik-Objekte (*figure*, *plot* und *transformation*) intern gespeichert werden. Danach wird auf die Implementierung des JSTerm in der Jupyter-Variante sowie in der standalone Variante eingegangen. Der letzte Unterabschnitt behandelt einige spezielle JSTerm-Methoden.

### 4.2.1. Event-Handler des JSTerms

Dieser Abschnitt beschreibt die Event-Listener, die innerhalb des JSTerms implementiert sind. Event-Listener sind JavaScript-Funktionen, die aufgerufen werden, wenn ein bestimmtes Ereignis (Event) auftritt. Wenn ein Event-Listener an ein Event gebunden wurde, so wird die JavaScript-Anwendung weiter ausgeführt, bis dieses Event auftritt, dann wird erst das Event behandelt, bevor die Anwendung weiter regulär ausgeführt wird. Mittels jQuery ist es sehr komfortabel möglich Event-Listener zu registrieren (bind) und zu entfernen (unbind). Das Listing 4.1 zeigt beispielhaft, wie auf einem DOM-Element ein Event registriert und entfernt werden kann. Es gibt viele verschiedene Events, die ausgelöst werden können. Im JSTerm werden die folgenden Maus-Events verwendet:

**`wheel`,** welches ausgelöst wird, wenn das Mausrad bewegt wird

**`mousemove`,** wenn die Maus sich bewegt

**`mousedown`,** wenn die linke Maustaste gedrückt

**`mouseup`,** wenn die linke Maustaste losgelassen

```
function _mouse_down_handler () {
    /*...*/
}
var example = $("#test-elem");

example.bind('mousedown', {"beispiel": "daten"},
 ↪  _mouse_down_handler);
example.unbind('mousedown', _mouse_down_handler);
```

Listing 4.1.: Beispielhaftes *bind* und *unbind* eines Event-Listeners.

Für die einzelnen Event-Listener wurden teilweise globale Variablen verwendet, um Daten zwischen mehreren Event-Listenern auszutauschen. Alternativ könnte auch einen privaten Kontext genutzt werden. So müssen bspw. beim Event-Listener zum Verschieben der *figure* die Startkoordinaten beim Drücken der Maustaste gesichert werden und beim Loslassen die Endkoordinaten um die Verschiebung der *figure* bestimmen zu können.

Im Folgenden werden die Event-Listener des JSTerm aufgelistet. Im Anschluss werden diese kurz erklärt. Die Event-Listener des JSTerms sind alle auf dem oberen Canvas registriert, welches für die Verarbeitung der Interaktionen auf Seiten des JSTerms zuständig ist. Neben einer Methode zum Zurücksetzen aller Event-Listener gibt es noch weitere Event-Listener für:

- die Legende
- den Box-Zoom (ROI)
- die Mauspositionsanzeige
- das Verschieben der *figure*
- die Datenverfolgung
- den Zoom mit dem Mausrad

**Legende:** Die Legende ist mithilfe von mehreren Buttons, die untereinander gruppiert sind, realisiert. Wenn auf einen Button geklickt wird, so wird dieser grau hinterlegt und der entsprechende *plot* ausgeblendet.

**Box-Zoom (ROI):** Für den Box-Zoom werden drei Event-Listener benötigt, nämlich für die Events `mousedown`, `mousemove` und `mouseup`. Beim Drücken der linken Maustaste wird die aktuelle Position innerhalb des Canvas-Objektes gespeichert und eine Variable auf den *Boolean*-Wert `true` gesetzt, die anzeigt, ob sich die Maus innerhalb des Canvas-Objekt bewegt und für die anderen beiden Event-Listener zur

Validierung benötigt wird. Der `mousemove`-Listener zeichnet, während sich die Maus bewegt, die Box, auf die die *figure* skaliert werden soll. Wie in den Grundlagen im Unterabschnitt 2.6 angeführt, werden hierfür zwingend zwei Canvas-Objekte pro *figure* benötigt. Im `mouseup`-Listener wird wiederum die aktuelle Mausposition gespeichert. Danach kann mit den Start- und Endkoordinaten entsprechend der gezoomte Bereich mit der `gr.setwindow` Methode angepasst werden.

**Mauspositionsanzeige:** Hierfür ist nur der Event-Listener für die Mausbewegung nötig. Es wird die aktuelle Position in das Koordinatensystem der *figure* umgerechnet und entsprechend angezeigt.

**Verschieben der *figure*:** Beim Verschieben der *figure* gibt es Event-Listener zum Drücken und Loslassen der linken Maustaste. Beim Drücken werden die Startkoordinaten gesichert, beim Loslassen werden entsprechend die Endkoordinaten gesichert. Die sich daraus ergebende Verschiebung wird dann von den aktuellen Window-Grenzen abgezogen.

**Datenverfolgung:** Für die Datenverfolgung wird nur das `mousemove` Event benötigt. Es wird von der aktuellen Mausposition für jedes *plot*-Objekt geprüft, welcher Datenpunkt den geringsten euklidischen Abstand zur Mausposition hat. Von diesem Datenpunkt wird der nächst kleinere sowie größere Datenpunkt überprüft. Zwischen diesen beiden Datenpunkten wird linear interpoliert, um eine stetige Verfolgung zu ermöglichen. Ein Label mit den Koordinaten dieses Datenpunktes wird an das *plot*-Objekt gezeichnet, der den kleinsten euklidischen Abstand zur aktuellen Mausposition hat.

**Zoom mit dem Mausrad:** Hierbei wird das Event `wheel` benötigt, das beim Drehen des Mausscrollrades ausgelöst wird. Es wird die Richtung (heraus- oder hineinzoomen) und die aktuelle Mausposition verarbeitet. Mit dieser Mausposition kann während des Scrollens beim Reinzoomen das Window entsprechend der Mausposition angepasst werden.

### 4.2.2. Speicherung der empfangenen Objekte im JSTerm

Der folgende Abschnitt erläutert, wie die *figure*, *plot* und *transformation*-Objekte intern gespeichert werden. Die empfangenen Objekte werden anders gespeichert, als mit dem Protokoll übertragen (s. Protokoll der Kommunikation in Abschnitt 3.3), da bspw. die optionalen Argumente mit Standardwerten beschrieben werden. Weiterhin werden die *figure*-Objekte so gespeichert, dass sie als Argument ein Array mit den zu der *figure* zugehörigen *plot*- und *transformation*-Objekten haben.

#### 4.2.2.1. Interne Speicherung einer *figure*

In diesem Abschnitt wird die interne Speicherung von *figure*-Objekten beschrieben. In der folgenden Auflistung sind die möglichen Werte aufgeführt und erläutert. Die *figure*-Objekte besitzen dabei intern keine `content` oder `meta` Attribute mehr, um eine möglichst flache Hierarchie zu erreichen.

**`figure_id:`** Das ist die ID des *figure*-Objekts vom Datentyp *Number*. Aus ihr ergeben sich die HTML-IDs der Canvas-Objekte. Diese IDs sind `"canvas_<figure_id>"` für das GR Canvas und `"canvas_<figure_id>_overlay"` für das Canvas mit den Interaktionen bzw. Events des JSTerms.

**`background_color:`** Dies ist ein Array von drei *Number*-Werten [r,g,b]. Diese entsprechen dem Rot-, Grün- und Blauanteil und es gilt $r, g, b \in [0, 1]$ für die Werte.

**`size:`** Als Wert wird hier ein Array von zwei *Number*-Werten übergeben, welche die Größe der Canvas-Objekte bestimmen. Die Angabe erfolgt in Pixeln auf dem Bildschirm.

**`transformation:`** Hiermit wird die ID eines *transformation*-Objektes übergeben. Der Wert ist vom Datentyp *Number*. Wenn dieser Wert gesetzt ist, so gilt die *transformation* für die gesamte *figure*, sonst gelten die Standardwerte des GR-Frameworks.

**`show:`** Zeigt an, ob die aktuelle *figure* gezeichnet werden soll oder nicht. Der Datentyp ist *Boolean*.

**`title:`** Hiermit wird ein *String* mit einem Titel, welcher angezeigt werden soll, übergeben.

**`x_org:`** Dies ist eine *Number* mit der x-Koordinate des Koordinatenursprungs.

**`y_org:`** Dies ist eine *Number* mit der y-Koordinate des Koordinatenursprungs.

**`x_tick:`** Mit dieser *Number* wird der Abstand der Markierungen (Ticks) auf der x-Achse angegeben.

**`y_tick:`** Mit dieser *Number* wird der Abstand der Markierungen (Ticks) auf der y-Achse angegeben.

**`major_x:`** Mit dieser *Number* wird angezeigt, jeder wievielter x-Tick ein großer x-Tick ist. Dabei sind große Ticks doppelt so lang wie kleine Ticks. Der Wert 2 bedeutet bspw., dass jeder zweite Tick größer gezeichnet wird.

**`major_y:`** Mit dieser *Number* wird angezeigt, jeder wievielter y-Tick ein großer y-Tick ist.

**keep_org:** Falls dieser Wert auf den *Boolean*-Wert *true* gesetzt ist, so ist der Schnittpunkt der Koordinatenachsen das Minimum des ersten empfangen *plots*. Wenn dieser Wert *false* ist, so wird dieser Schnittpunkt der Koordinatenachsen bei jedem empfangenen *plot* angepasst.

**axes_and_grid_drawn:** Mithilfe dieses *Boolean*-Wertes wird intern verarbeitet, ob die Achsen und das Gitter schon gezeichnet wurden.

**objects:** In einem Array sind hier alle weiteren Objekte also *transformations* und *plots* gespeichert, die der *figure* zugeordnet sind.

#### 4.2.2.2. Interne Speicherung eines *plots*

Dieser Abschnitt erläutert die Speicherung des *plot*-Objektes intern. Für die Argumente, die optional sind, werden hier Standardwerte gesetzt, wie bspw. für `content.transformation`.

**id:** Dies ist die ID des Objekts vom Typ *Number*.

**object_type:** In einer Zeichenkette steht der Typ des Objekts, also in diesem Fall `"plot"`. Damit kann innerhalb des JSTerm abgefragt werden, ob ein Objekt ein *plot* oder eine *transformation* ist, da alle Objekte in demselben Array der *figure* gesichert werden.

**content.label_coords:** Dies sind die Koordinaten als Array von vier *Number*-Werten des Labels aus der Legende.

**content.label_is_visible:** Mit dieser Variable vom Typ *Boolean* wird angezeigt, ob das *plot*-Objekt sichtbar ist oder nicht.

**content.kind:** Mit dem `kind` wird bestimmt, um welche Darstellungsart *plot* es sich handelt. Möglich sind bspw. `line`, `contour` oder `scatter`.

**content.data:** Die Daten werden aus den empfangenen Daten übernommen. Sie sind dabei von der jeweiligen Darstellungsart abhängig (s. `content.kind`). Die Verifizierung der Daten wird in der Methode zu dem jeweiligen `content.kind` innerhalb des JSTerms durchgeführt.

**content.transformation:** Damit wird die ID der zugehörigen *transformation* übergeben. Entweder gilt dieses für einen *plot* oder für die gesamte *figure*. Dabei wird vorrangig die *transformation* eines *plots* verwendet, bevor die der gesamten *figure* verwendet wird.

**meta:** Die Metadaten werden aus den empfangenen Daten übernommen.

#### 4.2.2.3. Interne Speicherung einer *transformation*

Das *transformation*-Objekt wird im Wesentlichen im Server so gespeichert, wie es auch empfangen wurde, also mit den `meta`- und `content`-Daten. Zusätzlich gibt es nur die ID und den Typ des Objektes vgl. *plot*.

`id:` Dies ist die ID des Objekts vom Typ *Number*.

`object_type:` In einer Zeichenkette steht der Typ des Objektes, also in diesem Fall `"transformation"`.

### 4.2.3. Implementierung mithilfe von Jupyter-Notebooks

Dieser Unterabschnitt beschreibt die Implementierung des JSTerms in der Jupyter-Variante. In Listing A.2 (s. Anhang) ist die konkrete Realisierung des RequireJS-Moduls, das für die Ausführung mit Jupyter benötigt wird, aufgeführt. Im Fall der Jupyter-Variante musste nur die JavaScript-Datei angepasst werden. In der Grafik 4.3 ist ein Flussdiagramm der Jupyter-Variante aufgeührt.

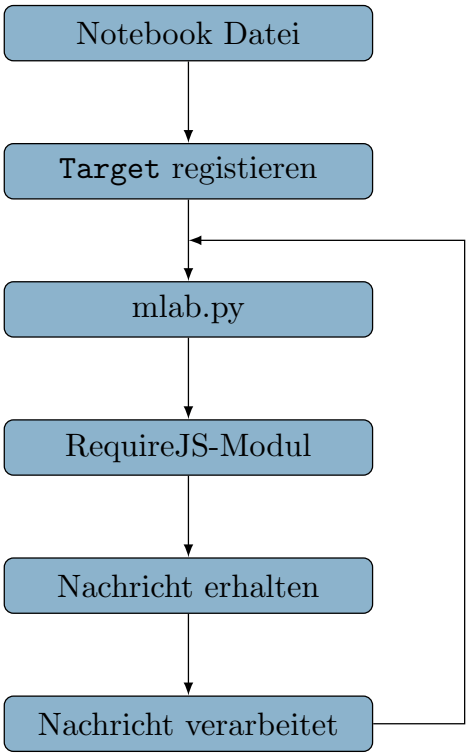



Abbildung 4.3.: Flussdiagramm des JSTerms in der Jupyter-Variante. Das Registrieren des `Targets` muss nur einmal erfolgen, nachdem der Jupyter-Kernel geladen ist.

Als erstes wird geprüft, ob die `define`-Methode des RequireJS zur Verfügung steht, um die Notebook-Extension mit dieser zu definieren. Diese Abfrage ist nötig, da die standalone Variante dieselbe Codebasis nutzt. In dieser gäbe es aber durch die unbekannte Methode beim Laden der JSTerm-JavaScript-Datei Fehlermeldungen. Dabei wird innerhalb der `define`-Methode nur das Jupyter spezifische implementiert. Die eigentliche JSTerm-Funktionalität liegt außerhalb, damit die standalone Variante diese auch nutzen kann. Das erste Argument der `define`-Methode ist eine Liste von Zeichenketten, welche die zu ladenden Abhängigkeiten darstellen. Das zweite Argument ist eine anonyme Funktion, welche die Realisierung der Notebook-Extension darstellt.

In der `_on_load`-Methode des Moduls wird zuerst die JSTerm-Methode zum Laden der weiteren Skripte aufgerufen. Damit wird z. B. Bootstrap, Font-Awesome und das GR.js geladen. Da dies u. a. mithilfe der `$.getScript`-Methode von jQuery durchgeführt wird, ist es sinnvoll, dies möglichst früh zu behandeln, da diese Methode die Skripte asynchron lädt. Im Anschluss wird ein Event-Listener für das `kernel_ready.Kernel`-Event registriert. Dieses Event wird ausgelöst, wenn der Kernel innerhalb der Jupyter-Umgebung geladen ist. In dem entsprechenden Event-Listener wird als Attribut des Kernels der `CommManager` initialisiert und mit der `register_target`-Methode ein sog. `Target` registriert. Dieses `Target` ist eine Zeichenkette und dient zur Identifizierung der Verbindung, über die beide `CommManager` im JSTerm und im mlab.py kommunizieren können. Das `Target` zur Identifizierung ist somit vergleichbar mit der WebSocket-URL in der standalone Variante. Diese Zeichenkette ist das erste Argument, das zweite Argument ist eine JavaScript-Funktion die aufgerufen werden soll, wenn diese Comm-Verbindung geöffnet wurde (`_handle_comm_opened`).

In der `_handle_comm_opened`-Methode wird für das `on_msg`-Event eine Methode definiert, die für jede Comm-Nachricht (CommMessage) aufgerufen wird. Innerhalb dieser werden die Daten mithilfe von `JSON.parse` zu JavaScript-Objekten interpretiert, da die Nachrichten als Zeichenkette empfangen werden. Danach werden die `content`- und `meta`-Daten an die Funktion `jsterm_on_message` übergeben, die die weitere Verarbeitung der Daten übernimmt.

### 4.2.4. Implementierung als standalone Anwendung

Die Implementierung der standalone Variante wird im folgenden Unterabschnitt beschrieben. Es wird hierbei auf den Tornado-Server und den WebSocket-Client eingegangen. Im JSTerm gibt es hier keinen spezifischen Code (vgl. Jupyter-Variante mit RequireJS-Modul), da der WebSocket-Client seinerseits direkt die `jsterm_on_message`-Methode aufruft. Der gesamte Ablauf der standalone Variante des JSTerms ist im Flussdiagramm 4.4 dargestellt.

Das Listing A.3 (s. Anhang) zeigt, wie der Tornado-Server implementiert ist. Dieser Server fungiert als Proxy, da er die Daten einfach mittels TCP-Sockets vom mlab.py empfängt und über WebSockets an das JSTerm schickt. Das `End-Of-Transmission-Block`-Zeichen wird im Proxy-Server entfernt. Technisch besteht der Proxy-Server aus dem TCP-Server und einem HTTP- / WebSocket-Server. Die Python-Klasse

`ProxyWebServer` dient zum Ausliefern des WebSocket-Clients. Es gibt innerhalb dieser Klasse dementsprechend nur die Python-Methode, die auf die `HTTP-GET`-Methode die entsprechende HTML-Seite mit dem WebSocket-Client ausliefert. Die CSS- und JavaScript-Dateien befinden sich in den Standardverzeichnissen (`static/js` und `static/css`), sodass es genügt, die Option `static_path="static"` für den HTTP-Server (s. Listing A.3 im Anhang) zu setzen.

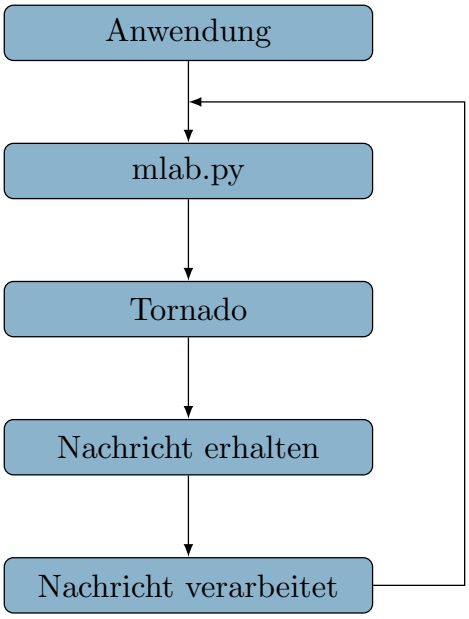



Abbildung 4.4.: Flussdiagramm des JSTerms in der standalone Variante.

Die Klasse `ProxyWebSocket` beinhaltet die Realisierung des WebSocket-Servers. Alle aktiven WebSocket-Verbindungen sind in dem Proxy-Server in einem Python-`set` gesichert (`websocket_connections`), um die Eindeutigkeit dieser Verbindungen sicher zu stellen. Beim Öffnen einer WebSocket-Verbindung wird die entsprechende Instanz in diese Menge eingetragen, beim Schließen entsprechend gelöscht. Die `on_message`-Methode ist im Server nicht realisiert, da diese Methode beim Empfang einer Nachricht vom WebSocket-Client zum WebSocket-Server aufgerufen wird. Es soll aber nur unidirektional vom Proxy-Server zum Client – also vom mlab.py über den Proxy-Server zum JSTerm – kommuniziert werden.

Die Verarbeitung der Datenpakete, die das mlab.py sendet, ist in der Klasse `ProxyTCPServer` realisiert. Es wird solange ein Datenpaket empfangen, bis das `ETB`-Zeichen gelesen wird, welches danach aus den Daten gelöscht wird. Die Daten werden als Zeichenkette übertragen und erst im WebSocket-Client als JavaScript-Objekte interpretiert. Es wird über die Menge der WebSocket-Verbindungen iteriert und an jede das Datenpaket mittels der `write_message`-Methode gesendet. Der Port, auf dem die Daten empfangen werden, wird durch die Umgebungs-Variable `GR_META_TARGET` definiert, die in mlab.py und diesem Server verwendet wird.

Der WebSocket-Client ist im Listing A.4 aufgeführt. Im `<head>` der Seite werden jQuery und das JSTerm geladen. Mittels zwei JSTerm-Methoden werden dann die Variablen des JSTerms initialisiert (bspw. die Liste mit allen verfügbaren *figure*-Objekten) und die weiteren Skripte (wie auch in der Jupyter-Variante) nachgeladen. Die WebSocket-Event-Listener zum Öffnen, Schließen und Empfang einer Nachricht werden entsprechend mit Ausgaben für die Entwicklerkonsole im `debug`-Modus realisiert. In der Methode, die beim Empfang eines Datenpaketes aufgerufen wird, werden die Daten als JavaScript-Objekte interpretiert und an das JSTerm über die `jsterm_on_message`-Methode übergeben.

### 4.2.5. Besondere JSTerm-Methoden

In diesem Abschnitt werden besondere Methoden des JSTerms näher beschrieben. In einer dieser Methoden, `jsterm_clearws`, wird das eigentliche `gr.clearws` durchgeführt, um die Workstation auf den Ursprungszustand zurückzusetzen. Weiterhin wird die Variable, die anzeigt, ob die Achsen und das Gitter gezeichnet sind, auf `false` gesetzt, damit beim nächsten Aufruf einer *plot*-Methode die Achsen und das Gitter gezeichnet werden. Damit ist gewährleistet, dass die Achsen und Gitterlinien nur einmal pro *figure* gezeichnet werden.

Die Methoden `jsterm_reservecontext`, `jsterm_selectcontext` und `jsterm_destroycontext` dienen zum Verwalten des GR-Kontextes. Die erste Methode legt den Kontext an, die zweite wird immer aufgerufen, wenn ein Kontext aktiv geschaltet werden soll, und die dritte dient zum Freigeben des Speichers für einen Kontext.

Die Event-Listener für die Maus-Events besitzen alle einen Namen, der mit einem Unterstrich beginnt, wie bspw. `_wheel_zoom`. Damit soll angezeigt werden, dass diese Methoden nicht wie gewöhnliche JavaScript-Methoden unmittelbar aufgerufen werden sollen, sondern nur bspw. durch die Methoden welche die Interaktionen des Nutzers verarbeiten. Diese Methoden, z. B. Zoom, beginnen alle mit `ctrl_` vor dem Namen der Methode.

## 4.3. Einrichten des JSTerms

Dieser Abschnitt beschreibt jeweils, wie beide Varianten des JSTerms eingerichtet und genutzt werden können. Das `mlab.py`-Modul erkennt automatisch, ob es in einer Jupyter-Umgebung ausgeführt wird oder nicht. Abhängig davon werden die Daten entweder über den `CommManager` oder TCP-Sockets versendet.

### 4.3.1. Einrichten als Jupyter-Notebook-Extension

Dieser Unterabschnitt beschreibt das Einrichten des JSTerms in der Jupyter-Variante. Hierfür ist – neben der Jupyter- und IPython-Software – keine weitere Software nötig, wie bpsw. Tornado in der standalone Variante. Nach dem Klonen des Repositories muss das GR.js in das Verzeichnis des JSTerms verlinkt werden. Dazu wird in dem

Unterorder `src/server/static/js/` ein Softlink erstellt. Danach muss das JSTerm selbst als sogenannte Notebook-Extension eingerichtet werden. Um das JSTerm für einen bestimmten Benutzer einzurichten, muss (auf unixoiden-Systemen) in den Ordner `~/.ipython/nbextensions/` gewechselt werden. Der Ordner für systemweit installierte Notebook Extensions ist bspw. `/usr/local/share/jupyter/nbextensions/-jupyter-js-widgets/`. Falls der Ordner nicht im `home`-Ordner des Nutzers existiert, muss mindestens einmal mit dem `ipython` Befehl (mit der Shell) der IPython-Interpreter gestartet werden. In dem Ordner muss das JSTerm mittels eines Softlinks oder einer Kopie verfügbar gemacht werden. Es wird mit dem Befehl `jupyter nbextension enable jsterm` aktiviert. Wichtig ist, dass hier der Name des Softlinks zum JSTerm ohne die Dateiendung `.js` angegeben wird. Danach kann ein Jupyter-Notebook gestartet werden. Im Unterordner `test/` befindet sich ein beispielhaftes Notebook, welches mit `jupyter notebook Tests.ipynb` gestartet werden kann. Das Listing 4.2 führt alle Kommandos nochmals zusammenhängend auf.

```
git clone git@ifflinux:f.macherey@fz-juelich.de/jsterm
cd ~/.ipython/nbextensions/
ln -s ~/pfad/zum/jsterm.js jsterm.js
jupyter nbextension enable jsterm

cd ~/beispiel-projekt/
jupyter notebook Beispiel.ipynb
```

Listing 4.2.: Einrichten des JSTerms als Jupyter-Notebook-Extension

### 4.3.2. Einrichten der standalone Variante

Dieser Unterabschnitt beschreibt das Einrichten des JSTerms in der standalone Variante. An der MATLAB-Seite muss nichts angepasst werden, da das mlab.py automatisch die richtige Variante des Sendens auswählt. Für den Tornado-Server muss das entsprechende Modul installiert sein. Es empfiehlt sich die Nutzung einer Python-`virtualenv`, damit der Server unabhängig von anderen Python-Paketen auf dem System ist. Durch die Verwendung einer `virtualenv` sind die Pakete und damit Versionen unabhängig von anderen Paketen auf dem System, sodass es keine Probleme mit Abhängigkeiten und verschiedenen Versionen gibt [Bic07]. Das Listing 4.3 zeigt, wie die `virtualenv` eingerichtet wird und wie das entsprechende Paket über `pip` installiert wird. Die Schritte, die zur Kommunikation ausgeführt werden müssen, sind dann:

1. Start des Servers (WebSocket, HTTP und TCP)
2. Aufbau der WebSocket-Verbindung durch den Client
3. Aufbau der TCP-Verbindung → Ausführen des mlab.py-Skripts

Dabei geschieht das Starten des Servers automatisch durch mlab.py. Der Aufbau der WebSocket-Verbindung wird durch das WebSocket-Modul und das JSTerm in der standalone Variante automatisch ausgeführt. Der letzte Schritt ist der Beginn des Sendens eines Datenpakets, im Fall des JSTerms einer *figure*, eines *plots* oder einer *transformation*. Dieser Sendevorgang verläuft so, wie es in Abbildung 3.3 bereits anschaulich dargestellt wurde.

Ein beispielhafter WebSocket-Client ist im Listing A.4 dargestellt. Dabei ist der gezeigte Client minimal. Es können je nach Anwendungsfall noch weitere Funktionen und HTML-Elemente eingefügt werden. Nachdem die Seite geladen ist, werden zwei JSTerm-Methoden aufgerufen. Eine Methode zum Anlegen der globalen Variablen, eine zweite zum Nachladen weiterer CSS- und JavaScript-Dateien, wie beispielsweise Bootstrap, Font-Awesome und das GR.js. In dem JavaScript-Callback zum Empfang einer Nachricht (`onmessage`) werden die erhaltenen Daten als JavaScript-Objekt interpretiert und im Anschluss zur weiteren Verarbeitung an das JSTerm geschickt.

```
pyvenv .venv
source .venv/bin/activate
pip install tornado
```

Listing 4.3.: Einrichten einer `virtualenv` und Installation von Tornado [DYW16].

# 5. Zusammenfassung

Das Ergebnis dieser Arbeit ist eine Webanwendung zur Darstellung von 2D-Grafiken im Browser. Die Webanwendung ist mit JavaScript realisiert und wird clientseitig ausgeführt. Zur Kommunikation mit Anwendungsschichten (mlab.py) wurde ein auf JSON basierendes Protokoll entwickelt. Dieses Protokoll ist hierarchisch aufgebaut. Eine Grafik wird durch eine *figure* abstrahiert. In dieser gibt es verschiedene *plot*-Objekte zur Darstellung von bspw. Linienplots und *transformation*-Objekte, welche ein *plot*-Objekt oder die gesamte *figure* modifizieren. Innerhalb des JSTerms werden die Grafiken mittels GR.js gezeichnet. Durch das neue Protokoll und die Speicherung der Objekte innerhalb des JSTerms bleibt die Semantik in den Daten vorhanden, sodass mit den Grafiken interagiert werden kann.

Das entwickelte JSTerm kann entweder als Notebook-Extension innerhalb einer Jupyter-Umgebung ausgeführt werden oder in der standalone Variante. Im zweiten Fall wird die Kommunikation über einen mit Tornado realisierten Proxy-Server durchgeführt.

Die Daten aus den physikalischen Experimenten können mithilfe des mlab.py ausgewertet werden. Die Darstellung erfolgt in der neuen Schicht des JSTerms durch JavaScript clientseitig im Browser des Benutzers.

JSTerm ermöglicht es mit den Grafiken zu interagieren. Es können bspw. einzelne *plots* über die Legende ein- und ausgeblendet werden, es kann mit dem Mausrad oder mit den entsprechenden Interaktionsmöglichkeiten gezoomt werden und die Grafik kann verschoben werden. Weiterhin ist es möglich die *figure* als PNG herunter zu laden. Die Datenverfolgung ermöglicht es, sich innerhalb der *figure* auf dem *plot* mit dem geringsten euklidischen Abstand zu der Mausposition stetig zu bewegen und damit Datenpunkte auszulesen.

# 6. Ausblick

Parallel zur Entwicklung des JSTerms wird im PGI/JCNS das sogenannte QtTerm entwickelt. Dieses soll die aktuelle Desktop-Standardanzeige ablösen. Zur Kommunikation wird auch hier das neue Protokoll genutzt, welches im Rahmen der vorliegenden Arbeit entwickelt wurde. Die Implementierungen des JSTerms und QtTerms sollen einfach miteinander austauschbar sein. Durch das JSTerm werden momentan Linienplots vollständig unterstützt. Im Rahmen einer aktuellen Seminararbeit am PGI/JCNS werden die weiteren Methoden des mlab.py, wie bspw. Streudiagramme und Histogramme, in das JSTerm eingepflegt, um so vollständig kompatibel mit mlab.py zu sein.

Weiterhin kann in Zukunft die Download-Funktion so angepasst werden, dass höher auflösende Grafiken herunterladbar sind. Aktuell werden beim Download lediglich Rastergrafiken als PNG unterstützt. Hier können in Zukunft auch andere bzw. vektorbasierte Grafikformate (SVG, PDF) unterstützt werden.

Momentan wird im Fall der Jupyter-Variante des JSTerms auch über diese Infrastruktur kommuniziert (`CommManager` statt Tornado und WebSocket-Client). Falls es in Zukunft bei größeren Datenmengen zu Performanz-Problemen mit dem Jupyter-Server kommt, so kann generell auf die Kommunikation mittels eines Proxy-Servers umgestellt werden.

Außerdem kann das Verschieben einer *figure* verbessert werden, sodass dies direkt beim Bewegen der Maus geschieht. In Zukunft auch das Einstellen der Eigenschaften und Attribute eines *plots* realisiert werden.

Die Methoden zum Registrieren und Löschen der Event-Listener (`bind` und `unbind`) sollen laut der offiziellen Dokumentation nicht mehr verwendet werden. An deren Stelle sollte der JSTerm-Code dahingehend überarbeitet werden, dass die neuen Varianten `on` und `off` verwendet werden.

Die Legende ist aktuell über mehrere Buttons (pro *plot* einer) realisiert. In Zukunft soll die Legende angepasst werden. Ihr Aussehen soll sich an der des mlab.py-Moduls orientieren. Es soll mit einem Klick auf ein Textlabel möglich sein, das zugehörige *plot*-Objekt ein- bzw. auszublenden. Weiterhin kann man im im mlab.py auch das Label der *plots* setzten. Diese Möglichkeit sollte auch durch das JSTerm gegeben sein. Hierfür muss dann ggf. das Protokoll erweitert werden.

# Literatur


[BB86] Jörg Bechlars und Rainer Buhtz. *GKS in der Praxis*. Berlin, Heidelberg, New York, Tokyo: Springer-Verlag, 1986. ISBN: 3-540-16139-2.

[Bic07] Ian Bicking. *virtualenv Python*. en. 2007. URL: `https://virtualenv.pypa.io/en/stable/` (besucht am 18. 11. 2016).

[Bre16] Thomas Breloff. *Plots in julia*. en. 2016. URL: `https://juliaplots.github.io/attributes/#plot` (besucht am 25. 07. 2016).

[Chu11] Andy Chung. *Require JS*. en. 2011. URL: `http://requirejs.org/` (besucht am 19. 10. 2016).

[Cor14] iMatix Corporation. *ZeroMQ offizielle Webseite*. en. 2014. URL: `http://zeromq.org/` (besucht am 19. 10. 2016).

[Dar16] Ben Darnell. *Tornado User's Guide*. en. 2016. URL: `http://www.tornadoweb.org/en/stable/guide/intro.html` (besucht am 17. 10. 2016).

[Dev16] Alexis Deveria. *Can I use "Websockets"?* en. Sep. 2016. URL: `http://caniuse.com/#feat=websockets` (besucht am 18. 10. 2016).

[Dro15] Steffen Drossard. „Integration des GR Frameworks in einen Browserkontext“. de. Diss. FH Aachen – University of Applied Sciences, 2015. URL: `https://pgi-jcns.fz-juelich.de/pub/doc/Bachelorarbeit_SteffenDrossard.pdf` (besucht am 23. 05. 2016).

[DS16] Alexis Deveria und Lennart Schoors. *Can I use "Canvas"?* en. 2016. URL: `http://caniuse.com/#search=canvas` (besucht am 14. 11. 2016).

[DYW16] Ben Darnell, Felix Yan und Greg Ward. *Tornado Install Guide*. en. 2016. URL: `http://www.tornadoweb.org/en/stable/#installation` (besucht am 10. 11. 2016).

[Enc94] José Luís Encarnação. *Graphical Kernel System (GKS)*. en. Techn. Ber. Okt. 1994, S. 164. URL: `http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=14915` (besucht am 05. 09. 2016).

[FM11] Ian Fette und Alexey Melnikov. *The WebSocket Protocol – RFC 6455*. en. Dez. 2011. URL: `https://tools.ietf.org/html/rfc6455` (besucht am 15. 07. 2016).

[Gra+16] Brian Granger u. a. *Jupyter offizielle Webseite*. en. Okt. 2016. URL: `https://jupyter.org/` (besucht am 18. 10. 2016).

[Hei16a] Josef Heinen. *GR offizielle Dokumentation*. en. 2016. URL: http://gr-framework.org/gr.html (besucht am 28. 09. 2016).

[Hei16b] Josef Heinen. *GR offizielles Repository*. en. Okt. 2016. URL: https://github.com/jheinen/gr (besucht am 17. 10. 2016).

[Hic] Ian Hicksen. *4.12.5 The canvas element*. en. URL: https://html.spec.whatwg.org/multipage/scripting.html#the-canvas-element (besucht am 06. 10. 2016).

[Hic+14] Ian Hicksen u. a. *WebSockets WHATWG*. en. 2014. URL: https://html.spec.whatwg.org/#network (besucht am 15. 07. 2016).

[Int16] Ecma International. *ECMAScript® 2016 Language Specification*. en. 2016. URL: http://www.ecma-international.org/ecma-262/7.0/index.html#sec-copyright-and-software-license (besucht am 10. 11. 2016).

[Mat16] The MathWorks. *MATLAB Dokumentation zu figure und plots*. en. 2016. URL: http://de.mathworks.com/help/matlab/creating_plots/using-high-level-plotting-functions.html (besucht am 28. 09. 2016).

[Mil+15] Chris Mills u. a. *Emscripten Kurzzusammenfassung MDN*. en. Dez. 2015. URL: https://developer.mozilla.org/en-US/docs/Mozilla/Projects/Emscripten (besucht am 17. 10. 2016).

[Net15] Mozilla Developer Network. *WebSocket Object*. en. 2015. URL: https://developer.mozilla.org/en-US/docs/Web/API/WebSocket (besucht am 13. 07. 2016).

[Net16] Mozilla Developer Network. *Document Object Model*. en. Okt. 2016. URL: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model (besucht am 17. 11. 2016).

[PG07] Fernando Pérez und Brian Granger. „IPython: A System for Interactive Scientific Computing“. en. In: *Computing in Science and Engineering*. 3. Ser. 9 (Juni 2007), S. 21–29. DOI: doi:10.1109/MCSE.2007.53. URL: http://ipython.org/ (besucht am 18. 10. 2016).

[Sch+16] Florian Scholz u. a. *AJAX Erklärung MDN*. en. Okt. 2016. URL: https://developer.mozilla.org/de/docs/AJAX (besucht am 27. 10. 2016).

[Sch97] Patrick Schnabel. *NAT - Elektronik Kompendium*. de. 1997. URL: http://www.elektronik-kompendium.de/sites/net/0812111.htm (besucht am 18. 11. 2016).

# Anhang

# A. Sourcecode-Listings

## A.1. Beispielhafter WebSocket-Client

```
<!DOCTYPE html>
<html>
  <head>
    <title>websocket minimal example</title>
    <meta charset="UTF-8">
    <script>
      window.onload = function() {
        var ws = new WebSocket("ws://localhost:8080/websocket/");

        ws.onopen = function(e) {
          console.log("websocket opened");
        }
        ws.onclose = function(e) {
          console.log("websocket closed");
        }
        ws.onmessage = function(e) {
          console.log("got data: \n", e.data);
        }
      };
    </script>
  </head>
    <body>
      <h1>Welcome to jsterm standalone</h1>
    </body>
</html>
```

Dieser beispielhafte Websocket-Client stammt von der folgenden Webseite und wurde in Teilen angepasst `https://www.tutorialspoint.com/html5/html5_websocket.htm` (letzter Zugriff *10. November 2016*).

## A.2. Implementierung des RequireJS-Moduls in der Jupyter-Variante

```
if (typeof define !== "undefined") {
    define(['base/js/namespace', 'services/kernels/kernel',
     ↪ 'services/kernels/comm', 'base/js/events'],
     ↪ function(IPython, Kernel, CommManager, events) {
        function _handle_comm_opened(comm) {
            comm.on_msg(function(msg) {
                if (msg.msg_type !== "comm_msg") {return;}

                var msg_data = JSON.parse(msg.content.data);
                var content = msg_data.content;
                var meta = msg_data.meta;

                jsterm_on_message(content, meta)
            });
        }

        function _on_load() {
            jsterm_load_external_scripts();
            events.on('kernel_ready.Kernel', function(event, data) {
                kernel = IPython.notebook.kernel;
                if (!kernel) {return;}

                comm_manager = kernel.comm_manager;
                comm_manager.register_target('jupyter.jsterm',
                 ↪ _handle_comm_opened);
            });
        }
        return {load_ipython_extension: _on_load};
    });
}
```

## A.3. Implementierung des Tornado-Servers in der standalone Variante

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
from __future__ import unicode_literals

import tornado.websocket
import tornado.tcpserver
import tornado.gen
import tornado.iostream
import tornado.web

import time
import logging
import os

TCP_PORT = 48100 # default value
WEB_PORT = 48000
websocket_connections = set()
time_on_startup = 0


class ProxyTCPSocket(tornado.tcpserver.TCPServer):
    @tornado.gen.coroutine
    def handle_stream(self, stream, address):
        while True:
            data = ''
            try:
                data = yield
                 ↪  stream.read_until(chr(23).encode('utf-8'))
            except(tornado.iostream.StreamClosedError,
             ↪  AssertionError):
                print("[{:10.4f}] TCP connection
                 ↪  closed...".format(time.time() - time_on_startup))
                break
            data = data.replace(chr(23).encode('utf-8'),
             ↪  ''.encode('utf-8'))
```

```
            for conn in websocket_connections:
                conn.write_message(data)


class ProxyWebSocket(tornado.websocket.WebSocketHandler):
    def open(self):
        print("[{:10.4f}] WebSocket opened".format(time.time() -
         ↪ time_on_startup))
        websocket_connections.add(self)


    def on_message(self, message):
        # channel for incomming messages client -> server, not
         ↪ needed at the moment
        #self.write_message(u"You said: " + message)
        pass

    def on_close(self):
        print("[{:10.4f}] WebSocket closed".format(time.time() -
         ↪ time_on_startup))
        websocket_connections.remove(self)

    def check_origin(self, origin):
        return True

class ProxyWebServer(tornado.web.RequestHandler):
    @tornado.web.asynchronous
    def get(self):
        self.render("index.html")


if __name__ == "__main__":
    if 'GR_META_TARGET' in os.environ:
        gr_target = os.environ['GR_META_TARGET']
        TCP_PORT = int(gr_target.split(':')[-1])

    time_on_startup = time.time()
    logging.getLogger('tornado.access').disabled = True

    # server for web socket communication to jsterm
    web_server = tornado.web.Application([
        (r"/websocket/", ProxyWebSocket),
        (r"/", ProxyWebServer),
```

```
    ], static_path="static")
    web_server.listen(WEB_PORT)

    # server to read data from metasend sender
80  tcp_server = ProxyTCPSocket()
    tcp_server.listen(TCP_PORT)

    print("jsterm-Proxyserver running ... listen for incomming on
     ↪  :{} and send to :{}".format(TCP_PORT, WEB_PORT))
    tornado.ioloop.IOLoop.current().start()
```

## A.4. WebSocket-Client in der standalone Variante

```
<!DOCTYPE html>
<html>
  <head>
    <title>jsterm standalone</title>
    <meta charset="UTF-8">
    <script src="./static/js/jquery.js"></script>
    <script src="./static/js/jsterm.js"></script>
    <script>
      window.onload = function() {
        jsterm_init_globals();
        jsterm_load_external_scripts();

        var ws = new WebSocket("ws://localhost:48000/websocket/");
        ws.onopen = function(e) {
          console.log("jsterm-proxy: websocket opened");
        }
        ws.onclose = function(e) {
          console.log("jsterm-proxy: websocket closed");
        }
        ws.onmessage = function(e) {
          var data = JSON.parse(e.data)
          console.log("jsterm-proxy: got data: \n", data);
          jsterm_on_message(data.content, data.meta);
        }
      };
    </script>
  </head>
    <body>
      <h1 id="standalone-h1">Welcome to jsterm standalone</h1>
    </body>
</html>
```