Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 239

Visualisierungspipeline für Simulationsworkflows auf Basis eines Datenstromverarbeitungssystems

Johannes Bohn

Studiengang:	Informatik
Prüfer/in:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Betreuer/in:	Dipl.-Inf. Pascal Hirmer, Dipl.-Inf. Peter Reimann
Beginn am:	20. Mai 2015
Beendet am:	20. November 2015
CR-Nummer:	D.3.3, J.3

Kurzfassung

Simulationen werden in vielen Feldern der Wissenschaft benötigt, um Abläufe der realen Welt zu untersuchen, die sich nicht durch Experimente reproduzieren lassen, wie zum Beispiel Katastrophenszenarien. Um die durch derartige Simulationen entstandenen Daten zu visualisieren werden häufig monolithische Programme mittels Skriptsprachen angefertigt. Diese werden von Nicht-Informatikern, wie z.B. von Naturwissenschaftlern im Bereich der Biologie oder von Ingenieuren, angefertigt. Fehlendes Wissen im Bereich der Softwareentwicklung, Softwarearchitektur und Softwaretests führt dabei jedoch oftmals zu Programmen, die nur sehr schwer oder gar nicht wartbar und erweiterbar sind.

Um derartige, schlecht wartbare Programme in eine sinnvoll gegliederte Struktur zu bringen, muss deren Kontroll- und Datenfluss zuerst umfassend analysiert werden, um eine anschließende Restrukturierung des Programmcodes zu ermöglichen. In dieser Arbeit wird untersucht, wie bestehende, monolithische Skript-basierte Programme restrukturiert und in einen Datenflussgraphen umgewandelt werden können. Auf Basis des entstandenen Datenflussgraphes kann der Programmablauf leichter verstanden werden und die Wartbarkeit somit verbessert werden.

Um die Ergebnisse dieser Arbeit zu verdeutlichen, werden die Konzepte anhand eines Beispielszenarios umgesetzt. Dieses beschäftigt sich mit der Analyse und Restrukturierung von Python-Skripten zur Visualisierung der Ergebnisse von Knochensimulationen. Nach einer umfassenden Analyse wird der Programmablauf der Skripte restrukturiert, in einem Datenflussgraphen modelliert und anschließend in einer passenden Ausführungsumgebung ausgeführt. Das Ergebnis ist ein gut strukturierter, erweiterbarer Datenfluss, der aus dem ursprünglichen, schlecht wartbaren Programm entstanden ist. Auf Basis dieses Anwendungsfalls können die in dieser Ausarbeitung entstandenen Konzepte auf weitere, skriptbasierte Programme angewendet werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Listings

1. Einführung

In vielen wissenschaftlichen Bereichen werden Simulationen benutzt, um Abläufe zu untersuchen, die sich nicht durch Experimente reproduzieren lassen, wie zum Beispiel Katastrophenszenarien [GVW02]. Für die Auswertung der Daten müssen die Ergebnisse der Simulationen häufig visualisiert werden. Da dies oftmals spezielle Software erfordert, werden die benötigten Programme oft in Skriptsprachen von Nichtinformatikern, wie z.B. Naturwissenschaftlern im Bereich der Biologie oder von Ingenieuren, angefertigt [Bea00][MSK+15]. Da diese selten Wissen im Bereich der Softwareentwicklung, Softwarearchitektur oder Softwaretests haben, sind ihre Programme oftmals schwer wartbar oder erweiterbar.

Um solche schlecht wartbaren Programme in eine sinnvoll gegliederte Struktur zu bringen, muss deren Kontroll- und Datenfluss umfassend analysiert werden. Da Datenflüsse für eine derartige Strukturierung helfen, bietet sich eine flussbasierte Programmierung an [BH95]. Bei flussbasierter Programmierung wird der Code in funktionale Blöcke aufgeteilt, die dann auf einer höheren Ebene in einem Datenflussgraphen verbunden werden. Dadurch wird der Aufbau eines Programms deutlich sichtbarer. In dieser Arbeit wird dies anhand eines Beispiels im Bereich Knochensimulationen umgesetzt.

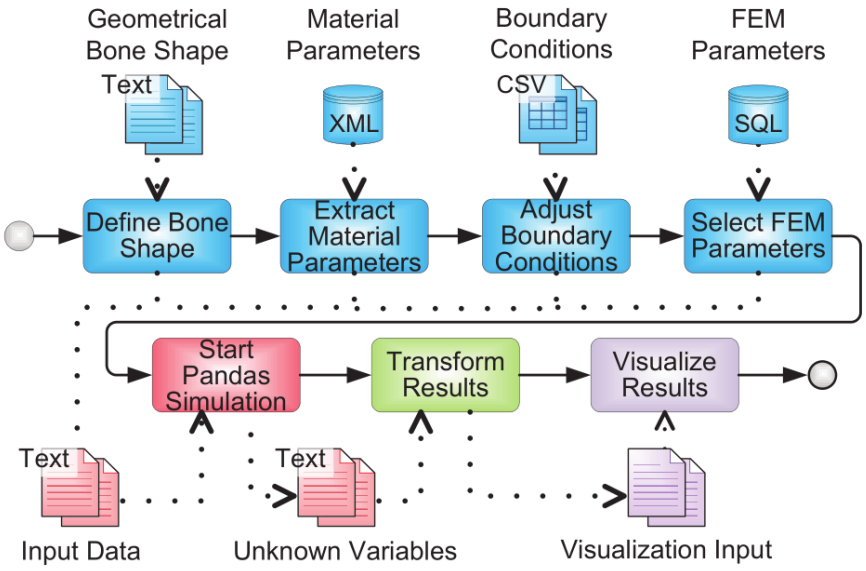

Abbildung 1.1.: Simulationsworkflow für Strukturänderungen in Knochen [RS14]

Bei dem Beispiel geht es um die Simulation von Strukturänderungen von Knochen [Kra14]. In Abb. 1.1 ist der gesamte Ablauf dieser Simulation zu sehen. Um etwas zu simulieren wird zuerst die Simulation vorbereitet. In diesem Beispiel in Abb. 1.1 sind das die ersten vier Schritte. Zu der Vorbereitung gehören das erstellen oder sammeln der Daten zu Form und Material des zu simulierenden Objekts. Danach kann simuliert werden. Die Resultate der Simulation werden dann konvertiert, um anschließend visualisiert zu werden. Das in dieser Arbeit vorgestellte Programm übernimmt die Visualisierung der Ergebnisse – in Abb. 1.1 der letzte Schritt.

Falls bei einer Simulation etwas schief läuft, kann das bisher erst nach Ende der oftmals langen Laufzeit festgestellt werden. Um dies zu verhindern ist das Ziel dieser Arbeit, darauf hinzuarbeiten, Simulationsergebnisse während der Simulation zu visualisieren, damit eine solche Fehlentwickelung frühzeitig erkannt werden kann.

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

In *Kapitel 2 – Grundlagen* werden die in dieser Arbeit verwendeten Technologien und Bibliotheken beschrieben. In *Kapitel 3 – Bestandsaufname* wird das bisher verwendete Programm beschrieben und analysiert. In *Kapitel 4 – Vorbereitung für die Umsetzung als Datenfluss* wird beschrieben, wie das originale Skript auf die wesentlichen Funktionen gekürzt und weiter analysiert wird. In *Kapitel 5 – Umsetzung als Datenfluss* wird die Umsetzung des Skriptes als Datenfluss beschrieben. In *Kapitel 6 – Zusammenfassung und Ausblick* wird eine Zusamenfassung der Arbeit mit Auswertung gegeben.

2. Grundlagen

In diesem Kapitel werden die in dieser Arbeit verwendeten Technologien beschrieben. Diese sind die Programmiersprache Python in Abschnitt 2.1, das Visualization Toolkit (VTK) in Abschnitt 2.2, die Bibliothek für Computer-Vision OpenCV in Abschnitt 2.3. In Abschnitt 2.4 werden die Technologien, die flussbasierte Programmierung implementieren oder damit in Zusammenhang stehen, beschrieben.

2.1. Python

Python[1] ist eine objektorientierte, skriptbasierte Programmiersprache, die 1990 von Guido van Rossum entwickelt wurde. Als solche wird sie interpretiert und hat dynamische Typenzuordung. Das bedeutet, dass der Typ einer Variable nicht explizit angegeben wird, sondern durch den darin gespeicherten Wert definiert wird. Python hat bereits höhere Datentypen wie assoziative sowie flexible Arrays eingebaut und kommt mit einer großen Anzahl Standardmodule, wie zum Beispiel *os* für die Interaktion mit dem Betriebssystem oder *math* für mathematische Funktionen. Um Beginn und Ende von Schleifen oder bedingte Verzweigungen zu definieren, wird in Python Einrückung statt Schlüsselwörter oder Klammern benutzt. In Python lässt sich Software durch die höheren Datentypen, die erforderte Einrückung und die fehlende Variablendeklaration lesbar und kompakt schreiben [Pyt].

```
import netCDF4
import numpy as np
from netCDF4 import ma
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages

def main(db_pth = '.', fmodel = 'clm'):
   g = netCDF4.Dataset(db_pth+'/land_water_mask/LandWaterMask_Global_CRUNCEP.nc', 'r')
   mask = g.variables['land_water_mask']
   mask = mask[:].swapaxes(0,1)

   f = netCDF4.Dataset(db_pth+'/NEE_first_year.nc', 'r')
   data = f.variables['NEE']
   data = data[:]
   data = data.swapaxes(0,2)
   adj = 60*60*24*(365/12)*1000
   data = data*adj
```

[1]Python: https://www.python.org/

```
   native = data.mean(2)
   latShape = mask.shape[0]
21 logShape = mask.shape[1]
   for x in range(latShape):
      for y in range(logShape):
         if mask[x,y] == 1 and ma.getmask(native[x,y]) == 1:
            for index in range(data.shape[2]):
26             data[x,y,index] = 0
   plt.imshow(np.mean(data,2))
   plt.xlabel("Mean 1982-2010 NEE [gC/m2/mon]")
   plt.title(fmodel + ":BG1")
   pp = PdfPages('result_NEE.pdf')
31 pp.savefig()
   pp.close()
```

Listing 2.1: Beispiel-Pythoncode [McP15]

Listing 2.1 enthält ein Beispiel für Python-Code in dem Daten graphisch dargestellt in eine PDF-Datei geschrieben werden. Im Beispiel in den Zeilen 12 und 22 werden zum Beispiel Variablen zugewiesen, ohne dass sie, beziehungsweise ihr Typ, deklariert wurden. In Zeilen 22 bis 26 sieht man, wie Python Einrückung benutzt, um bedingte Verzweigungen oder Schleifen zu beenden.

2.2. VTK

Das Visualization Toolkit (VTK)[2] ist ein Open-Source Software-System von Kitware für 3D-Computergrafik, Visualisierung und Bildverarbeitung. Es besteht aus einer C++-Bibliothek, die über Interface-Layers auch in Java und Python eingebunden werden kann. Mit VTK lassen sich Daten wie Punktemengen, Polygone, Bilder und Matrizen speichern und weiterverarbeiten [SAH00]. Unterstützt werden verschiedene Visualisierungsalgorithmen sowie einige Modellierungsfilter. Anwendung findet VTK zum Beispiel bei medizinischer Visualisierung oder industriellen Inspektionsprogrammen. Für verschiedene Anwendungen, wie zum Beispiel Volumenvisualisierung, Erdölexploration und Strömungsmechanik wird VTK außerdem durch kommerzielle Unternehmen erweitert [SAH00].

2.3. OpenCV

OpenCV (Open Source Computer Vision Library)[3] ist eine Software-Bibliothek in C und C++, die Algorithmen für maschinelles Lernen und Computer-Vision beinhaltet [Ope15]. Über Interfaces kann sie auch in Python, Java, Ruby und MATLAB eingebunden werden.

Ein Ziel von OpenCV ist eine einfach zu benutzende Bibliothek für umfangreiche Computer-Vision-Anwendungen bereitzustellen. Dafür beinhaltet OpenCV über 500 Funktionen aus verschiedenen Bereichen von Computer-Vision. OpenCV beinhaltet auch eine Bibliothek für maschinelles Lernen. Das

[2] VTK: http://www.vtk.org/
[3] OpenCV: http://opencv.org/

liegt daran, dass Computer-Vision und maschinelles Lernen oft zusammen verwendet werden. OpenCV wurde auf rechnerische Effizienz hin optimiert und kann Multicore-Prozessoren ausnutzen [BK08].

2.4. Flussbasierte Programmierung

In diesem Abschnitt werden Technologien beschrieben, die flussbasierte Programmierung implementieren. Bei flussbasierter Programmierung wird das Software-System als eine Menge von Komponenten, zwischen denen Nachrichten oder Daten ausgetauscht werden, angesehen [CM11].

2.4.1. PyF

PyF[4] ist eine Implementierung der flussbasierten Programmierung von Jonathan Schemoul[5] in Python. Die Plattform ist Open-Source und wurde für das Verarbeiten und Transformieren großer Daten entwickelt. Mit PyF lassen sich Codeblöcke definieren und deren Ein- und Ausgänge verbinden. Außerdem kann PyF über Plugins erweitert werden und lässt sich als Webservice betreiben. Wenn es als Webservice läuft, lässt sich der Datenfluss über eine graphische Oberfläche erstellen [PyF15].

2.4.2. Node-RED

Node-RED[6] ist ein Implementierung der flussbasierten Programmierung von IBM Emerging Technology, die auf Node.js basiert [Nod15]. Mit Node-RED lassen sich Programme auf einer graphischen Oberfläche verbinden. Unterstützt werden Interaktionen mit Hardware, Web Services, sowie APIs [MGR+15]. Die einzelnen Knoten des Graphen repräsentieren eigene oder vorinstallierte Codeblöcke. IBM unterhält auch eine Bibliothek benutzergenerierter Knoten und Graphen [Nod].

In Abb. 2.1 sieht man die graphische Oberfläche von Node-Red mit einem erstellten Beispiel-Datenfluss. In dem Datenfluss wird zuerst links eine Http-Anfrage empfangen. Auf den empfangenen Daten wird in *Funktion* Code ausgeführt. Der *switch*-Knoten entscheidet durch lesen der eingehenden Daten, auf welchem Ausgang der Datenfluss fortgesetzt werden soll. Es wird also hier aus den Daten ausgewählt, ob sie als Http-Antwort gesendet werden sollen (*http*) oder ob sie in eine Datei gespeichert werden sollen (*file*).

2.4.3. Kurator-Akka Framework

Bei der Verwaltung von wissenschaftlichen Daten kommt es vor, dass einige Daten Probleme wie Tippfehler in Namen oder Probleme mit Koordinatensystemen enthalten [Lud14]. Dies wird üblicherweise als Arbeitsablauf aus verschiedenen Werkzeugen und Services gelöst. Das Kurator-Projekt[7]

[4]PyF: http://pyfproject.org/
[5]Blog of Jonathan Schemoul: http://www.jondesign.net/
[6]Node-RED: http://nodered.org/
[7]Kurator: https://opensource.ncsa.illinois.edu/confluence/display/KURATOR/Kurator+Project+Home

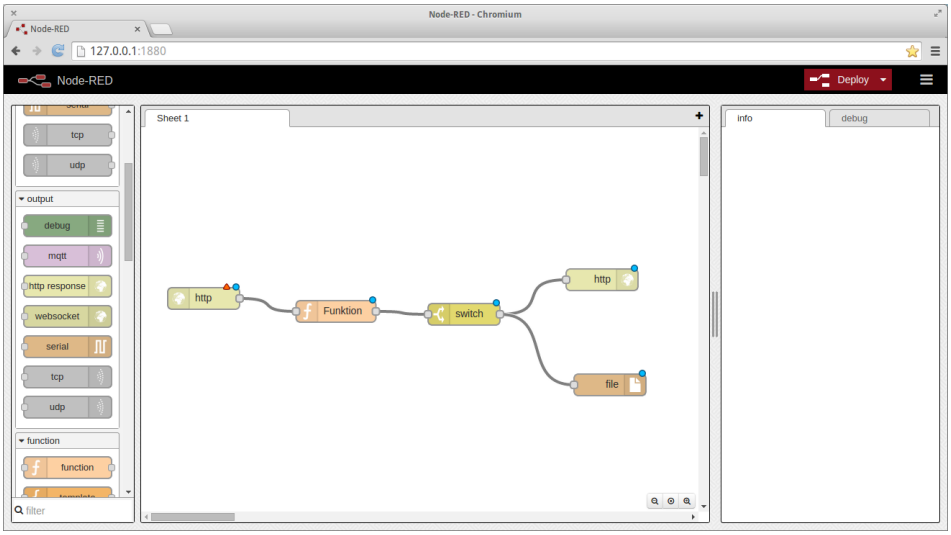

Abbildung 2.1.: Die graphische Oberfläche von Node-RED

von der University of Illinois und der Harvard University entwickelt ein Modul aus verschiedenen vorgefertigten Arbeitsschritten, die sich zu einem Workflow zusammenfügen lassen [DCM+12].

Akka[8] ist eine Ausführungsumgebung für verteilte, mitteilungsbasierte Java-Programme. Es wurde als Grundlage für das Kurator-Akka-Framework[9] benutzt, da es bereits die Verwaltung der Ausführungsreihenfolge und -zeit sowie des Datenflusses teilweise unterstützt [MLH+15].

Die einzelnen Codeblöcke für Kurator-Akka können in Java oder Python geschrieben werden. Die Informationen für die Einbindung in Workflows, sowie die Workflows selbst werden im YAML-Datenformat definiert.

2.4.4. YesWorkflow

Wissenschaftliche Workflow-Management-Systeme haben viele Vorteile, da sie Funktionen haben, die bei dem Erstellen komplizierter Abläufe aus modularen Bausteinen und der Ausführung dieser helfen. Trotzdem werden viele automatisierte Workflows außerhalb wissenschaftlicher Workflow-Management-Systeme implementiert und ausgeführt. Dies liegt daran, dass viele Wissenschaftler mit Skriptsprachen wie Python, Perl, R und MATLAB Erfahrung haben und damit produktiver sind. YesWorkflow[10] ist ein Werkzeug, mit dem ein Benutzer existierende Programme mit Informationen zu deren Workflow annotieren kann. Dadurch hat der Benutzer viele Vorteile von wissenschaftlichen Workflow-Management-Systemen, benötigt aber keine Workflow-Engine und muss den Code nicht anpassen. Aus der Annotation ergeben sich die Codeblöcke sowie eine Beschreibung des Datenflusses zwischen den Codeblöcken [MSK+15].

[8] Akka: http://akka.io
[9] Kurator-Akka: https://github.com/kurator-org/kurator-akka
[10] YesWorkflow: http://yesworkflow.org/wiki

Da die Annotationen in Kommentaren untergebracht sind, sind sie sprachenunabhängig. Um Wissenschaftlern den Einstieg in YesWorkflow zu erleichtern sind die Kommentare und das Modell von YesWorkflow absichtlich einfach gehalten [MSK+15].

Im Moment befindet sich das Projekt noch im Prototypstatus [McP15].

```
import netCDF4
import numpy as np
from netCDF4 import ma
import matplotlib.pyplot as plt
from matplotlib.backends.backend_pdf import PdfPages

# @BEGIN main
# @PARAM db_pth
# @PARAM fmodel
# @IN input_mask_file @URI file:{db_pth}/land_water_mask/LandWaterMask_Global_CRUNCEP.nc
# @IN input_data_file @URI file:{db_pth}/NEE_first_year.nc
# @OUT result_NEE_pdf @URI file:result_NEE.pdf

def main(db_pth = '.', fmodel = 'clm'):
   # @BEGIN fetch_mask
   # @PARAM db_pth
   # @IN g @AS input_mask_file @URI file:{db_pth}/land_water_mask/LandWaterMask_Global_CRUNCEP.nc
   # @OUT mask @AS land_water_mask
   g = netCDF4.Dataset(db_pth+'/land_water_mask/LandWaterMask_Global_CRUNCEP.nc', 'r')
   mask = g.variables['land_water_mask']
   mask = mask[:].swapaxes(0,1)
   # @END fetch_mask

   # @BEGIN load_data
   # @PARAM db_pth
   # @IN input_data_file @URI file:{db_pth}/NEE_first_year.nc
   # @OUT data @AS NEE_data
   f = netCDF4.Dataset(db_pth+'/NEE_first_year.nc', 'r')
   data = f.variables['NEE']
   data = data[:]
   data = data.swapaxes(0,2)
   adj = 60*60*24*(365/12)*1000
   data = data*adj
   # @END load_data

   # @BEGIN standardize_with_mask
   # @IN data @AS NEE_data
   # @IN mask @AS land_water_mask
   # @OUT data @AS standardized_NEE_data
   native = data.mean(2)
   latShape = mask.shape[0]
   logShape = mask.shape[1]
   for x in range(latShape):
      for y in range(logShape):
         if mask[x,y] == 1 and ma.getmask(native[x,y]) == 1:
            for index in range(data.shape[2]):
               data[x,y,index] = 0
   # @END standardize_with_mask
```

```
    # @BEGIN simple_diagnose
    # @PARAM fmodel
    # @IN data @AS standardized_NEE_data
    # @OUT pp @AS result_NEE_pdf @URI file:result_NEE.pdf
    plt.imshow(np.mean(data,2))
    plt.xlabel("Mean 1982-2010 NEE [gC/m2/mon]")
    plt.title(fmodel + ":BG1")
    pp = PdfPages('result_NEE.pdf')
    pp.savefig()
    pp.close()
    # @END simple_diagnose
# @END main
```

Listing 2.2: Beispielcode für YesWorkflow-Kommentare [McP15]

In Listing 2.2 ist der gleiche Code wie in Listing 2.1 zu sehen, der diesmal allerdings mit YesWorkflow annotiert ist. Die YesWorkflow-Kommentare definieren in Zeile 7 den Beginn des Programmes. In Zeilen 8 bis 12 werden Parameter und globale Ein- und Ausgabedaten definiert. Das dazugehörige Ende befindet sich in Zeile 61. Für Eingabedaten lassen sich auch Templates für URI-Dateipfade angeben.

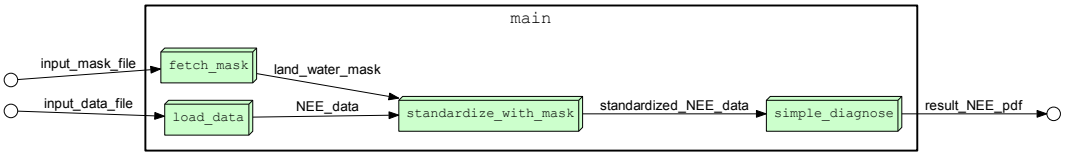

Abbildung 2.2.: Mit YesWorkflow aus Listing 2.2 generierter Graph

Dazwischen werden die einzelnen Codeblöcke definiert, zum Beispiel zwischen Zeilen 15 und 22. Genauso wie für das ganze Programm werden hier für jeden einzelnen Codeblock Anfang und Ende (Z.15,22) sowie Parameter und Ein- und Ausgabedaten (Z. 16-18) definiert. Aus diesen Namen oder alternativen Namen (*@AS*) ermittelt YesWorkflow, welche Codeblöcke über welche Datenkanäle miteinander verbunden sind und erstellt daraus einen Datenflussgraphen (Abb.2.2).

3. Bestandsaufname

Um Simulationen von Strukturveränderungen von Knochen zu visualisieren wird momentan PANPOST verwendet. Dieses wird in diesem Kapitel vorgestellt. PANPOST wurde am Institut für Mechanik der Universität Stuttgart geschrieben. Mit dem Programm lassen sich .dat-Dateien im Tecplot[1]-Format laden, rendern und anzeigen. Wenn eine Datei gerendert ist und das Modell angezeigt wird, kann man es drehen und so von allen Seiten betrachten. Über Menüs lassen sich Daten über Einfärbungen anzeigen und Einstellungen der Visualisierung wie zum Beispiel Belichtung ändern. Das angezeigte Bild lässt sich außerdem als Bilddatei abspeichern.

Zu dem PANPOST-Programm liegen 201 Tecplot-Dateien aus einer Simulationen sowie sieben weitere Beispieldateien vor. Die Tecplot-Dateien enthalten die relevanten 3D-Modelle. Da die Einlesemethode aus dem PANPOST-Programm weiterverwendet werden konnte, war es nicht notwendig, sich im Detail mit diesen Dateien, beziehungsweise der entsprechenden Datenstruktur zu beschäftigen.

Das PANPOST-Programm wurde in Python geschrieben und besteht aus 23 Dateien, von denen die Größte knapp 5000 Zeilen hat. Es benutzt VTK für die Visualisierung und wxPython[2] für die graphische Benutzeroberfläche. Nach kleineren Anpassungen läuft das Programm in Python 2.7.3, allerdings bleiben noch Probleme, wie zum Beispiel fehlende Informationen in der Leiste des Fensters (siehe Abb. 3.3), bestehen.

Es besteht aus einem Front- und einem Backend. Das Frontend beinhaltet die Benutzeroberfläche. Das Backend ist für die bei der Visualisierung durchgeführten Berechnungen zuständig. Diese Einteilung hilft aber nur bedingt, da der Code für beide Teile nicht sauber getrennt ist. Der größte Teil des Frontends wird zum Beispiel in der gleichen Datei und Klasse definiert, wie die wichtigsten Teile des Backends. Außerdem sind Kommentare größtenteils in Deutsch, teilweise aber auch in Englisch verfasst. Es gibt viel auskommentierten Code, sowie Kommentare, die auf fehlende Funktionen hindeuten.

Ein weiteres Problem des Programms besteht darin, dass es die verfügbaren Tecplot-Dateien aus der Simulation nicht fehlerfrei liest. Das Einlesen funktioniert an der betroffenen Stelle durch Abzählen der Werte anstatt den Namen der Werte in Betracht zu ziehen. Die Simulationsdateien haben in dieser Zeile einen Wert mehr als die Beispieldateien. Deshalb läuft das Programm ohne es zu ändern mit diesen nicht.

In Abschnitt 3.1 wird das Programm anhand der Python-Dateien, aus dem es besteht erklärt. Anschließend wird in Abschnitt 3.2 die Benutzeroberfläche des Programms beschrieben. In Abschnitt 3.3 wird eine kurze Zusammenfassung der Ergebnisse aus diesem Kapitel gegeben.

[1] Tecplot:http://www.tecplot.com/
[2] wxPython: http://www.wxpython.org/

3.1. Die Dateien des PANPOST-Programms

Abbildung 3.1 zeigt eine Übersicht über die Dateien aus denen das Programm besteht. Die zu analysierenden Python-Dateien lassen sich nach ihrer Funktion einteilen:

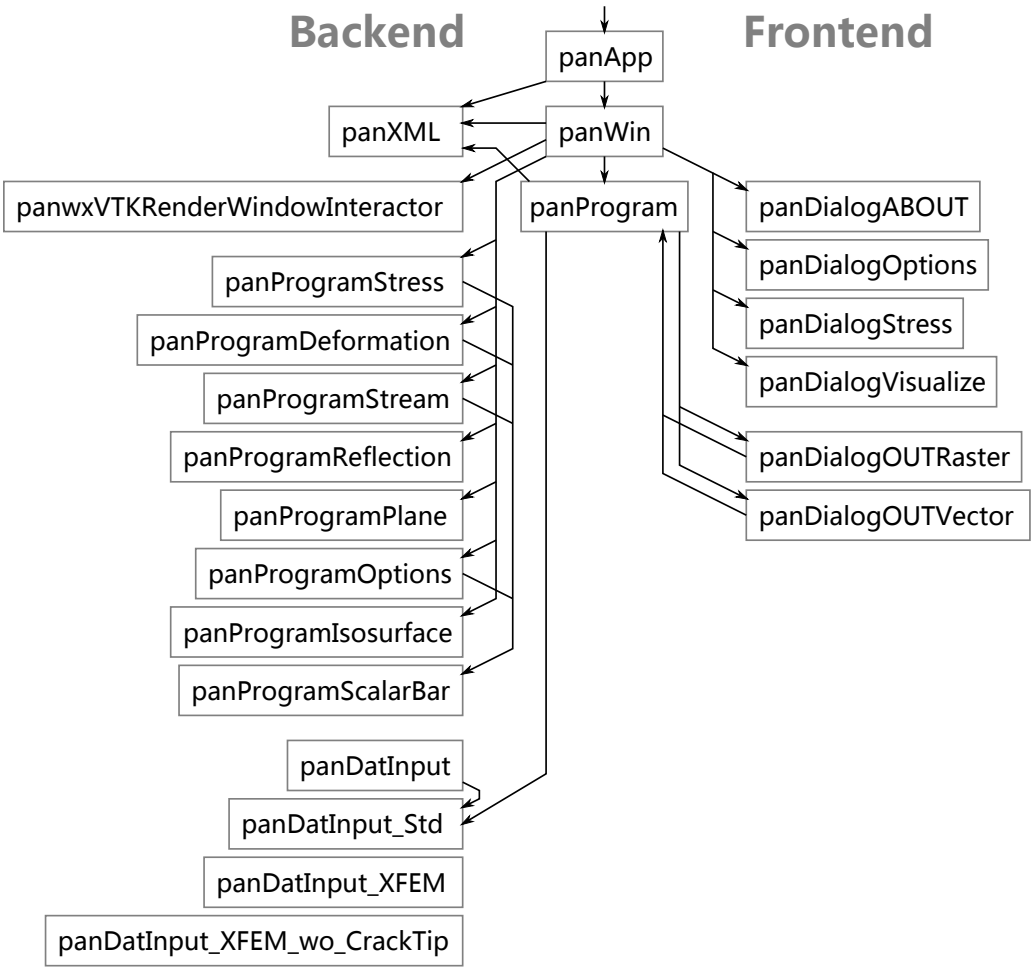

Abbildung 3.1.: Alle Dateien von PANPOST mit deren Abhängigkeiten

Hauptklasse

Die Main-Methode des Programms befindet sich in *panApp.py*. Hier werden außerdem noch eventuell mit übergebene Argumente ausgewertet. Die Standardeinstellungen für das Programm, wie zum Beispiel Kameraposition, befinden sich hart kodiert in dieser Datei. Diese Einstellungen werden verwendet, wenn sie nicht in den Dialogfenstern geändert werden. Abgesehen von den Programmeinstellungen und dem Aufruf des nächsten Skripts befindet sich in dieser Datei für diese Arbeit nichts Wesentliches.

Hauptfenster

Das Hauptfenster wird in der Datei *panWin.py* definiert. Außerdem wird hier das Rendern der Tecplot-Dateien vorbereitet und ausgeführt. Das ist eine zentrale Funktion für die Implementierung in den nachfolgenden Kapiteln. Der Ablauf davon wird in Abbildung 3.2 dargestellt.

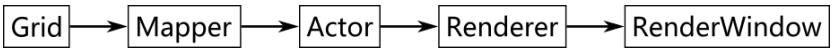

Abbildung 3.2.: Die Schritte, die zur Vorbereitung zum Rendern nötig sind

Um das *vtkUnstructuredGrid*, das aus der Tecplot-Datei ausgelesen wurde, zu rendern, wird zuerst ein *vtkDataSetMapper* erzeugt. Dieser Mapper erhält das Grid als Eingabedaten. Der Mapper ist eine Schnittstelle zwischen den Daten und den graphischen Primitiven. Als nächstes wird ein *vtkActor* erzeugt, der den Mapper als Eingabe erhält. Der Actor ist dafür zuständig Daten darüber zu speichern, wie das Objekt in der Renderumgebung platziert ist. Den Actor erhält wiederum der vtkRenderer als Eingabe, der in diesem Fall schon früher erzeugt wurde. Der Renderer ist für das Rendern eines einzelnen Objektes zuständig. Dieser wird dann einem vtkRenderWindow hinzugefügt. Das RenderWindow ist das Fenster in dem gerendert wird. Das RenderWindow hat die Methode *Render()*, die deren Renderer befiehlt, das Bild zu rendern.

Die meisten Klassen in den anderen Python-Dateien des Programms haben eine Referenz zurück auf die Klasse in *panWin.py*, von der sie aufgerufen wurden. Über diese Referenz sind auch außerhalb dieser Datei Renderaufrufe zu finden. Wenn eine Datei geöffnet wird, wird diese mehrmals gerendert.

Einlesen der Tecplot-Dateien

Für das Einlesen von Tecplot-Dateien sind in dem Programm vier Dateien vorgesehen. Drei der Dateien, *panDatInput_Std.py*, *panDatInput_XFEM.py* und *panDatInput_XFEM_wo_CrackTip.py*, lesen die Tecplot-Dateien ein. Dabei hat jede Datei eine andere Variante der Einlese-Methode. Die vierte Python-Datei, *panDatInput.py*, verweist auf *panDatInput_Std.py*. Die Idee dahinter ist vermutlich, dass in der Datei *panDatInput.py* auf die Datei mit dem zu verwendenden Einleseprogramm referenziert wird. Die referenzierte Datei wäre dann dafür zuständig, die Daten aus den Tecplot-Dateien einzulesen und in ein *vtkUnstructuredGrid* zu speichern. Tatsächlich wird allerdings in *panProgram.py* direkt auf *panDatInput_Std.py* referenziert und nicht auf *panDatInput.py*. Damit wird immer *panDatInput_Std.py* verwendet.

In der in den nachfolgenden Kapiteln vorgestellten Implementierung müssen genauso wie in dem PANPOST-Programm Tecplot-Dateien eingelesen werden, daher ist die hier verwendete Datei, *panDatInput_Std.py*, auch für den nächsten Abschnitt wichtig. Da im Ausgangsprogramm die anderen Einlesedateien nicht verwendet werden, können sie vernachlässigt werden.

Dialogfenster

Vier Dateien sind für je ein Dialogfenster zuständig. Die Datei *panDialogABOUT.py* stellt ein About-Fenster dar, *panDialogOptions.py* ein Optionsfenster, *panDialogStress.py* das Fenster für Stress-

Optionen und *panDialogVisualize.py* für die Schnitt-Optionen.
Das About-Fenster zeigt an, dass die Programme vom Institut für Mechanik an der Universität Stuttgart geschrieben wurden. Mit dem Optionsfenster und dem Stress-Fenster lassen sich Anzeigeeinstellungen, wie zum Beispiel Farbe, Beleuchtung, Reflexion und farbige Anzeige bestimmter Werte aus der Visualisierung ändern. Da es in dieser Arbeit um die Visualisierung der 3D-Objekte geht, ist die Funktion des Schnitt-Fensters, Schnittebenen durch das Objekt zu legen, nicht relevant.

Es gibt noch zwei weitere Dateien *panDialogOUTRaster.py* und *panDialogOUTVector.py*. Dies sind Einstellungsfenster, die beim Exportieren in Raster- und Vektorgrafikdateien erscheinen. Wenn als Format der zu speichernden Datei JPEG, PDF oder EPS gewählt wird, wird ein Dialogfenster mit Optionen geöffnet. Das Dialogfenster für JPEG-Dateien ist in *panDialogOUTRaster.py* definiert und bietet Optionen zu Qualität des Bildes und progressiver JPEG-Generierung. Das Dialogfenster für PDF- und EPS-Dateien ist in *panDialogOUTVector.py* definiert und bietet unter anderem die Wahl, ob die Datei komprimiert werden soll und ob die Hintergrundfarbe übernommen werden soll. Bei Wahl von PNG-, BMP- und PS-Dateien öffnet sich kein Optionsfenster.

Hilfsfunktionen

In *panProgram.py* befinden sich Hilfsfunktionen. Eine Klasse ist dabei für das Abspeichern in Bilddateien zuständig. Eine Weitere ruft zum Beispiel die Python-Dateien zum Einlesen von Tecplot-Dateien auf. Das Abspeichern in Bilddateien wird in der nachfolgenden Implementierung ebenfalls benötigt. Die Funktion dieser Datei ist daher für diese Arbeit von Bedeutung.

Berechnung von optionalen Visualisierungen

Die Berechnungen der zusätzlichen Visualisierungen, die über die Optionsmenüs ausgewählt und gestartet werden, werden in acht weiteren Dateien, *panProgramStress.py*, *panProgramDeformation.py*, *panProgramStream.py*, *panProgramReflection.py*, *panProgramPlane.py*, *panProgramOptions.py*, *panProgramIsosurface.py*, *panProgramScalarBar.py*, gemacht. Diese sind für die in den nachfolgenden Kapiteln vorgestellte Implementierung nicht relevant.

Bibliotheken

Die letzten beiden Dateien sind Bibliotheken von anderen Autoren, die in PANPOST verwendet werden. Die erste, *panXML.py*, bildet eine Schnittstelle zu XML. Die andere, *panwxVTKRenderWindowInteractor.py*, verbindet wxPython, die Bibliothek, die zum Erstellen der graphischen Oberfläche verwendet wurde, und VTK. Diese sind ebenfalls für die vorgestellte Implementierung nicht relevant.

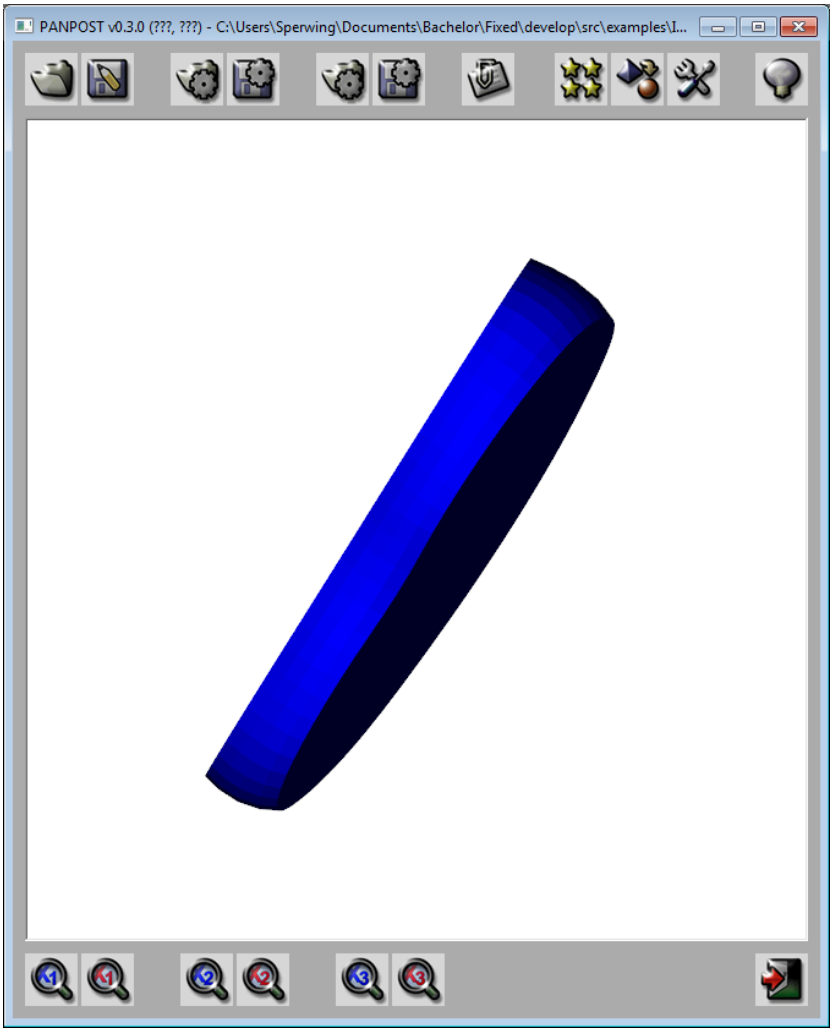

Abbildung 3.3.: Die Benutzeroberfläche von PANPOST

3.2. Benutzeroberfläche

In Abb. 3.3 sieht man die Benutzeroberfläche des bisher verwendeten Programmes. Diese besteht aus drei Teilen: der oberen Menüleiste, der Anzeigefläche in der Mitte und der unteren Menüleiste. Mit der oberen Leiste lassen sich mit den zwei linken Symbolen Tecplot-Dateien öffnen und als Bild exportieren. Die zwei Symbole daneben sind dafür da, VTK-Dateien zu importieren und exportieren, die nächsten Zwei, die Optionen als XML-Dateien zu importieren und exportieren.

Die vier Tasten oben rechts öffnen Dialogfenster. Die ersten zwei öffnen Fenster für Funktionen, die für die Weiterverwendung nicht relevant sind und teilweise auch nicht funktionieren. Die nächste Taste öffnet ein Fenster, in dem sich Schnitte durch das dargestellte Objekt machen lassen. Die dritte Taste öffnet das Optionen-Fenster. Hier lassen sich Einstellungen der Darstellung ändern, wie zum

Beispiel Hintergrundfarbe und Beleuchtung. Die letzte Taste zeigt ein Fenster mit Informationen über die Software an.

In der Mitte wird das Modell angezeigt. Es lässt sich mit der Maus drehen und zoomen. In der unteren Leiste befinden sich links sechs Tasten, um die Kameraposition des Modells zu ändern und rechts eine Taste, um das Programm zu beenden.

3.3. Zusamenfassung

Folgende Dateien sind für die Umsetzung als Visualisierungspipeline relevant:

panDatInput_Std.py In dieser Datei befindet sich die Funktion für das Einlesen der Tecplot-Dateien. Da das in einer eigenen Datei passiert und die Funktion keine Abhängigkeiten in anderen Dateien hat lässt sich der Originalcode direkt übernehmen.

panWin.py Hier befindet sich der in Abb. 3.2 dargestellte Ablauf für das Rendern der Tecplot-Dateien. Dieser kann nicht im alten Code aufgerufen werden, sondern muss neu umgesetzt werden.

panProgram.py In dieser Datei befinden sich die Funktionen für die Ausgabe als Bilddateien. Die Datei kann ebenfalls nicht eingebunden werden. Allerdings kann eine Funktion zum Abspeichern als PNG-Datei weitestgehend direkt übernommen werden.

4. Vorbereitung für die Umsetzung als Datenfluss

Das in Kapitel 3 beschriebene Programm soll flussbasiert umgesetzt werden. Um eine bessere Übersicht zu haben, wie das flussbasierte Programm aussehen muss, wurde das PANPOST-Programm zuerst auf die wesentlichen Funktionen gekürzt. Die notwendigen Funktionen, die auch im flussbasierten Programm erhalten bleiben sollen sind:

1. Einlesen einer Tecplot-Datei
2. Rendern der eingelesenen Datei mittels des in Abb. 3.2 gezeigten Ablaufs. Angezeigt werden muss das gerenderte Bild dabei nicht.
3. Speichern der gerenderten Grafik in eine Bilddatei

Für das Einlesen gibt es im PANPOST-Programm drei Funktionen, die in drei verschiedenen Dateien untergebracht sind. Da nur eine der Dateien, *panDatInput_Std.py* in PANPOST zum Einlesen tatsächlich benutzt wird, wird diese auch in der flussbasierten Implementierung verwendet. Anstatt den Code aus der Datei zu kürzen und als Teil des gekürzten Programms zu schreiben, wird hier nur die Datei eingebunden und die Funktion darin aufgerufen. Der Grund dafür ist, dass die Alternative – die Funktion neu zu schreiben und dabei auf das Wesentliche zu kürzen – sehr aufwendig wäre. Die von außen aufgerufene Funktion, die der Konstruktor derselben Klasse *Dat2Grid* in *panDatInput_Std.py* ist, ruft dabei viele andere Methoden der Klasse auf. Außerdem hat die Klasse keine Abhängigkeiten zu anderen Dateien, sodass sie isoliert verwendet werden kann.

Der Ablauf des Renderns in VTK wurde bereits in Kapitel 3 erläutert. Hierfür wurden bei der Erstellung des gekürzten Programms vorhandene Einstellungsmöglichkeiten untersucht und ausgewählt, welche sich im flussbasierten Programm ändern lassen sollen. Für die Ausrichtung der Kamera wurden die Methoden Azimuth und Elevation statt Yaw und Pitch ausgewählt. Die Funktion von Azimuth und Elevation erschien nach Test intuitiver zu verwenden. ResetCamera, eine Funktion, die die Kamera auf das visualisierte Objekt zentriert und den Zoom anpasst, sodass das ganze Objekt im Bild ist, soll sich an- und ausschalten lassen. Ausschalten soll möglich sein, damit der Benutzer mehr Freiheit beim Positionieren der Kamera hat. Genauso soll der Benutzer die Funktion anschalten können, weil sie es sehr einfach macht ein Bild einzustellen, auf dem man das vollständige gerenderte Modell erkennen kann. Sehr wichtige Einstellungen sind die Höhe und Breite des resultierenden Bildes und damit auch – letztendlich – des Videos. Andere Einstellungen, die der Benutzer ändern können soll, sind Zoom und ParallelProjection. Wenn ParallelProjection angeschaltet ist, wird das Bild mit paralleler anstatt perspektivischer Projektion gerendert.

Als Funktionen zum Abspeichern als Bilddatei gibt es im PANPOST-Programm Methoden für die Rastergrafikformate JPG, PNG und BMP sowie für die Vektorgrafikformate PDF, PS und EPS. Zum

Anzeigen sowie zur Erstellung eines Videos eignen sich Vektorgrafiken weniger, da sowohl Videos als auch Bildschirme auf Rastergrafik basieren. Von den drei Rastergrafikformaten wurde PNG gewählt, da PNG-Dateien komprimiert sind, ohne auf den Bildern Artefakte zu erzeugen.

In Abschnitt 4.1 wird das gekürzte Programm vorgestellt. Anschließend, in Abschnitt 4.2 werden die Funktionen, die das flussbasierte Programm haben soll, die aber in PANPOST nicht vorhanden waren beschrieben. In Abschnitt 4.3 wird die Annotation des gekürzten Programms mit Yesworkflow beschrieben.

4.1. Das gekürzte Programm

Das in diesem Abschnitt vorgestellte gekürzte Programm dient der Vereinfachung der Erstellung des flussbasierten Programms. Das flussbasierte Programm wird erst im nachfolgenden Kapitel 5 vorgestellt.

In diesem Programm sind manche der Einstellungen, wie zum Beispiel die Hintergrundfarbe, sowie Kameraposition und Bildgröße vorübergehend hart kodiert. Das liegt daran, dass die Möglichkeit zum Ändern der Einstellungen des PANPOST-Programms im gekürzten Programm nicht existiert und eine Alternative noch nicht benötigt wurde. Mit hart kodierten Einstellungen war es auch möglich deren Funktionen zu testen, da der Code des Programms schnell änderbar war. Die Ein- und Ausgabepfade sind ebenfalls hart kodiert, werden aber, wie auch die Einstellungen, im flussbasierten Programm wieder für den Benutzer änderbar sein.

Da das flussbasierte Programm automatisch mehrere Dateien verarbeiten soll, ist eine Benutzeroberfläche, mit der auf manuelle Eingaben gewartet werden müsste, nicht von Nutzen. Daher kann die gesamte Benutzeroberfläche und damit auch *wxPython* im gekürzten Programm eingespart werden.

Das gekürzte Programm wird in den folgenden Abschnitten vorgestellt.

4.1.1. Einlesen der Tecplot-Dateien

```
    print 'reading file'
    self.grid = panDatInput_Std.Dat2Grid(
       'C:\Users\Sperwing\Documents\Bachelor\Fixed -
           Kopie\develop\src\examples\IN_bandscheibe.dat');
4   print 'file sucessfully read'
```

Listing 4.1: Teil des gekürzten Programmes, in dem die Tecplot-Datei eingelesen wird

Zum Einlesen wird die Datei *panDatInput_Std.py* aus dem PANPOST-Programm verwendet. Der Code, der in dieser Version zum Einlesen benutzt wird und die entsprechende Funktion in *panDatInput_Std.py* nutzt, ist in Listing 4.1 zu sehen.

4.1.2. Rendern

```
self.mapper = vtk.vtkDataSetMapper()
self.mapper.SetInputData(self.grid())

self.actor = vtk.vtkActor()
self.actor.SetMapper(self.mapper)
self.actor.GetProperty().SetColor(0, 0, 1) #>settings

self.ren = vtk.vtkRenderer()
self.ren.SetBackground(1, 1, 1)
self.ren.AddActor(self.actor)
self.ren.SetLayer(0)

self.camera = self.ren.GetActiveCamera()
self.camera.SetFocalPoint(0.0, 0.0, 0.0)
self.camera.SetPosition(0, 0, 100)
self.camera.SetViewUp(0, 1, 0)
self.camera.Roll(0) #>settings
self.camera.Azimuth(30) #>settings
self.camera.Elevation(0) #>settings
self.camera.Zoom(1) #>settings
self.camera.ParallelProjectionOn() #>settings
self.ren.ResetCamera() #>settings

self.renWin = vtk.vtkRenderWindow()
self.renWin.AddRenderer(self.ren)
self.renWin.SetSize(1000, 1000) #>settings
self.renWin.OffScreenRenderingOn()
self.renWin.Render()
```

Listing 4.2: Teil des gekürzten Programmes, in dem das Rendern vorbereitet wird und gerendert wird

Wie bereits in Kapitel 3 beschrieben, ist der erste Schritt der Vorbereitung für das Rendern die Erstellung eines *vtkDataSetMapper* und die Eingabe des Grid in diesen (siehe Abb. 3.2). Das passiert auch in diesem Programm in Listing 4.2 in den ersten zwei Zeilen. Der Aufruf von *grid()* ist dabei nötig, da die zum Einlesen benötigte Klasse eine Instanz von sich selbst zurückgibt. Um das benötigte *vtkUnstructuredGrid* zu erhalten, wird hier mit *grid()* die *__call__()*-Methode der Klasse aufgerufen.

Als nächstes werden in Listing 4.2 in Zeilen 4 und 5 ein *vtkActor* erstellt und der Mapper übergeben. In der nächsten Zeile wird die Farbe des zu rendernden Objektes festgelegt. Der Kommentar dahinter, sowie ggf. analog bei weiteren Code-Zeilen gibt an, dass die Farbe des Objektes in der flussbasierten Implementierung zu den änderbaren Einstellungen gehören soll.

In Zeile 8 wird der *vtkRenderer* erstellt und in Zeile 10 diesem der *vtkActor* übergeben. In Zeile 9 wird der Hintergrund als weiß festgelegt und in Zeile 11 die Ebene des Renderers gesetzt.

Von Zeile 14 bis 21 wird die Kamera bearbeitet. Die Kamera ist der virtuelle Ort, von dem aus das Objekt im Bild visualisiert wird. Die Kamera ist hier als Teil des Bearbeiten des Renderers und nicht als zusätzlicher Schritt angesehen, da die Kamera nicht extra erstellt, sondern mit dem Renderer erstellt und daraus extrahiert wird. Die Zeilen sind teilweise redundant, um zu testen, wie genau die Einstellungen funktionieren. Die Kamera ist bereits im Renderer definiert und wird in Zeile 13 geladen. In Zeile 14 wird der Punkt definiert, auf den die Kamera zeigen soll. In Zeile 15 wird die

Position der Kamera definiert. Zeilen 16 bis 20 definieren Ausrichtung und Zoom der Kamera. Zeile 22 richtet die Kamera so aus, dass das Objekt in einer sinnvollen Größe im Mittelpunkt des Bildes ist. Zeile 21 ändert die Projektion von perspektivisch zu parallel.

Der Renderer wird nun in Zeile 25 einem neuen *vtkRenderWindow* hinzugefügt. Zeile 26 legt die Größe des RenderWindow und damit die Größe des gerenderten Bildes fest. Ohne Zeile 27 würde sich beim Rendern ein Fenster öffnen und das gerenderte Bild anzeigen. Im flussbasierten Programm nützt diese Funktion nicht, da sich beim Rendern mehrerer Bilder für jedes Bild ein neues Fenster öffnen und schließen würde. Daher wird das Bild nicht angezeigt, sondern nur abgespeichert (siehe Abschnitt 4.1.3). In Zeile 28 steht schließlich der Befehl zum Rendern.

4.1.3. Ausgabe

```
    try:
       w2if = vtk.vtkWindowToImageFilter()
3      w2if.SetInput(self.renWin)

       writer = vtk.vtkPNGWriter()
       writer.SetInputConnection(w2if.GetOutputPort())
       writer.SetFileName('output.png')
8      writer.Write()
    except:
       print 'Error writing png file'
```

Listing 4.3: Teil des gekürzten Programmes, in dem die Bilddatei gespeichert wird

Der Code in Listing 4.3 entspricht dem der Methode *SavePNG* in der Datei *panProgram.py* des PANPOST-Programms. Um aus dem *vtkRenderWindow* eine Bilddatei zu generieren, muss zuerst, in Zeile 2, ein *vtkWindowToImageFilter* erstellt werden, der das RenderWindow übergeben bekommt (Zeile 3). Dieser wird in Zeile 6 mit einem *vtkPNGWriter* verbunden. In Zeile 7 wird der Pfad der zu erstellenden Datei festgelegt. Mit dem Aufruf *Write()* schreibt der PNGWriter die Bilddatei. Falls das Schreiben Fehler ergibt wird in Zeile 10 eine Fehlermeldung in der Konsole ausgegeben.

4.2. Neue Funktionen

Gegenüber dem gekürzten Programm gibt es im flussbasierten Programm neben der Umsetzung als Datenfluss noch weitere Änderungen.

Die Einstellungen, die im gekürzten Programm hart kodiert waren, werden im flussbasierten Programm in eine Settings-Datei ausgelagert. Für das Einlesen der Settings-Datei wird das bereits mitgelieferte Paket ConfigParser verwendet.

Ebenfalls im gekürzten Programm hart kodiert war die Eingabedatei. Im flussbasierten Programm werden alle Dateien in einem Ordner verarbeitet. Der Pfad des Ordners, in dem sich die Eingabedateien befinden, sowie ein Pfad zum Zwischenspeichern der erzeugten Bilder und der Pfad und Name der Ausgabedatei werden dabei in der Settings-Datei festgelegt.

Das gekürzte Programm gibt am Ende der Ausführung eine PNG-Datei zurück. Im flussbasierten Programm soll nicht ein Bild, sondern ein Video aus den Bildern vieler Dateien entstehen. Um das zu bewerkstelligen wird die Bilddatei zuerst wie im gekürzten Programm abgespeichert. Anschließend wird sie wieder eingelesen und mittels OpenCV zu dem Video als Frame hinzugefügt. Bei der Umsetzung stellte sich heraus, dass das Hinzufügen im Datenfluss möglich ist und nicht nach der Visualisierung aller Bilder passieren muss. In der Umsetzung als Datenfluss wird also ein Bild visualisiert und an das Video angefügt, bevor das nächste Bild visualisiert wird. Das erlaubt es außerdem, das visualisierte Bild nach dem Anfügen an das Video zu löschen. Damit wird immer nur eine Bilddatei zwischengespeichert, was ressourcenschonender ist.

4.3. Annotation mit YesWorkflow

Um das gekürzte Programm in ein flussbasiertes Programm zu konvertieren, müssen sinnvolle Codeblöcke und der Datenfluss zwischen diesen ermittelt werden. Dafür wurde YesWorkflow[1] benutzt.

Da YesWorkflow noch ein Prototyp ist, lassen sich damit nur die Kommentare aus einer einzelnen Datei auslesen. Deshalb war es nicht praktikabel YesWorkflow bereits im ursprünglichen PANPOST-Code zu verwenden, sondern das in Abschnitt 4.1 gekürzte Skript zu annotieren.

Bisher wurde in dieser Arbeit das Programm in drei Arbeitsschritte eingeteilt:

1. Einlesen einer Tecplot-Datei
2. Vorbereiten zum Rendern und Rendern der eingelesenen Datei
3. Speichern der gerenderten Grafik in eine Bilddatei

Diese sind als Grundlage für die Aufteilung in Codeblöcke hilfreich. Der Code zum Einlesen und Speichern ist hierbei kurz genug, um daraus jeweils einen Codeblock zu formen (siehe Listing 4.1 und 4.3). Der Code für das Rendern (siehe Listing 4.2) hat allerdings viele Funktionen und ist damit für einen Codeblock zu unübersichtlich. Deswegen soll dieser in mehrere Codeblöcke unterteilt werden. Entsprechend der Beschreibung in Abschnitt 3 wird im ersten Schritt der Mapper erstellt und diesem das Grid übergeben. Im zweiten Schritt wird der Actor erstellt, diesem der Mapper übergeben und die Farbe des Objektes eingestellt. Der dritte Schritt besteht aus der Erstellung des Renderers und dem Übergeben des Actors an diesen. Außerdem werden in diesem Abschnitt die Einstellungen für Hintergrundfarbe, Zentrierungspunkt und der Kameraeinstellungen festgelegt. Der vierte und letzte Schritt besteht aus der Erstellung des RenderWindow, dem Hinzufügen des Renderers, dem Einstellen der Fenstergröße und schließlich dem Rendern. Wenn man nun das Einlesen der Tecplot-Datei und Abspeichern der Bilddatei hinzunimmt ergibt sich folgende Einteilung für den Ablauf:

1. Einlesen einer Tecplot-Datei (Listing 4.1)
2. Erstellung des Mappers (Listing 4.2, Zeilen 1 und 2)
3. Erstellung des Actors und Setzen von Einstellungen (Listing 4.2, Zeilen 4 bis 6)

[1] YesWorkflow: http://yesworkflow.org/wiki

4. Erstellung des Renderers und Setzen weiterer Einstellungen (Listing 4.2, Zeilen 8 bis 23)
5. Erstellung des RenderWindows und Rendern (Listing 4.2, Zeilen 25 bis 29)
6. Speichern der gerenderten Grafik in eine Bilddatei (Listing 4.3)

Zusätzlich zu den Codeblöcken aus dem gekürzten Programm werden noch weitere, leere Codeblöcke eingefügt, die im flussbasierten Programm mit Code gefüllt werden sollen. Diese sind:

1. *prepare_settings*, ein Codeblock für das Einlesen der Einstellungen aus einer Datei (siehe Listing 4.4, Zeilen 12 bis 17)
2. *get_filename*, als Platzhalter für eine Funktion, die aus dem Eingabeordner die Dateinamen aller Tecplot-Dateien zurückgibt (siehe Listing 4.4, Zeilen 6 bis 10)
3. *make_video*, ein Codeblock für das Erstellen des Videos aus der Bilddatei
4. *remove_imagefile*, für das Löschen der nun nicht mehr verwendeten Bilddatei

```
   # @BEGIN main
   # @IN dat-file @URI file:{dat-file-name}
   # @IN settings-file @URI file:{op_pth}/settings.ini
4  # @OUT video-file @URI file:{video-file-name}

      # @BEGIN prepare_settings
      # @IN settings-file @URI file:{op_pth}/settings.ini
      # @OUT settings
9     # @OUT path-settings
      ### INSERT CODE HERE ###
      # @END prepare_settings

      # @BEGIN get_filename
14    # @PARAM path-settings
      # @OUT filename
      ### INSERT CODE HERE ###
      # @END prepare_setting

19 def __init__(self, params):
      # @BEGIN read_file
      # @IN dat-file @URI file:{dat-file-name}
      # @PARAM filename
            # @PARAM path-settings
24    # @OUT grid
      print 'reading file'
      self.grid = panDatInput_Std.Dat2Grid(
         'C:\Users\Sperwing\Documents\Bachelor\Fixed -
             Kopie\develop\src\examples\IN_bandscheibe.dat');
      print 'file sucessfully read'
29    # @END read_file
```

Listing 4.4: Der Anfang des gekürzten Programms mit YesWorkflow-Kommentaren

In Listing 4.4 sieht man den Anfang des gekürzten Programms mit eingefügten YesWorkflow-Kommentaren. Die ersten vier Zeilen definieren den Anfang des Programms und die Ein- und Ausgabedaten. Als Eingabe bekommt das Programm die Tecplot-Datei (Zeile 2), und die Einstellungsdatei (Zeile 3). Als Ausgabe gibt das Programm die Videodatei zurück (Zeile 4). Über *@URI* sind für die Dateien die Pfade angegeben, wobei der Teil in geschweiften Klammern variabel ist.

In Zeilen 6 bis 11 wird der erste Codeblock definiert. In diesem Fall handelt es sich um den leeren Codeblock, der später für die Verarbeitung der Einstellungsdatei zuständig sein soll. Er bekommt die Einstellungsdatei übergeben und liest daraus die verschiedenen Einstellungen ein, die in *path-settings*, die Dateipfade, und *settings*, die sonstigen Einstellungen, aufgeteilt sind.

Der nächste Codeblock, Zeilen 13 bis 17, ist ebenfalls noch nur ein Platzhalter. Hier soll die Funktion, die aus dem Eingabeordner die Dateinamen aller Tecplot-Dateien ausliest eingefügt werden. *@PARAM* in Zeile 14 sagt aus, dass der Wert ein Eingabeparameter ist.

Der nächste Block, von Zeile 20 bis 29, liest die Tecplot-Datei ein. Zeile 25 bis 28 davon sind der Code aus Listing 4.1. Dieser bekommt die Tecplot-Datei und im flussbasierten Programm auch den Pfad der Tecplot-Datei sowie die Pfad-Einstellungen und gibt das Grid an den nächsten Codeblock weiter, der hier nicht mehr dargestellt ist.

In Abbildung 4.1 sieht man nun den aus der gesamten Datei generierten Graphen.

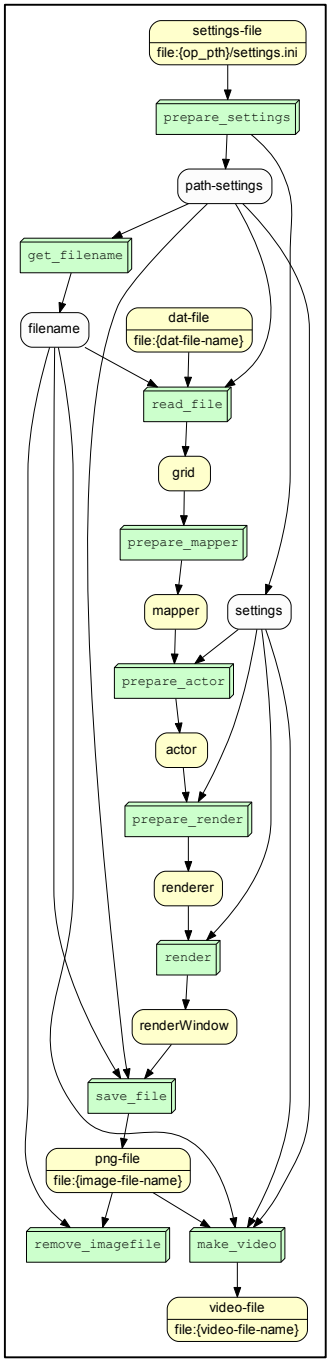

Abbildung 4.1.: Mit YesWorkflow generierter Graph des Datenflusses

5. Umsetzung als Datenfluss

In diesem Kapitel wird die Umsetzung als flussbasiertes Programm beschrieben. In Abschnitt 5.1 werden Ausführungsumgebungen für die flussbasierte Ausführung untersucht und verglichen. In Abschnitt 5.2 wird die Umsetzung als flussbasiertes Programm beschrieben. In Abschnitt 4.3 werden dann Probleme mit der Umsetzung aufgezeigt.

5.1. Vergleich von Datenflussplattformen

Um ein flussbasiertes Programm zu schreiben benötigt man eine Ausführungsumgebung, in der das Programm läuft. Für diese Arbeit wurden dafür drei Möglichkeiten betrachtet: Node-RED[1], Kurator-Akka[2] und PyF[3].

5.1.1. Node-RED

In der Aufgabenstellung wurde bereits Node-RED als Ausführungsumgebung vorgeschlagen. Node-RED erlaubt es dem Benutzer zusätzlich zu voreingestellten Knotentypen selbst Knoten mit Code zu definieren. Allerdings basiert Node-RED auf JavaScript und unterstützt Python nicht. Das Problem hierbei ist nicht, dass es unmöglich wäre Python-Code von Node-RED aus zu starten, denn dies kann mittels Knoten für externen Programmaufruf oder Web-Service-Aufruf erreicht werden. Schwieriger ist es die Python-Objekte von einem Codeblock an den nächsten weiterzugeben. Zwar hat Python Funktionen um Python-Objekte in Text zu serialisieren, allerdings muss das in dem zu serialisierenden Objekt auch implementiert sein. In den verwendeten VTK-Objekten ist das nicht der Fall. Als letzte Möglichkeit wäre es möglich die Serialisierung selbst zu implementieren. Da dafür nicht nur die Objekte die weitergegeben werden, sondern auch alle darin enthaltenen Objekte serialisiert werden müssten, wäre der Aufwand dafür gegenüber dem Aufwand die Implementierung mit einer Python-basierten Plattform umzusetzen nicht angemessen. Aus den genannten Gründen wurde Node-RED als Datenflussplattform verworfen.

[1] Node-RED: http://nodered.org/
[2] Kurator-Akka: https://github.com/kurator-org/kurator-akka
[3] PyF: http://pyfproject.org/

5.1.2. Kurator-Akka

Im Gegensatz zu Node-RED unterstützt Kurator-Akka Java und über Jython, einer Java-Implementierung von Python, auch Python. Mit Kurator-Akka werden die Datenflüsse in YAML definiert. Da Kurator-Akka nicht auf Python basiert und dessen Datenfluss in einer anderen Sprache separat von dem sonstigen Code definiert werden muss, wurde Kurator-Akka hier nicht verwendet. Eine Umsetzung mit Kurator-Akka wäre allerdings auch möglich gewesen.

5.1.3. PyF

PyF unterstützt nur Python. Es lässt sich auf verschiedene Arten ausführen; am einfachsten und für diese Anwendung ausreichend, lassen sich die Codeblöcke im Code anstatt in einer graphischen Oberfläche definieren und verbinden. Hauptnachteil ist allerdings, dass PyF seit längerer Zeit nicht gewartet wurde, was die Installation erschwert. Vermutlich wurde der Support von PyF eingestellt.

Da es Python besser unterstützt und der Datenfluss direkt im Code implementiert werden kann wurde PyF ausgewählt.

5.2. Umsetzung mittels PyF

In Listing 5.1 sieht man die Main-Methode des flussbasierten Programms. Im ersten Abschnitt (Zeilen 2 bis 8) wird das Programm vorbereitet, im zweiten und dritten (Zeilen 9 bis 26) werden die Codeblöcke definiert und verbunden und im letzten wird das Programm gestartet (Zeilen 28 bis 30). Zum Schluss wird das Schreiben des Videos beendet (Zeile 31).

```
def main():
    global pathsettings, rendersettings, videosettings
    print "start"
    # read settings from "settings.ini" file in operating path
    dir = dirname(__file__)
    path = join(dir,"settings.ini")
    pathsettings, rendersettings, videosettings = prepare_settings(path)
    source = get_filenames(pathsettings['inputfolder'])

    read_runner = runner (read_file)
    mapper_runner = runner (prepare_mapper)
    actor_runner = runner (prepare_actor)
    prepare_runner = runner (prepare_render)
    render_runner = runner (render)
    save_runner = runner (save_file)
    video_runner = runner (make_video)
    remove_runner = runner (remove_imagefile)

    read_runner.connect_in('values', iter(source))
    mapper_runner.connect_in('values',read_runner('out'))
    actor_runner.connect_in('values',mapper_runner('out'))
    prepare_runner.connect_in('values',actor_runner('out'))
```

```
    render_runner.connect_in('values',prepare_runner('out'))
24  save_runner.connect_in('values',render_runner('out'))
    video_runner.connect_in('values', save_runner('out'))
    remove_runner.connect_in('values', video_runner('out'))

    for item in remove_runner('out'):
29      #outfile.write( item )
        print item + " done!"
    writer.release
```

Listing 5.1: Die ersten zwei Codeblöcke der Umsetzung

In diesem Abschnitt wird das flussbasierte Programm beschrieben. Dazu wird zuerst in 5.2.1 die Vorbereitung des flussbasierten Programms erklärt. Dann wird in 5.2.2 die flussbasierte Ausführung beschrieben. In 5.2.3 werden die verwendeten Hilfsfunktionen beschrieben. In Abschnitt 5.2.4 wird schließlich noch die Einstellungsdatei erklärt.

5.2.1. Vorbereitung für die flussbasierte Ausführung

Bevor die Daten flussbasiert verarbeitet werden können, sind zwei Aktionen notwendig: Die Einstellungen müssen ausgelesen werden und die Pfadnamen der zu verarbeitenden Dateien müssen gesammelt werden. In Listing 5.1 werden zuerst die Einstellungen ausgelesen. Dazu wird in Zeile 5 der Ausführungspfad ausgelesen und in Zeile 6 an diesen „settings.ini“ angehängt. Der Grund dafür ist, dass das Programm erwartet, dass sich im Ausführungsordner des Programms die Einstellungsdatei mit dem Namen settings.ini befindet. In Zeile 7 wird mit dem Dateipfad eine Hilfsfunktion aufgerufen, die die Einstellungsdatei ausliest und die Einstellungen in drei assoziativen Arrays nach Kategorie sortiert ausgibt. Die drei Arrays werden in den Variablen *pathsettings*, *rendersettings* und *videosettings* gespeichert. Diese sind globale Variablen, sodass in den Codeblöcken direkt darauf zugegriffen werden kann. In Zeile 8 wird eine Hilfsfunktion aufgerufen. Diese hat die Aufgabe die Namen der Dateien in dem Eingabeordner, der in den Einstellungen angegeben ist, zurückzugeben.

Von Zeile 10 bis Zeile 17 wird angegeben, dass die Funktionen *read_file*, *prepare_mapper*, usw. (vgl. Listing 5.2) Codeblöcke für die flussbasierte Ausführung sind. Von Zeile 19 bis 26 werden die Codeblöcke zu einem Datenfluss verbunden, indem jedem Codeblock die Ausgabedaten des zuvor auszuführenden Codeblock als Eingabedaten definiert werden.

In Zeile 19 wird dabei angegeben, dass als Quelle für den ersten Block, *read_file*, über *source*, die Liste mit den Dateinamen, iteriert wird.

Nachdem die Dateinamen und Einstellungen für den flussbasierten Teil zugänglich gemacht wurden und der Ablauf des Datenflusses definiert wurde, wird der flussbasierte Programmteil in Listing 5.1 in Zeilen 28 bis 30 gestartet. Das wird dadurch implementiert, dass über den Ausgang des letzten Codeblocks iteriert wird. In Zeile 30 wird der Dateipfad des dann nicht mehr vorhandenen zwischengespeicherten Bildes in die Konsole ausgegeben, um beim Debuggen anzuzeigen, dass der Ablauf für eine Datei fertig ist.

Am Ende der Ausführung wird in Zeile 31 noch der *VideoWriter*, der die Videodatei erstellt, beendet.

5.2.2. Flussbasierte Ausführung

Da viel von dem Code identisch mit dem in Listing 4.1, 4.2 und 4.3 vorgestellten Code ist, werden hier nur beispielhaft zwei Codeblöcke vorgestellt, um die Änderungen durch die änderbaren Einstellungen und die Codeblöcke, die vorher noch nicht implementiert wurden, sowie die Änderungen, die durch die flussbasierte Umsetzung nötig sind, zu zeigen. Der vollständige Code ist im Anhang in Listing A.1 zu sehen.

```
@component('IN', 'OUT')
def read_file(values, out):
    for item in values:
        print "read: " + item
        grid = panDatInput_Std.Dat2Grid(pathsettings['inputfolder'] + '\\'+ item)
        yield (out, grid(), item)
```

Listing 5.2: Der erste Codeblock der Umsetzung: *read_file*

Listing 5.2 zeigt die Implementierung des Codeblocks *read_file*. Der Codeblock besteht einerseits aus Code, der für die Definition als Codeblock und das Verbinden der Daten notwendig ist, und andererseits aus Code, der die Funktion des Codeblocks implementiert.

Für einen Codeblock in PyF benötigt man eine Funktion, die zwei Eingabewerte bekommt. Einer davon, in Zeile 2 *values*, enthält die Eingabewerte als Liste, der andere, *out*, wird nur von PyF verwendet und wird im *yield* (siehe Zeile 6) weitergegeben. Über den Eingabewerten läuft eine Schleife. Da allerdings in PyF die Werte mit *yield* anstatt *return* zurückgegeben werden wird die Schleife für einen Wert in *values* erst ausgeführt, wenn er gebraucht wird. So läuft der Datenfluss für einen Eingabewert erst ganz durch, bevor der nächste verarbeitet wird. Ebenfalls nötig für einen Codeblock in PyF ist die Definition als Komponente, die eine Liste von Werten, anstatt eines Arrays von Listen von Werten, bekommt und weitergibt in Zeile 1.

Die Funktion dieses Codeblocks ist in Zeile 5 zu sehen. Dort wird, wie in 4.1, *panDatInput_Std.Dat2Grid* aufgerufen. Hier werden allerdings die Dateipfade aus dem Pfad des Eingabeordners sowie den Namen der Eingabedateien zusammengesetzt. In Zeile 6 wird außerdem die *__call__()*-Methode aufgerufen, um das *vtkUnstructuredGrid* zu erhalten. Weitergegeben werden die *vtkUnstructuredGrid* sowie die Dateinamen.

```
@component('IN', 'OUT')
def prepare_mapper(values, out):
    for item in values:
        num, grid, filename = item
        print "prepare mapper: "+filename
        mapper = vtk.vtkDataSetMapper()
        mapper.SetInputData(grid)
        yield (out, mapper,filename)
```

Listing 5.3: Der zweite Codeblock: *prepare_mapper*

In Listing 5.3 ist der Codeblock *prepare_mapper* zu sehen, in dem der Mapper erstellt wird und diesem das Grid aus Listing 5.2 übergeben wird.

Da zum Zwischenspeichern der Datei und zum Nachverfolgen in der Konsole auch ein Dateiname benötigt wird, wird der Name der Datei zwischen den Codeblöcken weitergereicht. Dazu enthält die Liste *values* in Listing 5.3 Tupel. Diese bestehen, wie in Zeile 4 sichtbar aus *num*, *grid* und *filename*. Die letzten beiden sind die letzten beiden Werte im *yield* in Zeile 6 in Listing 5.2. Der erste Wert enthält die verwendete Port-Nummer und wird nicht benötigt.

```
@component('IN', 'OUT')
def make_video(values, out):
   global writer
   for filepath in values:
      print "make video: "+filepath
      image= cv2.imread(filepath)
      if not writer:
         try:
            codec = cv2.VideoWriter_fourcc(*videosettings['codec'])
         except:
            print "no valid codec specified"
            codec = -1
         writer =
              cv2.VideoWriter(pathsettings['outputfile'],codec,float(videosettings['framerate']),
                     (int(videosettings['width']),int(videosettings['height'])))
      writer.write(image)
      yield (out,filepath)
```

Listing 5.4: Der Codeblock für *make_video*

In Listing 5.4 sieht man den Codeblock, in dem das Video mit OpenCV erstellt wird. Dazu wird in Zeile 6 für jede Tecplot-Datei das zuvor generierte Bild mit dem Bildleser von OpenCV wieder eingelesen. Falls es noch keinen *VideoWriter* gibt, also im ersten Durchlauf, wird in Zeile 9 der Codec aus den Einstellungen ausgelesen. Falls dort kein FourCC-Code angegeben wurde, wird in Zeile 12 "-1“, der Wert für keinen angegebenen Codec, als Codewert festgelegt. Wenn das passiert öffnet sich ein Fenster, in dem aus vorhandenen Codecs einer ausgewählt werden kann. In Zeile 13 wird dann ein *VideoWriter* definiert, wenn noch keiner existiert. Die nötigen Einstellungen werden dabei aus den assoziativen Arrays, in denen sie aus der Datei eingelesen wurden, entnommen. In Zeile 15 wird das Bild schließlich an das Video angefügt.

```
@component('IN', 'OUT')
def remove_imagefile(values, out):
   for filepath in values:
      print "remove "+ filepath
      remove(filepath)
      yield filepath
```

Listing 5.5: Der Codeblock für *remove_imagefile*

Der letzte Codeblock der flussbasierten Ausführung löscht die Bilddatei, da sie nachdem sie an das Video angefügt wurde nicht mehr benötigt wird. Dies passiert, wie in Listing 5.5 in Zeile 5 sichtbar, über einen Aufruf.

5.2.3. Hilfsfunktionen

```
def prepare_settings(filename):
   # reads settings from file at filename and returns ordered by section
   configpars = ConfigParser.ConfigParser()
   configpars.read(filename)
   pathsettings = config_get("path",configpars)
   rendersettings = config_get("render",configpars)
   videosettings = config_get("video",configpars)
   return pathsettings, rendersettings, videosettings

def config_get(section,configpars):
   # returns one section of the file as a list
   dict = {}
   options = configpars.options(section)
   for option in options:
      try:
         dict[option] = configpars.get(section,option)
      except:
         print("exception on %s!" % option)
         dict[option] = None
   return dict

def try_int(string):
   try:
      return int(string)
   except:
      return string

def sort_key(string):
   return [try_int(char) for char in split('([0-9]+)', string) ]

def get_filenames(folder):
   #returns the names of all .dat-files in the specified folder
   filenames = [ f for f in listdir(folder) if isfile(join(folder,f)) and (f.endswith('.dat'))]
   filenames.sort(key=sort_key)
   return filenames
```

Listing 5.6: Die verwendeten Hilfsfunktionen

In der Umsetzung wurden fünf Hilfsfunktionen verwendet. Zwei davon sind dafür zuständig die Settings-Datei einzulesen, die anderen drei die Dateinamen der zu verarbeitenden Tecplot-Dateien herauszufinden. Für das Einlesen der Einstellungen wird zuerst *prepare_settings* in Listing 5.6 in Zeile 1 aufgerufen. Diese Funktion bekommt den Dateipfad der Settings-Datei übergeben und gibt die Einstellungen als drei assoziative Arrays zurück. Dazu wird in Zeile 3 ein *ConfigParser* definiert, der in Zeile 4 die Settings-Datei ausliest. Danach wird drei mal die Hilfsfunktion *config_get* aufgerufen, die für einen Abschnitt in der Settings-Datei alle Optionswerte als assoziatives Array zurückgibt.

Die andere Funktion der Hilfsfunktionen ist es, eine Liste der Dateinamen in dem in den Einstellungen angegebenen Eingabeordner zu erstellen. Dafür wird die Methode *get_filenames* benutzt. In Zeile 33 wird eine Liste aller Dateien in dem Ordner erstellt, die mit ".dat“ enden. Da die Dateien im Video in der richtigen Reihenfolge sein müssen werden in Zeile 34 die Dateien sortiert. Dazu wird die

Hilfsfunktion *sort_key* benutzt, die mithilfe der Methode *try_int* die Dateien nicht alphanumerisch, sondern auch nach Zahlen sortiert, sodass zum Beispiel "2.dat“ vor "10.dat“ kommt.

5.2.4. Die Settings-Datei

```
[path]
Inputfolder: F:\bone_results\bc_standing
Workingfolder: C:\Users\Sperwing\Documents\Bachelor\OutputTecstat
Outputfile: C:\Users\Sperwing\Documents\Bachelor\OutputTecstat\outputStanding.avi

[render]
#in deg (float)
Roll: 100
Azimuth: 30
Elevation: 0

Zoom: 1
ParallelProjection: true

# The camera will reposition itself to view the center point of the actors,
# and move along its initial view plane normal (i.e., vector defined from
# camera position to focal point) so that all of the actors can be seen.
ResetCamera: true

Color: 0 0 1

[video]
# Height and width of video and images
# Can not be bigger than possible window size
Height: 850
Width: 1000

Framerate: 5

# FourCC (or -1 for choose from existing)
Codec: -1
```

Listing 5.7: Die Settings-Datei

Die Settings-Datei ist im INI-Format geschrieben. Sie besteht aus drei Abschnitten, *path*, *render* und *video*, die in Listing 5.7 in Zeilen 1, 6 und 22 beginnen. Diese dienen hauptsächlich der Übersichtlichkeit der Datei, werden allerdings auch im Programm in drei verschiedene Arrays gespeichert. Im ersten Abschnitt werden die Pfade von dem Eingabeordner (Zeile 2), dem Ordner zum Zwischenspeichern der Bilder (Zeile 3), sowie der Ausgabedatei (Zeile 4) definiert.

Der zweite Abschnitt enthält Einstellungen, die bei der Vorbereitung für das Rendern verwendet werden. In Zeilen 8 bis 10 wird die Richtung der Kamera in Grad festgelegt. In Zeile 12 wird der Zoom eingestellt und in Zeile 13 ob die Projektion parallel oder perspektivisch sein soll. In Zeile 18 wird eingestellt, ob die Funktion *ResetCamera*, die automatisch die Richtung und den Zoom der Kamera anpasst, aufgerufen werden soll. In Zeile 20 wird die Farbe des Objektes eingestellt. Die drei Zahlen stehen dabei für die Farben rot, grün und blau mit jeweils einem Wert zwischen 0 und 1.

Im letzten Abschnitt werden die Optionen für das Video eingestellt. Die Einstellungen für Höhe und Breite des Bildes in Pixeln in Zeilen 25 und 26 werden allerdings auch beim Rendern benötigt und sind nur für die Übersichtlichkeit unter *video* kategorisiert. Weitere Einstellungen sind die Bildfrequenz in Bilder pro Sekunde in Zeile 28 und der Codec in Zeile 31, der im FourCC-Format angegeben wird.

5.3. Evaluation

Das vorgestellte Programm liest alle Tecplot-Dateien in einem Ordner ein und macht daraus ein Video. Damit ist die Vorgabe teilweise erfüllt. In der Aufgabenstellung ist allerdings auch davon die Rede, dass die Ergebnisdaten der Simulation kontinuierlich verarbeitet werden sollen, also dass Dateien sobald sie vom Simulationsprogramm erstellt wurden zur Ausführung hinzugefügt werden können. Das ist mit dieser Implementierung nicht gegeben, da in der Implementierung die Dateien im Ordner zu Beginn aufgelistet und später zu dem Ordner hinzugefügte Dateien ignoriert werden. Um das umzusetzen müsste das Programm den Eingabeordner überwachen und sobald dort eine neue Datei erstellt wird den Datenfluss durchlaufen, um diese zu visualisieren. Für das Überwachen von Ordnern in Python gibt es Möglichkeiten, wie zum Beispiel Watchdog[4] oder pyinotify[5].

Ein Problem bereitet allerdings die Wahl von PyF als Datenflussengine. Da PyF die im Datenfluss zu verarbeitenden Objekte auf Basis von Iteration über Listen behandelt, wäre eine Umsetzung der kontinuierlichen Verarbeitung nur möglich, wenn der Datenfluss für jedes zu verarbeitende Objekt ganz neu gestartet wird. Das ist mit PyF möglich, würde PyF allerdings nicht optimal nutzen und den Code außerhalb des Datenflusses komplizierter machen. Für ein solches Programm wäre daher eine erneute Überprüfung der möglichen Datenflussplattformen sinnvoll.

Für die Umsetzung der kontinuierlichen Verarbeitung wäre es außerdem vorteilhaft, die Bilddateien anzuzeigen anstatt sie in ein Video zu speichern. Dies könnte mittels OpenCV oder auch direkt mit VTK implementiert werden.

Ein Problem mit der aktuellen Umsetzung ist, dass die Einlesefunktion aus PANPOST, die auch in der Umsetzung verwendet wurde, ohne Anpassung nicht alle Tecplot-Dateien einliest. Wie in Kapitel 3 bereits beschrieben, ist der Hintergrund, dass beim Einlesen der Dateien in einer bestimmten Zeile nicht die in der Datei angegebenen Werte angeschaut werden, sondern die Werte nur abgezählt. Wenn eine Datei in jener Zeile eine andere Anzahl an Werten hat, misslingt das Einlesen. An dieser Stelle wurde das Problem bemerkt, da Tecplot-Dateien mit verschieden vielen Werten vorlagen. Es ist allerdings auch möglich, dass die Einlesefunktion weitere Probleme dieser Art hat.

Da YesWorkflow mit Absicht einfach gehalten ist, hat es nur wenige Befehle. Im flussbasierten Programm gibt es sowohl Knoten, wie zum Beispiel *prepare_settings*, die nur einmal ausgeführt werden als auch Knoten die für jede zu visualisierende Datei einmal ausgeführt werden. Dafür gibt es in YesWorkflow allerdings keine Unterscheidung, was dessen Verwendung für flussbasierte Programme erschwert.

[4]Watchdog:https://pythonhosted.org/watchdog/
[5]https://github.com/seb-m/pyinotify

6. Zusammenfassung und Ausblick

In dieser Arbeit wurde ein Programm, das Simulationsergebnisse visualisiert, analysiert und beispielhaft als Datenfluss umgesetzt. Das Programm wird für die Visualisierung von Strukturänderungen in Knochen eingesetzt [Kra14]. Dazu liest es Tecplot-Dateien ein und zeigt die darin enthaltenen 3D-Modelle an. Dabei wurde das zuvor sehr umfangreiche Programm um unnötige Funktionen gekürzt, was es erlaubt, die notwendige Interaktion mit dem Programm zu reduzieren. Dazu wurde das analysierte Programm zuerst um die überflüssigen Funktionen, wie zum Beispiel die Möglichkeit sich das visualisierte Objekt von allen Seiten anzusehen oder die Möglichkeit Daten durch Einfärbungen anzuzeigen, gekürzt. Dieses gekürzte Programm wurde mit YesWorkflow annotiert, was den Datenfluss des Programms aufzeigte. Anschließend wurde das Programm als Datenfluss mit PyF umgesetzt. Dabei wurde es um die Funktionen, mehrere Bilder automatisch zu verarbeiten und die Bilder in ein Video zu konvertieren, erweitert.

Ausblick

In Abschnitt 5.3 wurde bereits beschrieben, dass das flussbasierte Programm die Simulationsergebnisse noch nicht kontinuierlich verarbeitet. Hier wäre eine Erweiterung sinnvoll, die die Tecplot-Dateien nach deren Erstellung sofort verarbeitet und das resultierende Bild anzeigt. Ebenso wäre es sinnvoll die Einlesefunktion zu korrigieren oder neu und kürzer umzusetzen, damit auch dieser Teil übersichtlicher und korrekter wird. Außerdem ist nicht klar ob PyF weiterentwickelt wird. Wenn das nicht der Fall ist wird es wahrscheinlich nötig sein, auf eine andere Datenflussplattform zu wechseln. Die Verwendung von Kurator-Akka wurde als Alternative noch nicht getestet.

A. Anhang – Der Code des flussbasierten Programms

```
'''
Created on 30.07.2015

@author: bohnjs
'''
# --------------------------------------------------------------------------
# Global config-variables
# =======================
import panDatInput_Std

DEBUG = True # type debug info

# --------------------------------------------------------------------------
# Import libraries
# ================
import vtk
from pyf.dataflow import runner, component
import ConfigParser
from os import listdir, remove
from os.path import isfile, join, dirname, splitext
from re import split

import cv2

# --------------------------------------------------------------------------
# Import modules
# ==============
# import panWin
import panXML


@component('IN', 'OUT')
def read_file(values, out):
   for item in values:
      print "read: " + item
      grid = panDatInput_Std.Dat2Grid(pathsettings['inputfolder'] + '\\'+ item)
      yield (out, grid(), item)

@component('IN', 'OUT')
def prepare_mapper(values, out):
   for item in values:
      num, grid, filename = item
      print "prepare mapper: "+filename
      mapper = vtk.vtkDataSetMapper()
```

```
      mapper.SetInputData(grid)
      yield (out, mapper,filename)

@component('IN', 'OUT')
def prepare_actor(values, out):
   for item in values:
      num, mapper, filename = item
      print "prepare actor: " + filename
      actor = vtk.vtkActor()
      actor.SetMapper(mapper)
      color = [float(x) for x in rendersettings['color'].split()]
      actor.GetProperty().SetColor(color)
      yield (out, actor, filename)

@component('IN', 'OUT')
def prepare_render(values, out):
   for item in values:
      num, actor, filename = item
      print "prepare render: " + filename
      ren = vtk.vtkRenderer()
      ren.SetBackground(1, 1, 1)
      ren.AddActor(actor)
      ren.SetLayer(0) #?
      camera = ren.GetActiveCamera()
      camera.SetFocalPoint(0,0,0) #>settings
      camera.SetPosition(0, 0, 100) #>settings
      camera.SetViewUp(0, 1, 0)
      camera.Roll(float(rendersettings['roll']))
      camera.Azimuth(float(rendersettings['azimuth']))
      camera.Elevation(float(rendersettings['elevation']))
      camera.Zoom(float(rendersettings['zoom']))
      if rendersettings['parallelprojection'] :
         camera.ParallelProjectionOn()
      if rendersettings['resetcamera'] :
         ren.ResetCamera()
      yield (out, ren, filename)

@component('IN', 'OUT')
def render(values, out):
   for item in values:
      num, ren, filename = item
      print "render: "+filename
      renwin = vtk.vtkRenderWindow()
      renwin.AddRenderer(ren)
      renwin.SetSize(int(videosettings['width']),int(videosettings['height']))
      #print "SetSize: "+ videosettings['width'] + " x " + videosettings['height']
      renwin.OffScreenRenderingOn()
      renwin.Render()
      yield (out, renwin, filename)

@component('IN', 'OUT')
def save_file(values, out):
   for item in values:
      num, renwin, filename = item
      print "save: "+filename
```

```
      try:
         w2if = vtk.vtkWindowToImageFilter()
         w2if.SetInput(renwin)
         writer = vtk.vtkPNGWriter()
         writer.SetInputConnection(w2if.GetOutputPort())
         filename = pathsettings['workingfolder']+'\\'+filename+'.png'
         writer.SetFileName(filename)
         writer.Write()
      except:
         print ' PanpostSave.SavePNG() - error writing png file:'
      yield (out,filename)


@component('IN', 'OUT')
def make_video(values, out):
   global writer
   for filepath in values:
      print "make video: "+filepath
      image= cv2.imread(filepath)
      if not writer:
         try:
            codec = cv2.VideoWriter_fourcc(*videosettings['codec'])
         except:
            print "no valid codec specified"
            codec = -1
         writer =
              cv2.VideoWriter(pathsettings['outputfile'],codec,float(videosettings['framerate']),
                     (int(videosettings['width']),int(videosettings['height'])))
      writer.write(image)
      yield (out,filepath)

@component('IN', 'OUT')
def remove_imagefile(values, out):
   for filepath in values:
      print "remove "+ filepath
      remove(filepath)
      yield filepath



def prepare_settings(filename):
   # reads settings from file at filename and returns ordered by section
   configpars = ConfigParser.ConfigParser()
   configpars.read(filename)
   pathsettings = config_get("path",configpars)
   rendersettings = config_get("render",configpars)
   videosettings = config_get("video",configpars)
   return pathsettings, rendersettings, videosettings

def config_get(section,configpars):
   # returns one section of the file as a list
   dict = {}
   options = configpars.options(section)
   for option in options:
      try:
```

```
            dict[option] = configpars.get(section,option)
        except:
            print("exception on %s!" % option)
            dict[option] = None
    return dict

def try_int(string):
    try:
        return int(string)
    except:
        return string

def sort_key(string):
    return [try_int(char) for char in split('([0-9]+)', string) ]

def get_filenames(folder):
    #returns the names of all .dat-files in the specified folder
    filenames = [ f for f in listdir(folder) if isfile(join(folder,f)) and (f.endswith('.dat'))]
    filenames.sort(key=sort_key)
    return filenames


pathsettings = None
rendersettings = None
videosettings = None

writer = None


# ----------------------------------------------------------------------------
# Starting mainloop
# =================
def main():
    global pathsettings, rendersettings, videosettings
    print "start"
    # read settings from "settings.ini" file in operating path
    dir = dirname(__file__)
    path = join(dir,"settings.ini")
    pathsettings, rendersettings, videosettings = prepare_settings(path)
    source = get_filenames(pathsettings['inputfolder'])

    read_runner = runner (read_file)
    mapper_runner = runner (prepare_mapper)
    actor_runner = runner (prepare_actor)
    prepare_runner = runner (prepare_render)
    render_runner = runner (render)
    save_runner = runner (save_file)
    video_runner = runner (make_video)
    remove_runner = runner (remove_imagefile)

    read_runner.connect_in('values', iter(source))
    mapper_runner.connect_in('values',read_runner('out'))
    actor_runner.connect_in('values',mapper_runner('out'))
    prepare_runner.connect_in('values',actor_runner('out'))
    render_runner.connect_in('values',prepare_runner('out'))
```

```
209     save_runner.connect_in('values',render_runner('out'))
        video_runner.connect_in('values', save_runner('out'))
        remove_runner.connect_in('values', video_runner('out'))

        for item in remove_runner('out'):
214        #outfile.write( item )
           print item + " done!"
        writer.release

    if __name__ == '__main__':
219     main()
```

Listing A.1: Das in dieser Arbeit vorgestellte flussbasierte Programm

Literaturverzeichnis

[Bea00] D. Beazly. Scientific Computing with Python. *Astronomical Data Analysis Software and Systems*, 216:49, 2000. URL http://adsabs.harvard.edu/abs/2000ASPC..216...49B. (Zitiert auf Seite 7)

[BH95] E. Baroth, C. Hartsough. Experience Report: Visual Programming in the Real World. *Visual Object Oriented Programming*, S. 21–42, 1995. (Zitiert auf Seite 7)

[BK08] G. Bradski, A. Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Inc., 1 Auflage, 2008. (Zitiert auf Seite 11)

[CM11] M. Cieślik, C. Mura. A lightweight, flow-based toolkit for parallel and distributed bioinformatics pipelines. *BMC Bioinformatics*, 12(1), 2011. doi:10.1186/1471-2105-12-61. (Zitiert auf Seite 11)

[DCM+12] L. Dou, G. Cao, P. Morris, R. Morris, B. Ludäscher, J. Macklin, J. Hanken. Kurator: A Kepler Package for Data Curation Workflows. *Procedia Computer Science*, 9:1614 – 1619, 2012. doi:10.1016/j.procs.2012.04.177. (Zitiert auf Seite 12)

[GVW02] S. T. Grilli, S. Vogelmann, P. Watts. Development of a 3D numerical wave tank for modeling tsunami generation by underwater landslides. *Engineering Analysis with Boundary Elements*, 26(4):301–313, 2002. doi:10.1016/S0955-7997(01)00113-8. (Zitiert auf Seite 7)

[Kra14] R. F. Krause. *Growth, modelling and remodelling of biological tissue*. Dissertation, Universität Stuttgart, Holzgartenstr. 16, 70174 Stuttgart, 2014. URL http://elib.uni-stuttgart.de/opus/volltexte/2015/9904. (Zitiert auf den Seiten 8 und 37)

[Lud14] B. Ludäscher. KURATOR: A Provenance-enabled Workflow Platform and Tool to Curate Biodiversity Data, 2014. URL http://de.slideshare.net/ludaesch/eresearch-round-kurator-project. Präsentation. (Zitiert auf Seite 11)

[McP15] T. McPhillips. YesWorkflow, 2015. URL http://yesworkflow.org/wiki. (Zitiert auf den Seiten 6, 10, 13 und 14)

[MGR+15] J. Maccallum, R. Gottfried, I. Rostovtsev, J. Bresson, A. Freed. Dynamic Message-Oriented Middleware with Open Sound Control and Odot. In *International Computer Music Conference*. 2015. (Zitiert auf Seite 11)

[MLH+15] T. McPhillips, D. Lowery, J. Hanken, B. Ludäscher, J. A. Macklin, P. J. Morris, R. A. Morris, T. Song, J. Wieczorek. Data cleaning with the Kurator toolkit, Bridging the gap between conventional scripting and high-performance workflow automation, 2015. URL http://www.slideshare.net/TimothyMcPhillips. Präsentation für TDGW 2015. (Zitiert auf Seite 12)

[MSK+15] T. McPhillips, T. Song, T. Kolisnik, S. Aulenbach, K. Belhajjame, R. K. Bocinsky, Y. Cao, J. Cheney, F. Chirigati, S. Dey, J. Freire, C. Jones, J. Hanken, K. W. Kintigh, T. A. Kohler, D. Koop, J. A. Macklin, P. Missier, M. Schildhauer, C. Schwalm, Y. Wei, M. Bieda, B. Ludäscher. YesWorkflow: A User-Oriented, Language-Independent Tool for Recovering Workflow Information from Scripts. *International Journal of Digital Curation*, 10(1):298–313, 2015. doi:10.2218/ijdc.v10i1.370. (Zitiert auf den Seiten 7, 12 und 13)

[Nod] Library - Node-RED. URL http://flows.nodered.org/. (Zitiert auf Seite 11)

[Nod15] Node-RED, 2015. URL http://nodered.org/. (Zitiert auf Seite 11)

[Ope15] OpenCV, 2015. URL http://opencv.org/. (Zitiert auf Seite 10)

[PyF15] PyF, flow-based python programming, 2015. URL http://pyfproject.org/. (Zitiert auf Seite 11)

[Pyt] The Python Tutorial. URL https://docs.python.org/3/tutorial/index.html. (Zitiert auf Seite 9)

[RS14] P. Reimann, H. Schwarz. Simulation Workflow Design Tailor-Made for Scientists. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*, S. 49. ACM, 2014. (Zitiert auf den Seiten 6 und 7)

[SAH00] W. J. Schroeder, L. S. Avila, W. Hoffman. Visualizing with VTK: A Tutorial. *IEEE Computer Graphics and Applications*, S. 20–27, 2000. URL http://www.doc.ic.ac.uk/~dr/teaching/Visualization/VTK_Tutorial.pdf. (Zitiert auf Seite 10)

Alle URLs wurden zuletzt am 18. 11. 2015 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift