der Bundeswehr
Universität München

Masterarbeit

Über die Kopplung von Laborexperimenten
mit MATLAB

Vorgelegt von:	Cand. M. Sc. Philipp Heinemann
Betreuer:	Hptm Dipl.-Ing. Ivo Baselt
Erstprüfer:	Univ. -Prof. Dr.-Ing. Andreas Malcherek
Zweitprüfer:	Dr.-Ing. Helmut Kulisch
Abgabetermin:	30. Juni 2013

Institut für Wasserwesen
Professur für Hydromechanik und Wasserbau
Univ.-Prof. Dr.-Ing. A. Malcherek
Dipl.-Ing. Ivo Baselt

Telefon +49 89 6004-2479
Telefax +49 89 6004-3858
E-Mail ivo.baselt@unibw.de

Universität der Bundeswehr München

Universität der Bundeswehr München · 85577 Neubiberg · Germany

Neubiberg, den 18.03..2012

Masterarbeit 2013

Thema: **Über die Kopplung von Laborexperimenten mit MATLAB**

Bearbeiter: Philipp Heinemann

Betreuer: Dipl.-Ing. Ivo Baselt

Inhalt: Die Professur für Hydromechanik und Wasserbau der Universität der Bundeswehr München nutzt derzeit zur digitalen Datenerfassung, -verarbeitung und –auswertung von Laborexperimenten die Software LabView. Um zukünftig sowohl für die Datenerfassung und Verarbeitung als auch für die Steuerung und Regelung des Wasserbaulabors die Möglichkeiten der MATLAB – Umgebung zu nutzen, soll in dieser Masterarbeit eine Kopplung von MATLAB mit der Hardware des Wasserbaulabors erfolgen.

Die wesentlichen Aufgaben der Masterarbeit sind:

- Einarbeitung in die MATLAB „Data Acquisition Toolbox“ und in die Erstellung von GUI`s
- Kopplung der MATLAB Software mit der Hardware des Wasserbaulabors
- Programmierung von Testprogrammen zur Verifikation und Validierung der Kopplung
- Durchführung der Experimente zum Vergleich der Testprogramme
- Vergleich der Ergebnisse der Testprogramme zwischen LabView und MATLAB
- Auswertung, Diskussion und Bewertung der Ergebnisse
- Konstruktive Hinweise zur Datenverarbeitung von aktuellen Projekten

Abgabe: nach Vereinbarung

Erstkorrektor: Prof. Dr.-Ing. A. Malcherek
Zweitkorrektor: Dr.-Ing. H. Kulisch

Vorwort

Diese Arbeit entstand im Rahmen meines Masterstudiums in Bauingenieurwesen und Umweltwissenschaften an der Universität der Bundeswehr München. Bedanken möchte ich mich zunächst bei Univ.-Prof. Dr.-Ing. Andreas Malcherek und der Professur für Hydromechanik und Wasserbau.

Mein besonderer Dank gilt meinem Betreuer Hptm Dipl.-Ing. Ivo Baselt und dem Laborleiter Dr.-Ing. Helmut Kulisch, die mir bei jeglichen Problemen weiterhelfen konnten und mich so in meiner Masterarbeit unterstützten.

Des Weiteren möchte ich mich bei Herrn Capek bedanken, der mir stets für Fragen bezüglich der Implementierung von Software mit Hardwarekomponenten des Wasserbaulabors zur Seite stand.

Inhaltsverzeichnis

Vorwort I

Inhaltsverzeichnis II

1. Einleitung 1

2. Simulationssoftware 2

2.1 MATLAB Guide 2

2.2 Simulink 7

2.3 Data Acquisition Toolbox (DAT) 8

2.4 Multifunktions-Datenerfassungsmodul 13

3. Implementierung einer Benutzeroberfläche 16

3.1 Erstellen des Simulationsmodells 16

3.2 Erstellen der Benutzeroberfläche 18

3.3 Steuern des Simulationsmodells 19

3.4 Verändern des Verstärkungsfaktors 22

3.5 Darstellen der Ergebnisse 24

3.6 Alternative Steuerung des Simulationsmodells 26

3.7 Real-Time Plot 29

4. Hardwarekopplung über DAT 40

4.1 Versuchsaufbau 40

4.2 Das Simulationsmodell 41

4.3 Die Benutzeroberfläche 46

4.4 Auswertung der Messreihe 48

5. Durchflussmessung mit MATLAB 51

5.1 Versuchsaufbau 51

5.2 Das Simulationsmodell 52

5.3 Die Benutzeroberfläche 55

5.4 Auswertung der Messreihe 56
5.5 Vergleich mit LabView 58
6. Durchflussregelung mit MATLAB 60
6.1 Versuchsaufbau 60
6.2 Das Simulationsmodell 61
6.2.1 Durchflussmessung 63
6.2.2 PID-Controller 64
6.2.3 Impulserzeugung 65
6.2.4 Signalausgabe 68
6.3 Die Benutzeroberfläche 69
6.4 Auswertung der Ergebnisse 70
6.5 Vergleich mit LabView 75
7. Fazit und Ausblick 78
Abbildungsverzeichnis IV
Tabellenverzeichnis VIII
Quellenverzeichnis IX
Anlagenverzeichnis X
Erklärung XIII

1. Einleitung

Die Professur für Hydromechanik und Wasserbau der Universität der Bundeswehr München nutzt derzeit zur digitalen Datenerfassung, -verarbeitung und –auswertung von Laborexperimenten die Software LabView. Um zukünftig sowohl für die Datenerfassung und Verarbeitung, als auch für die Steuerung und Regelung von Versuchsanlagen im Wasserbaulabor die Möglichkeiten der MATLAB (Version 2012a) Umgebung zu nutzen, soll mithilfe dieser Masterarbeit eine Kopplung von MATLAB mit den Hardwarekomponenten des Wasserbaulabors realisiert werden.

Ein großer Bestandteil der Arbeit ist daher die Einarbeitung in die MATLAB „Data Acquisition Toolbox“, welche zur Kopplung von Hardwarekomponenten mit der MATLAB Software bzw. der Toolbox „Simulink“ genutzt werden kann. Über die erstellten Simulationsmodelle erfolgen die Datenverarbeitung von erfassten Messwerten und die Steuerung bzw. Regelung von Hardwarekomponenten. Anhand von Testprogrammen wird veranschaulicht, wie die Kopplung mit der Hardware des Wasserbaulabors erfolgen kann und welche Einstellungen dabei zu berücksichtigen sind. Zu den Testprogrammen gehören unter anderem Programme zur Messung und Regelung des Durchflusses in einer Versuchsanlage des Wasserbaulabors.

Um die Programme anwenderfreundlich zu gestalten, werden mit der Toolbox „MATLAB Guide“ grafische Benutzeroberflächen erstellt, mit denen die Steuerung der Prozesse zur Datenaufnahme und –verarbeitung möglich ist. Diese Toolbox basiert auf der objektorientierten Programmierung und bietet daher auch viele Möglichkeiten zur Datenauswertung. Um die Erstellung von Benutzeroberflächen nachzuvollziehen, werden einige Beispielprogramme geschrieben, welche die wichtigsten Funktionen zur Steuerung und Auswertung der Testprogramme zur Durchflussmessung und -regelung enthalten.

Um die Testprogramme zu verifizieren und zu validieren, erfolgt im Rahmen dieser Masterarbeit eine Auswertung der Ergebnisse verschiedener Szenarien. Die gewonnenen Erkenntnisse dienen zum Vergleich zwischen der MATLAB und LabView Software, um die Kopplung mit der Hardware des Wasserbaulabors diskutieren und bewerten zu können.

Alle geschriebenen Beispiel- und Testprogramme befinden sich auf der beiliegenden CD und können zu Testzwecken im Wasserbaulabor ohne weitere Einstellungen ausgeführt werden.

2. Simulationssoftware

In diesem Kapitel erfolgt eine Übersicht und Beschreibung der verwendeten MATLAB Toolboxen, mit denen Testprogramme zur Kopplung von MATLAB Software mit der Hardware des Wasserbaulabors erstellt wurden. Die Testprogramme erhalten für den Endnutzer eine grafische Benutzeroberfläche zur Steuerung der Simulationsmodelle und Auswertung der erfassten Daten. Über die „Data Acquisition Toolbox" soll die Kopplung der Simulationsmodelle mit der Hardware des Wasserbaulabors erfolgen. Zur Hardware gehören unter anderem Messsensoren und Steuermotoren.

Für die Erstellung der grafischen Benutzeroberflächen wird „MATLAB Guide" benutzt. Die Simulationsmodelle, welche zur Datenerfassung oder Steuerung bzw. Regelung der Hardware verwendet werden können, lassen sich mit „Simulink" erzeugen. Die „Data Acquisition Toolbox" dient dabei als Schnittstelle zwischen Software und Hardware.

2.1 MATLAB Guide

MATLAB Guide (Graphical User Interface Development Environment) oder kurz GUI ist eine Toolbox von MATLAB mit der sich grafische Benutzeroberflächen erstellen lassen. Ein GUI dient zur Steuerung von MATLAB Skripten. Dazu gehören auch die mit Simulink erstellten Simulationsmodelle. Die Erstellung der Benutzeroberflächen basiert auf der objektorientierten Programmierung und wird auf Grundlage des Corporate Windows-Designs erzeugt.

MATLAB Guide lässt sich über die Kommandozeile mit dem Befehl *„guide"* öffnen. Daraufhin erscheint ein Auswahlfenster mit vordefinierten GUI's (Abb. 2.1.1). Hier lassen sich auch neue GUI's direkt abspeichern oder bestehende GUI's öffnen (*„Open Existing GUI"*). Zur Vorauswahl gehört ein „Blank GUI", ein neues bzw. leeres Fenster. Dieses Fenster wird von MATLAB als „Figure" (.fig) abgespeichert und enthält alle grafischen Elemente, welche mit dem **Dialog Editor** hinzugefügt werden können. Die Funktionen dieser Elemente werden in einem zugehörigen MATLAB Skript abgespeichert (.m). Die anderen vordefinierten GUI's enthalten Elemente, wie **Control Panels**, **Dialog Boxes, Menüleisten** oder **Diagramme**.

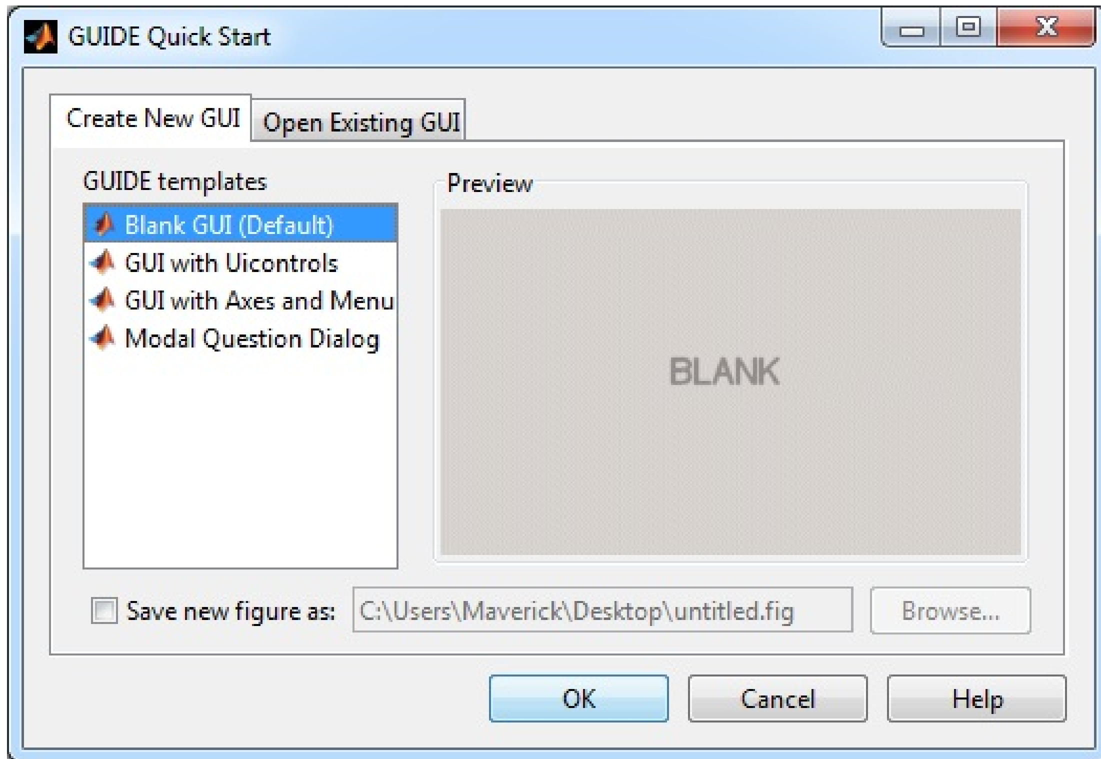

Abb. 2.1.1: Auswahlfenster (GUIDE Quick Start) mir vordefinierten GUI's

Der **Dialog Editor** (Abb. 2.1.2) besteht aus einer Menüleiste mit Schnellstartsymbolen und einer Toolbar, welche die Komponenten zur objektorientierten Programmierung enthält. Die Toolbar befindet sich auf der linken Seite vom **Dialog Editor**. Die einzelnen Steuerelemente lassen sich per „Drag&Drop"-Verfahren von der Toolbar auf die Fensterfläche vom **Dialog Editor** ziehen. Jedes Element besitzt individuelle Eigenschaften, die sich mit dem **Property Inspector** bearbeiten lassen. Um in den **Property Inspector** zu gelangen, genügt ein Doppelklick auf das ausgewählte Element.

Die Menüleiste befindet sich am oberen Rand vom **Dialog Editor** und enthält unter anderem Schnellstartsymbole zum Speichern, Laden und Starten eines GUI, sowie den **Menu Editor**, **Property Inspector** und **m-File Editor**. In der Menüleiste kann auch das Layout vom **Dialog Editor** und der zu erstellenden GUI's bearbeitet werden.

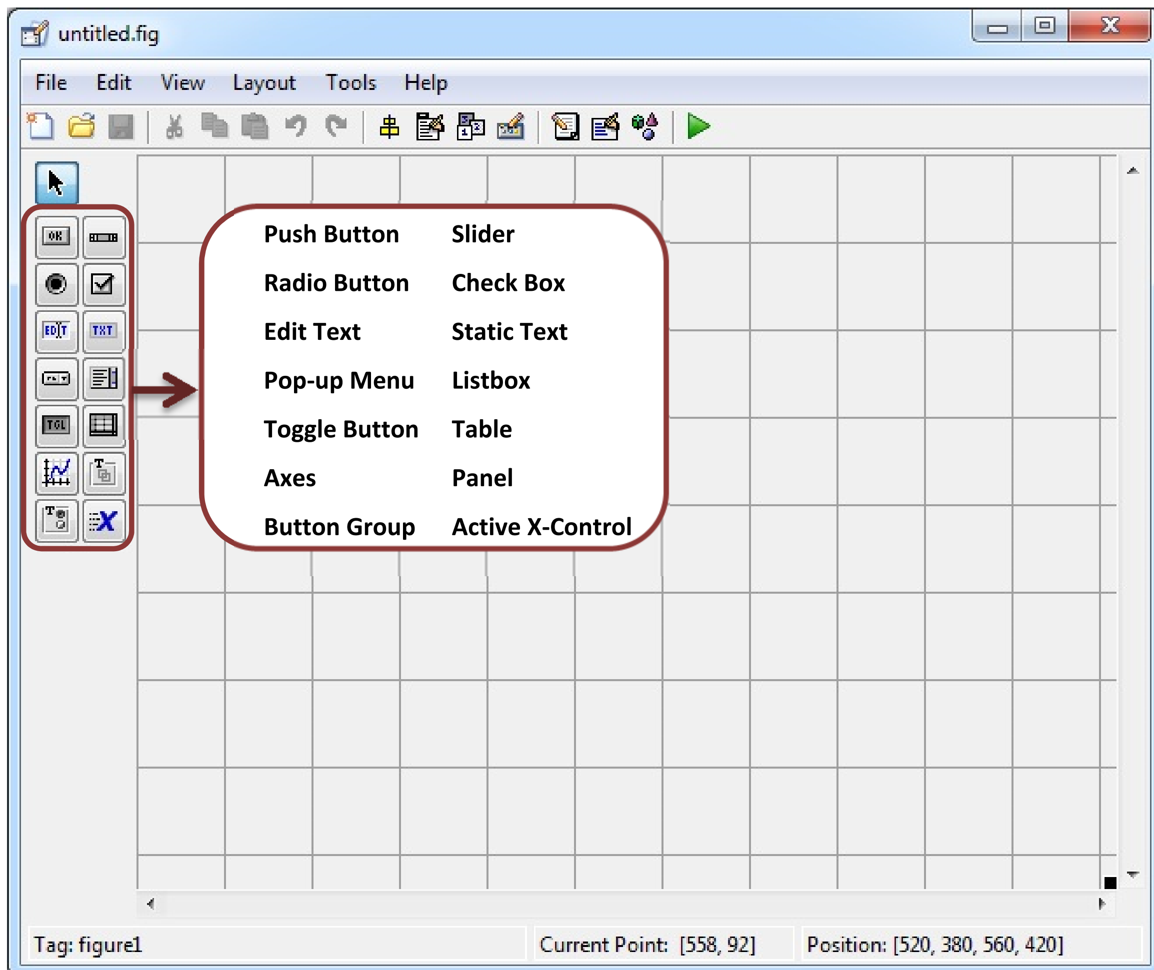

Abb. 2.1.2: Dialog Editor mit Darstellung der Steuerelemente

Der **M-File Editor** erstellt ein MATLAB Skript zu der Benutzeroberfläche und erzeugt dabei den Quelltext für die einzelnen Steuerelemente. Es wird also ein MATLAB Code generiert, welcher zu jedem Element Funktionen erstellt. Dabei wird jedem grafischen Objekt eine eindeutige Zahl (*„Pointer“*) zugewiesen. Diese Zahl wird in MATLAB *„handle“* genannt und dient zum Zugriff auf die grafischen Attribute.

In der Anlage 1.1 befindet sich ein vordefiniertes Skript zu einem GUI, welches noch keine Steuerelemente enthält. Dennoch befindet sich in diesem Skript ein Code zur Generierung von dem GUI mit dem abgespeicherten Namen und dem zugehörigen *„„Tag““* (*„Beispiel1_gui“*), sowie eine *„Opening Function“* und eine *„Output Function“*. Der „Tag“ eines GUI ist die Bezeichnung für die Fensterfläche bzw. der Objektname vom „Figure“, welches die grafischen Elemente des GUI beinhaltet. Über diese eindeutige Zuweisung kann direkt Zugriff auf die Fensterfläche genommen werden, um beispielsweise mit einem *„Close“*-Befehl die Fensterfläche zu schließen und so das GUI zu beenden.

Der „Tag“ von einem GUI, sowie von allen grafischen Objekten, kann im **Property Inspector** geändert werden (Anlage 1.2). Es empfiehlt sich für den abgespeicherten Namen und dem „Tag“ der Benutzeroberfläche die gleiche Bezeichnung zu geben, um Bezeichnungsfehler zu vermeiden.

Im Folgendem sind zwei Befehle aufgelistet, mit denen sich das GUI *„Beispiel1_gui“* schließen lässt:

```
close Beispiel1_gui
delete(handles.Beispiel1_gui)
```

Im *„Beispiel1_gui“* wird ein **Push Button** mit der Funktion zum Schließen der Oberfläche hinzugefügt. Der Button erhält den „Tag“ *„pushbutton1“*. Im **Property Inspector** kann anschließend der angezeigte Text (String) von dem Button eingegeben werden. Da der Button zum Schließen des GUI dienen soll, wurde als String *„Beenden“* gewählt (Abb. 2.1.3).

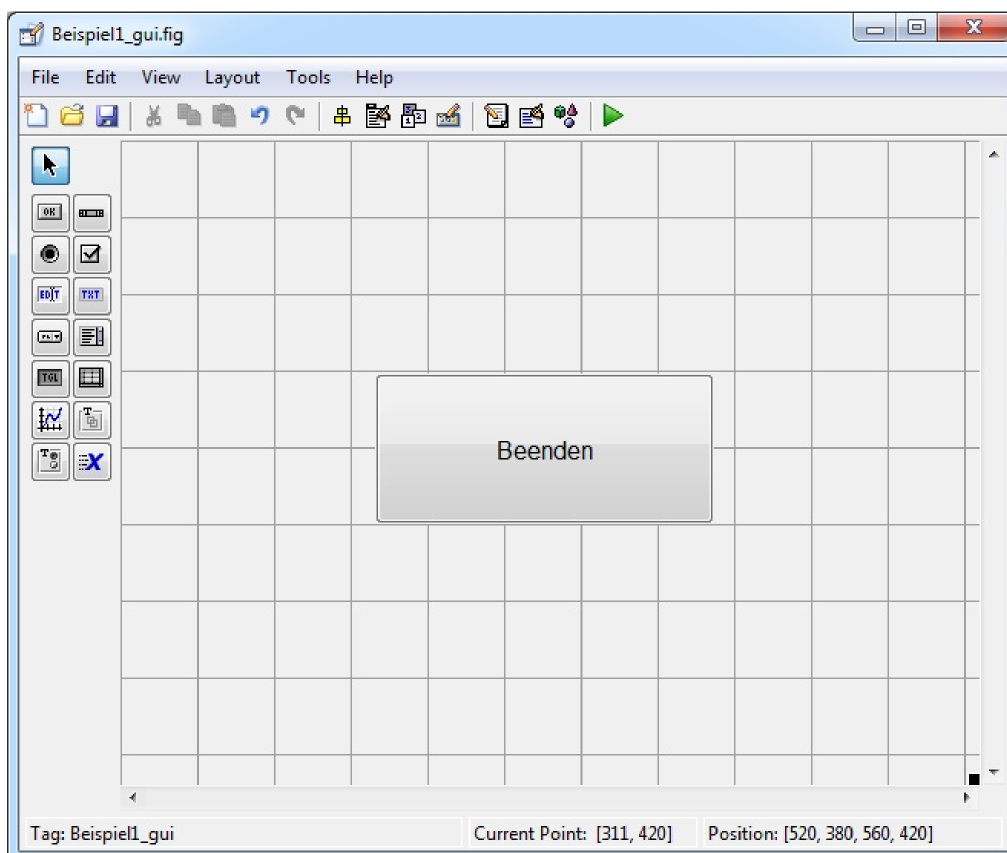

Abb. 2.1.3: Benutzeroberfläche von „Beispiel1“ im Dialog Editor mit dem Button „Beenden“

In dem zugehörigen .m-File wird folgender Quellcode für den Button *„pushbutton1“* erzeugt und kann mit Hilfe vom **M-File Editor** bearbeitet werden. So kann man den Steuerelementen bestimmte Funktionen zuteilen, welche nach ausgewählten Aktionen ausgeführt werden.

```
% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

In diesem Quelltext repräsentiert „hObjekt“ den „handle“ vom Button von dem ein „Callback“ ausgelöst wird. Manche „Callbacks“ verwenden die Struktur „eventData“, um zum Beispiel alle Callbacks auf dieselbe Callback-Funktion zu verknüpfen. Mithilfe der „eventData“-Werte kann in der Funktion unterschieden werden, welche Callbacks zu welchen Button gehören. Diese Struktur ist derzeit nur reserviert und soll erst in einer späteren MATLAB Version in MATLAB Guide implementiert werden. Dennoch besteht bereits die Möglichkeit „eventData“-Werte an die einzelnen Komponenten von MATLAB Guide zu vergeben und sich eine „eventData“-Struktur anzulegen. (MathWorks, 2013)[1]

Die Struktur „handles“ beinhaltet die Verweise auf die jeweiligen Steuerelemente des GUI, welche über den zugehörigen Objektnamen aufgerufen werden können.

Dem Button *„pushbutton1“* wird ein Befehl zugewiesen, mit dem die grafische Oberfläche geschlossen werden kann. Zum Schließen der Benutzeroberfläche wird ein *„Close“*-Befehl verwendet, bei der die Struktur „handles“ den Verweis auf den Button *„pushbutton1“* liefert. Die Funktion des *„pushbutton1“* sieht folgendermaßen aus.

```
% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

close Beispiel1_gui
```

[1] MathWorks Documentation Center, verfügbar auf: http://www.mathworks.de/de/help/matlab/creating_guis/customizing-callbacks-in-guide.html

Im *„Beispiel1_gui“* wird demnach eine Oberfläche erstellt, welche sich mit einem **Push Button** schließen lässt und so das Prinzip der „handles“-Struktur aufzeigt. Die MATLAB Files vom *„Beispiel1_gui“* befinden sich auf der beiliegenden CD und der zugehörige Quelltext ist dem Anhang 1.4 beigefügt.

Abschließend ist zu erwähnen, dass MATLAB einen eigenen Workspace verwendet, in dem Variablen ausgelagert werden können. Um bestimmte Variablen aus dem MATLAB Workspace in Funktionen zu verwenden, können diese mit dem Befehl *„evalin“* geladen werden. In den MATLAB Workspace lassen sich die Variablen mit dem Befehl *„assignin“* auslagern.

Die einzelnen Funktionen eines GUI können auch nicht ohne Weiteres auf die Variablen anderer Funktionen zugreifen. Damit mehrere Funktionen auf dieselbe Variable zugreifen können, muss diese Variable in jeder Funktion als global deklariert sein. Der Befehl dafür lautet „global“.

2.2 Simulink

Simulink ist eine von MathWorks entwickelte Toolbox in MATLAB. Diese Toolbox dient zur Modellierung von mathematischen und physikalischen Systemen mit Hilfe von grafischen Blöcken, welche mit Verbindungslinien verknüpft werden können. Jeder Block besitzt bestimmte Eigenschaften, mit denen Gleichungssysteme aufgestellt werden können. Für die numerische Simulation stehen verschiedene Lösungsverfahren zur Verfügung, sogenannte *„Solver“*. Simulink verwendet für die Berechnung Zeitschritte (*„Time Steps“*), welche in Abhängigkeit von gewählten Einheiten zu interpretieren sind. Die Datenflüsse der einzelnen Simulationsblöcke können an jeder Stelle erfasst und ausgegeben werden. Öffnen lässt sich diese Toolbox über die Kommandozeile mit dem Befehl *„simulink“*.

2.3 Data Acquisition Toolbox (DAT)

Die „Data Acquisition Toolbox“ (DAT) ist eine Toolbox in MATLAB bzw. in Simulink und bietet Funktionen, um die MATLAB Software über ein Multifunktions-Datenerfassungsmodul mit Hardwarekomponenten zu koppeln. Somit dient die Data Acquisition Toolbox zum Erfassen von Messergebnissen über eine Messkarte oder auch zum Ansteuern von Stellmotoren. So können zum Beispiel Messergebnisse von einem Durchflussgeber[2] initialisiert werden oder es kann ein Schieber von einem Ventil angesteuert werden, um den Durchfluss im Rohr zu regeln. Diese Möglichkeiten werden im weiteren Verlauf der Arbeit anhand von Testprogrammen vorgestellt und beschrieben.

In Simulink ist die DAT im Library Browser als Toolbox eingebunden und kann in Simulationsmodellen mit anderen Blöcken verknüpft werden. Die Übertragung von Daten erfolgt anhand der Übermittlung von aufgenommenen Spannungen und kann entweder auf analogem oder digitalem Wege erfolgen. Die DAT fungiert dabei als ADU bzw. DAU[3]. In der DAT stehen zur Kopplung diverse Komponenten zur Verfügung, über die mit einer Messkarte erfasste Daten in ein Simulationsmodell übertragen werden können.

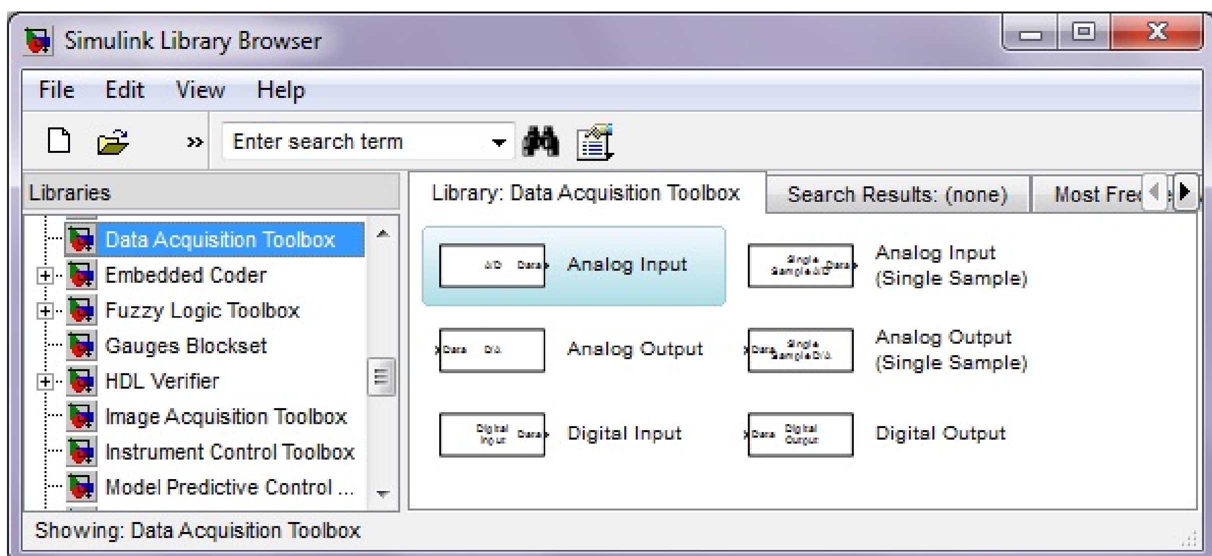

Abb. 2.3.1: Data Acquisition Toolbox im Simulink Library Browser mit den zur Verfügung stehenden Komponenten

[2] Zur Durchflussmessung wurde ein IDM (induktiver Durchflussmesser) mit Analoganzeige verwendet.

[3] Ein Analog-Digital-Umsetzer (ADU) setzt analoge Eingangssignale in digitale Daten um. Das Gegenstück ist ein Digital-Analog-Umsetzer (DAU) und setzt digitale Signale oder einzelne Werte in analoge Signale um.

Für analoge Datensätze, also stufenlose Signale, stehen ein **Analog Input**-Block und ein **Analog Output**-Block zur Verfügung. Somit können Daten während eines laufenden Simulationsmodells vom **Analog Input**-Block aufgenommen bzw. vom **Analog Output**-Block an bestimmte Hardwarekomponenten übermittelt werden. Dieser Prozess kann entweder synchron oder asynchron ablaufen. Bei einer asynchronen Datenübertragung wird ein Zeitpuffer implementiert. Auch ist es möglich von analogen Signalen einzelne Datensätze („*Single Samples*“) zu erfassen bzw. zu übertragen. Dazu stehen die Blöcke **Analog Input (Single Sample)** und **Analog Output (Single Sample)** zur Verfügung. Über diese Blöcke können die Daten nur synchron, also ohne Zeitverzögerung, übermittelt werden.

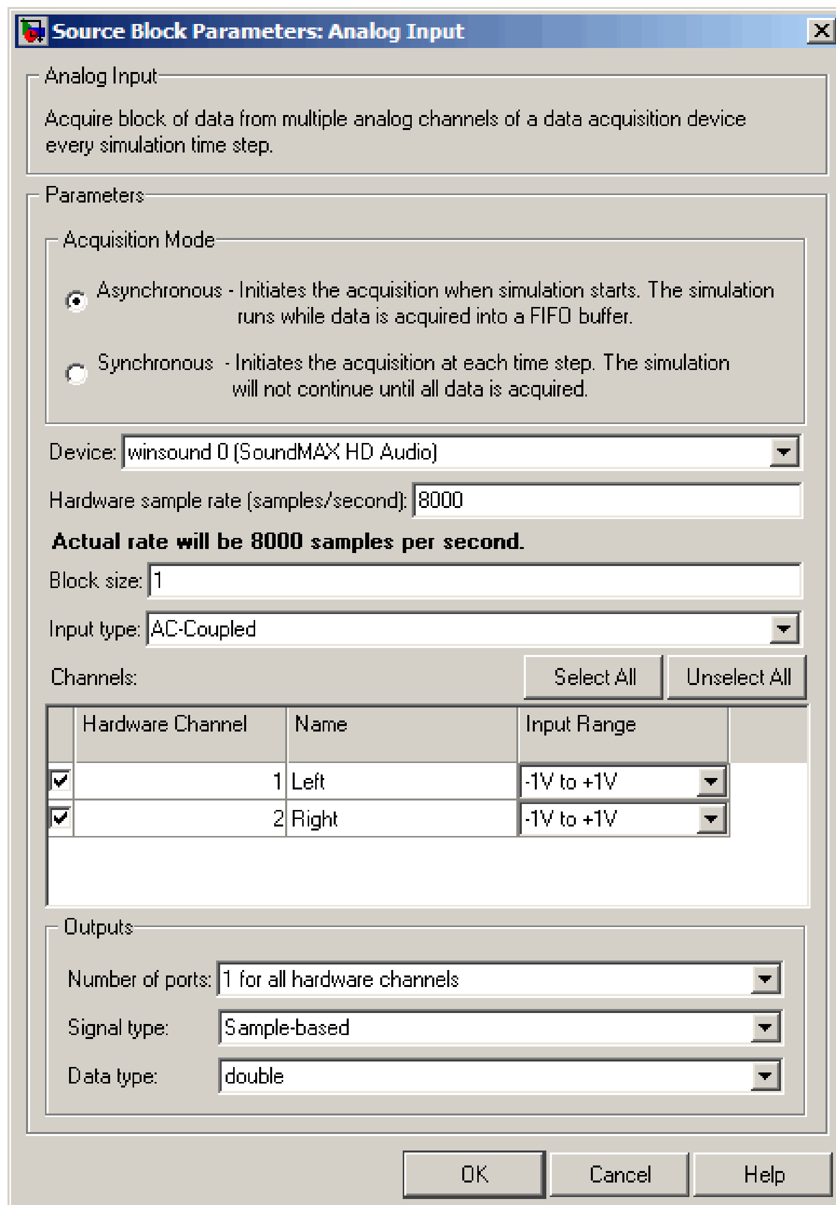

Abb. 2.3.2: Parameterfenster vom „Analog Input“-Block

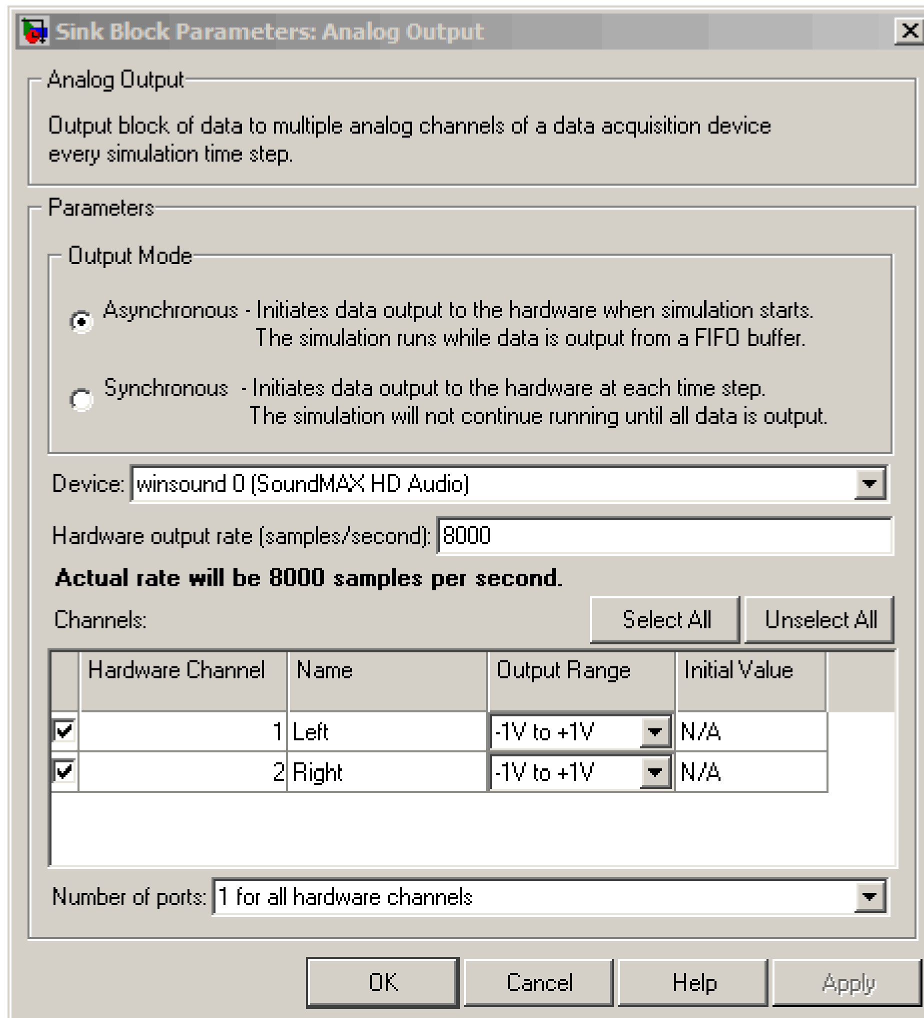

Abb. 2.3.3: Parameterfenster vom „Analog Output“-Block

In den Abbildungen 2.3.2 und 2.3.3 sind die Parameterfenster vom **Analog Input**-Block und **Analog Output**-Block dargestellt[4]. Über diese beiden Blöcke können mehrere Hardware Channel einer Messkarte gleichzeitig erfasst werden. Wenn mehrere Hardware Channel benutzt werden, können jedem Channel eigene Ports zugeteilt werden, um deren Datensätze in einem Simulationsmodell einzeln zu verarbeiten. Die *„Input Range“* bzw. *„Output Range“* ist die Begrenzung für die Spannungen, welche von der Messkarte aufgenommen oder an die Messkarte übermittelt werden. Die *„Sample Rate“* gibt an, wie viele Daten pro Sekunde aufgenommen bzw. übergeben werden sollen. Über die *„Sample Rate“* wird somit die Auflösung für analoge Datensätze bestimmt.

[4] Zur Veranschaulichung der Parameterfenster wurde eine Soundkarte mit dem Treiber *„SoundMAX HD Audio“ verwendet. Die DAT kann über die Soundkarte Tonsignale aufnehmen bzw. an die Soundkarte übergeben.*

Da ein Computer nur mit digitalen Daten arbeitet, müssen analog erfasste Daten in digitale Daten umgewandelt werden. Heutzutage sind Computer aber so leistungsstark, dass analoge Signale sehr hoch aufgelöst werden können.

Bei der Erfassung von Daten kann über die Einstellung *„Signal type“* bestimmt werden, ob Daten auf Grundlage von Samples oder der Simulationszeit erfasst werden sollen. Die Datenstruktur kann über die Einstellung *„Data Type“* bestimmt werden. Über *„Input Type“* kann ausgewählt werden auf welchen Bezugspunkt die Messdaten bzw. die davon abgeleiteten Spannungen bezogen werden sollen (*„single ended“*[5] oder *„AC-Coupled“*[6]). Für die Testversuche und Messungen wird für die Aufnahme der Daten der Massepunkt (0 V) als Bezugspunkt gewählt. Von der DAT werden die Daten so in das Simulationsmodell implementiert, dass sie vom Typ „double“[7] sind.

In der Abbildung 2.3.4 ist eine Sinusfunktion dargestellt, die von der DAT jeweils hoch aufgelöst und niedrig aufgelöst erfasst wurde.

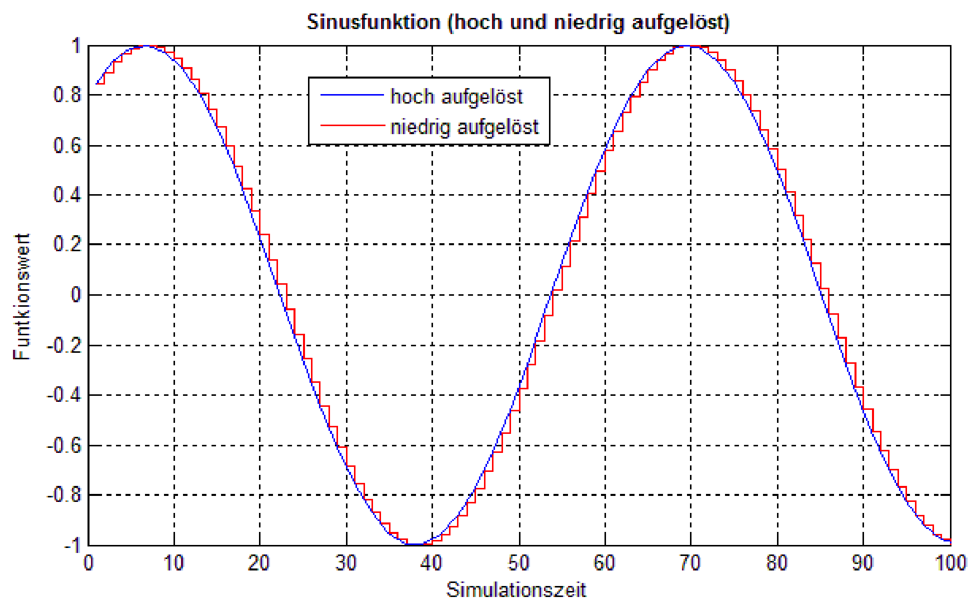

Abb. 2.3.4: Darstellung einer Sinusfunktion (hoch und niedrig aufgelöst)

[5] Single ended = Spannungsmessung über einen Bezugspunkt (meist Massepunkt = 0V)

[6] AC-Coupled = Differenz der Spannungen von zwei Eingängen an der Messkarte

[7] Double = 16 Bit-Zahl

Für eine digitale Übertragung werden die Komponenten **Digital Input** und **Digital Output** benutzt. So lassen sich binäre Datensätze in Vektorform zwischen Simulink und einer Messkarte übertragen. Der **Digital Input**-Block nimmt zu jedem Simulationszeitschritt die digitalen Daten der Messkarte auf und erfasst diese synchron zum realen Prozess. Ungepuffert wird aus diesen Daten ein binärer Vektor erstellt, mit dem Simulink während einer laufenden Simulation rechnen kann.

Der **Digital Output**-Block überträgt hingegen digitale Daten an gekoppelte Hardwarekomponenten, welche über Simulink erzeugt werden können. Dieser Block gibt zu jedem Simulationszeitschritt die aktuellste Reihe von Daten an die Hardware synchron zur Simulation weiter. Die Ausgangsdaten liegen dabei immer als binärer Vektor vor und sind ungepuffert.

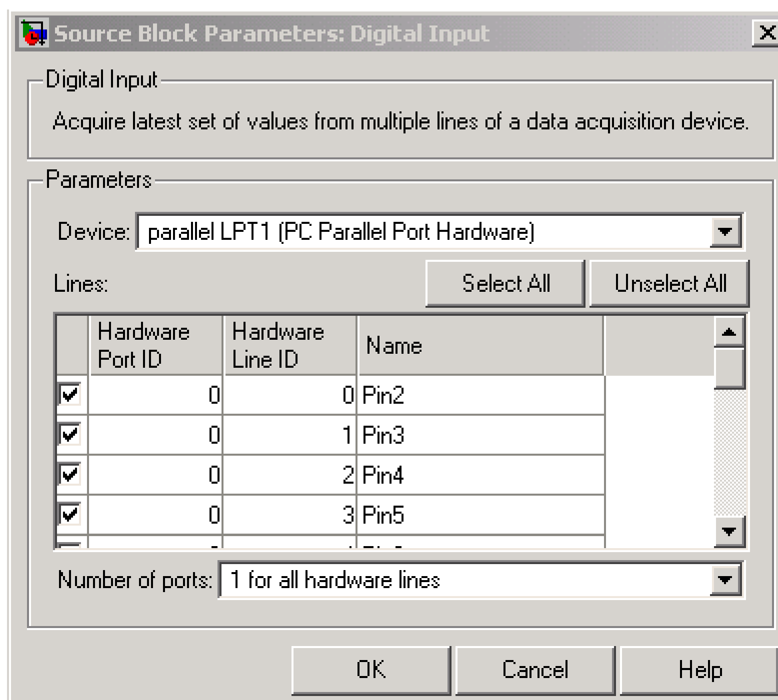

Abb. 2.3.5: Parameterfenster vom „Digital Input“-Block mit Auflistung der verfügbaren „Hardware Channels“

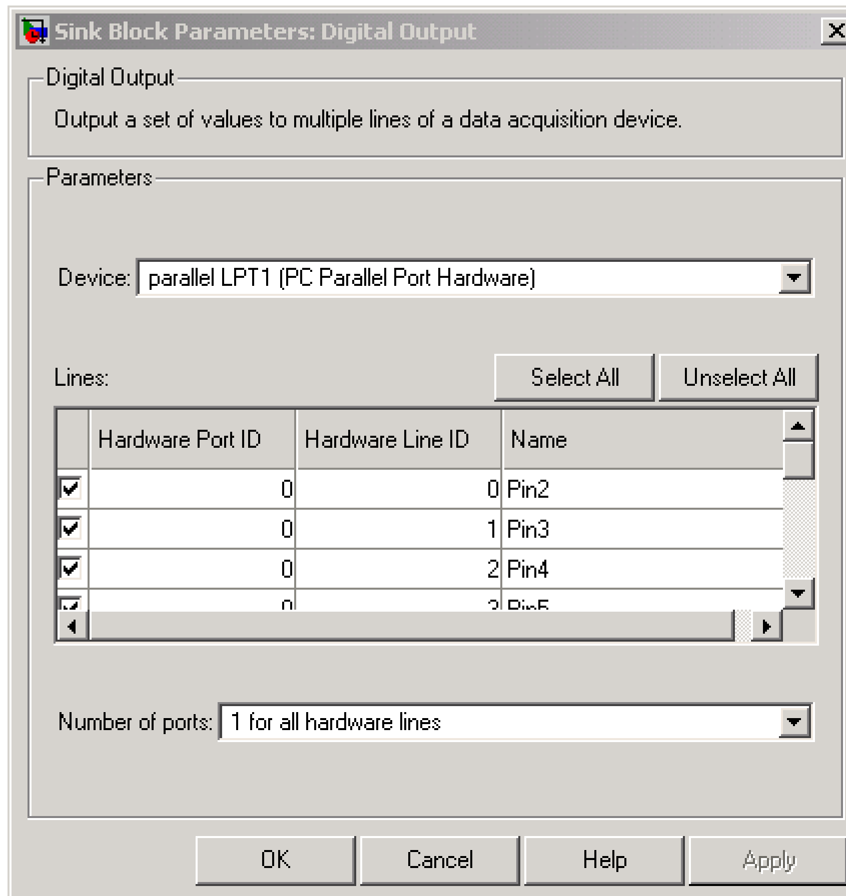

Abb. 2.3.6: Parameterfenster vom „Digital Output"-Block mit Auflistung der verfügbaren „Hardware Channels"

In den Abbildungen ist zu erkennen, dass jede *„Hardware Line"* eine eigenständige ID (Identifikationsnummer) zugeteilt bekommt, welche separat angesprochen werden kann. Zudem besteht die Möglichkeit, dass allen *„Hardware Lines"* ein eigener Anschluss im Simulationsmodell zugeordnet wird oder alle über denselben Port gekoppelt werden.

2.4 Multifunktions-Datenerfassungsmodul

Im Institut für Hydromechanik und Wasserbau wird ein Multifunktions-Datenerfassungsmodul vom Typ NI PCI-MIO-16E-4 *(National Instruments)* verwendet und dient als Schnittstelle zur Übertragung von Daten zwischen der MATLAB Software und der Messhardware. Die Messkarte trägt aktuell die Bezeichnung PCI-6040 E und verwendet die Treibersoftware NI-DAQmx[8]. Zur Datenübermittlung stehen 16 Analogeingänge und zwei

[8] In Simulink wird der Treiber mit *„nidaq Dev1" angezeigt.*

Analogausgänge mit einer 12-bit Auflösung sowie acht Digital-I/O-Kanäle und zwei Counter/Timer mit 24-bit-Auflösung zur Verfügung. Unter Verwendung eines Kanals besitzt die Karte eine Übertragungsrate von 500 kS/s. Werden mehrere Kanäle belegt, liegt die Übertragungsrate bei 250 kS/s. Vom Hersteller wird eine nahtlose Integration mit LabView garantiert. (National Instruments Germany GmbH, 2013)

Abb. 2.4.1: Multifunktions-Datenerfassungsmodul NI PCI-MIO-16E-4 **(National Instruments Germany GmbH, 2013)**

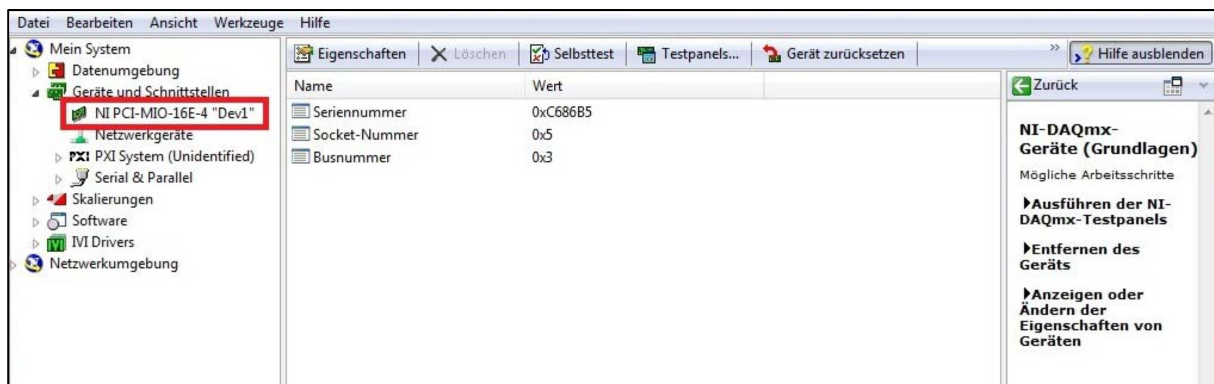

Abb. 2.4.2: Anzeige des Measurement & Automation Explorer von dem Multifunktions-Datenerfassungsmodul NI PCI-MIO-16E-4 (National Instruments)

Da das Multifunktions-Datenerfassungsmodul nicht über direkte Signalverbindungen verfügt, wird ein Anschlussblock benötigt, welcher als Schnittstelle zwischen Sensoren bzw. Signalen und dem Modul fungiert. Ein Anschlussblock bietet einen einfachen Zugriff auf die Ein- und Ausgänge von Messkarten. Für die Kopplung von Hardware mit Software wird im

Wasserbaulabor ein ungeschirmter 68-Pin-I/O-Anschlussblock vom Typ NI CB-68LP genutzt. Über die 68 Schraubklemmenanschlüsse können Messsensoren oder Steuereinheiten mit der Messkarte verknüpft werden. Eine Darstellung der Kanalbelegung vom Data Acquisition-Board befindet sich im Anhang 1.3.

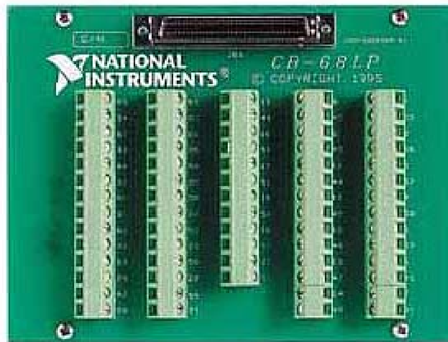

Abb. 2.4.3: Data Acquisition-Board NI CB-68LP **(National Instruments Germany GmbH, 2013)**

3. Implementierung einer Benutzeroberfläche

Dieses Kapitel ist wie ein Tutorial zum Einstieg in die „MATLAB Guide“ Programmierung aufgebaut und soll einen Einblick in die Erstellung von grafischen Oberflächen geben, welche für die Testprogramme erstellt werden. Anhand von Beispielen wird beschrieben, wie ein GUI aufgebaut werden kann, um Simulationsmodelle zu steuern und auszuwerten.

Dazu wird ein Simulationsmodell erstellt, welches eine Sinuskurve mit Verstärkungsmöglichkeit erzeugt. Anschließend soll die Simulation in ein MATLAB Guide implementiert werden, mit dessen Hilfe sich die Simulation über eine grafische Benutzeroberfläche steuern bzw. auswerten lässt. Damit die einzelnen Skripte miteinander kommunizieren können, muss darauf geachtet werden, dass sich alle verwendeten Dateien im selben Ordner befinden und dieser Ordner in MATLAB aktiviert ist.

3.1 Erstellen des Simulationsmodells

Zur Beschreibung der Erstellung von grafischen Oberflächen wird in Simulink ein neues Modell mit dem Namen *„Beispiel2_sim“* erstellt. Anhand einer erzeugten Sinuskurve soll gezeigt werden, wie Daten zwischen Simulink und MATLAB Guide ausgetauscht werden können. Durch diese Verknüpfung können später laufende Simulationen gesteuert oder Ergebnisse der Simulationen in Echtzeit auf der grafischen Benutzeroberfläche dargestellt werden.

Um eine Sinuskurve zu erzeugen, wird aus dem Simulink Library Browser der Block **Sine Wave** mit den voreingestellten Standardwerten verwendet. Hinzu kommen ein **Gain-**Block und ein **Scope**, zum Verstärken der Funktion und zum Darstellen der Simulationsergebnisse. Im **Scope** wird festgelegt, dass die eingehenden Daten unter dem Namen *„SineWave“* im Workspace als Array gespeichert werden sollen. Daraufhin lassen sich die Ergebnisse im GUI als Graph darstellen.

Die Daten werden aber erst an den Workspace übermittelt, nachdem die Simulation gestoppt wurde. Im Kapitel 3.7 wird eine Variante beschrieben, wie sich Simulationsdaten auch während einer laufenden Simulation an ein GUI übergeben und in diesem darstellen lassen. Es handelt sich demnach um ein *„Real-Time Update“.*

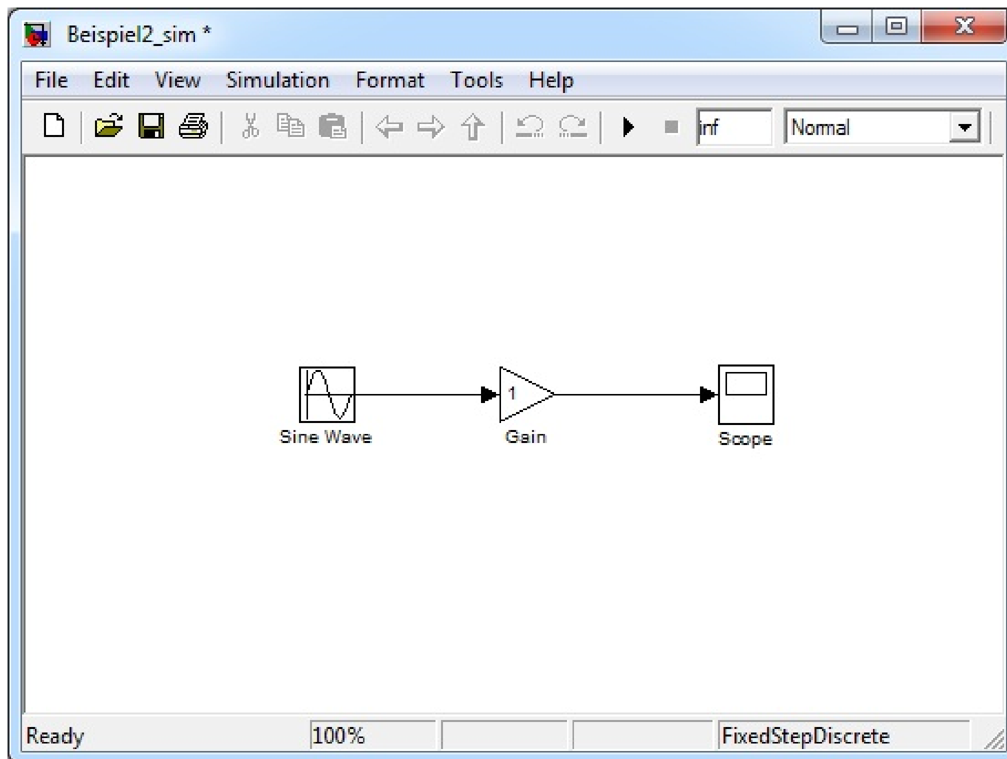

Abb. 3.1.1: Simulationsmodell „Beispiel2_sim.mdl“. Hinzugefügt wurde ein „Sine Wave“-Block zum Erzeugen der Sinusfunktion, ein „Gain“ zum Verstärken der Funktion und ein „Scope“ zum Darstellen und Speichern der Ergebnisse.

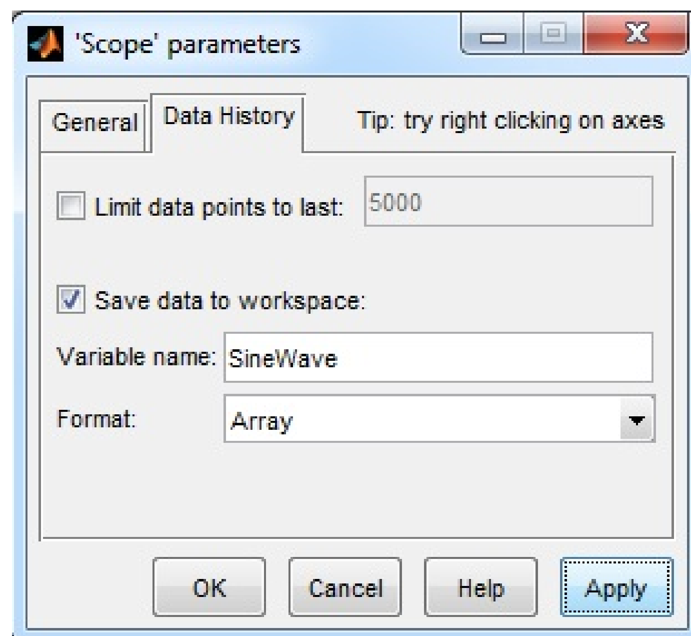

Abb. 3.1.2: Parameterfester vom „Scope“ zum Speichern der eingehenden Daten. Die Daten werden als Array (als Vektor) unter dem Namen „SineWave“ gespeichert.

Die Simulationszeit wird in diesem Beispiel auf unendlich gesetzt[9], da das Starten und Stoppen der Simulation über ein GUI gesteuert werden soll. Zur Berechnung des Modells wird der Solver *„FixedStepDiscrete“* mit einer „Sample Time“ von 1e-5 verwendet.[10] Die Simulationszeit, der Solver und die Schrittweite einer Simulation lassen sich in Simulink unter *„Configuration Parameters“* einstellen.

3.2 Erstellen der Benutzeroberfläche

Das MATLAB GUI bekommt die Bezeichnung *„Beispiel2_gui“* und sollte auch unter diesen Namen abgespeichert werden, damit keine Komplikationen aufgrund verschiedener Bezeichnungen auftreten. Im **Dialog Editor** werden folgende Objekte hinzugefügt: zwei Button, *„Startbutton“* und *„Stopbutton“*, zum Steuern der Simulation, ein Fenster für den Graphen (**Axes**) zum Anzeigen der Simulationsergebnisse, ein Textfeld (**Edit Text**) mit dem „Tag“ *„edit_Gain“* zum Festlegen des Verstärkungsfaktors und ein *„Closebutton“* zum Beenden der Simulation und der Benutzeroberfläche (Abb. 3.2.1).

[9] Um die Simulationszeit eines Modells auf unendlich zu setzen wird das zugehörige Feld *„inf“* eingetragen.

[10] Simulink berechnet die erstellten Simulationsmodelle nach Zeitschritten. Da ein einfaches Modell, wie die Erzeugung einer Sinusfunktion relativ schnell berechnet werden kann, wäre nach kurzer Zeit schon eine große Menge an Daten, also an Sinuswellen, vorhanden. Bei der Darstellung dieser Werte, wäre in einem klein aufgelösten *„Scope“* oder *„Axes“*-Feld für den Nutzer nur ein ausgefülltes Rechteck zu erkennen. Erst bei einer höheren Auflösung würde man die einzelnen Sinuskurven erkennen. Um für dieses Beispiel eine Sinuskurve für eine geeignete Darstellung im GUI zu erzeugen, wird eine Schrittweite von 1e-5 gewählt. Dadurch werden deutlich mehr Daten (1x10^5) pro Zeitschritt berechnet. Dies führt zu einer höheren Rechenauslastung, wodurch weniger Zeitschritte pro Sekunde simuliert werden.

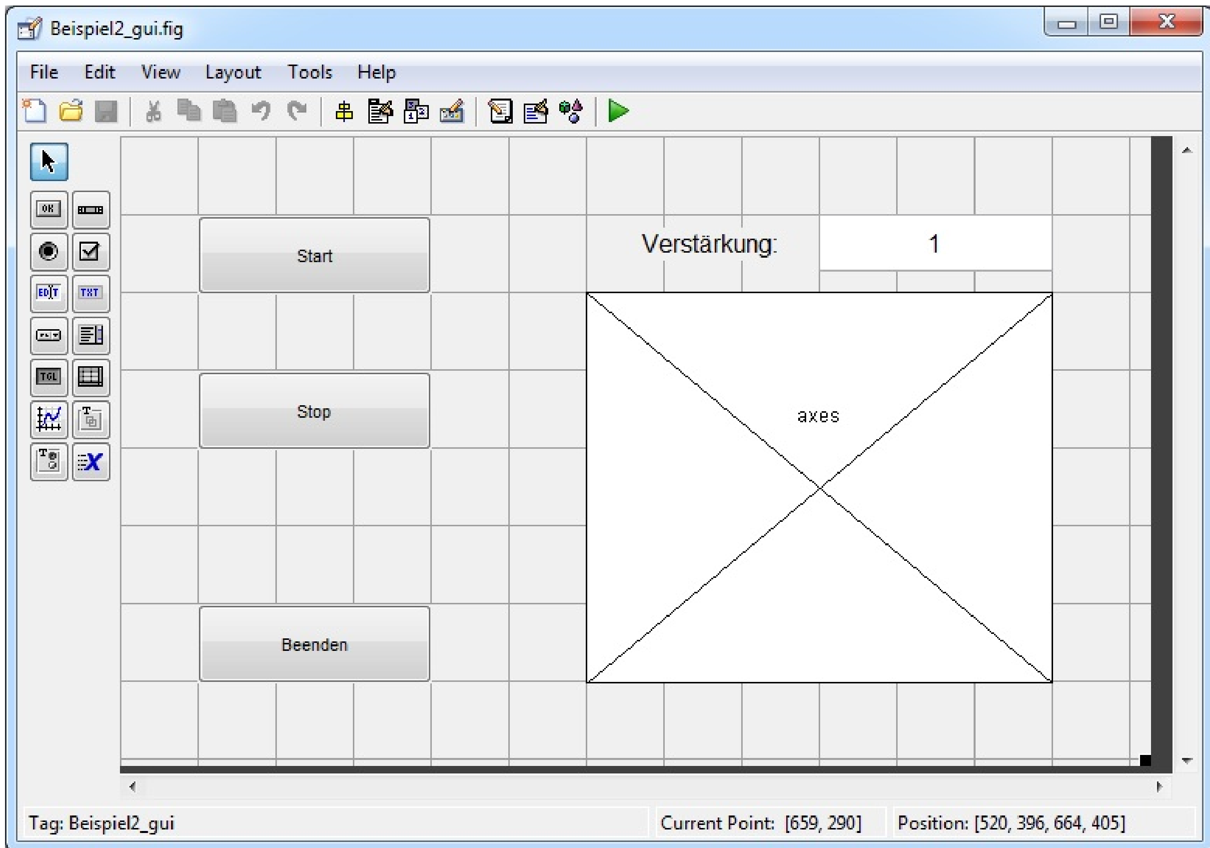

Abb. 3.2.1: Benutzeroberfläche von „Beispiel2“ im „Dialog Editor“. Über die Buttons wird das Simulationsmodell gesteuert. Über das Eingabefeld kann der Verstärkungsfaktor geändert werden. Die aufgenommenen Daten werden nach Ende der Simulation im Diagramm „axes“ grafisch dargestellt.

Das GUI kann nun über das zugehörige .m-File *„Beispiel2_gui.m“* gestartet werden. Um Programmfehler zu vermeiden, sollten GUI's generell immer über die zugehörigen .m-Files gestartet werden und nicht über die *„Figures“* selbst. Sonst könnte es beispielsweise vorkommen, dass das Simulationsmodell nicht mit geöffnet wird, obwohl die Aufforderung in der *„Opening Function“* des GUI steht.

3.3 Steuern des Simulationsmodells

Um ein Simulationsmodell gleichzeitig mit einer Benutzeroberfläche zu öffnen, verwendet man den Befehl *„open_system“*. Dieser Befehl wird mit Bezug auf das Simulationsmodell in die *„OpeningFcn“* vom GUI eingetragen. Für das Beispiel 2 sieht der Befehl wie folgt aus:

```
open_system('Beispiel2_sim')
```

Der *„Stopbutton“* soll zu Beginn der Anwendung deaktiviert sein. Dazu kann im **Property Inspector** die Einstellung *„Enable“* auf „off“ gesetzt werden.

Durch Betätigen des *„Startbutton“* soll die Simulation gestartet werden. Um Parameter, wie z.B. den Simulationsstatus, auslesen bzw. verändern zu können, werden die Befehle *„get_param“* bzw. *„set_param“* verwendet. Damit eine Callback-Funktion den Bezug zum Simulationsmodell herstellen kann, wird nach den Befehlen die Bezeichnung des Modells genannt (*„Beispiel2_sim“*) und die abzurufende Eigenschaft. Alternativ kann auch mit dem Befehl *„bdroot“* das Modell angesprochen werden, welches gerade geöffnet ist. Dies empfiehlt sich nicht unter Verwendung mehrerer Simulationsmodelle. Mit *„SimulationStatus“* wird der Simulationsstatus ausgelesen und mit *„SimulationCommand“* kann der Simulationsstatus bestimmt werden. Möglich sind dabei Starten, Stoppen oder Pausieren (*„start“*, *„stop“*, *„pause“*).

```
get_param('Beispiel2_sim','SimulationStatus');
set_param('Beispiel2_sim','SimulationCommand','start/stop/pause');
```

Durch Betätigen des *„Startbutton“* soll das Modell gestartet werden. Folglich wird die Eigenschaft von *„SimulationCommand“* auf *„start“* gesetzt.

```
set_param('Beispiel2_sim','SimulationCommand','start');
```

Gleichzeitig soll der *„Startbutton“* deaktiviert und der *„Stopbutton“* aktiviert werden, damit eine sinnvolle Steuerung des Programmes gegeben ist. Die Eigenschaft *„Enable“* muss also bei beiden Objekten geändert werden. Beim *„Startbutton“* wird die Eigenschaft auf „off“ und beim *„Stopbutton“* auf „on“ gesetzt.

```
set(handles.Startbutton,'Enable','off');
set(handles.Stopbutton,'Enable','on');
```

Der Quelltext für den *„Startbutton"* sieht folgendermaßen aus:

```
% --- Executes on button press in Startbutton.
function Startbutton_Callback(hObject, eventdata, handles)
% hObject    handle to Startbutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Turn off the Startbutton
set(handles.Startbutton,'Enable','off');
% Turn on the Stopbutton
set(handles.Stopbutton,'Enable','on');
% Start the model
set_param('Beispiel2_sim','SimulationCommand','start');
```

Analog dazu kann der *„Stopbutton"* programmiert werden. Die Simulation wird gestoppt und der *„Startbutton"* wird aktiviert bzw. der *„Stopbutton"* deaktiviert.

```
% --- Executes on button press in Stopbutton.

function Stopbutton_Callback(hObject, eventdata, handles)
% hObject    handle to Stopbutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Turn on the Startbutton
set(handles.Startbutton,'Enable','on');
% Turn off the Stopbutton
set(handles.Stopbutton,'Enable','off');
% Start the model
set_param('Beispiel2_sim','SimulationCommand','stop');
```

Der *„Closebutton"* soll das Simulationsmodell und die Benutzeroberfläche schließen. Um ein Modell bedingungslos schließen zu können, wird der Befehl *„bdclose"* verwendet. Da in diesem Beispiel der Verstärkungsfaktor von dem Modell verändert werden kann, wird man beim Versuch das Modell zu schließen von MATLAB aufgefordert erst das Modell zu speichern.

```
bdclose('Beispiel2_sim')
```

Ein bedingungsloses Schließen hat nicht nur den Vorteil, dass zum Beenden des Simulationsmodells kein Hinweisfenster zum Speichern erscheint, sondern dass auch bei einem erneuten Aufruf des Programms die festgelegten Standardwerte eingetragen sind.

```
% --- Executes on button press in Closebutton.
function Closebutton_Callback(hObject, eventdata, handles)
% hObject    handle to Closebutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Closes Simulink model unconditionally
bdclose('Beispiel2_sim')
% Closes GUI
close 'Beispiel2_gui'
```

3.4 Verändern des Verstärkungsfaktors

Zum Beginn des Programms bzw. der Simulation wird der Wert vom **Gain**-Block aus dem Modell ausgelesen und in der Variable *„value“* gespeichert. Anschließend wird dieser Wert als String in das Textfeld mit dem „Tag“ *„edit_Gain“ übergeben*. Dazu werden folgende Programmzeilen in die *„OpeningFunction“* des GUI geschrieben:

```
value = get_param(['Beispiel2_sim' '/Gain'],'Gain');
set(handles.edit_Gain,'String',value);
```

Die Identifizierung des gewünschten Parameters funktioniert also nach folgendem Prinzip: Zuerst steht die Bezeichnung des Simulationsmodells und nach einem „Slash“ („/“) folgt die Bezeichnung von dem Block, aus dem der Wert geladen werden soll. Nach diesem Prinzip, analog zur Windows Ordnerstruktur, wird folglich der Pfad zu einem gewünschten Wert aufgestellt. Würde man den Simulationsblock *„Verstaerkung“* nennen, müsste man schreiben:

```
get_param(['Beispiel2_sim' '/Verstaerkung'],..);
```

Anschließend folgt die Typbezeichnung des Parameters. Der Parameter eines **Gain**-Blocks ist vom Typ *„Gain“*. Demnach wird als Typbezeichnung *„Gain“* gewählt. Bei einem **Constant**-Block ist die Typbezeichnung *„Value“*.[11]

```
get_param(['Beispiel2_sim' '/Gain'],'Gain');
```

[11] Die Parameterbezeichnungen der Blöcke „Gain“ und „Constant“ wurden mithilfe des MATLAB Documentation Center ermittelt. (MathWorks, 2013), verfügbar auf:
http://www.mathworks.de/de/help/matlab/creating_guis/customizing-callbacks-in-guide.html

Durch diesen Programmierschritt bekommt der Nutzer den aktuellen Verstärkungsfaktor in dem Textfeld *„edit_Gain“* angezeigt und kann diesen über die Benutzeroberfläche im Simulationsmodell ändern. Nach Bestätigung des neuen Wertes mit der Taste **ENTER,** wird der Wert aus dem Textfeld *„edit_Gain“* ausgelesen und unter der Variable *„value“* abgespeichert.

```
value = get(hObject,'String');
```

Anschließend wird der Wert des **Gain**-Blocks von dem neuen Wert der Variable *„value“* ersetzt, indem die Variable *„value“* an den **Gain**-Block übergeben wird und damit den Wert *„Gain“* ersetzt.

```
set_param(['Beispiel2_sim' '/Gain'],'Gain',value)
```

Während einer laufenden Simulation würde der Wert des **Gain**-Blocks sich nicht ändern, auch wenn er erfolgreich an den Block übergeben wurde. Der Wert sitzt sozusagen in der Warteschleife. Damit während einer laufenden Simulation der **Gain**-Block aktualisiert wird und so der neue Wert in der Simulation aufgenommen wird, muss das Simulationsmodell einer Aktualisierung unterzogen werden. Folglich wird zuerst der Simulationsstatus überprüft. Bei laufender Simulation kann das Modell mit dem Befehl „SimulationCommand“ und der Eigenschaft „Update“ aktualisiert werden. Durch die Aktualisierung wird nun der in der Warteschleife sitzende Verstärkungsfaktor vom **Gain**-Block aufgenommen und in der Simulation verwendet. Es handelt bei diesem Prozess sozusagen um ein *„Real-Time Update“* von einem Simulationsblock über eine Benutzeroberfläche.

```
status = get_param('Beispiel2_sim','SimulationStatus');
if strcmp(status,'running')
	set_param('Beispiel2_sim', 'SimulationCommand', 'Update')
end
```

```
function edit_Gain_Callback(hObject, eventdata, handles)
% hObject    handle to edit_Gain (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit_Gain as text
%        str2double(get(hObject,'String')) returns contents of edit_Gain as
a double

value = get(hObject,'String');

% Update the model's gain value
set_param(['Beispiel2_sim' '/Gain'],'Gain',value)

% Update simulation if the model is running
status = get_param('Beispiel2_sim','SimulationStatus');
if strcmp(status,'running')
    set_param('Beispiel2_sim', 'SimulationCommand', 'Update')
end
```

Die *„edit_Gain_CreateFcn"* beschreibt lediglich die Darstellung des Textfeldes *„edit_Gain"*. In diesem Beispiel wurde der Hintergrund des Textfeldes mithilfe des **Property Inspector** auf die Farbe weiß eingestellt.

```
% --- Executes during object creation, after setting all properties.
function edit_Gain_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit_Gain (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

3.5 Darstellen der Ergebnisse

Zum Abschluss der Simulation sollen die Werte in der Benutzeroberfläche, also dem GUI, in einem Graphen dargestellt werden. Die erzeugte Sinusfunktion wird von einem **Scope** dargestellt und aufgrund der vorgenommenen Einstellungen im **Scope** nach Beenden der Simulation als Array an den MATLAB Workspace übergeben.

In diesem Array stehen in der ersten Spalte die Zeitschritte („Time Steps") und in der zweiten Spalte die Funktionswerte der Sinuskurve. Da dieses Array erst nach Ende der Simulation

erzeugt wird, können die Werte erst nach Betätigen des *„Stopbutton“*, in das GUI eingelesen werden. Zum Einlesen der Werte wird der Befehl *„evalin“* verwendet. Mithilfe dieses Befehls können Daten jeglicher Struktur aus Variablen ausgelesen und in einer neuen GUI Variable abgespeichert werden. In diesem Beispiel werden die Daten aus dem Array *„SineWave“* in die GUI Variable *„SineWave“* geladen. Nach dem Befehl *„evalin“* folgt der Bezug zu dem Array *„SineWave“*. Mit der Bezeichnung *„Base“* stellt man die Verknüpfung zum MATLAB Workspace her und mit dem Namen der Variable den Bezug zu den Simulationsergebnissen.

```
SineWave = evalin('Base','SineWave');
```

Im Graphen der Benutzeroberfläche sollen anschließend auf der x-Achse die Zeitschritte und auf der y-Achse die Funktionswerte dargestellt werden. Dies lässt sich mit dem Befehl *„plot“* ermöglichen. Auf den Befehl *„plot“* folgt die Bezeichnung der Variable, von der die Daten auf der x-Achse dargestellt werden sollen und anschließend die Bezeichnung der Variable für die y-Achse. Bei der Verwendung von Arrays muss angegeben werden, welche Zeile bzw. Spalte ausgelesen werden soll. Nach der Variablenbezeichnung kommen zuerst die Einträge für die Zeile und nach einem Semikolon für die Spalte. Um alle Zeilen aus einer Spalte einzulesen, wird ein Doppelpunkt verwendet. Die Programmierung zur Graphenerstellung in einem GUI ist also ähnlich zur allgemeinen MATLAB Software.

Ein Unterschied ist, dass eine *„handels“*-Struktur erzeugt werden muss („pHandles“), welche die Ausführung des *„plot“*-Befehls enthält und zusätzlich den Bezug zum Objekt herstellt, in dem der Graph dargestellt werden soll („handles.axes“). Mit der Eigenschaft „Parent“ kann der Bezug zum Objekt „handles.axes“ hergestellt werden.

```
pHandles = plot(SineWave(:,1),SineWave(:,2),'Parent',handles.axes);
```

Die beiden Programmzeilen werden der Funktion vom *„Stopbutton“* hinzugefügt, um die Ergebnisse nach der Simulation im GUI darzustellen. Der vollständige Quelltext befindet sich in der Anlage 1.5 und das Programm auf der CD unter *„Beispiel2“*.

Die Abb. 3.5.1 zeigt die grafische Benutzeroberfläche nach einer Simulation. Die Sinuskurve wird als Graph nach ca. 12 Simulationszeitschritten mit dem Verstärkungsfaktor von „1“.

Dem Nutzer bleibt nach gestoppter Simulation die Möglichkeit eine neue Simulation zu starten oder das Programm zu beenden.

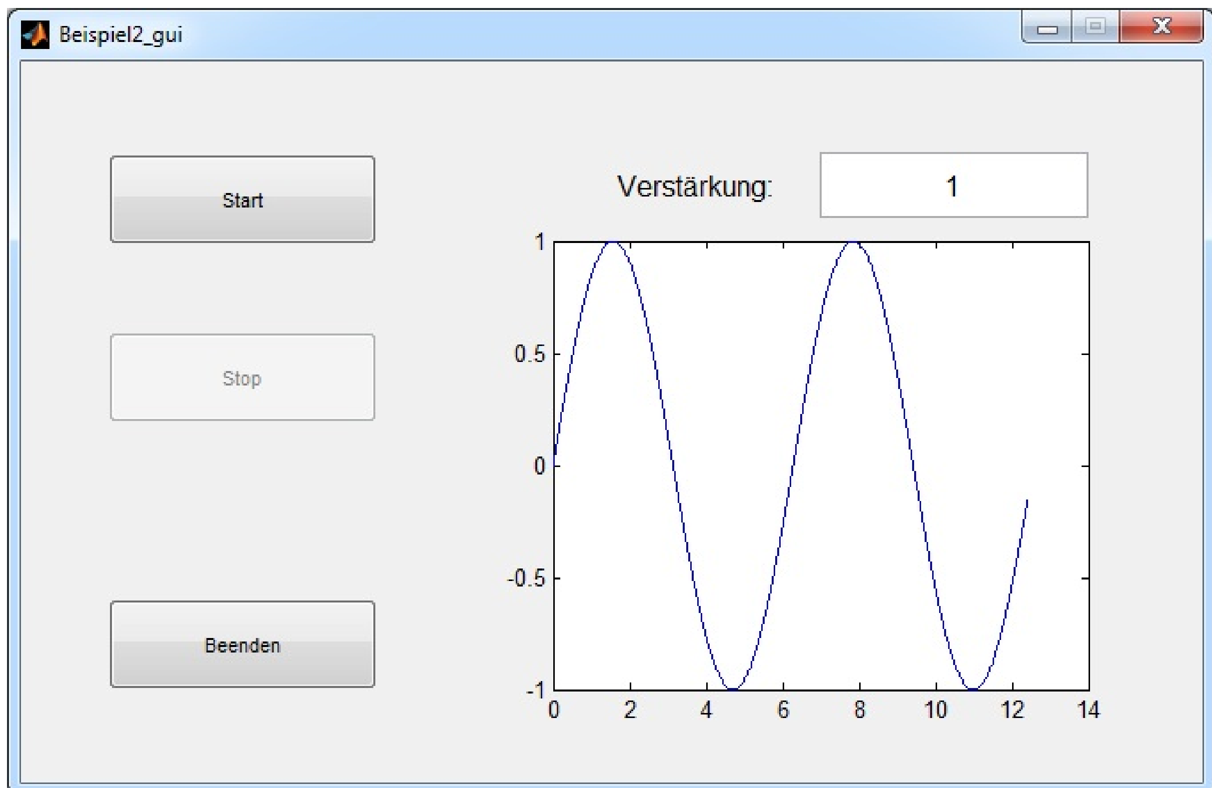

Abb. 3.5.1: Programmfenster „Beispiel2_gui“ nach einer durchlaufenden Simulation mit Verstärkungsfaktor „1“.

3.6 Alternative Steuerung des Simulationsmodells

Da in der Programmierung mehrere Wege möglich sind, wird noch eine weitere Option zur Steuerung eines Simulationsmodells erläutert. Im *„Beispiel3_gui“* wird nur noch ein Button (*„Startstopbutton“*) zum Starten und Stoppen des Modells verwendet. Dieser Button soll nach erstmaligem Betätigen, also nach dem Starten der Simulation, den String von *„Start“* auf *„Stop“* ändern und bei erneutem Betätigen die Simulation stoppen.

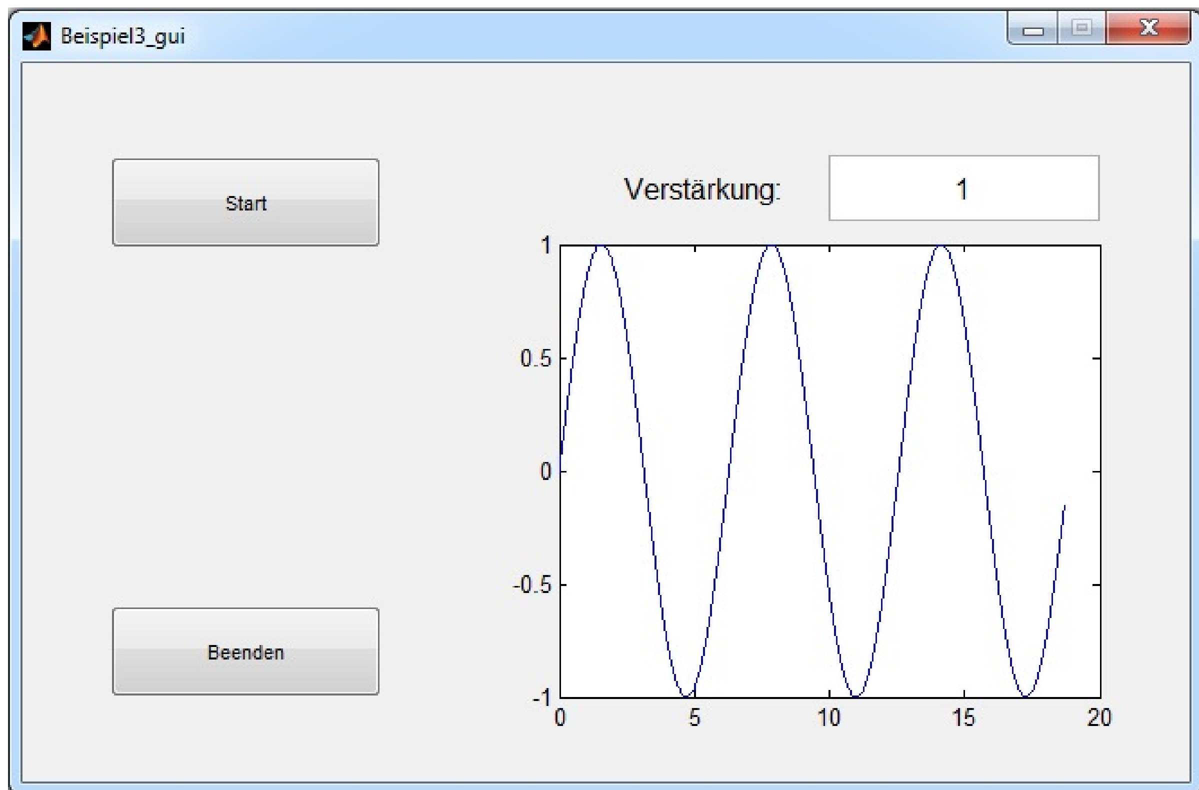

Abb. 3.6.1: Programmfenster „Beispiel3_gui" mit einem Button zum Starten und Stoppen der Simulation.

Um die Bedingung aufzustellen, dass ein Button (*„Startstopbutton")* zum Starten und Stoppen dient, werden die Strings aus dem Button ausgelesen und verglichen. Der String eines Objekts lässt sich mit folgendem Befehl auslesen:

```
get(hObject,'String');
```

Wenn der Button den String *„Start"* besitzt, soll die Simulation nach Betätigen des Button gestartet werden. Zuerst wird eine neue Variable mit der Bezeichnung *„mystring"* eingeführt. Diese Variable soll den Wert des Strings vom *„Startstopbutton"* zugeteilt bekommen. Gleichzeitig wird der Simulationsstatus von *„Beispiel3_sim"* in der Variable *„status"* als String geladen.

```
mystring = get(hObject,'String');
status = get_param('Beispiel3_sim','SimulationStatus');
```

Anschließend wird überprüft, ob der String auf dem Button *„Start“* oder *„Stop“* lautet. Mit dem Befehl *„strcmp“* lassen sich Texte miteinander vergleichen (*„compare strings“*). Wird der *„Startstopbutton“* betätigt und lautet der String auf dem Button *„Start“*, dann erfolgt die Prozedur unter der Bedingung, dass der Wert der Variable *„mystring“* gleich *„Start“* ist, und lässt die Simulation starten. Zuvor wird geprüft, ob die Simulation auch wirklich gestoppt ist. Zum Abschluss der ersten Prozedur wird noch der String auf dem Button von *„Start“* auf *„Stop“* geändert und die Programmierung für den Zustand *„Start“* ist abgeschlossen.

```
if strcmp(mystring,'Start')

    % Check the status of the simulation and start it if it's stopped
    if strcmp(status,'stopped')
        set_param('Beispiel3_sim','SimulationCommand','start')
    end

    % Update the string on the pushbutton
    set(handles.Startstopbutton,'String','Stop')
```

Für den Zustand *„Stop“* muss folglich der Quelltext angepasst werden, damit durch Betätigen des *„Startstopbutton“* unter der Bedingung, dass der String auf dem Button *„Stop“* lautet, die Simulation angehalten wird. Gleichzeitig sollen die Simulationsergebnisse in der Benutzeroberfläche dargestellt werden.

```
elseif strcmp(mystring,'Stop')

    % Check the status of the simulation and stop it if it's running
    if strcmp(status,'running')
        set_param('Beispiel3_sim', 'SimulationCommand', 'Stop')
        % Load and plot results after simulation
        SineWave = evalin('base','SineWave');
        pHandles = plot(SineWave(:,1),SineWave(:,2),'Parent',handles.axes);
    end

    % Update the string on the pushbutton
    set(handles.Startstopbutton,'String','Start')

end
```

Der vollständige Quelltext vom Beispiel 3 und dem *„Startstopbutton“* befindet sich in der Anlage 1.6.

3.7 Real-Time Plot

Zuvor wurde zur Darstellung von Simulationsergebnissen in der grafischen Benutzeroberfläche nur die Variante erläutert, dass Daten nach Ende einer Simulation aus dem Workspace geladen und im Diagramm geplottet werden.

In diesem Abschnitt wird nun beschrieben, wie Daten auch während einer laufenden Simulation an ein GUI übergeben werden können, ohne die Simulation dafür zu unterbrechen. Um die Simulationswerte zu übergeben, wird ein „EventListener" benutzt. Mit einem „EventListener" lassen sich zu jedem Zeitschritt der Simulation gewünschte Daten an MATLAB Skripte übergeben. So ist es möglich, Daten bzw. Variablen von MATLAB Skripten an GUI's weiterzuleiten bzw. diese in bestimmten Objekten des **Dialog Editors** anzuzeigen oder darzustellen.

In dem nachfolgendem Beispiel 4 soll der aktuelle Funktionswert einer erzeugten Sinuskurve in einem Textfeld des GUI angezeigt werden. Gleichzeitig sollen alle bis zum aktuellen Zeitschritt errechneten Werte in einem Graphen dargestellt werden. Es handelt sich demnach um ein *„Real-Time Plot"*.

Zuerst wird eine Benutzeroberfläche mit der Bezeichnung *„Beispiel4_gui"* erstellt. Diese soll zwei Textfelder beinhalten, ein Textfeld zum Verändern der Verstärkung und eines zum Anzeigen des aktuellen Funktionswertes der Sinusfunktion. Zusätzlich kommen noch ein Button zum Steuern der Simulation und einer zum Beenden des Programms hinzu. Ein **„Axes"**-Feld dient zum Darstellen der Simulationsergebnisse in Echtzeit. Die aktuellen Simulationswerte werden dafür zu jedem Zeitschritt an den Graphen übergeben und dargestellt. Der Graph hält nach jedem Schritt an dem zuvor dargestellten Punkt fest und zeichnet den aktuell übergebenen Wert auf.[12]

[12] Dieser Vorgang lässt sich mit dem Befehl *„hold on"* verwirklichen.

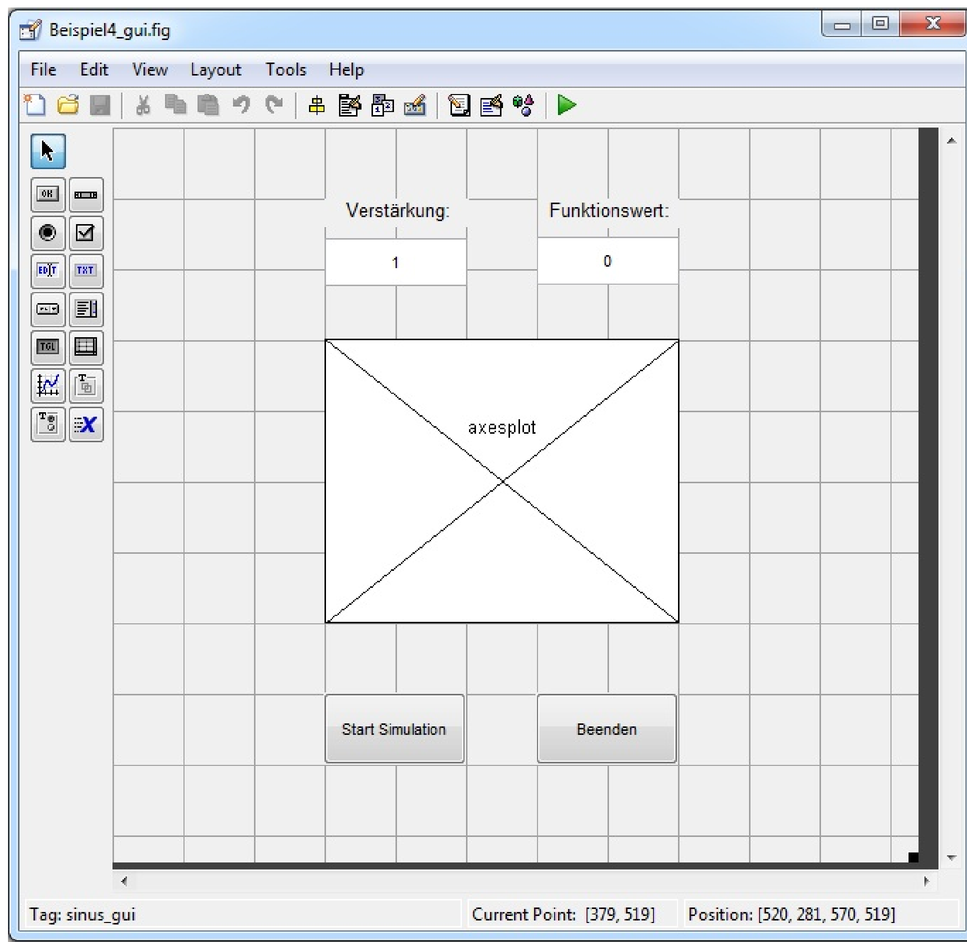

Abb. 3.7.1: Benutzeroberfläche „Beispiel4_gui" im „Dialog Editor" mit Textfeld zum Anzeigen des aktuellen Funktionswerts und „Plot"-Fenster zum Aufzeichnen aller Punkte.

Im nächsten Schritt wird ein Simulationsmodell erstellt, mit dem eine Sinusfunktion mit Verstärkungsmöglichkeit durch einen **Gain**-Block erzeugt wird. Die Simulationsergebnisse werden im Modell zur Überprüfung von einem **Scope** und einem **Display** dargestellt. Zum Übergeben der Funktionswerte wird ein **Outport**-Block mit der Bezeichnung *„SinusOut"* und der Portnummer „1" hinzugefügt. Die Simulationszeit kann während einer laufenden Simulation problemlos im GUI ausgelesen werden. Die Sinusfunktion wird mit einer Amplitude von „1" und einer Frequenz von 0,1 s^{-1} erzeugt.

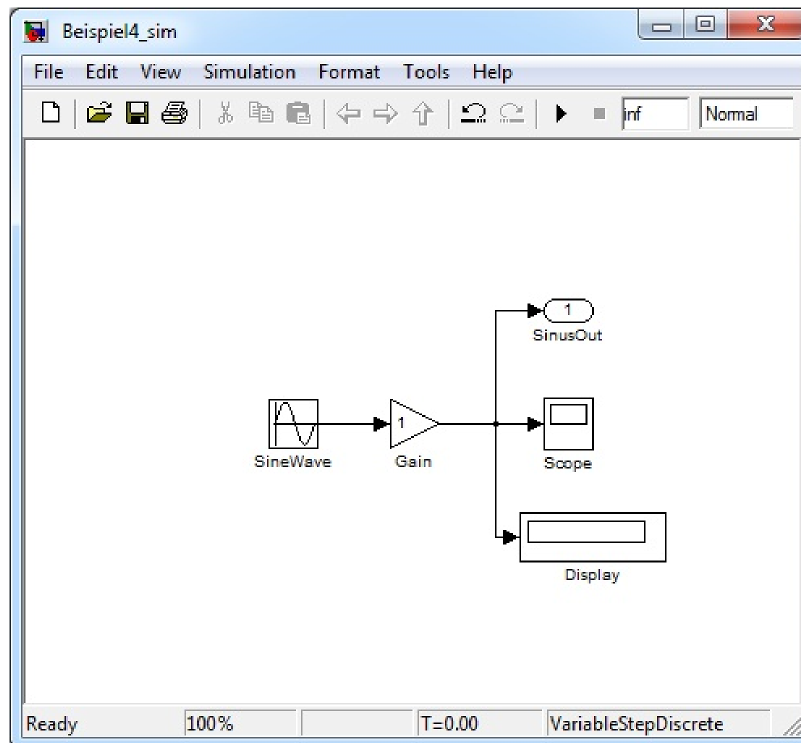

Abb. 3.7.2: Simulationsmodell „Beispiel4_sim“ mit einem Block zum Erzeugen der Sinusfunktion, einem „Gain“ zur Verstärkungsmöglichkeit und einem „Outport“-Block zum Übergeben der Werte mithilfe des „EventListener“.

Der „EventListener“ wird als Callback des Simulationsmodells in der *„StartFunction“* erstellt. Ein „EventListener“ funktioniert nach dem Prinzip, dass er Werte nach einem festgelegten Event von einem Simulationsblock aufnimmt. In diesem Beispiel sollen die Daten von dem **Outport**-Block mit der Bezeichnung *„SinusOut“* aufgenommen und übergeben werden. Als Event wird festgelegt, dass die ausgehenden Werte dieses Blocks übermittelt werden sollen. Dieses *„Event“* nennt sich *„PostOutputs“*. Die Werte sollen anschließend an ein MATLAB Skript übermittelt werden. Dazu wird ein zweites Skript mit dem Namen *„updategui“* genutzt, welches als „Listener“ dient und anschließend die Daten an das GUI übergibt. Die Verwendung eines weiteren Skriptes ist notwendig, da Daten permanent von einem Simulationsmodell übergeben werden und dadurch das Skript bei jeder Übergabe aufgerufen wird. Würde man die Daten direkt an ein GUI übergeben, würde sich die Oberfläche ständig neu aufrufen und für den Benutzer „scheinbar flackern“.

Für die Übergabe müssen die „handles“ des GUI für Simulink sichtbar gemacht werden, da diese standardmäßig als versteckt eingestellt sind.

Dies ist mit folgendem Eintrag in der *„StartFcn“* des Simulationsmodells möglich:

```
set(0,'ShowHiddenHandles','on');
```

Anschließend werden die Argumente zum Übergeben der Simulationswerte bestimmt. Der Block, von dem der „Listener“ die Werte beziehen soll, ist der **Outport**-Block mit der Bezeichnung *„SinusOut“*. Dieses Argument muss zu Beginn der Simulation erzeugt werden und wird im Workspace ausgelagert. Der „EventListener“ kann daraufhin den Bezug zum Block herstellen.

```
blk = 'Beispiel4_sim/SinusOut';
```

Ein „Event“ wird nach bestimmten Bedingungen spezifiziert, also wie Werte aus einem Block von einem „EventListener“ aufgenommen werden sollen. Im *„Beispiel4“* soll das „Event“ des „Listeners“ die Übergabe der Werte sein, nachdem der **Outport**-Block durchlaufen wurde.

```
event = 'PostOutputs';
```

Die Werte sollen also immer zu dem Zeitpunkt übermittelt werden, nachdem die Berechnung in der Prozessstruktur eines Blocks durchgeführt wurde. Bei einem **Outport**-Block macht es prinzipiell keinen Unterschied, ob vor oder nach der Berechnung Werte übermittelt werden, da die Werte durch den **Outport**-Block nicht verändert bzw. beeinflusst werden. Beispielsweise wären bei einem **Gain**-Block die Werte vor der Berechnung unterschiedlich zu denen, die durch die Bearbeitung des Blocks (Multiplikation mit einem Faktor) nach der Berechnung vorliegen.

Tabelle 3.7-1: Liste der „Event“-Bedingungen zum Übertragen von Daten während einer laufenden Simulation (MathWorks, 2013)[13]

Event	Occurs…
‘PreDerivates‘	Before a block’s Derivates method executes
‘PostDerivates’	After a block’s Derivates method executes
‘PreOutputs’	Before a block’s Outputs method executes
‘PostOutputs’	After a blocks’s Outputs method executes
‘PreUpdate’	Before a block’s Update method executes
‘PostUpdate’	After a block’s Update method executes

Als „Listener“, also Empfänger der zu übermittelnden Daten, dient ein zusätzliches MATLAB Skript mit dem Namen *„updategui.m“*. Dieses Skript wird zu jedem Zeitschritt von Simulink angesprochen und empfängt die Daten von dem **Outport**-Block *„SinusOut“*. Bei jeder Übergabe eines Wertes wird demnach das Skript geöffnet und lässt den Quelltext vollständig durchlaufen. Um das Skript aufzurufen, wird folgendes Argument erstellt:

```
listener = @updategui;
```

Der „EventListener“ selbst, welcher den Ablauf zum Übermitteln der Daten regelt, muss auch in der „StartFunction“ erstellt werden und wird ebenfalls in den Workspace ausgelagert. Der „EventListener“ enthält die Argumente, welche zuvor definiert wurden und lädt diese aus dem Workspace.

```
EventListener = add_exec_event_listener(blk, event, listener);
```

Die „StartFunction“ des Simulationsmodells sieht für das Beispiel 4 unter Verwendung eines „EventListeners“ folgendermaßen aus (Abb. 3.7.3):

[13] MathWorks Documentation Center verfügbar auf: http://www.mathworks.de/de/help/simulink/slref/add_exec_event_listener.html?searchHighlight=add_exec_event_listener

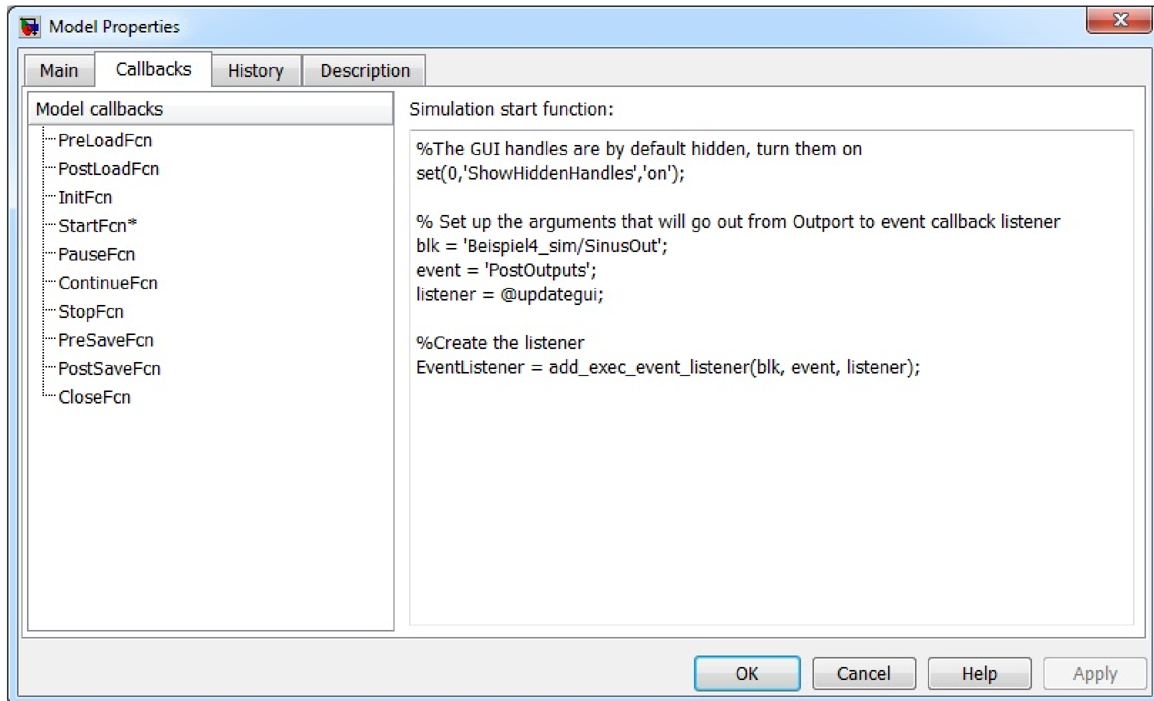

Abb. 3.7.3: „StartFcn" von „Beispiel4_sim" zur Erstellung des „EventListener"

Der „EventListener" wird während einer laufenden Simulation mit einem roten Antennensymbol an dem verwendeten Block angezeigt. Derzeit ist es unter Ausschluss von Komplikationen noch nicht möglich mehrere „Eventlistener" in einem Simulationsmodell zu verarbeiten, da sie sich auch unter Verwendung verschiedener Bezeichnungen selbst überschreiben würden.

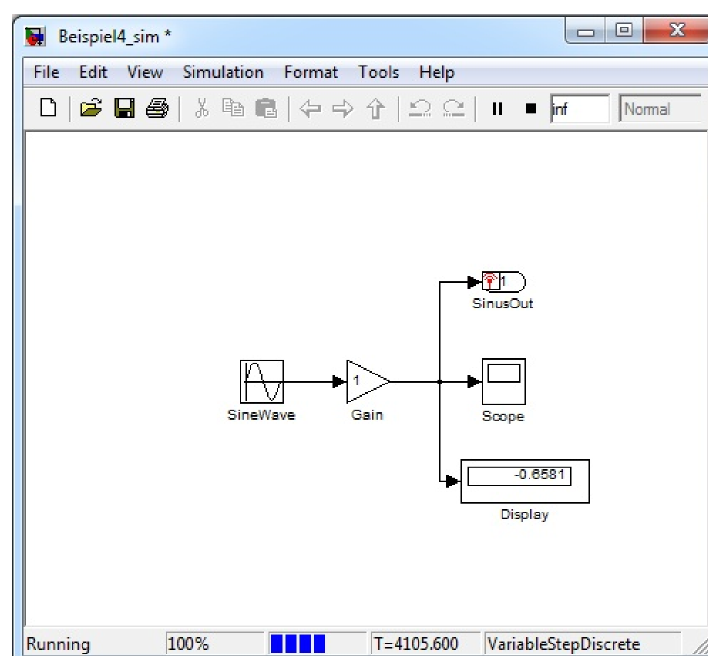

Abb. 3.7.4: Darstellung des „EventListener" am „Outport"-Block "SinusOut"

Der aktuelle Wert von dem Block *„SinusOut“* wird von dem Skript *„updategui.m“* aufgenommen. Dazu wird eine Variable mit der Bezeichnung *„rto1“* erzeugt, welche als Hilfsvariable dienen soll, um einen übersichtlichen Quelltext zu erzeugen. Diese Variable enthält die Aufforderung den Bezug zum „RuntimeObject“ vom Block „SinusOut“ herzustellen bzw. den Wert von diesem Block einzulesen und so an das Skript *„updategui.m“* zu übergeben.

```
rto1 = get_param('Beispiel4_sim/SinusOut','RuntimeObject');
```

Im nächsten Schritt wird der Wert von diesem „RuntimeObject“ an die Variable *„str“* übergeben und gleichzeitig in Textform umgewandelt (*„num2str“*). Dadurch lässt sich der Wert später in einem Textfeld darstellen.

Der Block *„SinusOut“* besitzt im Simulationsmodell nur einen Eingang. Dieser Eingang ist mit *„InputPort(1)“* bezeichnet. Aus diesem Port wird der eingehende Wert des „RuntimeObjects“, welcher vom Typ *„Data“* ist, eingelesen.

```
str = num2str(rto1.InputPort(1).Data);
```

Um diesen Wert in dem Textfeld mit dem „Tag“ *„edit_display“* darzustellen, muss das Objekt zunächst in das MATLAB Skript eingelesen bzw. das „handle“ dazu erzeugt werden. Dies lässt sich mit dem Befehl *„findobj“* ermöglichen. Das Textfeld bzw. dessen „handle“ kann daraufhin an die Variable *„display“* gegeben werden.

```
display = findobj('Tag','edit_display');
```

Als Nächstes kann von dem Textfeld die Eigenschaft *„string“* verändert werden, um so den angezeigten Wert vom Textfeld mit dem aktuellen Wert der Simulation, welcher in der Variable *„str“* gespeichert ist, zu ersetzen.

```
set(display,'string',str);
```

Darüber hinaus sollen die Simulationswerte in einem Graphen dargestellt werden. Die Simulationszeit soll auf der x-Achse angezeigt werden und die Funktionswerte der Sinusfunktion auf der y-Achse. Dazu müssen zunächst zwei Variablen (hier: mit der Bezeichnung *„XData“* und *„YData“*) erstellt werden. Den beiden Variablen werden anschließend die zugehörigen Werte zugeteilt. *„XData“* erhält die Simulationszeit, welche direkt aus dem Simulationsmodell ausgelesen werden kann. Die Funktionswerte der Sinusfunktion werden über den „EventListener“ von dem Block *„SinusOut“* an die Variable *„YData“* übergeben. Dazu muss wieder der Bezug zum Block sowie zu dessen Port und Datentyp erfolgen.

```
XData = get_param('Beispiel4_sim','SimulationTime');
YData = rto1.InputPort(1).Data;
```

Die Variablen können anschließend an den Workspace übergeben werden, enthalten aber immer nur den aktuellen Wert der Simulation. Der Wert der Variablen ändert sich demnach automatisch, weil die Variable bei jedem Durchlauf des Skriptes neu abgespeichert wird. Die Auslagerung der Variablen an den Workspace ist nicht notwendig für einen *„Real-Time Plot“,* zeigt aber, dass so weitere Möglichkeiten für eine Programmierung bzw. einer alternativen Darstellung von Simulationsergebnissen möglich sind.

```
assignin('base','XData',XData)
assignin('base','YData',YData)
```

Normalerweise genügt es den Bezug zu dem Graphen *„plotaxes“* über eine Hilfsvariable (*„guiplot“*) herzustellen, um anschließend die Variablen *„XData“* und *„YData“* zu plotten. Der Befehl *„hold on“* dient zum Erhalt der zuvor gezeichneten Punkte.

```
guiplot=findobj(0, 'Tag','axesplot');
plot(guiplot,XData,YData)
hold on
```

Für das Beispiel 4 muss „guiplot" als persistente Variable[14] deklariert werden, da während der Ausführung des Beispiels der Bezug zum Diagramm „axesplot" verloren geht. Mit einer „if"-Schleife kann überprüft werden, ob sich ein Wert in der Variable *„guiplot"* befindet und der Bezug noch besteht. Falls nicht, wird der Bezug während der Schleife wiederhergestellt.

```
persistent guiplot
if isempty(guiplot)
    guiplot=findobj(0, 'Tag','axesplot');
end
```

Anschließend können die Simulationswerte aus den Variablen *„XData"* und *„YData"* im Objekt *„axesplot"* dargestellt werden.

```
plot(guiplot,XData,YData)
```

Nach dem Befehl zum Plotten folgt der Befehl *„hold on"*, um an den gezeichneten Punkten des Diagramms festzuhalten. Sonst würden die zuvor gezeichneten Punkte überschrieben werden und man würde immer nur den aktuellen Punkt im Graphen sehen.
Mit *„grid on"* lassen sich zur besseren Veranschaulichung des Graphen automatisierte Gitterlinien erzeugen.

Damit ist die Programmierung für ein *„Real-Time Plot"* prinzipiell abgeschlossen. Ein wichtiger Punkt muss aber beachtet werden. Beim erneuten Starten der Simulation, ohne dass das Programm zuvor geschlossen wurde, würden durch den Befehl *„hold on"* die Punkte im Graphen erhalten bleiben. Um diese Punkte zu löschen, wird der Befehl *„cla"* verwendet. Dieser Befehl steht für *„clear current axes"* und bedeutet, dass alle Daten aus dem aktuellen Graphen gelöscht werden. Beispielsweise kann der Bezug zu den Graphen *„axes1"* und *„axes2"* mit folgender „handels"-Struktur hergestellt werden.

```
cla (handles.axes1,'reset')
cla (handles.axes2,'reset')
```

[14] Der Sinn einer persistenten Variable ist es, dass die Variable ihren Wert in dem Workspace der Funktion beibehält. Eine persistente Variable ist sozusagen das Gegenstück zu einer globalen Variable.

Die Funktion zum Leeren von Graphen bei einem erneuten Start einer Simulation muss dem *„Startstopbutton“* hinzugefügt werden, damit die Punkte zum Start der Simulation gelöscht werden. Wird nur ein Graph im GUI verwendet, reicht auch der Ausdruck *„cla“*.

Die Befehle *„varargout“* bzw. *„varargin“* dienen lediglich zum Übergeben bzw. Übernehmen von Variablen. Der Quelltext des MATLAB Skriptes *„updategui.m“* sieht im Gesamten folgendermaßen aus:

```
function varargout = updategui(varargin)

%create a run time object that can return the value of the outport block's
%input and then put the value in a string

rto1 = get_param('Beispiel4_sim/SinusOut','RuntimeObject');

% create str from rto1
str = num2str(rto1.InputPort(1).Data);

% get a handle to the GUI's 'current state' window
display = findobj('Tag','edit_display');

% update the gui
set(display,'string',str);


% get Data from Simulation
XData = get_param('Beispiel4_sim','SimulationTime');
YData = rto1.InputPort(1).Data;

% save Data to workspace
assignin('base','XData',XData)
assignin('base','YData',YData)


% real-time plot
persistent guiplot

if isempty(guiplot)
    guiplot=findobj(0, 'Tag','axesplot');
end
plot(guiplot,XData,YData)
hold on
grid on
```

Die grafische Benutzeroberfläche während einer laufenden Simulation ist in der nachfolgenden Abbildung dargestellt.

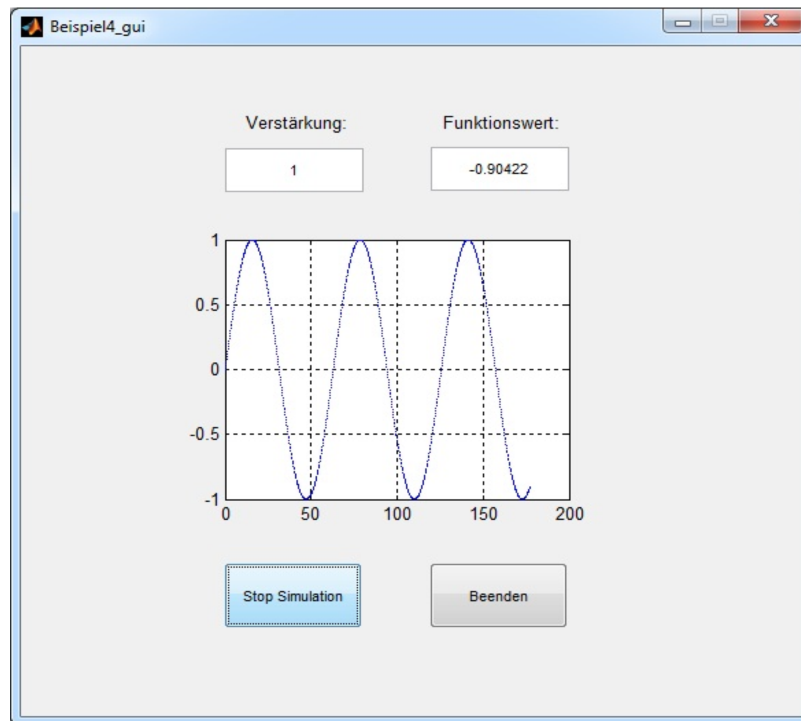

Abb. 3.7.5: Programmfenster zu „Beispiel4_gui" während einer laufenden Simulation. Angezeigt werden der akutelle Funktionswert der erzeugten Sinuskurve in einem Textfeld und die aufgezeichneten Punkte im Diagrammfenster. Der Verstärkungsfaktor der Sinuskurve kann über ein weiteres Textfeld verändert werden.

Der Quelltext von der Benutzeroberfläche, zum Steuern der Simulation, hat sich im Vergleich zum Beispiel 3 nur marginal geändert. Die Implementierung eines *„Real-Time Plot"* wird hauptsächlich über ein eigenständiges Skript gesteuert und bedarf daher kaum Änderungen in dem Skript vom GUI. Um die Funktionsweisen der einzelnen Objekte nachzuvollziehen, können die vorherigen Kapitel und Beispiele herangezogen werden.

Eine der Neuerungen ist, dass nach dem Schließen der Benutzeroberfläche und der Simulation der MATLAB Workspace gelöscht werden muss, da sonst Komplikationen beim nächsten Start der Anwendung auftreten könnten.

```
% clear Workspace
evalin('base','clear');
```

Die Skripte *„Beispiel4_gui.m"* und *„updategui.m"* befinden sich auf der CD unter „Beispiel4 (Real_Time_Plot)" und die zugehörigen Quelltexte sind außerdem in der Anlage 1.7 und 1.8 zu finden.

4. Hardwarekopplung über DAT

In diesem Kapitel wird anhand eines Beispielprogramms ***„SinusGUI“*** ein Simulationsmodell mithilfe der Data Acquisition Toolbox (DAT) mit einem Multifunktions-Datenerfassungsmodul vom Typ NI PCI-MIO-16E-4 gekoppelt und beschrieben, wie Datensätze mit der DAT übergeben werden können. Zusätzlich wird eine Benutzeroberfläche erstellt, mit der das Simulationsmodell gesteuert werden kann und die aufgenommenen Daten grafisch darstellt.

4.1 Versuchsaufbau

An dem Beispiel *„Sinus_GUI“* wird gezeigt, wie ein Simulationsmodell über die Data Acquisition Toolbox mit einer Messkarte gekoppelt wird. Von dem Simulationsmodell wird eine Sinusfunktion erzeugt, deren Signale über den Block **Analog Output (Single Sample)** an die Messkarte weitergegeben werden. Die Messkarte leitet die Signale an das Data Acquisition-Board der Messkarte weiter, an dem ein Oszilloskop[15] angeschlossen ist. Das Oszilloskop stellt die Werte in einem Graphen dar und kann sie gleichzeitig über einen analogen Ausgang an andere Komponenten weiterleiten. In diesem Versuch werden die aufgenommenen Daten vom Oszilloskop an die selbe Messkarte zurückgeschickt und vom Simulationsmodell über ein **Analog Input (Single Sample)**-Block importiert.

Über eine grafische Benutzeroberfläche kann der Prozess gestartet bzw. gestoppt werden. Simulink nimmt während der laufenden Simulation die Signale der erzeugten Sinuskurve auf und die Signale, welche von dem Oszilloskop importiert werden. Am Ende der Simulation werden die Daten an den MATLAB Workspace ausgelagert, woraufhin sie zur grafischen Darstellung in der Benutzeroberfläche vom GUI geladen werden können. Um bei diesem Prozess die erwartete Synchronität der erzeugten und der aufgenommenen Sinusfunktion zu gewährleisten, ist eine hohe Datenübertragungsrate und Rechenleistung von Hardware und Software erforderlich. Die Messkarte hat dabei eine Schlüsselfunktion mit einer Datenübertragungsrate von 250 kS/s pro Kanal.

[15] Ein Oszilloskop ist ein elektronisches Messgerät zur optischen Darstellung von elektrischen Spannungen.

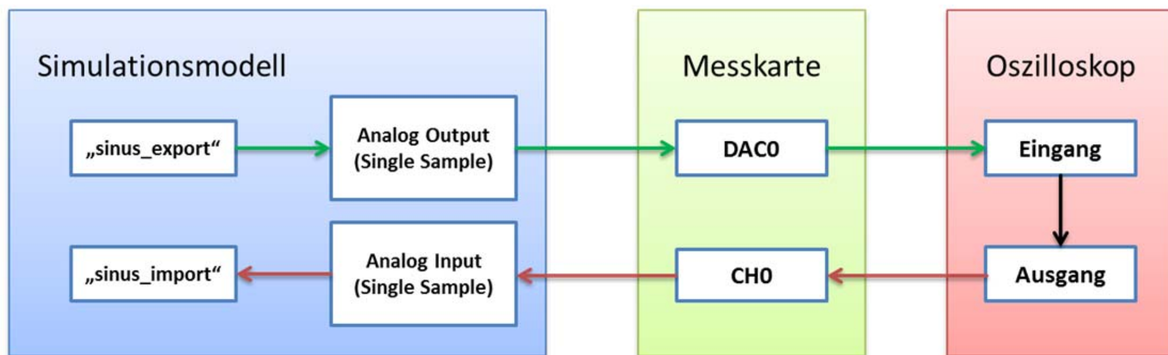

Abb. 4.1.1: Schematische Darstellung vom Versuchsaufbau zur Kopplung des Simulationsmodells mit einem Oszilloskop

4.2 Das Simulationsmodell

Das Simulationsmodell besteht aus einem **Sine Wave**-Block, mit dem die Sinusfunktion erzeugt wird, einem **To File**-Block, welcher Daten am Ende einer Simulation an den MATLAB Workspace übertragt, zwei **Scope**-Blöcke, zur Darstellung der Datenflüsse während einer laufenden Simulation, und den beiden Komponenten der Data Acquisition Toolbox zur Kopplung mit dem Datenerfassungsmodul.

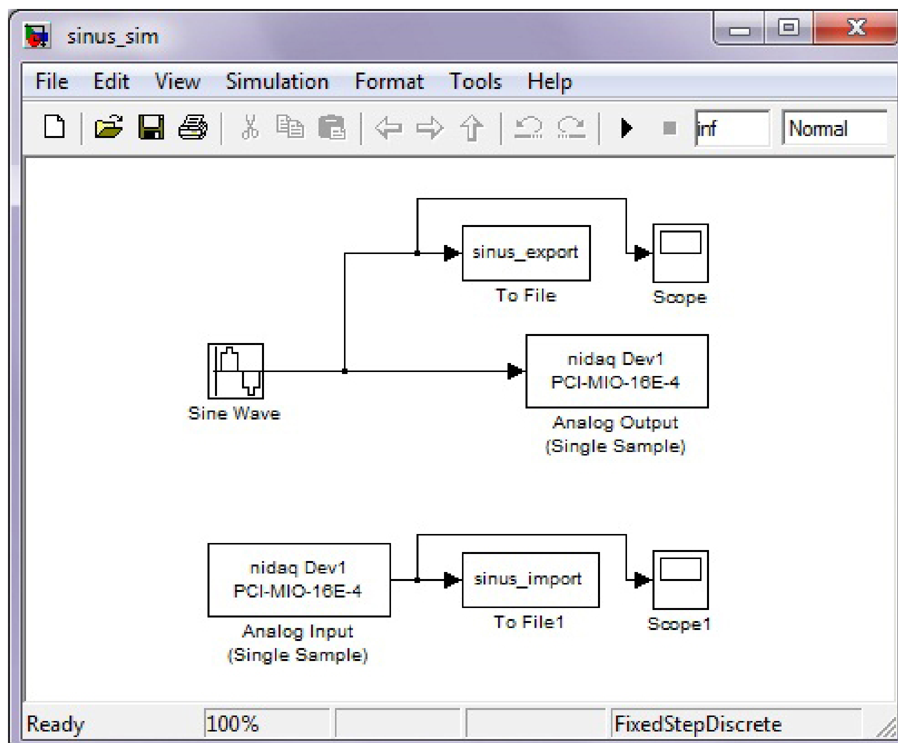

Abb. 4.2.1: Simulationsmodell „sinus_sim.mdl“ mit analogem Ein- und Ausgang, einem Block zum Erzeugen der Sinusfunktion, zwei To File Blöcken zum Auslagern der aufgenommenen Daten und zwei Blöcken zum Darstellen der Datenflüsse

Für die Berechnung des Simulationsmodells wird der Solver „FixedStepDiscrete“ verwendet. Die Schrittweite vom Solver wird mit „1,0“ festgelegt und ist die Grundlage für die Anzahl der Messungen in Abhängigkeit zur Laufzeit des Programms. Das bedeutet, dass pro Simulationszeitschritt ein Wert erzeugt bzw. ein Messwert erfasst wird. Die Simulationszeit wird auf unendlich gesetzt, damit die Steuerung des Modells über die grafische Benutzeroberfläche erfolgen kann. Alle weiteren Einstellungen vom Solver werden auf den Standardeinstellungen belassen.

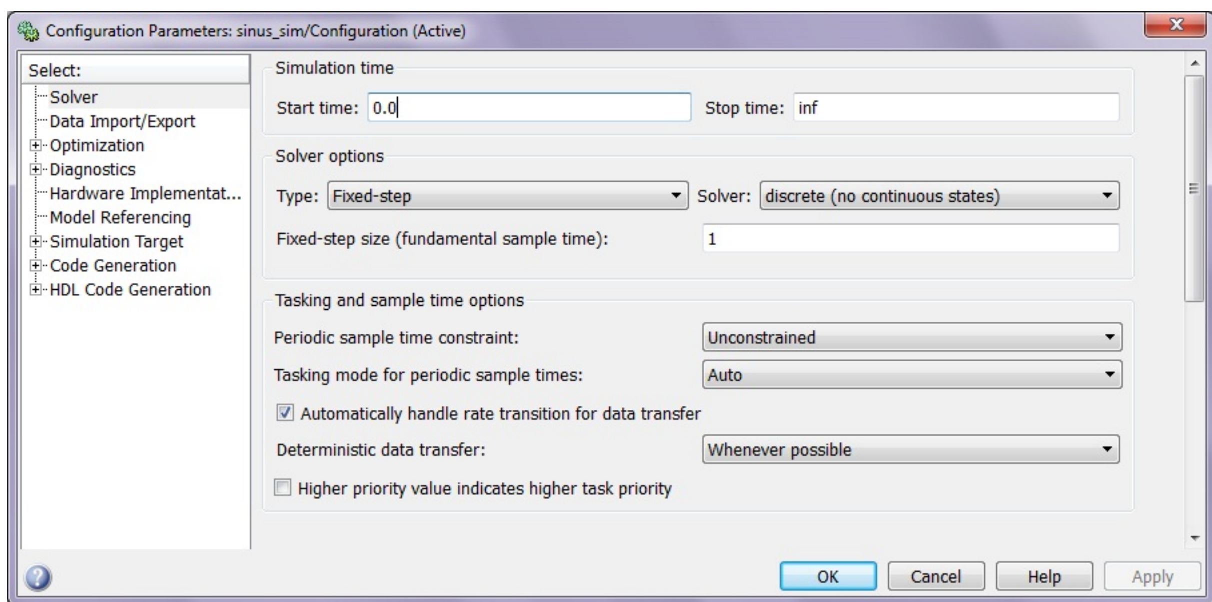

Abb. 4.2.2: Parameterfenster des „Solver“ mit der Einstellung „Fixed-step discrete“ und einer Schrittweite von „1“.

Die Sinusfunktion wird mit dem Block **Sine Wave** in Abhängigkeit zur Simulationszeit mit der Amplitude „1“ erzeugt. Die Frequenz wird auf 0,01 s^{-1} gesetzt, damit die Sinusfunktion auch nach längerer Simulationszeit gut erkennbar dargestellt werden kann. Eine Sinusfunktion, welche auf der Simulationszeit basieren soll, wird mit folgender Formel erzeugt:

$$O(t)=Amp*Sin(Freq*t+Phase)+Bias$$

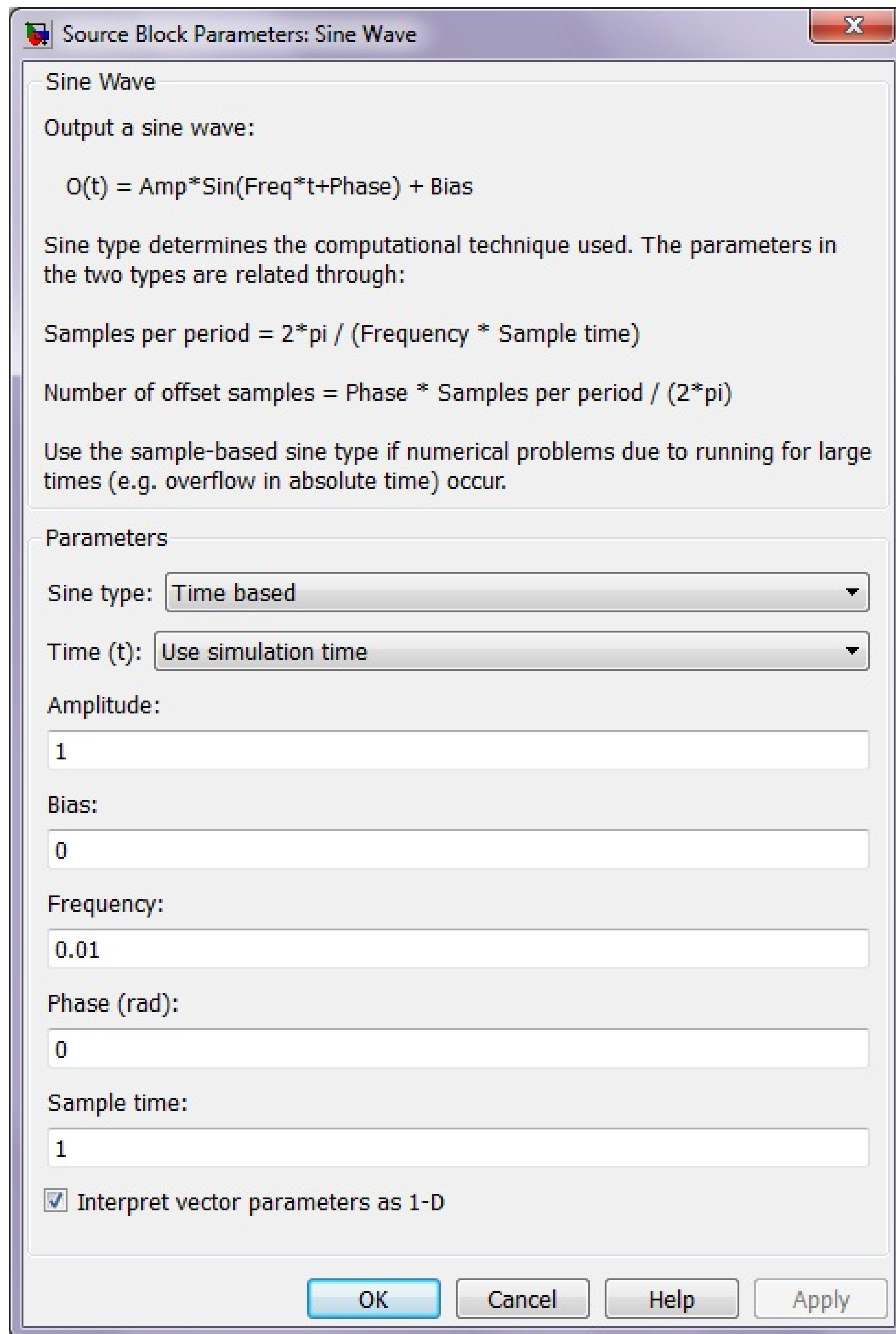

Abb. 4.2.3: Parameterfenster vom „Sine Wave"-Block zum Erzeugen der Sinusfunktion auf Basis der Simulationszeit mit einer Amplitude von „1" und einer Frequenz von 0,01 s^{-1}.

Die erzeugte Sinusfunktion wird an einen **To File**-Block weitergeleitet, welcher am Ende der Simulation die Daten unter dem Namen *„sinus_export"* als Array an den MATLAB Workspace ausgibt, und an den Block **Analog Output (Single Sample)**, welcher die Signale an eine Messkarte exportiert. Gleichzeitig empfängt der **„Analog Input (Single Sample)"**-Block die Signale der Sinuskurve, welche von einem Oszilloskop importiert werden. Das Oszilloskop

zeigt simultan zum Übertragungsprozess und zur Simulationszeit die erzeugte Sinusfunktion an und gibt die dargestellten Werte an die Messkarte weiter. Von der Messkarte werden die Daten an die Simulation zurückgegeben.

Abb. 4.2.4: Von einem Oszilloskop gemessene Sinuskurve während einer laufenden Simulation.

Von dem Block **Analog Input (Single Sample)** werden die Daten an einen weiteren **To File**-Block gegeben, welcher am Ende der Simulation die aufgenommenen Daten als Variable *„sinus_import“* im MATLAB Workspace abspeichert.

Die Sinusfunktion wird von dem **Analog Output (Single Sample)**-Block über den Kanal „DAC0“[16] an die Messkarte übertragen. Dazu werden eingehende Daten in Spannungen (im Verhältnis 1:1) umgewandelt. Die „Output Range“ beschränkt die zu übergebenden Spannungen in einem Bereich von „-10“ bis „+10“ Volt. Da die Amplitude der erzeugten Sinusfunktion „1“ ist, werden aufgrund des eingestellten Intervalls alle aufgezeichneten Daten an die Messkarte übergeben.

[16] In Simulink wird der Kanal mit der Bezeichnung *„Hardware Channel 0“* angezeigt. Simulink erkennt Kanäle automatisch und zeigt die zu verwendenden Kanäle an. Im *„Analog Output (Single Sample)*-Block werden daher nur die zwei Kanäle der Messkarte angezeigt, mit denen analoge Signale ausgegeben werden können („DAC0“ und „DAC1“). Die beiden Kanäle dienen zugleich zum Ausgeben digitaler Signale.

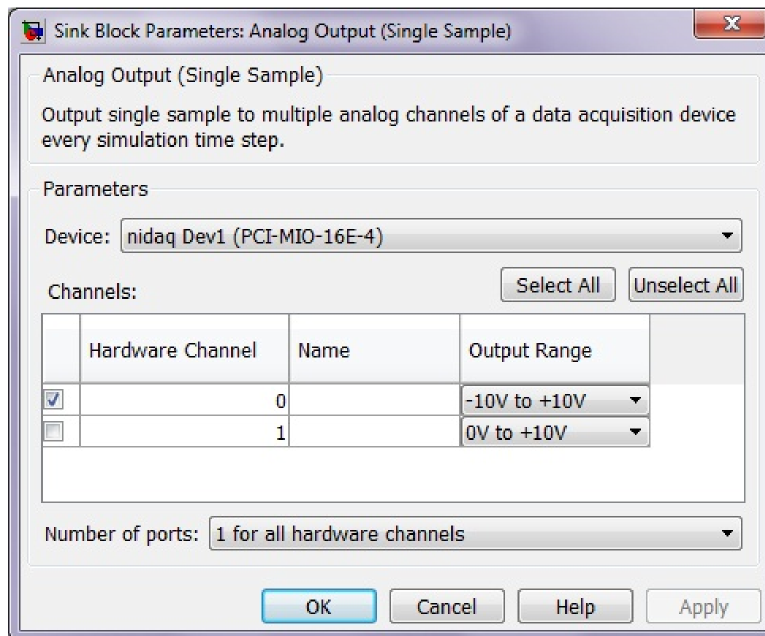

Abb. 4.2.5: Parameterfenster vom „Analog Output (Single Sample)“-Block mit einer „Output Range“ von „-10V“ bis „+10V“ am aktivierten Hardware Channel „DAC0“

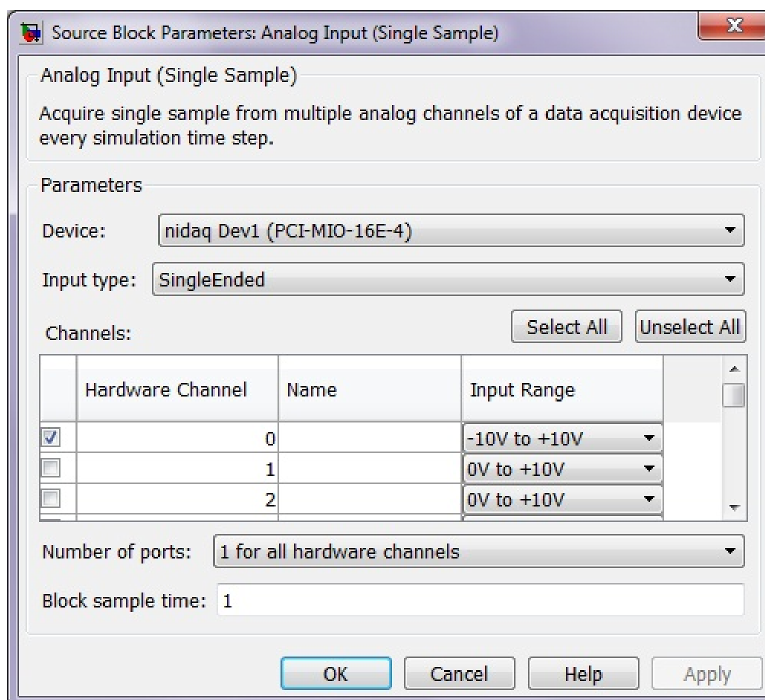

Abb. 4.2.6: Parameterfenster vom „Analog Input (Single Sample)“-Block mit einer „Input Range“ von „-10V“ bis „+10V“ am aktivierten Hardware Channel „CH0“

Die Signale der Sinusfunktion werden von dem Oszilloskop über den **Analog Input (Single Sample)**-Block wieder in das Simulationsmodell eingespeist.

Die Begrenzung der aufzunehmenden Signale liegt wie beim **Analog Output (Single Sample)**-Block zwischen „-10“ und „+10“ Volt. Die Übergabe der Signale erfolgt über den Kanal „CH0“[17]. Die Kanalbelegung für das Beispiel *„Sinus_GUI“* befindet sich in der Anlage 1.10.

4.3 Die Benutzeroberfläche

Zur Steuerung der Simulation und Auswertung der Messerergebnisse wird eine grafische Benutzeroberfläche erstellt. Diese enthält zwei Button zum Steuern der Simulation und zwei Diagramme zum Darstellen der erzeugten bzw. aufgenommenen Sinuskurve (Abb.4.3.1). Das Starten und Stoppen der Simulation erfolgt über einen Button. Der zweite Button dient zum Beenden der Oberfläche und der Simulation. Der Quelltext der Benutzeroberfläche *„sinus_gui.m“* ist im Anhang 1.9 abgebildet.

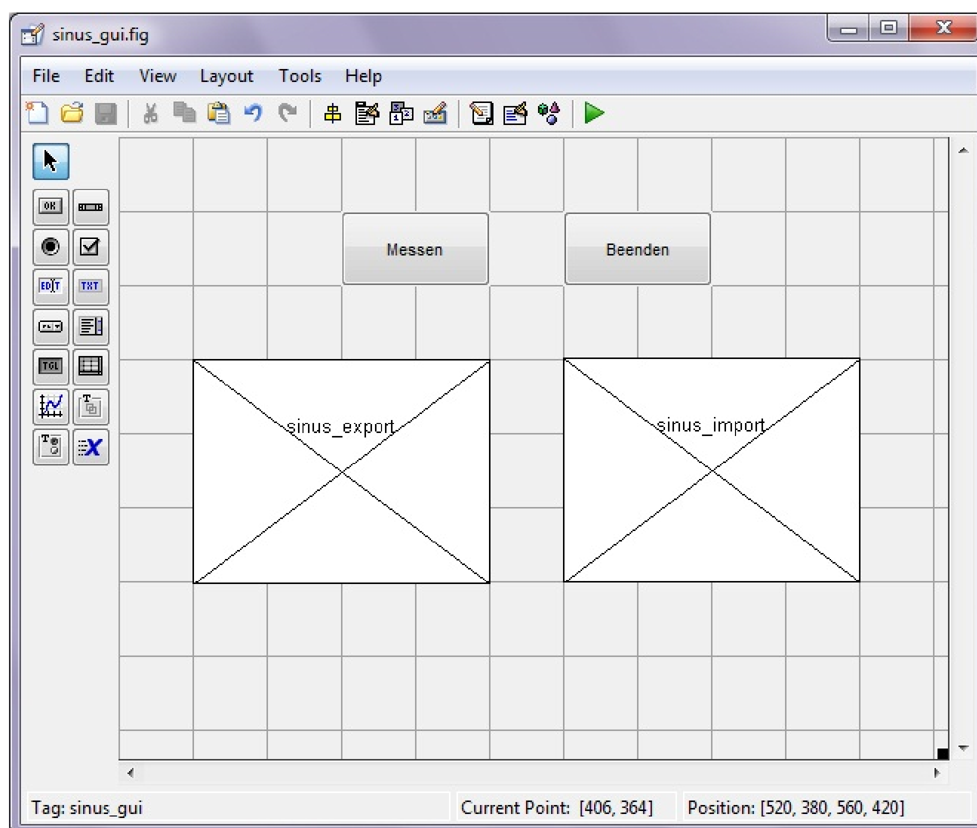

Abb. 4.3.1: Benutzeroberfläche von „Sinus_GUI“ im „Dialog Editor“ mit zwei Button zum Steuern des Simulationsmodells und zwei Plot-Fenster zur Darstellung der erzeugten und aufgenommenen Sinuskurve.

[17] Der analoge Input-Channel „CH0“ wird in Simulink als „Hardware Channel 0“ angezeigt. Die Messkarte besitzt insgesamt 16 analoge Input-Channel.

Nach einer durchlaufenden Simulation werden in der Benutzeroberfläche die ein- und ausgehenden Daten der Sinusfunktion über zwei Diagramme dargestellt. Die im GUI dargestellten Graphen zeigen einen fast simultanen Verlauf der Sinusfunktion und beweisen so eine erfolgreiche Übermittlung von Daten zwischen Messhardware und MATLAB Software unter Verwendung einer grafischen Benutzeroberfläche.

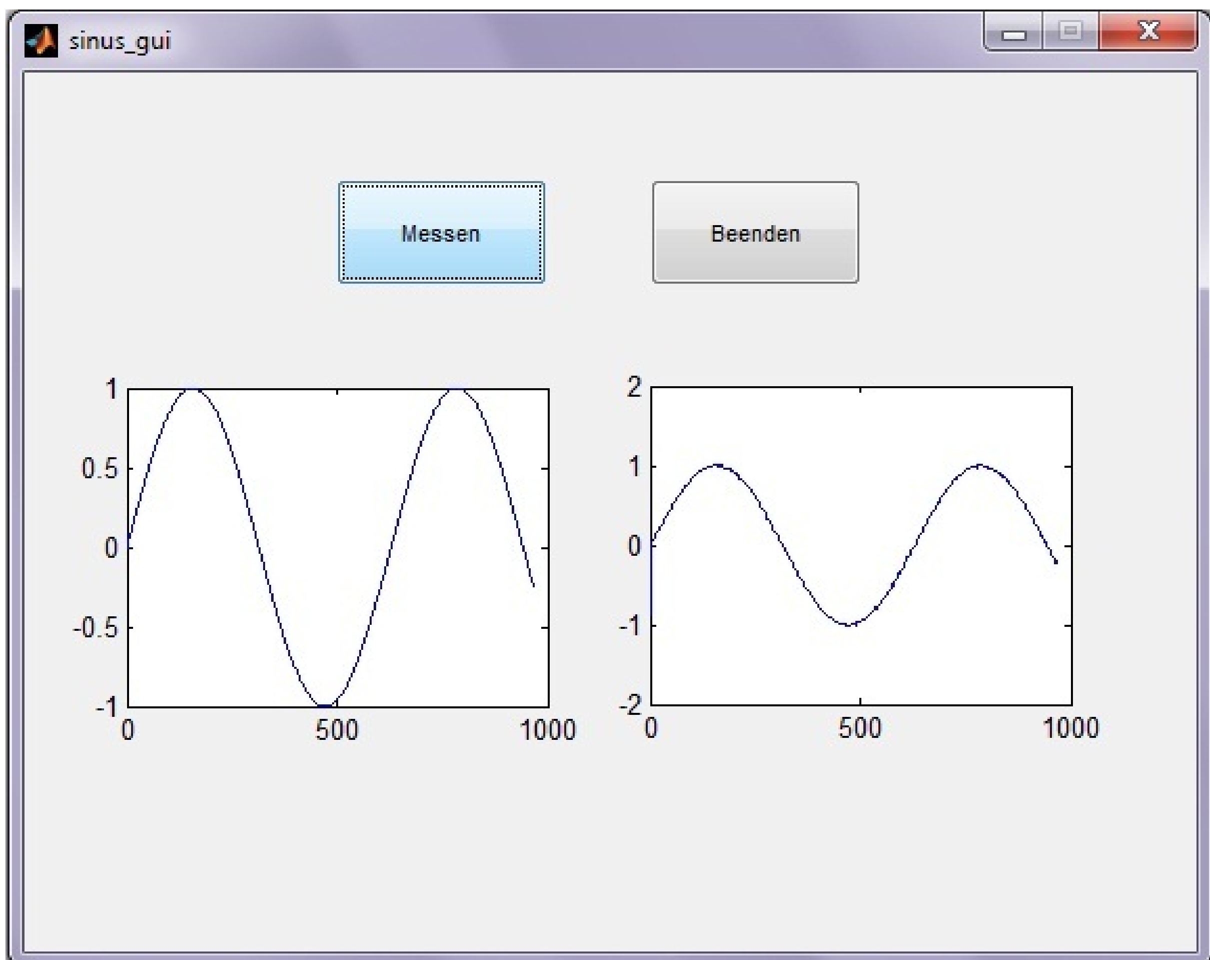

Abb. 4.3.2: Darstellung der Benutzeroberfläche nach einer durchlaufenden Simulation mit erzeugter und aufgenommener Sinuskurve.

4.4 Auswertung der Messreihe

Aufgrund von Stromschwankungen in der Messkarte können auch vereinzelt Messfehler auftreten, die als Fluktuationen erkenntlich sind. Diese führen zu der unterschiedlichen Skalierung der beiden Graphen, da sich bei der aufgenommenen Sinusfunktion vereinzelt Werte außerhalb des Bereiches der erzeugten Funktionswerte [-1; +1] befinden.

Tabelle 4.4-1: Min. und Max. der erfassten Sinusfunktion ("sinus_import")

„sinus_import“	Funktionswert
Max.	1,0254
Min.	-1,0059

Sieht man sich die erfassten Daten genauer an (Tab. 4.4-2), stellt man fest, dass sich eine Verschiebung von der erzeugten zur aufgenommenen Sinusfunktion einstellt.

In diesem Fall spricht man von einem *„delay“*, welcher mit der Rechenleistung der Messkarte (250 kS/s) bzw. der Datenübertragungsgeschwindigkeit von Anschlusskabeln zu begründen ist. Deshalb wird über den **Analog Input (Single Sample)**-Block, welcher nur einzelne *Samples* einer Messreihe (bezogen auf die *Sample Rate*) aufzeichnet, ein Versatz der Sinusfunktion importiert, um alle Werte in einer Simulation auf eine gemeinsame Zeitbasis zu bringen. Für die Realisierung einer gemeinsamen Zeitbasis kommt es zu einer Interpolation zwischen den Signalen, die von der erzeugten Sinusfunktion (*„sinus_export“*) an die Messkarte gesendet werden.

Eine Synchronität ist trotzdem gegeben, da sich die zeitliche Verschiebung der eingehenden zu den ausgehenden Werten gleichmäßig verhält. Die mittlere Abweichung von den ein- und ausgehenden Funktionswerten bzw. den daraus erzeugten Spannungen einer gleichen Zeitreihe beträgt 0,0086. Aufgrund der geringen Verschiebung der einzelnen Werte kann die Messwerterfassung mit MATLAB und somit auch die Kopplung der MATLAB Software mit den Hardwarekomponenten des Wasserbaulabors als validiert angesehen werden. Die erfassten Werte der Versuchsreihe und deren Auswertung befinden sich auf der beiliegenden CD unter *„Auswertung SinusGUI“*.

Tabelle 4.4-2: Auszug der Datenerfassung vom Beispiel „Sinus_GUI“[18]

t	sinus_export	sinus_import	t	sinus_export	sinus_import
0	0,0000	-0,8789	**25**	0,2474	0,2441
1	0,0100	0,0000	**26**	0,2571	0,2490
2	0,0200	0,0098	**27**	0,2667	0,2588
3	0,0300	0,0195	**28**	0,2764	0,2734
4	0,0400	0,0342	**29**	0,2860	0,2783
5	0,0500	0,0293	**30**	0,2955	0,2881
6	0,0600	0,0488	**31**	0,3051	0,2979
7	0,0699	0,0537	**32**	0,3146	0,3027
8	0,0799	0,0684	**33**	0,3240	0,3076
9	0,0899	0,0781	**34**	0,3335	0,3223
10	0,0998	0,0879	**35**	0,3429	0,3320
11	0,1098	0,0977	**36**	0,3523	0,3369
12	0,1197	0,1074	**37**	0,3616	0,3516
13	0,1296	0,1367	**38**	0,3709	0,3613
14	0,1395	0,1318	**39**	0,3802	0,3857
15	0,1494	0,1416	**40**	0,3894	0,3857
16	0,1593	0,1563	**41**	0,3986	0,3906
17	0,1692	0,1611	**42**	0,4078	0,4004
18	0,1790	0,1709	**43**	0,4169	0,4102
19	0,1889	0,1855	**44**	0,4259	0,4150
20	0,1987	0,1904	**45**	0,4350	0,4248
21	0,2085	0,2002	**46**	0,4439	0,4346
22	0,2182	0,2148	**47**	0,4529	0,4492
23	0,2280	0,2197	**48**	0,4618	0,4541
24	0,2377	0,2295	**49**	0,4706	0,4639
25	0,2474	0,2441	**50**	0,4794	0,4688

[18] „Sinus_export“ ist dabei die vom Simulationsmodell erzeugte Sinusfunktion, welche an die Messkarte übermittelt wird. „Sinus_import“ ist die Sinusfunktion, die von dem Oszilloskop angezeigt und an die Messkarte übermittelt wird. Anschließend wird diese Datenreihe von der Simulation aufgenommen.

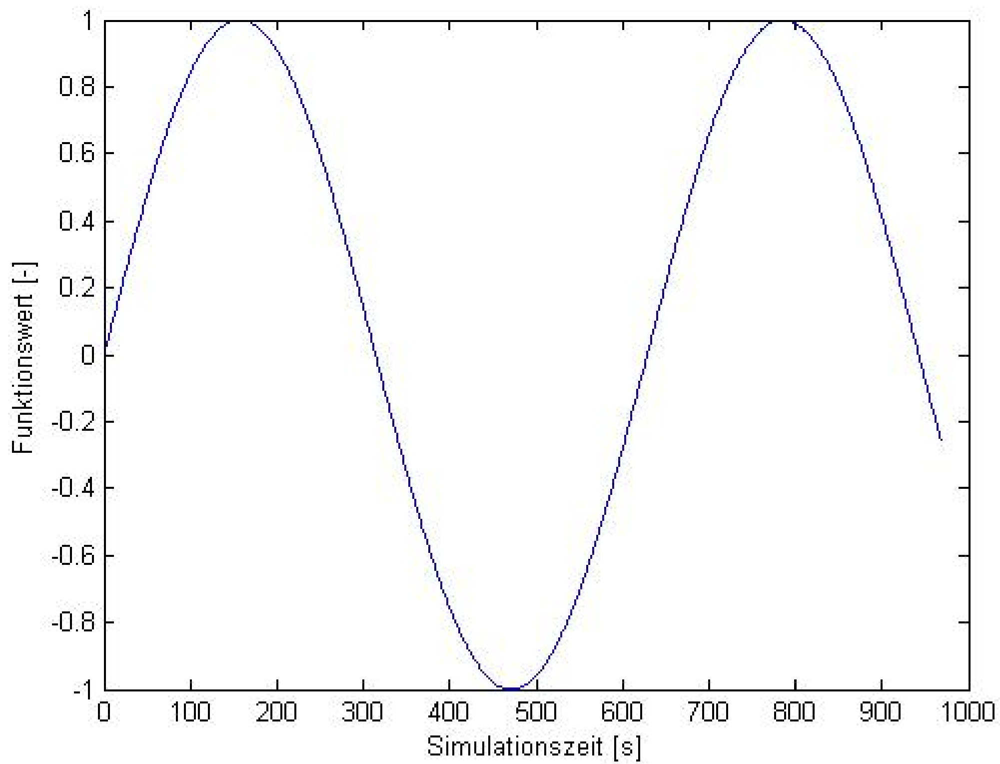

Abb. 4.4.1: Darstellung der erzeugten Sinusfunktion ("sinus_export")

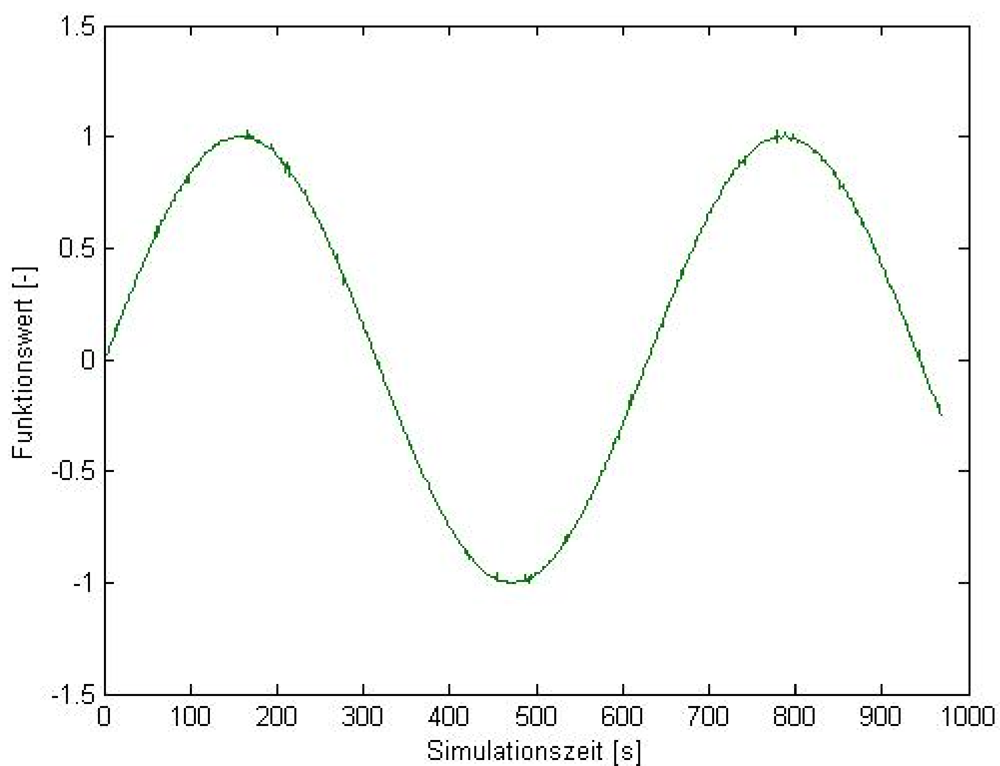

Abb. 4.4.2: Darstellung der erfassten Sinusfunktion ("sinus_import")

5. Durchflussmessung mit MATLAB

5.1 Versuchsaufbau

Für den nächsten Versuch wird ein Testprogramm ***„Q_Messen"*** erstellt, welches den Durchfluss in einem Versuchsrohr aufnehmen und speichern soll. Das Versuchsrohr vom Typ DN 150 befindet sich im Wasserbaulabor und wird über einen Hochbehälter angesteuert. Über einen Schieberegler wird ein Durchfluss von 25 l/s eingestellt.

Für dieses Experiment werden ein Simulationsmodell und eine Benutzeroberfläche erstellt. Das Simulationsmodell dient zu Aufnahme des gemessenen Durchflusses und gibt die Daten an eine Benutzeroberfläche weiter. In der Benutzeroberfläche wird der gemessene Durchfluss in einem Graphen dargestellt. Weiterhin kann über die Benutzeroberfläche zum Steuern der Simulation, also zum Starten und Stoppen der Durchflussmessung verwendet werden. Die aufgenommenen Daten sollen zudem über die Oberfläche als Array (.mat) und als Ascii-Datei (.dat) gespeichert werden können. Der Nutzer kann dafür die Bezeichnung für die zu speichernden Daten über ein Eingabefeld selbst wählen.

5.2 Das Simulationsmodell

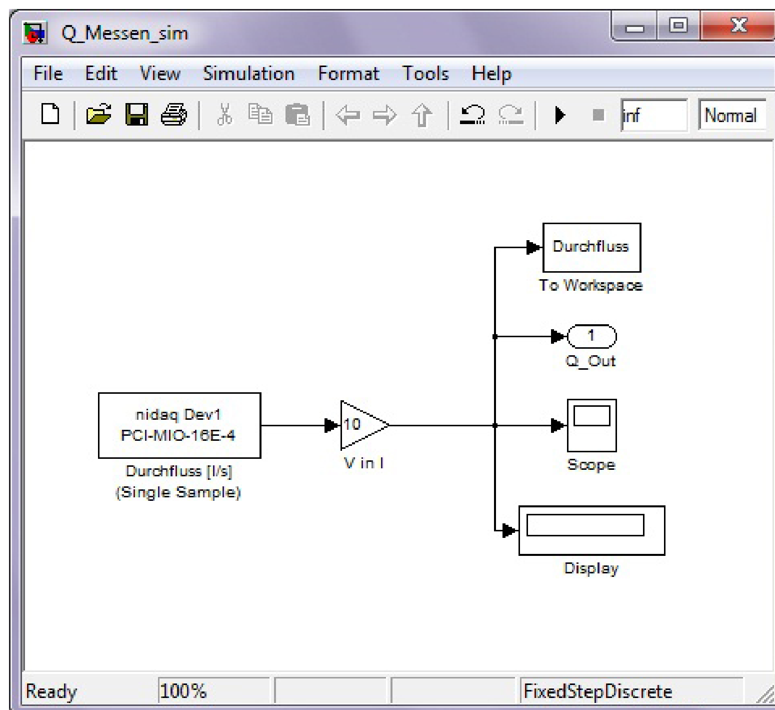

Abb. 5.2.1: Simulationsmodell „Q_Messen_sim.mdl" mit „Analog Input (Single Sample)" zum Aufnehmen der Durchflussmessung mit einem IDM, „Gain"-Block zum Umrechnen der Spannung [V] in Durchfluss [l/s], und Blöcken zum Anzeigen, Darstellen und Übergeben der aufgenommen Werte.

Das Simulationsmodell enthält einen **Analog Input (Single Sample)**-Block, ein **Display** und **Scope** sowie ein **Output**-Block. Von dem **Output**-Block werden die Daten über einen „EventListener" an die Benutzeroberfläche übergeben, um sie in Echtzeit grafisch darzustellen. Für den „EventListener" werden in der *„StartFcn"* der Simulation folgende Einstellungen vorgenommen:

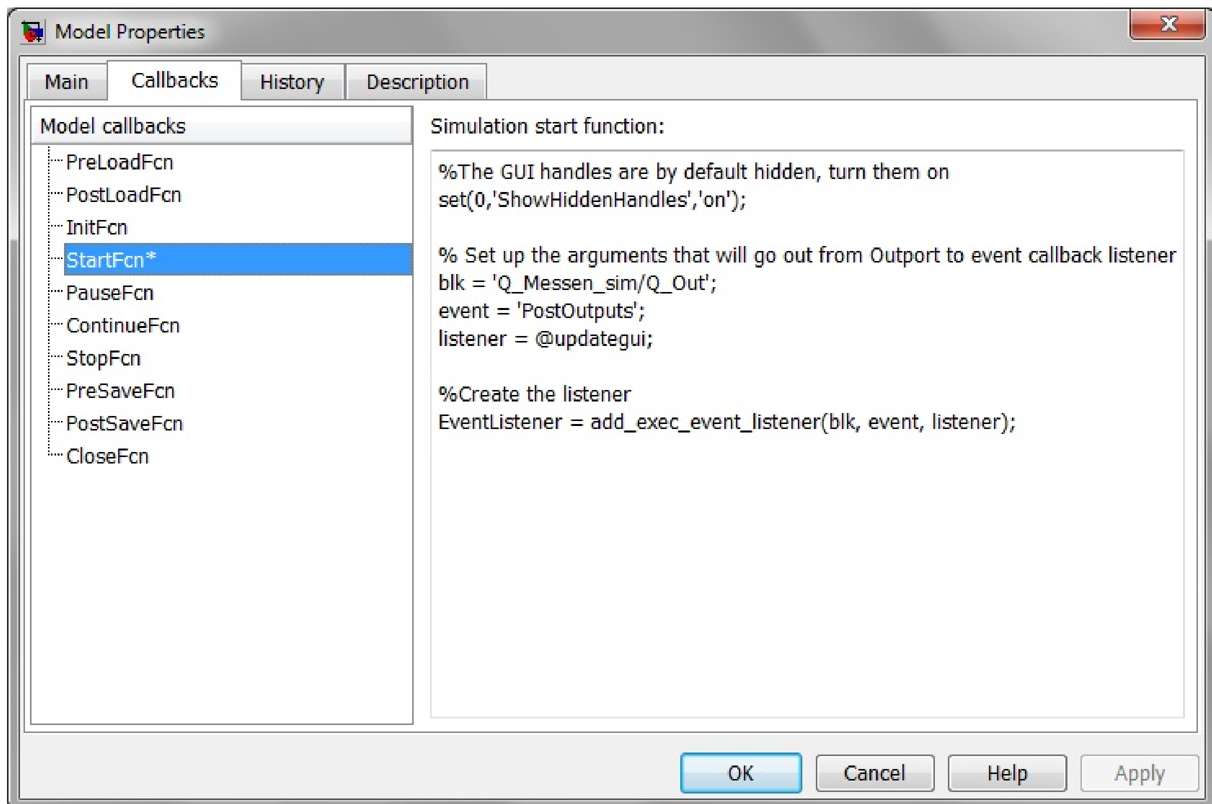

Abb. 5.2.2: „StartFcn“ von „Q_Messen_sim“ mit Quelltext zur Erstellung des „EventListener“

Der gemessene Durchfluss wird von der Messkarte über den analogen Datenerfassungskanal „CH8“ an die Simulation übergeben. Dabei entsprechen 10 V einem Durchfluss von 100 l/s. Der **Analog Input (Single Sample)**-Block übernimmt Signale in einem Bereich von „0 V“ bis „+10 V“. Daher kann über das Testprogramm lediglich ein Durchfluss von bis zu 100 l/s aufgenommen werden. Die Kanalbelegung der Messkarte bzw. vom Data Acquisition-Board kann der Anlage 1.13 entnommen werden.

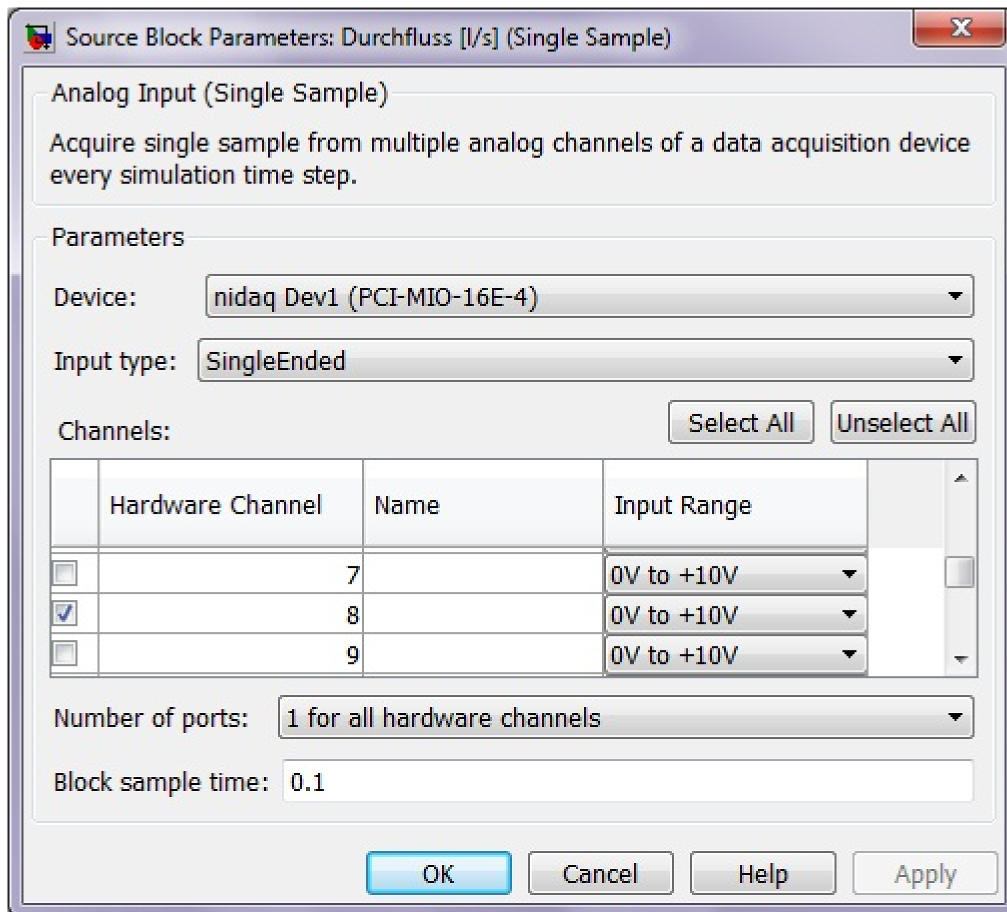

Abb. 5.2.3: Parameterfenster vom „Analog Output (Single Sample)"-Block mit einer „Input Range" von „0V" bis „+10V" am aktivierten Hardware Channel „CH8" bei einer Sample Time von 0,1. Als Bezugspunkt ist der Massepunkt (0 V) gewählt (single ended).

Für einen Testlauf des Programms wird ein Durchfluss von 25 l/s eingestellt. Die Benutzeroberfläche stellt während der Messwerterfassung die einzelnen Durchflussmessungen als Punktmasse in einem Diagramm dar. Auf der x-Achse ist die Anzahl der erfassten Messwerte dargestellt.[19] Eine Tabelle zu den erfassten Werten bei einem Durchfluss von 25 l/s befindet sich auf der beiliegenden CD unter „Auswertung Q_Messen".

[19] Im Simulationsmodell „Q_Messen_sim.mdl" ist eine Schrittweite von „1,0" eingestellt. Daher wird zu jedem Zeitschritt ein Wert erfasst.

5.3 Die Benutzeroberfläche

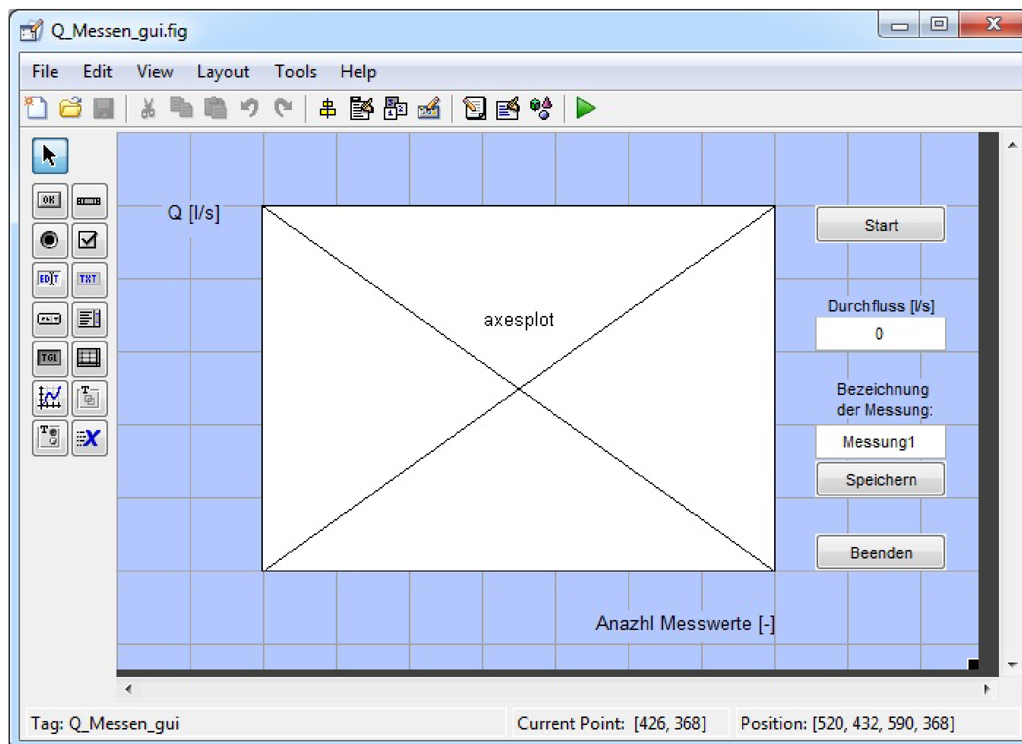

Abb. 5.3.1: Benutzeroberfläche von „Q_Messen" im „Dialog Editor" mit Textfeld zum Anzeigen des aktuellen Durchflusses und „Plot"-Fenster zum Darstellen der aufgezeichneten Durchflussmessung.

Über die Benutzeroberfläche kann die Durchflussmessung über das Simulationsmodell gestartet werden. Die aufgenommenen Daten werden in Echtzeit in einem Graphen dargestellt. Der aktuelle Durchfluss wird zudem in einem Textfeld angezeigt. Nach abgeschlossener Durchflussmessung besteht die Möglichkeit zum Abspeichern der aufgenommenen Daten. Der Durchfluss und die Simulationszeit werden nach der Simulation im MATLAB Workspace gespeichert. Über den „Speichern"-Button werden die Vektoren in das GUI geladen und anschließend im Programmordner als Array (.mat) und als Ascii-Datei (.dat) abgelegt. Die Datenreihe vom Durchfluss wird zu einer besseren Darstellung vor dem Speicherprozess umgewandelt.

Der Quelltext zum Speichern der aufgenommenen Daten ist nachfolgend aufgeführt:

```
Bezeichnung = get(handles.edit_Bezeichnung,'String');
Durchfluss = evalin('base','Durchfluss');
STIME = evalin('base','STIME');

Durchfluss=Durchfluss(1,:);
Durchfluss=Durchfluss';

Ergebnis=[STIME,Durchfluss];
save(Bezeichnung,'Durchfluss','STIME')
dlmwrite(Bezeichnung,Ergebnis,';')
```

5.4 Auswertung der Messreihe

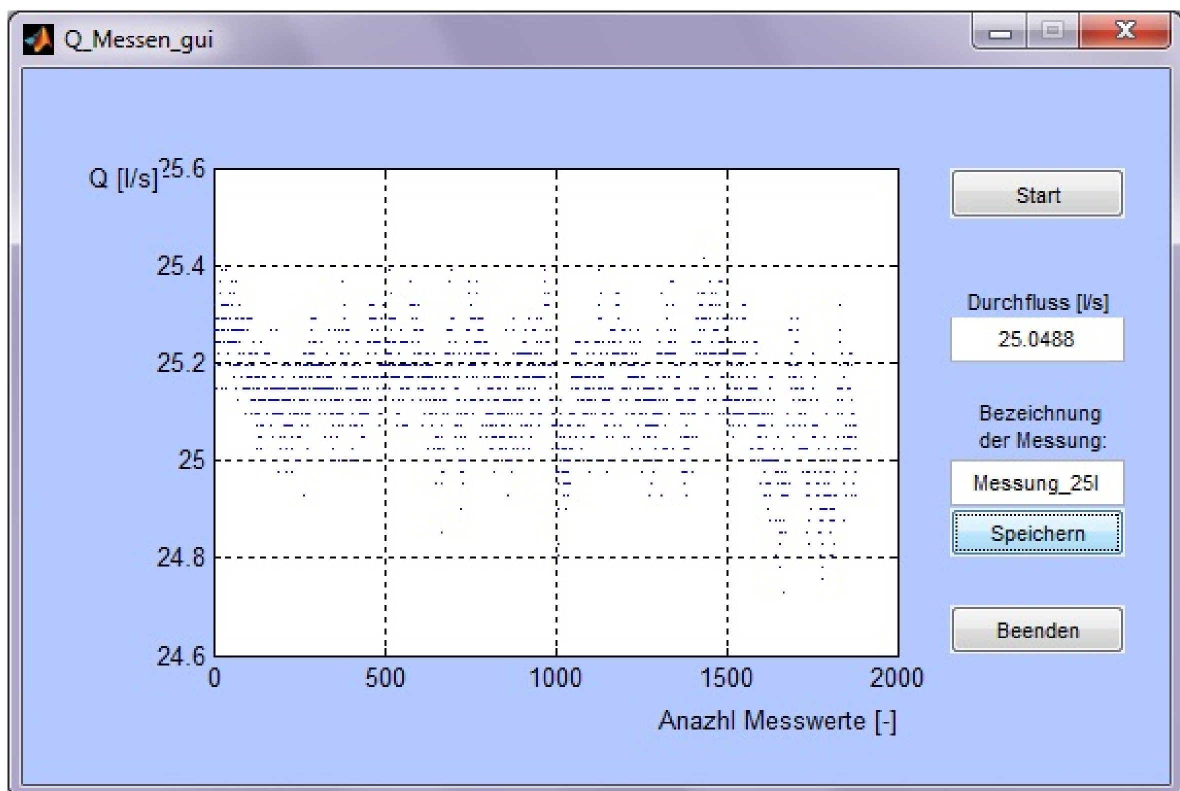

Abb. 5.4.1: Programmfenster „Q_Messen_gui“ nach einer Simulation (bei 25 l/s) mit der aufgezeichneten Durchflussmessung.

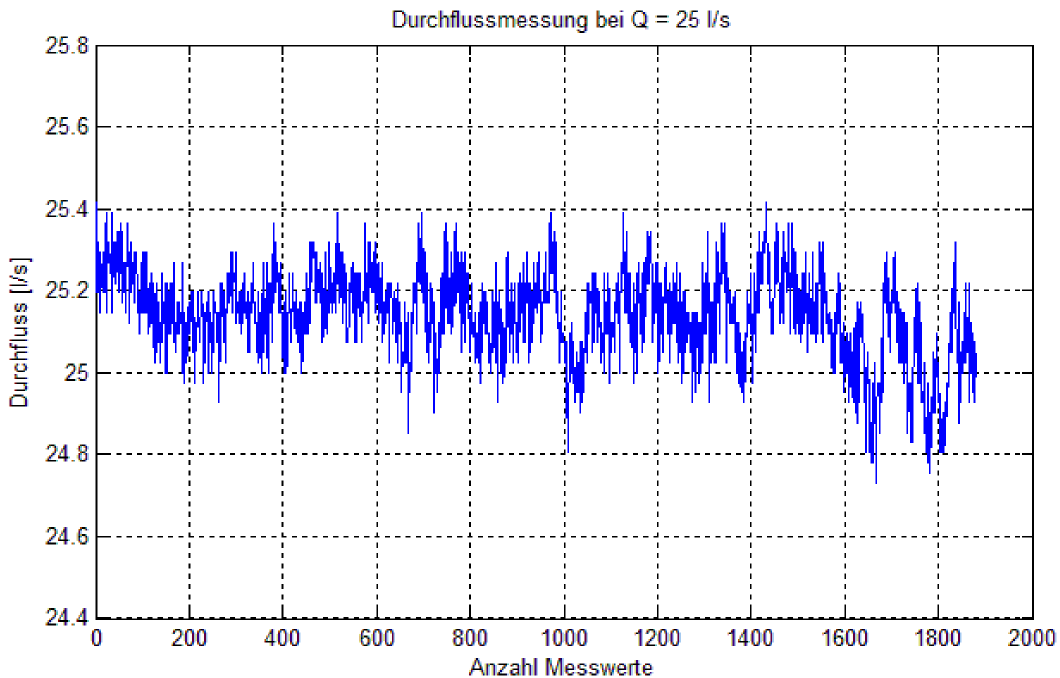

Abb. 5.4.2: Graph der Durchflussmessung bei Q = 25 l/s

Im Graphen werden die Schwankungen in der Durchflussmessung ersichtlich. Um die Güte der Messung einzuschätzen, wird der Mittelwert, die mittlere Abweichung sowie die Standardabweichung berechnet. Der gemittelte Durchfluss von 1879 Einzelmessungen liegt bei 25,14 l/s. Diese geringe Abweichung ist darauf zurückzuführen, dass der Durchfluss im Rohr manuell mithilfe der Analoganzeige eines Durchflussgebers (IDM) eingestellt wurde.

Interessanter ist es, die Schwankungen der Einzelmessungen zum Mittelwert zu überprüfen. Die mittlere Abweichung der Messreihe liegt bei 0,084 und ergibt eine prozentuale Abweichung von 0,33 % vom gemittelten Durchfluss. Die Standardabweichung beträgt hingegen 0,108. Anhand der geringen Abweichungen ist eine Durchflussmessung mit MATLAB und somit die Kopplung mit der Data Acquisition Toolbox im Rahmen der Messwerterfassung validiert.

Tabelle 5.4-1-1: Auswertung der Messung mit MATLAB bei Q = 25 l/s

Auswertung	Mittelwert [l/s]	mittl. Abweichung	Standardabweichung
IST-Durchfluss	25,14	0,084	0,108

Die Verifikation von dem MATLAB Programm *„Q_Messen“* zur Durchflussmessung ergibt sich anhand weiterer Testläufe, bei denen ähnliche Abweichungen vom gemittelten Durchfluss erfasst wurden. Die weiteren Messungen erfolgten im Rahmen der Durchflussregelung auf 5 l/s, 10 l/s und 30 l/s, nachdem sich der SOLL-Durchfluss eingestellt hat. Die Auswertung der Daten erfolgt in Kapitel 6.4 und ist auf der CD unter *"Auswertung Q_Regelung"* als Excell-Tabelle zu finden.

5.5 Vergleich mit LabView

Im nächsten Schritt wird die Messwerterfassung von MATLAB mit LabView verglichen. Für die Messung mit LabView wird wie bei der MATLAB Messung ein Durchfluss von 25 l/s eingestellt. Die Auswertung der aufgezeichneten Messreihe mit LabView, ergibt einen gemittelten Durchfluss von 25,4 l/s. Die mittlere Abweichung von den Messwerten beträgt 0,071 und ergibt eine prozentuale Abweichung von 0,28 % vom Mittelwert. Die Standardabweichung liegt bei 0,084. Eine Tabelle der gesamten Messreihe mit LabView ist auf der beiliegenden CD enthalten.

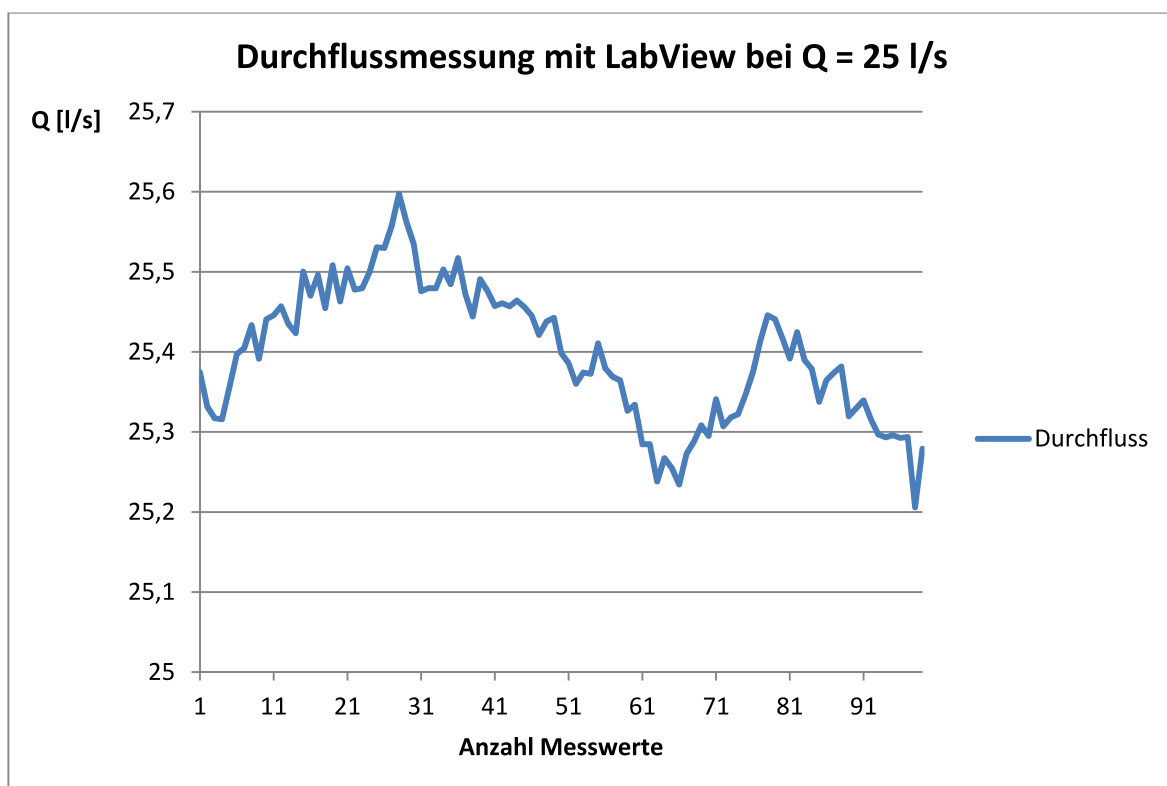

Abb. 5.5.1: Graph der Durchflussmessung mit LabView bei Q =25 l/s

Zur Untersuchung der Qualität von Messwerterfassungen über MATLAB und LabView werden die aufgenommen Messreihen des eingestellten Durchflusses von 25 l/s verglichen. Der Mittlerwert von der LabView Messung liegt bei 25,4 l/s. Die Abweichung zum eingestellten Durchfluss ist damit zu begründen, dass der Durchfluss separat für diese Messung eingestellt wurde.

Für einen Vergleich von Programmen zur Qualität einer Messwerterfassung ist es entscheidend, die Auswertung der Messreihen auf den gemittelten Durchfluss zu beziehen. So werden die Schwankungen, also die Abweichungen der Einzelmessungen, ersichtlich. Daraus kann ermittelt werden, wie groß die Unterschiede der Messschwankungen und Abweichungen sind, welche sich aufgrund der Datenverarbeitung ergeben.

Tabelle 5.5-1: Vergleich der Messungen von MATLAB und LabView bei Q = 25 l/s

Auswertung	Mittelwert [l/s]	mittl. Abweichung	Standardabweichung
LabView	25,40	0,071 (0,28 %)	0,084
MATLAB	25,14	0,084 (0,33 %)	0,108

Bezogen auf die mittlere Abweichung bzw. auf die Standardabweichung der Durchflussmessung bei Q = 25 l/s besteht nur ein geringer Unterschied zwischen der Messwerterfassung mit MATLAB und LabView (Tab. 5.4.1-2). Daher kann die Schlussfolgerung gezogen werden, dass eine Durchflussmessung mit MATLAB qualitativ gleichwertig zu der Messung mit LabView ist. Somit kann das MATLAB Programm *„Q_Messen"* als validiert angesehen werden.

6. Durchflussregelung mit MATLAB

In diesem Kapitel wird beschrieben, wie ein Regelkreis mithilfe der DAT und einem Simulationsmodell erstellt werden kann. Eine Benutzeroberfläche wird zum Steuern und zum Auswerten des Regelprozesses dienen und so den Regelkreis als Benutzerprogramm vervollständigen. Über das Programm ***„Q_Regelung“*** soll der Durchfluss in einem Rohr des Wasserbaulabors geregelt werden können. Das Testprogramm zur Durchflussregelung befindet sich auf der beiliegenden CD.

6.1 Versuchsaufbau

Im Wasserbaulabor der Universität der Bundeswehr München soll der Durchfluss in einem DN 150 Rohr über ein MATLAB Programm geregelt werden. An dem Rohr sind ein steuerbares Ventil (Ringkolbenschieber) und ein Durchflussmesser (IDM) angeschlossen. Der Durchflusssensor gibt den IST-Durchfluss im Versuchsrohr an eine Messkarte weiter, welche die Werte an das Simulationsmodell, welches den Regelkreis darstellt, leitet. Über den Regelkreis werden die Stellgrößen zum Öffnen und Schließen des Ventils bestimmt, solange bis sich der gewünschte Durchfluss eingestellt hat. Über die Benutzeroberfläche kann der Nutzer seinen SOLL-Durchfluss für die Regelung einstellen und bekommt während des laufenden Prozesses über eine grafische Darstellung der Durchflussmessung die Rückmeldung zum aktuellen IST-Durchfluss. Nach dem erfolgreichen Regelprozess kann der Nutzer die aufgenommenen Daten in einem .mat-File und einem .dat-File abspeichern.

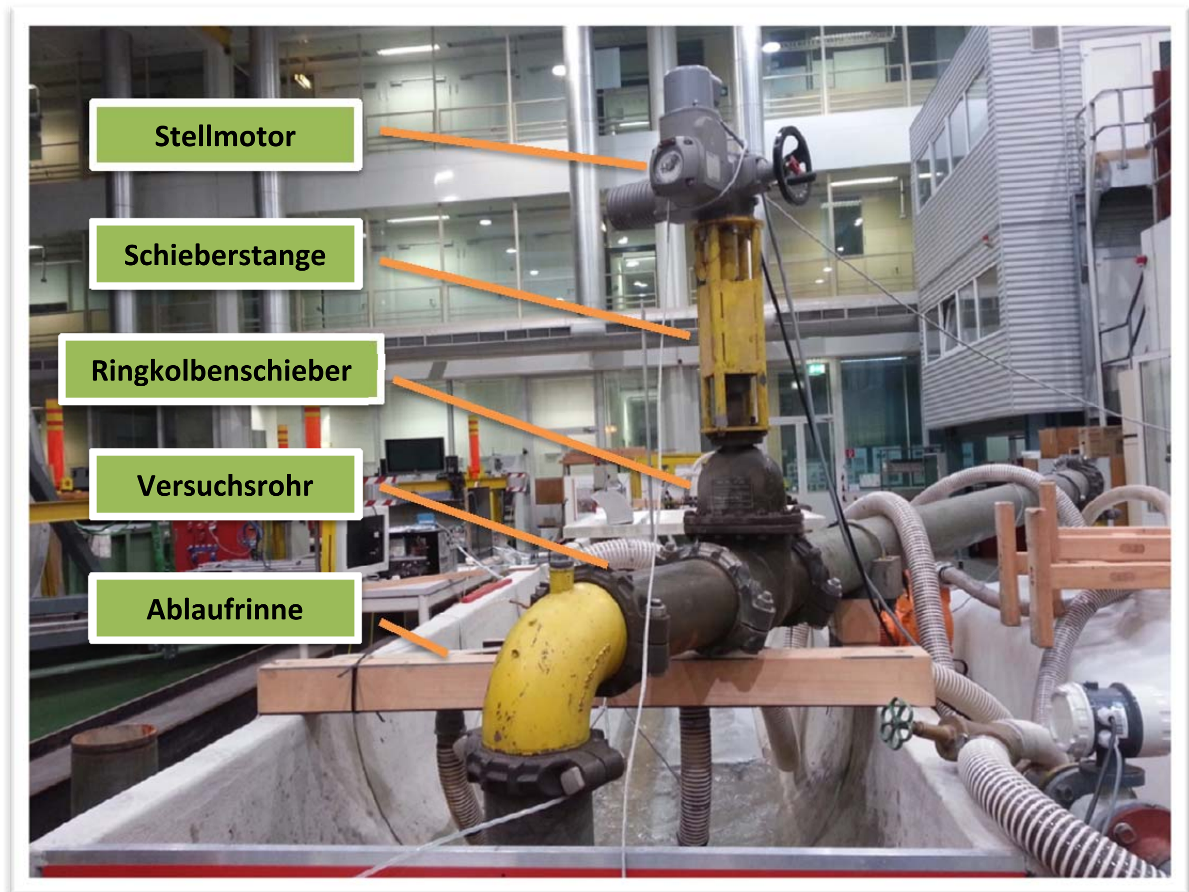

Abb. 6.1.1: Foto vom Versuchsaufbau zur Durchflussregelung im Wasserbaulabor. Zur Durchflussregelung im Versuchsrohr wird ein Simulationsmodell (Digital Output-Block) mit dem Stellmotor eines Schiebers gekoppelt, um das Ventil zu öffnen oder zu schließen.

6.2 Das Simulationsmodell

Das Simulationsmodell dient als Regelkreis zum Ansteuern des Ventils, um den gewünschten Durchfluss zu erhalten. Der SOLL-Durchfluss kann über ein **Edit Text**-Feld in der Benutzeroberfläche eingestellt werden und wird anschließend an das Simulationsmodell übergeben. Der IST-Durchfluss wird von einem Durchflussgeber im Rohr aufgenommen und über eine Messkarte an das Regelungsprogramm übermittelt. Über einen **Mux**-Block[20] können die Werte von SOLL- und IST-Durchfluss in einem **Scope** grafisch dargestellt werden.

[20] Ein *„Mux“*-Block kombiniert in Simulink mehrere Vektoren, sodass diese über das gleiche Anschlusssignal weitergeleitet werden können, um zum Beispiel in einem *„Scope“* gemeinsam dargestellt zu werden.

Ein **PID-Controller** berechnet eine Stellgröße, mit der ein Impuls zum Öffnen oder Schließen des Schiebers erzeugt werden kann. Ist die Stellgröße positiv, wird ein Impuls erzeugt, der an den Anschluss des Schiebers zum Öffnen gesendet wird. Ist die Stellgröße hingegen negativ, wird ein Impuls erzeugt, mit dem das Ventil geschlossen wird.

Als Sicherungselement ist ein **Manual Switch** eingebaut, mit dem der Regelkreis sofort unterbrochen werden kann. Über den *„Stop“*-Button der Benutzeroberfläche kann der Nutzer diese Funktion aufrufen. Daraufhin wird ein Impuls zum Schließen des Ventils ausgesendet. Gleiches gilt für den Button zum Schließen der Anwendung.

Für das Simulationsmodell zur Durchflussregelung wird der Solver *„FixedStepDiscrete“* mit einer Schrittweite von 0,1 verwendet. Die Simulationszeit ist dabei als Angabe für die Anzahl der erfassten Messwerte zu interpretieren. Aus der Schrittweite von 0,1 ergibt sich, dass in einem Zeitschritt 10 Messwerte erfasst werden.

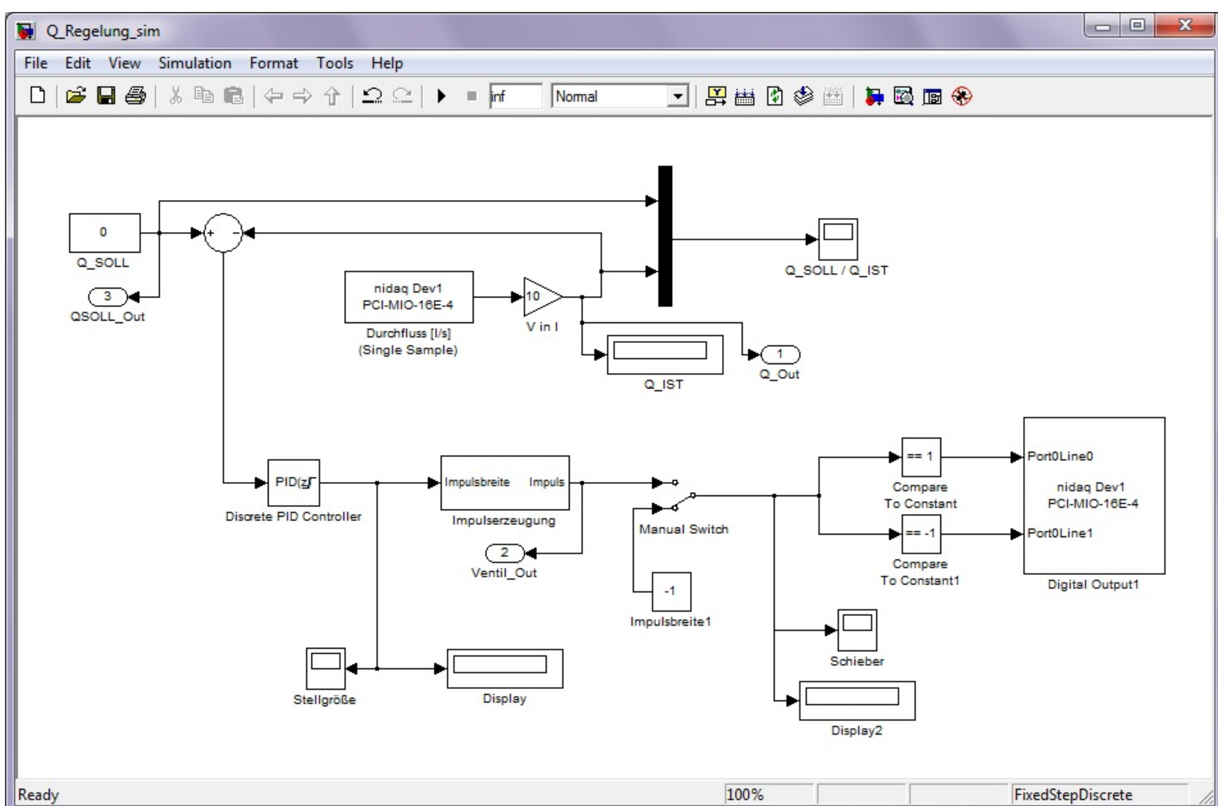

Abb. 6.2.1: Simulationsmodell "Q_Regelung_sim", welches als Regelkreis die Stellgröße bzw. Impulsbreite zum Öffnen oder Schließen des Schiebers ermittelt.

6.2.1 Durchflussmessung

Der aktuelle Durchfluss im Rohr wird über einen Durchflussgeber als Spannung an ein Multifunktions-Datenerfassungsmodul vom Typ NI PCI-MIO-16E-4 *(National Instruments)* übergeben. Der Block **Analog Input (Single Sample)** liest die aufgenommen Werte von der Messkarte in das Simulationsmodell ein. Im Modell wird die übertragene Spannung in den zugehörigen Durchflusswert zurückgerechnet. Die Umrechnung steht in einem Verhältnis von 1:10. Daraus ergibt sich die Auflösung der Messwerterfassung. So wird ein Durchfluss von 10 l/s in eine Spannung von „1 V" konvertiert. Die Auflösung der Durchflussmessung kann im Durchflussgeber eingestellt werden.

Die analogen Komponenten der Data Acquisition Toolbox können Spannungen in einem Bereich von „-10 V" bis „+10 V" aufnehmen. Aufgrund der gewählten Auflösung wird der maximal erfassbare Durchfluss auf 100 l/s begrenzt. Mit dem Programm *„Q_Regelung"* ist daher eine Durchflussregelung auf maximal 100 l/s möglich.

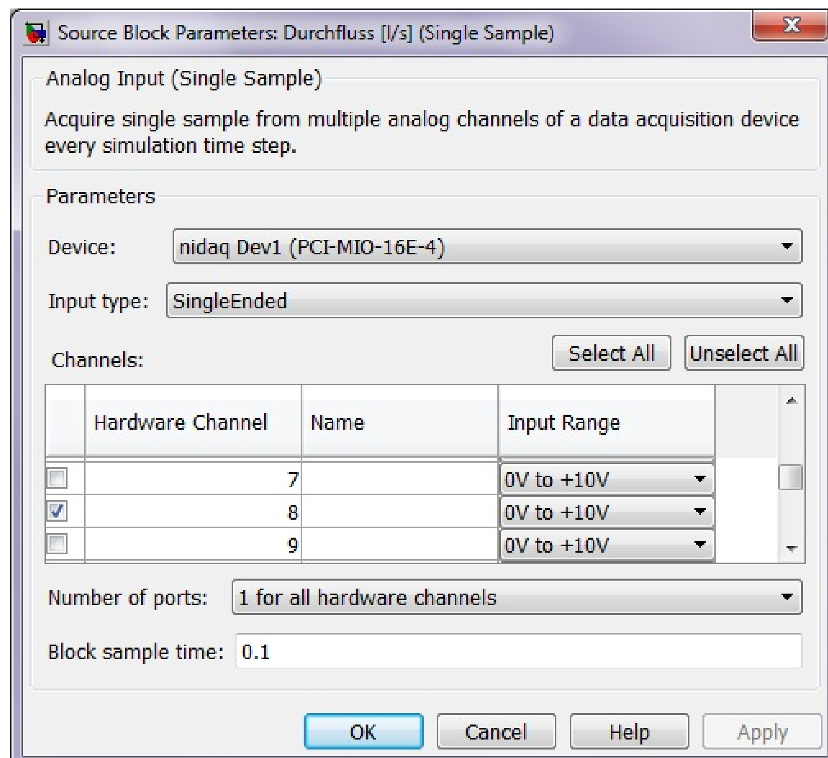

Abb. 6.2.1.1: Parameterfenster vom „Analog Input (Single Sample)"-Block. Die Messdaten werden über den Hardware Channel „CH8" der Messkarte in einem Bereich von „0 V" bis „+10 V" mit einer Sample Time von 0,1 eingelesen. Als Bezugspunkt ist der Massepunkt (0 V) (single ended) gewählt.

6.2.2 PID-Controller

Als Eingangsgröße für den **PID-Controller** wird die Differenz zwischen SOLL- und IST-Durchfluss berechnet. Anhand des errechneten Werts berechnet der **PID-Controller** eine Stellgröße, mit der die Regelung erfolgen kann. Der **PID-Controller** ist so eingestellt, dass er nur einen Proportionalanteil besitzt.[21] So kann zu Ungunsten der Regelzeit ein Überschießen des Durchflusses vermieden werden. Der Regler ist diskret mit einer „Sample Time“ von 0,1 eingestellt und rechnet nach *„Forward Euler“*.

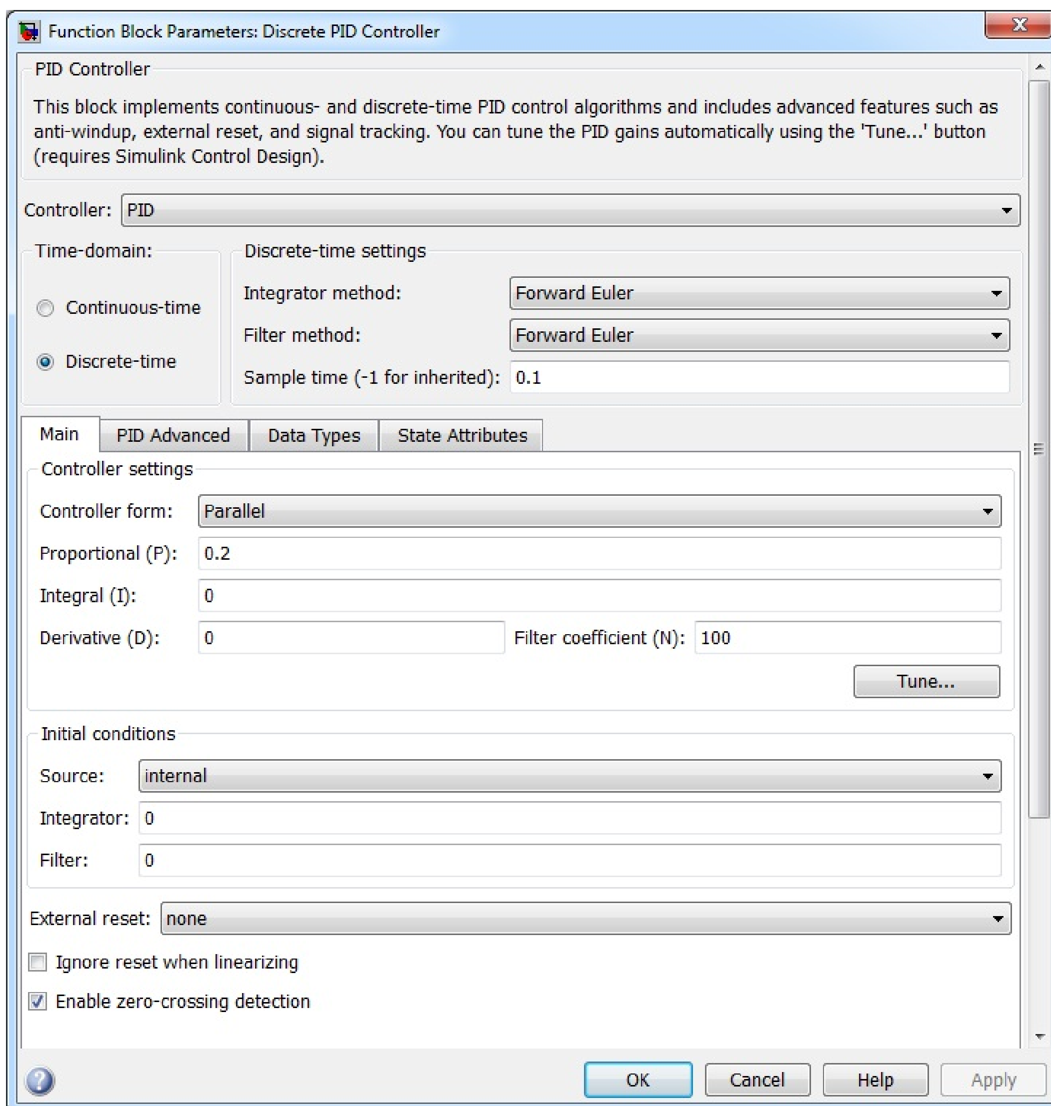

Abb. 6.2.2.1: Parameterfenster zum Einstellen des PID-Controller (eingestellt mit einem Proportionalanteil von 0,2 bei einer Sample Time von 0,1).

[21] Als Proportionalanteil wird ein Wert von 0,2 verwendet. Dieser Wert ist das Resultat einer Feinjustierung für die Soll-Durchflüsse von 5 l/s, 10 l/s, 20 l/s, 25 l/s, 30 l/s. Auf einen integrierenden und differenzierenden Anteil wird aufgrund der Stabilität des Regelkreises verzichtet.

6.2.3 Impulserzeugung

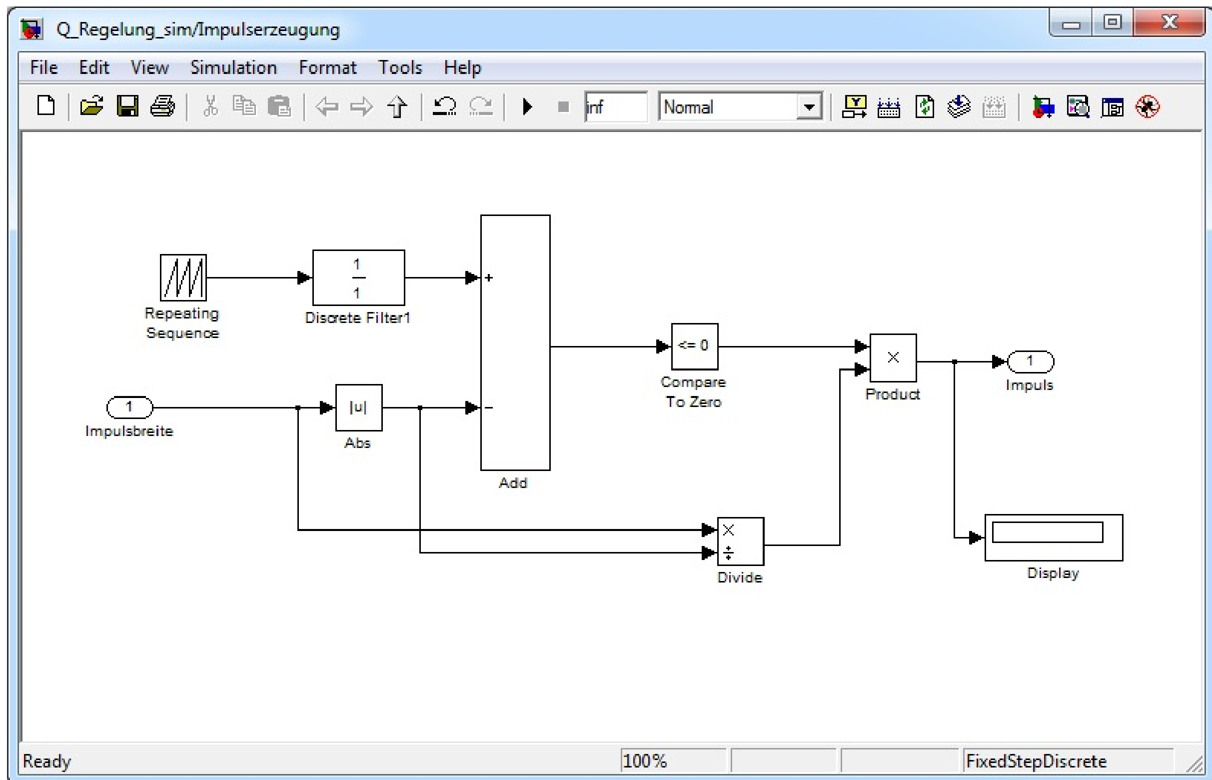

Abb. 6.2.3.1: Subsystem "Impulserzeugung" von "Q_Regelung_sim" zur Modifikation der Stellgröße zu einer Impulsbreite.

Aufgrund der berechneten Stellgröße wird ein Impuls erzeugt, mit dem der Schieber vom Ventil angesteuert wird. Es handelt sich dabei um ein frequenzmoduliertes Rechteck-Signal. Eine **Repeating Sequence** erzeugt eine sich mit einer Periodendauer von 10 Zeitschritten wiederholende Funktion, welcher proportional ansteigende Werte von „0“ bis „11“ zugeteilt werden. Ein Filter diskretisiert anschließend diese Funktion, damit sie in der Simulation unter dem verwendeten Solver (*„FixedStepDiscrete“*[22]) benutzt werden kann. Daraufhin wird der Betrag der errechneten Stellgröße von der ansteigenden Funktion abgezogen. Dies führt zu einer Verschiebung der Funktion nach unten, also in den negativen Bereich.

[22] Eine Diskretisierung der Funktion zur Impulserzeugung muss erfolgen, um die Daten auf eine gemeinsame *Sample Rate* mit den aufgenommenen Daten vom *Analog Input (Single Sample)*-Block zu bringen. Grund ist, dass die *Repeating Sequence* mehr Daten pro Zeitschritt erzeugt, d. h dass für einige Daten der Funktion kein Wert (Stellgröße) vom *PID-Controller* vorhanden ist und somit nicht subtrahiert werden kann. Durch eine Diskretisierung werden die erzeugten Daten der Funktion auf die *Sample Time* des Simulationsmodells, also auch auf die Anzahl, der vom *„Analog Input (Single Sample)“*-Block aufgenommen Daten pro Zeitschritt, reduziert.

Ein **Compare To Zero**-Block erzeugt aus dieser Funktion den entsprechenden Impuls, wobei die Impulsbreiten durch die Nulldurchgänge der Funktion bestimmt werden. Dabei ist die Impulsbreite die Länge der Funktion, in der sie im negativen Bereich verläuft, also vom ersten Null-Durchgang der Funktion bis zum nächsten Null-Durchgang, wonach die Funktion wieder in den positiven Bereich steigt. Für diesen Prozess wurde der **Compare To Zero**-Block mit „<= 0“ eingestellt. Läuft die Funktion also im positiven Bereich, dann wird ein Null-Signal über den **Digital Output**-Block der Data Acquisition Toolbox an den Schieber übertragen.

Das Verhältnis der Stellgröße und der von der **Repeating Sequence** erzeugten Funktion bestimmen die Qualität der Durchflussregelung und die Impulspausen, welche nötig sind, damit sich der Durchfluss im Rohr bis zum nächsten Impuls einstellen kann.

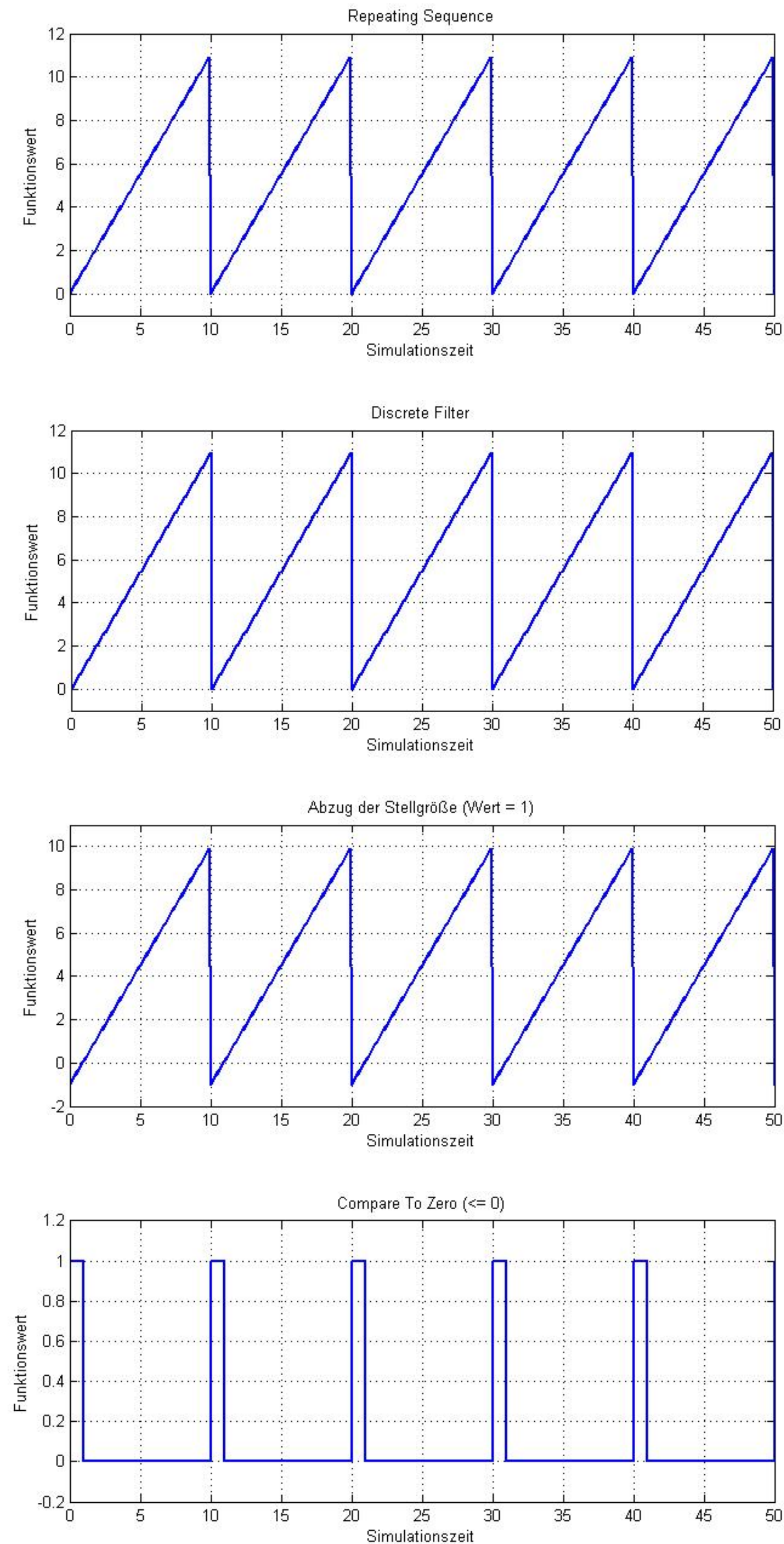

Abb. 6.2.3.2: Darstellung der einzelnen Schritte zur Impulserzeugung und Modifikation der Stellgröße zu einer Impulsbreite[23]

[23] Zur Veranschaulichung wurde eine konstante Stellgröße mit dem Wert „1“ verwendet.

6.2.4 Signalausgabe

Die erzeugten Impulssignale werden an den **Digital Output**-Block weitergeleitet**.** Dieser Block besitzt zwei Eingänge. Einen Eingang über den die Impulse zum Öffnen des Ventils an den Schieber übertragen werden und einen zum Schließen des Ventils. Als Schnittstelle zum Übertragen der digitalen Signale dient ebenfalls die Messkarte vom Typ NI PCI-MIO-16E-4 *(National Instruments)*. So kann über den jeweiligen Anschluss des Schiebers bestimmt werden, in welche Stellrichtung der Schieber gesteuert werden soll. Die Impulsbreite gibt dabei an, wie lange der Schieber angesteuert werden soll.

Um zwischen Öffnen und Schließen des Ventils zu differenzieren, wird das Vorzeichen der Stellgröße an den Impulswert angehängt. Mithilfe von **Compare To Constant**-Blöcken wird unterschieden, an welchen Anschluss des **Digital Output**-Block der Impuls gesendet wird. In den **Compare To Constant**-Blöcken wird gleichzeitig die Digitalisierung der Impulsgrößen vorgenommen. Das bedeutet, dass die mathematischen Werte der Simulation vom Typ *„uint8“* [24] in binäre Zustandsvariablen vom Typ *„boolean“* [25] umgewandelt werden. Die Kanalbelegung für die Anschlüsse ist in der Anlage 1.16 abgebildet.

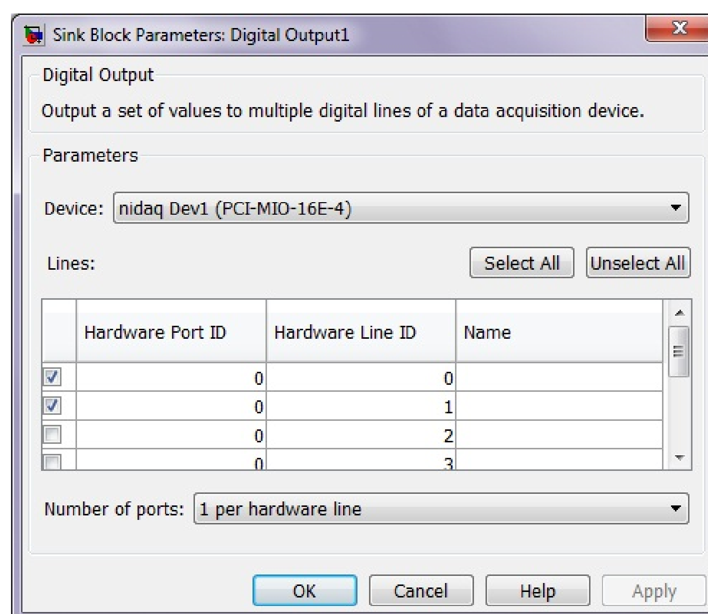

Abb. 6.2.4.1: Parameterfenster vom „Digital Output“-Block Block mit den aktivierten Kanälen „DIO0“ und „DIO1“ zur Ausgabe der Impulsbreite an den Schieberegler

[24] Daten vom Typ *uint8* sind „unsigned 8-bit (1-byte) integers“

[25] Daten vom Typ *boolean* sind Daten mit endlich vielen Werte oder Zuständen („true“ oder „false“)

6.3 Die Benutzeroberfläche

Die Benutzeroberfläche *„Q_Regelung_gui“* dient zum Starten und Stoppen der Durchflussregelung. Bei dem Befehl zum Stoppen ist eine Verzögerung in der Simulation eingebaut, damit über ein **Manual Switch** vor Abbruch der Simulation ein Impuls von „0“ an die Messkarte gegeben werden kann. Bei einem bestehenden Impuls würde sonst die Messkarte nach Beenden des Programms noch weiter Signale an den Schieber des Ventils übertragen.

Der Nutzer kann den gewünschten SOLL-Durchfluss in einem **Edit Text**-Feld oder über eine **Scrollbar** einstellen. Dieser Wert wird unverzüglich und auch während einer laufenden Simulation an das Modell übergeben. Der IST-Durchfluss wird während eines laufenden Prozesses in einem Textfeld angezeigt und in einem Diagramm grafisch dargestellt. Der Status, ob das Ventil geschlossen oder geöffnet wird, zeigt ein Textfeld mit farblicher Kennzeichnung an. Nach beendeter Simulation wird zu dem IST-Durchfluss auch der SOLL-Durchfluss im Diagramm angezeigt.

Die Programmierung der Benutzeroberfläche mit den einzelnen Komponenten befindet sich in der Anlage 1.14.

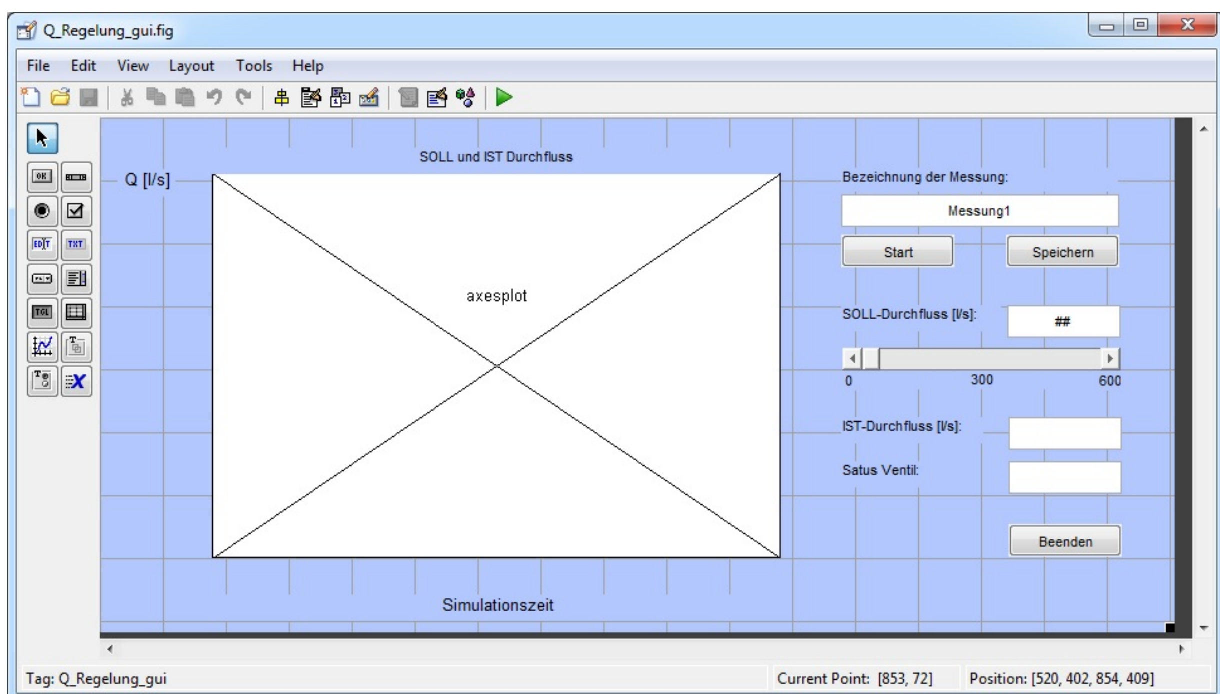

Abb. 6.3.1: Benutzeroberfläche von „Q_Regelung“ im „Dialog Editor“ mit Diagramm zum Anzeigen des IST- und SOLL-Durchflusses und Textfeld zum Einstellen des SOLL-Durchflusses.

6.4 Auswertung der Ergebnisse

Für einen Testlauf der Durchflussregelung wurde in der Benutzeroberfläche ein SOLL-Durchfluss von 25 l/s eingestellt. Zu Beginn des Testlaufes ist das Ventil komplett geschlossen, sodass kein Anfangsdurchfluss vorhanden ist. Nachdem der **PID-Controller** die Stellgröße berechnet hat, wird die Impulsbreite zum Öffnen des Ventils ermittelt. Abhängig vom proportionalen Anteil des **PID-Controller** und der vorgegebenen Periodendauer benötigt der Vorgang zum Einstellen des SOLL-Durchflusses von 25 l/s ca. 50 Sekunden.

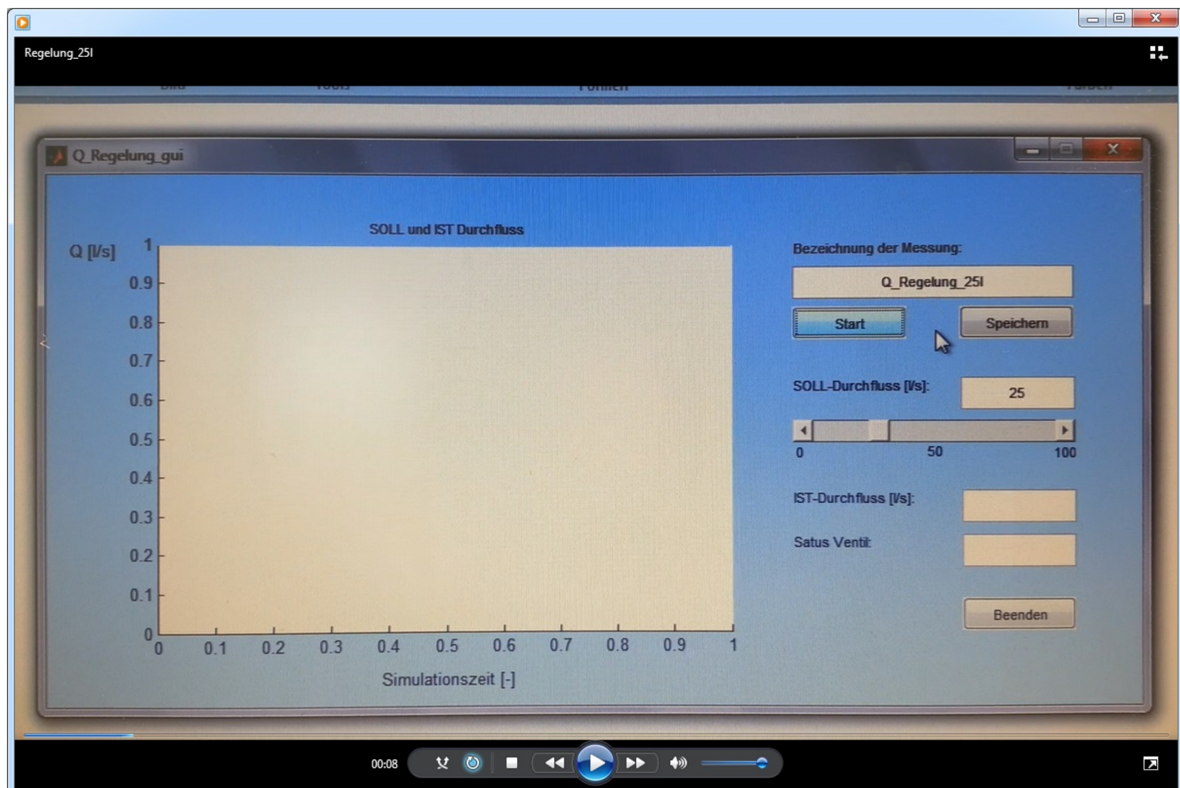

Abb. 6.4.1: Ausschnitt aus der Videoaufnahme zur Durchflussregelung auf Q =25 l/s. Die Videoaufnahme befindet sich auf der beiliegenden CD unter „Auswertung Q_Regelung".

Eine Videoaufnahme, welche sich auf der CD unter *„Auswertung Q_Regelung"* befindet, zeigt die Benutzeroberfläche während der Regelung. Der IST-Durchfluss wird in einem Textfeld der Oberfläche angezeigt. Zusätzlich wird ein Graph gezeichnet, der alle Durchflussmessungen darstellt (grün). Ein weiteres Textfeld zeigt mit farblicher Unterlegung den Status des Ventils, also ob der Schieber öffnet oder schließt. Nachdem das Programm zur Durchflussregelung gestoppt wurde, wird zusätzlich der SOLL-Durchfluss angezeigt (rot).

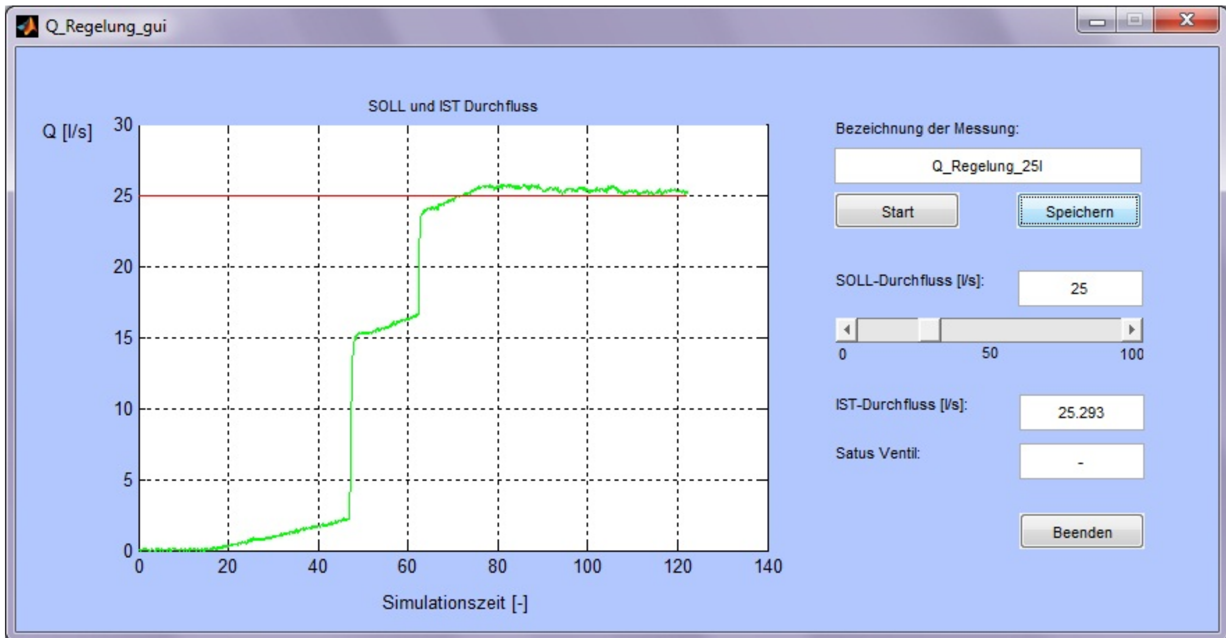

Abb. 6.4.2: Programmoberfläche nach erfolgreicher Durchflussregelung mit einem SOLL-Durchfluss von 25 l/s (rot) und der Darstellung des IST-Durchflusses (grün).

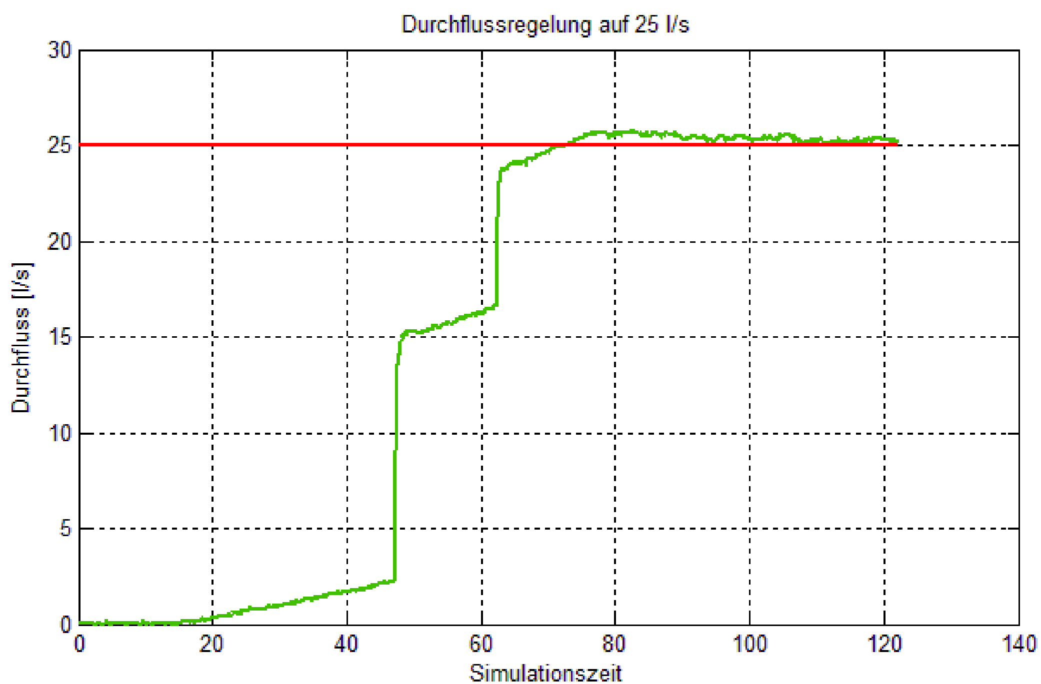

Abb. 6.4.3: Darstellung vom IST- und SOLL-Durchfluss der Durchflussregelung auf Q =25 l/s

In der Abbildung 6.4.3 ist der IST-Durchfluss zu erkennen, welcher in Abhängigkeit von dem Proportionalanteil des **PID-Controllers** (0,2) und der Impulsbreite bzw. der Periodendauer geregelt wird. Da der Schieber nicht proportional öffnet, ist in der Anfangsphase des Regelungsprozesses der Anstieg des Durchflusses noch gering. Erst bei späteren Öffnungen des Ringkolbenschiebers treten stärkere Durchflusssteigerungen ein, die sich im Graphen durch sprunghafte Anstiege abzeichnen. Für den Regelungsprozess ist aufgrund der Impulsbreite und der gesamten Periodendauer eine Pause eingebaut, wodurch sich der Durchfluss bis zum nächsten Impuls einstellen kann.

Nach einem geringen Überschießen des Durchflusses regelt der **PID-Controller** langsam nach, sodass es nicht zu einer Hysterese kommt. Im weiteren Verlauf der Messung bleibt der Durchfluss bei einem Mittelwert von 25,26 l/s konstant. Die erzeugten Impulse, welche den Schieber ansteuern, sind in der Abbildung 6.4.4 dargestellt.

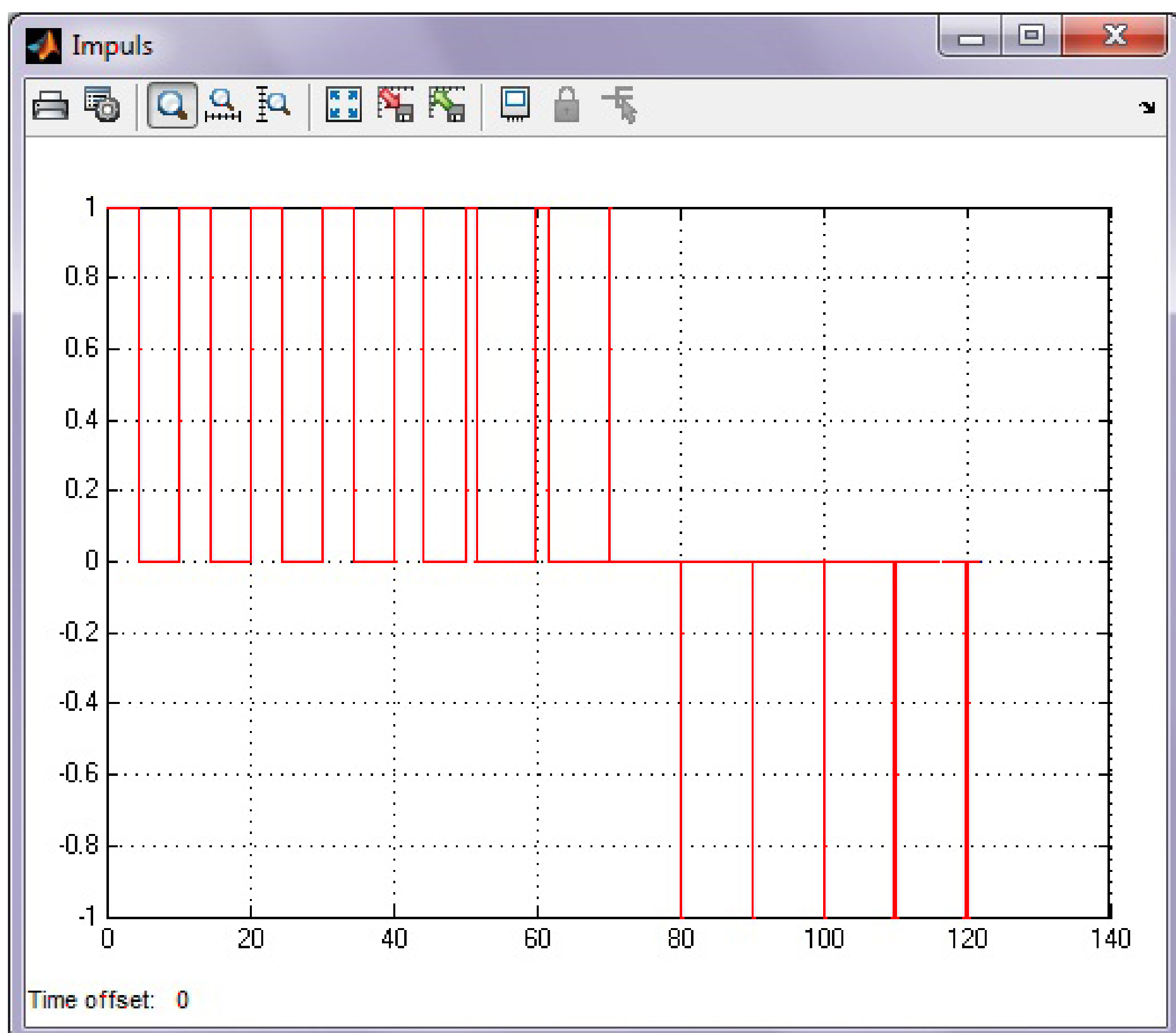

Abb. 6.4.4: Darstellung der Impulsübertragung an den Schieber für Regelung auf Q = 25 l/s

In dem Simulationsmodell wurde für den Solver eine Schrittweite von „0,1“ eingestellt. Daher werden pro Zeitschritt zehn Durchflussmessungen aufgenommen. Die Auswertung der Daten bezieht sich auf die Messreihe, nachdem sich der IST-Durchfluss eingestellt hat und kein Signal mehr an den Schieber gesendet wird. Der Mittelwert des gemessenen Durchflusses nach 115 Simulationszeitschritten (real: 50 Sekunden mit 1150 Messungen) beträgt 25,26 l/s, bei einer mittleren Abweichung von 0,08. Die Standardabweichung der Messreihe liegt bei 0,101. Durch eine Abweichung des gemittelten Durchflusses nach Ende der Regelung von 0,64 % vom SOLL-Durchfluss ist das Programm *„Q_Regelung“* validiert.

Tabelle 6.4-1: Auswertung der Durchflussmessung nach Erreichen SOLL-Durchfluss 25 l/s

Auswertung Durchflussmessung nach Regelung	
SOLL-Durchfluss [l/s]	25
Mittelwert IST-Durchfluss [l/s]	25,26
prozentuale Abweichung [%]	1,05
mittl. Abweichung [-]	0,080
Standardabweichung [-]	0,101
Q erreicht nach [STIME]	115

Zur Verifizierung des Programms wurden weitere Testläufe mit den SOLL-Durchflüssen von 5 l/s, 10 l/s und 30 l/s durchgeführt. Eine Excell-Tabelle zu der Messwerterfassung von allen Testläufen sowie die Ascii-Dateien und die MATLAB Vektoren der aufgenommenen Werte (Simulationszeit, SOLL-Durchfluss, IST-Durchfluss) befinden sich ebenfalls auf der CD unter *„Auswertung Q_Regelung“*. Die Graphen der weiteren Testläufe sind in der Abb. 6.4.5. dargestellt.

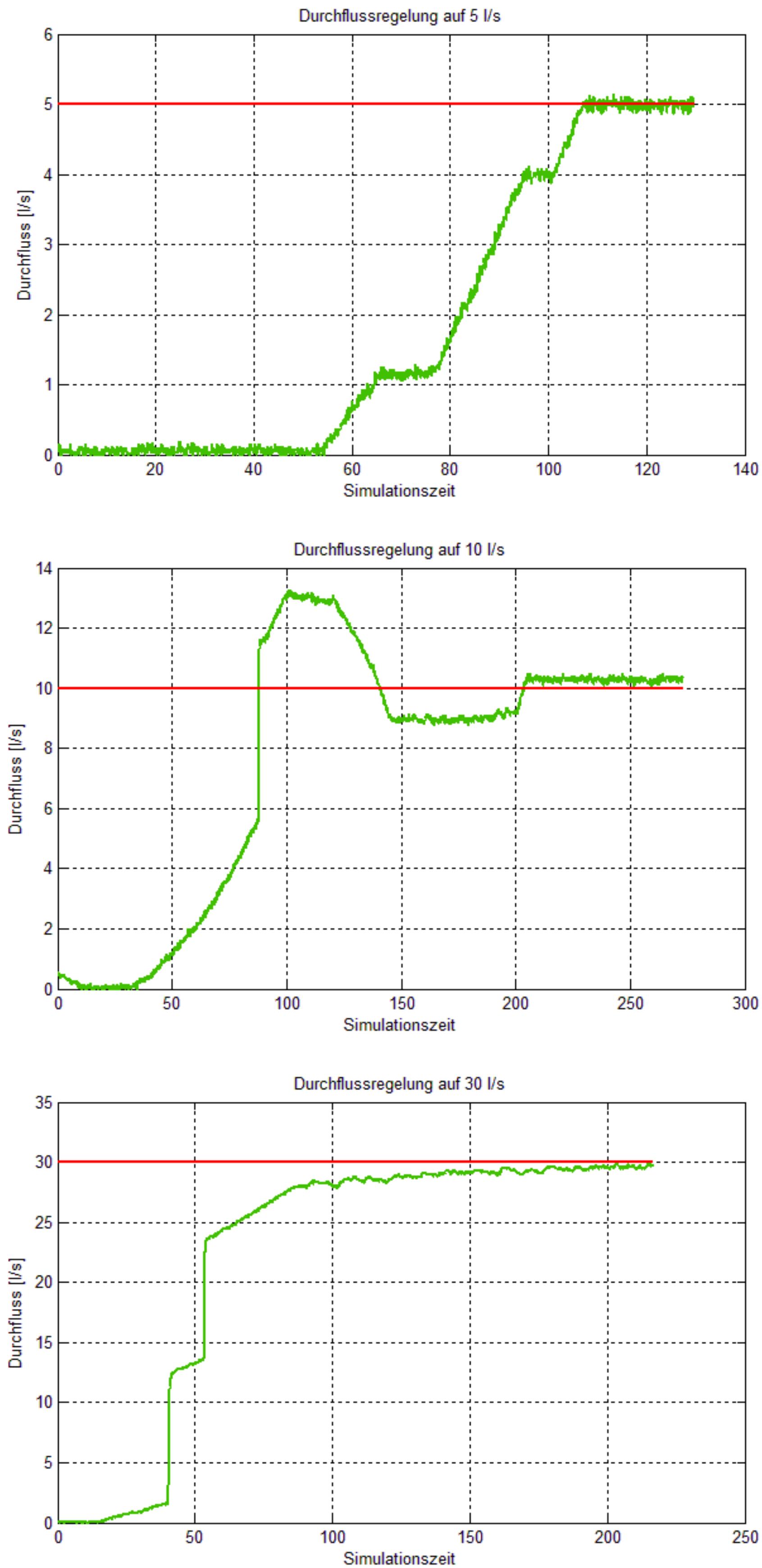

Abb. 6.4.5: Darstellung der IST- und SOLL-Durchflüsse für 5 l/s, 10 l/s und 30 l/s

Die Auswertung der Testläufe nach erfolgreicher Durchflussregelung zeigt, dass das Programm zur Durchflussregelung *„Q_Regelung“* für andere SOLL-Durchflüsse reproduzierbar ist. Die Schlussfolgerung der Versuchsreihe ist, dass das Programm *„Q_Regelung“* verifiziert ist und die MATLAB Software mit der Kopplung von Hardwarekomponenten des Wasserbaulabors zu einer Durchflussregelung geeignet ist.

Tabelle 6.4-2: Auswertung der Durchflussmessungen aller Testläufe nach Erreichen vom SOLL-Durchfluss

SOLL-Durchfluss [l/s]	Mittelwert IST-Durchfluss [l/s]	Abweichung [l/s]	mittlere Abweichung [-]	Standardabweichung [-]
5	4,99	-0,01 (0,16 %)	0,047	0,059
10	10,29	+0,29 (2,91 %)	0,064	0,078
25	25,26	+0,26 (1,05 %)	0,080	0,101
30	29,64	-0,36 (1,20 %)	0,110	0,134

6.5 Vergleich mit LabView

Zur Validierung und Verifizierung des Programms *„Q_Regelung“* wurde eine Durchflussregelung mit LabView durchgeführt. Verglichen wird dabei die Regelung auf 25 l/s. In der Abb. 6.5.1 ist ein ähnliches Verhalten des Regelungsprozesses zu der Durchflussregelung mit dem MATLAB Programm zu erkennen. Die Dauer, bis zum Einstellen des SOLL-Durchflusses, mit ca. 55 Sekunden ist in etwa dieselbe.

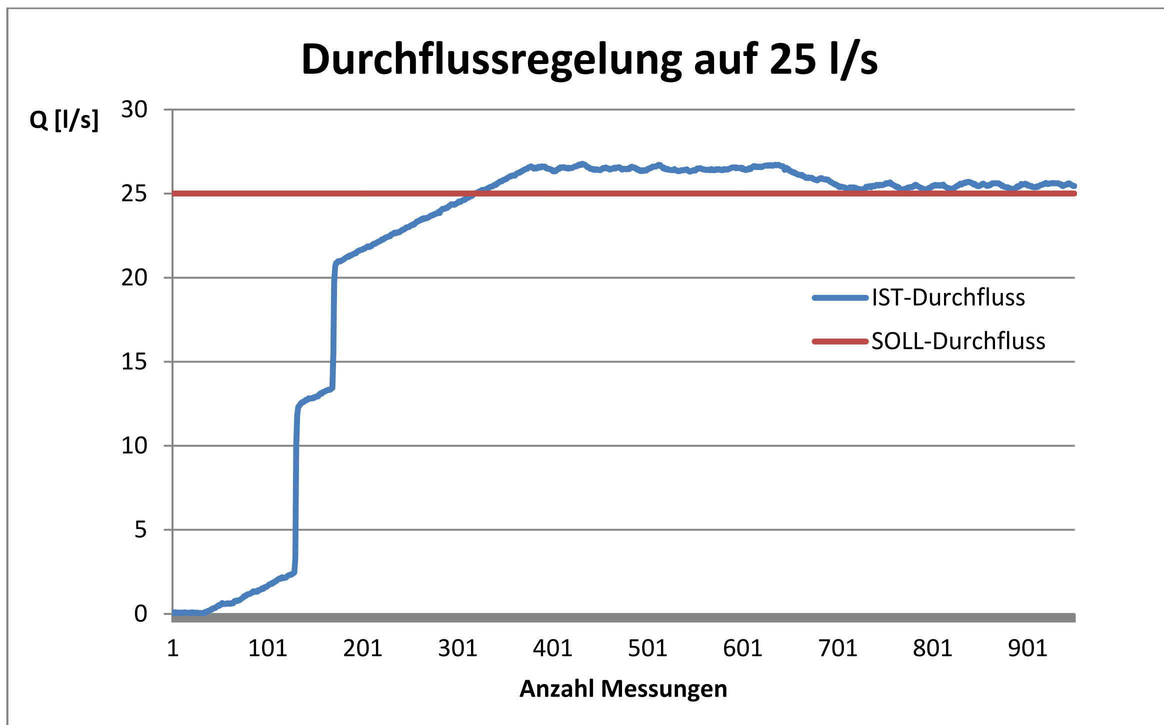

Abb. 6.5.1: Darstellung vom IST- und SOLL-Durchfluss der Regelung mit LabView auf Q =25 l/s

Zum Vergleich der beiden Anwendungsprogramme werden die Durchflussmessungen nach dem Einstellen des SOLL-Durchflusses ausgewertet. Der gemittelte Durchfluss nach der LabView Regelung liegt mit 25,46 l/s um 0,46 l/s (1,84 %) nur marginal über dem eingestellten SOLL-Durchfluss von 25 l/s.

Der gemittelte Durchfluss nach der LabView Regelung ist um 0,79 % (0,2 l/s) höher, als der Durchfluss, der sich nach der MATLAB Regelung einstellt. Die Abweichung ist unter anderem damit zu begründen, dass der maximal mögliche Durchfluss der Versuchsanlage manuell und somit nicht immer exakt eingestellt werden konnte.[26] Dies führt zu ungleichen Druckbelastungen im Versuchsrohr und somit zu einem unterschiedlichen Verhalten des Regelkreises.

[26] Das Versuchsrohr im Wasserbaulabor wurde über einen Hochbehälter angesteuert, welcher von einer Pumpe gefüllt wird. Vor dem Versuchsrohr befindet sich ein Ringkolbenschieber, welcher für einen maximalen Durchfluss von ca. 35 l/s geöffnet wurde.

Tabelle 6.5-1: Vergleich der Durchflussregelungen auf Q = 25 l/s mit MATLAB und LabView

Software	SOLL-Durchfluss [l/s]	Mittelwert IST-Durchfluss [l/s]	Prozentuale Abweichung [%]
LabView	25	25,46	1,82
MATLAB	25	25,26	1,04

Als Schlussfolgerung der Testversuche mit beiden Programmen kann festgehalten werden, dass eine Durchflussregelung mit MATLAB genauso erfolgsversprechend ist, wie eine Regelung mit LabView. Mit dem Ergebnis wird auch bestätigt, dass sich die MATLAB Software mit den Hardwarekomponenten des Wasserbaulabors (Durchflusssensoren, Schiebereglern) koppeln lässt. Aufgrund der geringen Abweichungen vom SOLL-Durchfluss und der Vergleichsmessungen mit LabView, ist eine Durchflussregelung mit den erstellten MATLAB Programmen validiert und verifiziert.

7. Fazit und Ausblick

In dieser Arbeit wurde gezeigt, dass eine Kopplung der Messhardware im Wasserbaulabor der Universität der Bundeswehr München mit der MATLAB Software über die „Data Acquisition Toolbox“ möglich ist. Über grafische Benutzeroberflächen kann so ein anwenderfreundliches Programm entworfen werden. Zu Testzwecken wurden MATLAB Programme zur Durchflussmessung und Durchflussregelung geschrieben, welche anhand von mehreren Versuchsläufen und einem Vergleich mit der LabView Software validiert und verifiziert werden konnten.

Die Messergebnisse der Testläufe zur Durchflussregelung mit den Programmen von LabView und MATLAB weichen in einem Bereich von unter einem Prozent ab. Somit ist eine Kopplung über die MATLAB Software mit den Hardwarekomponenten des Wasserbaulabors als Alternative zu LabView anwendbar. Unter der Verwendung von MATLAB bestehen zudem noch vielfältige Darstellungs- und Auswertungsmöglichkeiten, da die grafischen Benutzeroberflächen auf Basis der objektorientierten Programmierung erzeugt werden.

Einige Komponenten der MATLAB Oberfläche zur Datenübertragung und –verarbeitung sind derzeit noch nicht vollständig von MathWorks implementiert worden. Daher kann zum Beispiel mithilfe des „EventListener“ nur ein Wert von einem laufenden Simulationsmodell an eine Benutzeroberfläche übergeben werden. Wenn diese Funktion auf eine gleichzeitige Übertragung von mehreren Daten erweitert werden würde, wäre es in Zukunft möglich, ein umfangreiches Analyseprogramm über die Netzeinspeisung einer Steffturbine zu erstellen.

Weiterhin könnte sogar eine Optimierung der Steffturbine während des laufenden Betriebs über eine Regelung einzelner Komponenten möglich sein, um einen maximalen Wirkungsgrad der Turbine zu erreichen.

Im Anhang 1.17 ist dargestellt, wie ein mögliches Analyseprogramm aufgebaut sein könnte. Im Rahmen dieser Arbeit entstanden zudem ein Simulationsmodell zum Erfassen der Messwerte und die Programmierung zur Übergabe der Werte an die Benutzeroberfläche. So besteht die Möglichkeit an den Vorarbeiten des Analyseprogramms anzuknüpfen, falls MathWorks die Verwendung mehrerer „EventListener“ ermöglicht.

Abbildungsverzeichnis

Abb. 2.1.1: Auswahlfenster (GUIDE Quick Start) mir vordefinierten GUI's 3

Abb. 2.1.2: Dialog Editor mit Darstellung der Steuerelemente 4

Abb. 2.1.3: Benutzeroberfläche von „Beispiel1" im Dialog Editor mit dem Button „Beenden" 5

Abb. 2.3.1: Data Acquisition Toolbox im Simulink Library Browser mit den zur Verfügung stehenden Komponenten 8

Abb. 2.3.2: Parameterfenster vom „Analog Input"-Block 9

Abb. 2.3.3: Parameterfenster vom „Analog Output"-Block 10

Abb. 2.3.4: Darstellung einer Sinusfunktion (hoch und niedrig aufgelöst) 11

Abb. 2.3.5: Parameterfenster vom „Digital Input"-Block mit Auflistung der verfügbaren „Hardware Channels" 12

Abb. 2.3.6: Parameterfenster vom „Digital Output"-Block mit Auflistung der verfügbaren „Hardware Channels" 13

Abb. 2.4.1: Multifunktions-Datenerfassungsmodul NI PCI-MIO-16E-4 (National Instruments Germany GmbH, 2013)..14

Abb. 2.4.2: Anzeige des Measurement & Automation Explorer von dem Multifunktions-Datenerfassungsmodul NI PCI-MIO-16E-4 (National Instruments) 14

Abb. 2.4.3: Data Acquisition-Board NI CB-68LP (National Instruments Germany GmbH, 2013)...........15

Abb. 3.1.1: Simulationsmodell „Beispiel2_sim.mdl". Hinzugefügt wurde ein „Sine Wave"-Block zum Erzeugen der Sinusfunktion, ein „Gain" zum Verstärken der Funktion und ein „Scope" zum Darstellen und Speichern der Ergebnisse. 17

Abb. 3.1.2: Parameterfester vom „Scope" zum Speichern der eingehenden Daten. Die Daten werden als Array (als Vektor) unter dem Namen „SineWave" gespeichert. 17

Abb. 3.2.1: Benutzeroberfläche von „Beispiel2" im „Dialog Editor". Über die Buttons wird das Simulationsmodell gesteuert. Über das Eingabefeld kann der Verstärkungsfaktor geändert werden. Die aufgenommenen Daten werden nach Ende der Simulation im Diagramm „axes" grafisch dargestellt. 19

Abb. 3.5.1: Programmfenster „Beispiel2_gui" nach einer durchlaufenden Simulation mit Verstärkungsfaktor „1". 26

Abb. 3.6.1: Programmfenster „Beispiel3_gui" mit einem Button zum Starten und Stoppen der Simulation. 27

Abb. 3.7.1: Benutzeroberfläche „Beispiel4_gui" im „Dialog Editor" mit Textfeld zum Anzeigen des aktuellen Funktionswerts und „Plot"-Fenster zum Aufzeichnen aller Punkte. 30

Abb. 3.7.2: Simulationsmodell „Beispiel4_sim“ mit einem Block zum Erzeugen der Sinusfunktion, einem „Gain“ zur Verstärkungsmöglichkeit und einem „Outport“-Block zum Übergeben der Werte mithilfe des „EventListener“. 31

Abb. 3.7.3: „StartFcn“ von „Beispiel4_sim“ zur Erstellung des „EventListener“ 34

Abb. 3.7.4: Darstellung des „EventListener“ am „Outport“-Block "SinusOut" 34

Abb. 3.7.5: Programmfenster zu „Beispiel4_gui“ während einer laufenden Simulation. Angezeigt werden der akutelle Funktionswert der erzeugten Sinuskurve in einem Textfeld und die aufgezeichneten Punkte im Diagrammfenster. Der Verstärkungsfaktor der Sinuskurve kann über ein weiteres Textfeld verändert werden. 39

Abb. 4.1.1: Schematische Darstellung vom Versuchsaufbau zur Kopplung des Simulationsmodells mit einem Oszilloskop 41

Abb. 4.2.1: Simulationsmodell „sinus_sim.mdl“ mit analogem Ein- und Ausgang, einem Block zum Erzeugen der Sinusfunktion, zwei To File Blöcken zum Auslagern der aufgenommenen Daten und zwei Blöcken zum Darstellen der Datenflüsse 41

Abb. 4.2.2: Parameterfenster des „Solver“ mit der Einstellung „Fixed-step discrete“ und einer Schrittweite von „1“. 42

Abb. 4.2.3: Parameterfenster vom „Sine Wave“-Block zum Erzeugen der Sinusfunktion auf Basis der Simulationszeit mit einer Amplitude von „1“ und einer Frequenz von 0,01 s^{-1}. 43

Abb. 4.2.4: Von einem Oszilloskop gemessene Sinuskurve während einer laufenden Simulation. 44

Abb. 4.2.5: Parameterfenster vom „Analog Output (Single Sample)“-Block mit einer „Output Range“ von „-10V“ bis „+10V“ am aktivierten Hardware Channel „DAC0“ 45

Abb. 4.2.6: Parameterfenster vom „Analog Input (Single Sample)“-Block mit einer „Input Range“ von „-10V“ bis „+10V“ am aktivierten Hardware Channel „CH0“ 45

Abb. 4.3.1: Benutzeroberfläche von „Sinus_GUI“ im „Dialog Editor“ mit zwei Button zum Steuern des Simulationsmodells und zwei Plot-Fenster zur Darstellung der erzeugten und aufgenommenen Sinuskurve. 46

Abb. 4.3.2: Darstellung der Benutzeroberfläche nach einer durchlaufenden Simulation mit erzeugter und aufgenommener Sinuskurve. 47

Abb. 4.4.1: Darstellung der erzeugten Sinusfunktion ("sinus_export") 50

Abb. 4.4.2: Darstellung der erfassten Sinusfunktion ("sinus_import") 50

Abb. 5.2.1: Simulationsmodell „Q_Messen_sim.mdl“ mit „Analog Input (Single Sample)“ zum Aufnehmen der Durchflussmessung mit einem IDM, „Gain“-Block zum Umrechnen der Spannung [V] in Durchfluss [l/s], und Blöcken zum Anzeigen, Darstellen und Übergeben der aufgenommen Werte. 52

Abb. 5.2.2: „StartFcn“ von „Q_Messen_sim“ mit Quelltext zur Erstellung des „EventListener“ 53

Abb. 5.2.3: Parameterfenster vom „Analog Output (Single Sample)“-Block mit einer „Input Range“ von „0V“ bis „+10V“ am aktivierten Hardware Channel „CH8“ bei einer Sample Time von 0,1. Als Bezugspunkt ist der Massepunkt (0 V) gewählt (single ended). 54

Abb. 5.3.1: Benutzeroberfläche von „Q_Messen“ im „Dialog Editor“ mit Textfeld zum Anzeigen des aktuellen Durchflusses und „Plot“-Fenster zum Darstellen der aufgezeichneten Durchflussmessung. 55

Abb. 5.4.1: Programmfenster „Q_Messen_gui“ nach einer Simulation (bei 25 l/s) mit der aufgezeichneten Durchflussmessung. 56

Abb. 5.4.2: Graph der Durchflussmessung bei Q = 25 l/s 57

Abb. 5.5.1: Graph der Durchflussmessung mit LabView bei Q =25 l/s 58

Abb. 6.1.1: Foto vom Versuchsaufbau zur Durchflussregelung im Wasserbaulabor. Zur Durchflussregelung im Versuchsrohr wird ein Simulationsmodell (Digital Output-Block) mit dem Stellmotor eines Schiebers gekoppelt, um das Ventil zu öffnen oder zu schließen. 61

Abb. 6.2.1: Simulationsmodell "Q_Regelung_sim", welches als Regelkreis die Stellgröße bzw. Impulsbreite zum Öffnen oder Schließen des Schiebers ermittelt. 62

Abb. 6.2.1.1: Parameterfenster vom „Analog Input (Single Sample)“-Block. Die Messdaten werden über den Hardware Channel „CH8“ der Messkarte in einem Bereich von „0 V“ bis „+10 V“ mit einer Sample Time von 0,1 eingelesen. Als Bezugspunkt ist der Massepunkt (0 V) (single ended) gewählt. 63

Abb. 6.2.2.1: Parameterfenster zum Einstellen des PID-Controller (eingestellt mit einem Proportionalanteil von 0,2 bei einer Sample Time von 0,1). 64

Abb. 6.2.3.1: Subsystem "Impulserzeugung" von "Q_Regelung_sim" zur Modifikation der Stellgröße zu einer Impulsbreite. 65

Abb. 6.2.3.2: Darstellung der einzelnen Schritte zur Impulserzeugung und Modifikation der Stellgröße zu einer Impulsbreite 67

Abb. 6.2.4.1: Parameterfenster vom „Digital Output“-Block Block mit den aktivierten Kanälen „DIO0“ und „DIO1“ zur Ausgabe der Impulsbreite an den Schieberegler 68

Abb. 6.3.1: Benutzeroberfläche von „Q_Regelung“ im „Dialog Editor“ mit Diagramm zum Anzeigen des IST- und SOLL-Durchflusses und Textfeld zum Einstellen des SOLL-Durchflusses. 69

Abb. 6.4.1: Ausschnitt aus der Videoaufnahme zur Durchflussregelung auf Q =25 l/s. Die Videoaufnahme befindet sich auf der beiliegenden CD unter „Auswertung Q_Regelung“. 70

Abb. 6.4.2: Programmoberfläche nach erfolgreicher Durchflussregelung mit einem SOLL-Durchfluss von 25 l/s (rot) und der Darstellung des IST-Durchflusses (grün). 71
Abb. 6.4.3: Darstellung vom IST- und SOLL-Durchfluss der Durchflussregelung auf Q =25 l/s 71
Abb. 6.4.4: Darstellung der Impulsübertragung an den Schieber für Regelung auf Q = 25 l/s 72
Abb. 6.4.5: Darstellung der IST- und SOLL-Durchflüsse für 5 l/s, 10 l/s und 30 l/s 74
Abb. 6.5.1: Darstellung vom IST- und SOLL-Durchfluss der Regelung mit LabView auf Q =25 l/s..........76

Tabellenverzeichnis

Tabelle 3.7-1: Liste der „Event“-Bedingungen zum Übertragen von Daten während einer laufenden Simulation (MathWorks) 33
Tabelle 4.4-1: Min. und Max. der erfassten Sinusfunktion ("sinus_import") 48
Tabelle 4.4-2: Auszug der Datenerfassung vom Beispiel „Sinus_GUI“ 49
Tabelle 5.4-1-1: Auswertung der Messung mit MATLAB bei Q = 25 l/s 57
Tabelle 6.4-1: Auswertung der Durchflussmessung nach Erreichen SOLL-Durchfluss 25 l/s 73
Tabelle 6.5-1: Vergleich der Durchflussregelungen auf Q = 25 l/s mit MATLAB und LabView 77

Quellenverzeichnis

MathWorks. (03. April 2013). *www.mathworks.de.* Abgerufen am 03. April 2013 von http://www.mathworks.de/de/help/matlab/creating_guis/customizing-callbacks-in-guide.html

MathWorks. (14. April 2013). *www.mathworks.de.* Abgerufen am 14. April 2013 von http://www.mathworks.de/de/help/simulink/slref/add_exec_event_listener.html?searchHighlight=add_exec_event_listener

MathWorks. (12. Juni 2013). *www.mathworks.de.* Abgerufen am 12. Juni 2013 von http://www.mathworks.de/de/help/matlab/creating_guis/customizing-callbacks-in-guide.html

MathWorks, Inc. (16. August 2012). MATLAB R2012b.

National Instruments Germany GmbH. (01. Mai 2002). DAQ - AT E Series User Manual.

National Instruments Germany GmbH. (18. Mai 2013). *ni.com*. Abgerufen am 18. Mai 2013 von http://sine.ni.com/nips/cds/view/p/lang/de/nid/1187

Anlagenverzeichnis

Anlage 1: Abbildungen und Quelltexte XI

1.1 Vordefiniertes M-File („Beispiel1_gui.m“) 1

1.2 Property Inspector 3

1.3 68-Pin MIO Connector Pin Assignments 4

1.4 Quelltext „Beispiel1_gui.m“ 5

1.5 Quelltext „Beispiel2_gui.m“ 7

1.6 Quelltext „Beispiel3_gui.m“ 10

1.7 Quelltext „Beispiel4_gui.m“ 13

1.8 Quelltext „update_gui.m“ von Beispiel4 17

1.9 Quelltext „Sinus_gui.m“ 18

1.10 Kanalbelegung Data Acquisition-Board „Sinus_GUI“ 21

1.11 Quelltext „Q_Messen_gui.m“ 22

1.12 Quelltext „update_gui.m“ von „Q_Messen“ 26

1.13 Kanalbelegung Data Acquisition-Board „Q_Messen“ 27

1.14 Quelltext „Q_Regelung_gui.m“ 28

1.15 Quelltext „update_gui.m“ von „Q_Regelung“ 34

1.16 Kanalbelegung Data Acquisition-Board „Q_Regelung“ 35

1.17 Benutzeroberfläche vom Analyseprogramm Steffturbine 36

1.18 Quelltext „update_gui.m“ von „Steff_gui“ 37

Anlage 2: CD mit Beispiel- und Testprogrammen XII

Anlage 1: Abbildungen und Quelltexte

1.1 Vordefiniertes M-File („Beispiel1_gui.m“)

```
function varargout = Beispiel1_gui(varargin)
% BEISPIEL1_GUI MATLAB code for Beispiel1_gui.fig
%      BEISPIEL1_GUI, by itself, creates a new BEISPIEL1_GUI or raises the
existing
%      singleton*.
%
%      H = BEISPIEL1_GUI returns the handle to a new BEISPIEL1_GUI or the
handle to
%      the existing singleton*.
%
%      BEISPIEL1_GUI('CALLBACK',hObject,eventData,handles,...) calls the
local
%      function named CALLBACK in BEISPIEL1_GUI.M with the given input
arguments.
%
%      BEISPIEL1_GUI('Property','Value',...) creates a new BEISPIEL1_GUI or
raises the
%      existing singleton*.  Starting from the left, property value pairs
are
%      applied to the GUI before Beispiel1_gui_OpeningFcn gets called.  An
%      unrecognized property name or invalid value makes property
application
%      stop.  All inputs are passed to Beispiel1_gui_OpeningFcn via
varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help Beispiel1_gui

% Last Modified by GUIDE v2.5 23-Dec-2012 04:59:13

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @Beispiel1_gui_OpeningFcn, ...
                   'gui_OutputFcn',  @Beispiel1_gui_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT
```

```
% --- Executes just before Beispiel1_gui is made visible.
function Beispiel1_gui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to Beispiel1_gui (see VARARGIN)

% Choose default command line output for Beispiel1_gui
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes Beispiel1_gui wait for user response (see UIRESUME)
% uiwait(handles.Beispiel1_gui);


% --- Outputs from this function are returned to the command line.
function varargout = Beispiel1_gui_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;
```

1.2 Property Inspector

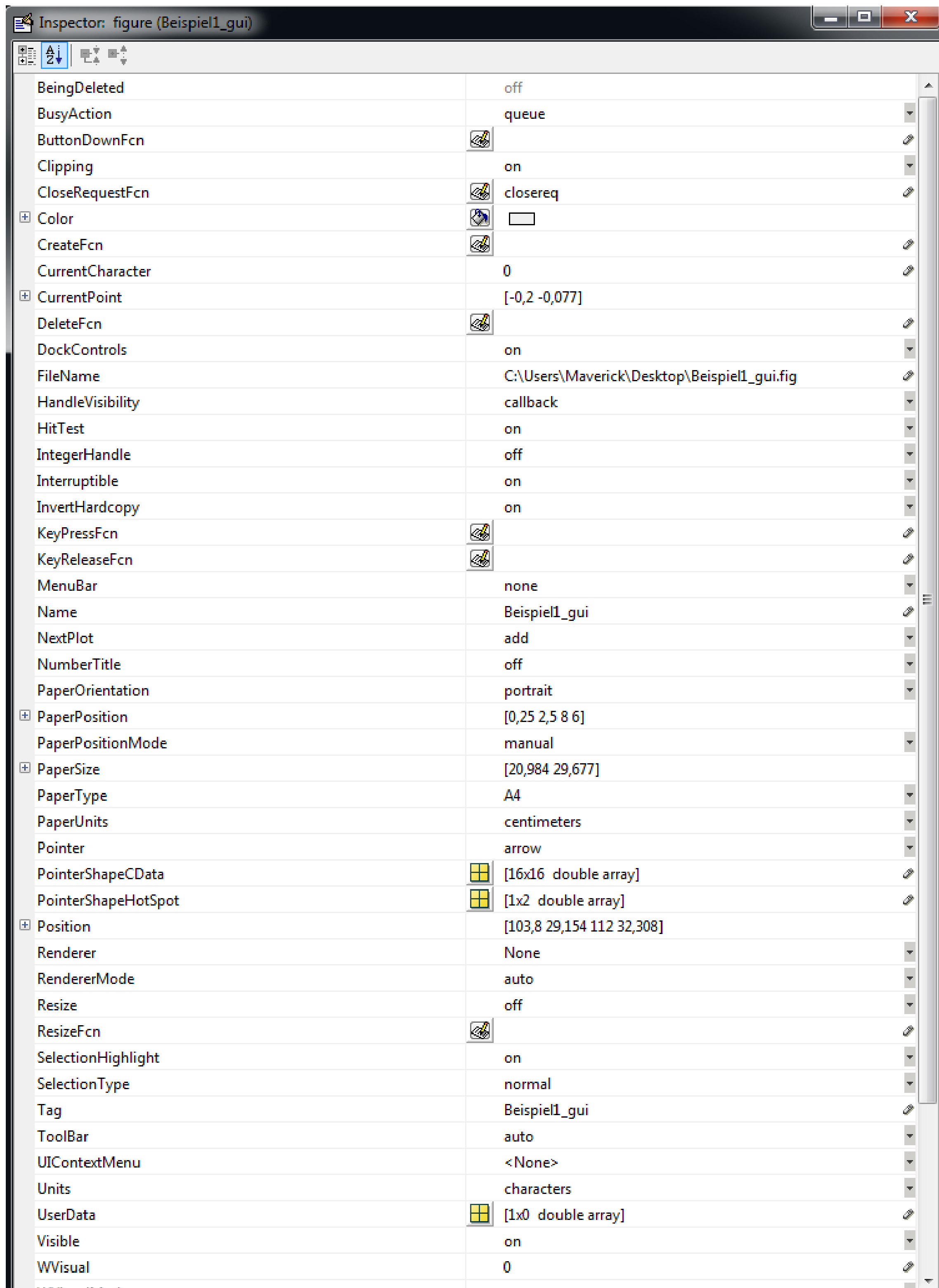

Abb. 1.2.1: Property Inspector von "Beispiel1_gui"

1.3 68-Pin MIO Connector Pin Assignments

Signal	Pin	Pin	Signal
ACH8	34	68	ACH0
ACH1	33	67	AIGND
AIGND	32	66	ACH9
ACH10	31	65	ACH2
ACH3	30	64	AIGND
AIGND	29	63	ACH11
ACH4	28	62	AISENSE
AIGND	27	61	ACH12
ACH13	26	60	ACH5
ACH6	25	59	AIGND
AIGND	24	58	ACH14
ACH15	23	57	ACH7
DAC0OUT[1]	22	56	AIGND
DAC1OUT[1]	21	55	AOGND
EXTREF[2]	20	54	AOGND
DIO4	19	53	DGND
DGND	18	52	DIO0
DIO1	17	51	DIO5
DIO6	16	50	DGND
DGND	15	49	DIO2
+5V	14	48	DIO7
DGND	13	47	DIO3
DGND	12	46	SCANCLK
PFI0/TRIG1	11	45	EXTSTROBE*
PFI1/TRIG2	10	44	DGND
DGND	9	43	PFI2/CONVERT*
+5V	8	42	PFI3/GPCTR1_SOURCE
DGND	7	41	PFI4/GPCTR1_GATE
PFI5/UPDATE*	6	40	GPCTR1_OUT
PFI6/WFTRIG	5	39	DGND
DGND	4	38	PFI7/STARTSCAN
PFI9/GPCTR0_GATE	3	37	PFI8/GPCTR0_SOURCE
GPCTR0_OUT	2	36	DGND
FREQ_OUT	1	35	DGND

[1]Not available on AT-AI-16XE-10

[2]Not available on AT-MIO-16XE-10, AT-AI-16XE-10, or AT-MIO-16XE-50

Abb. 1.3.1: 68-Pin MIO Connector Assignments **(National Instruments Germany GmbH, 2002)**

1.4 Quelltext „Beispiel1_gui.m“

```
function varargout = Beispiel1_gui(varargin)
% BEISPIEL1_GUI MATLAB code for Beispiel1_gui.fig
%      BEISPIEL1_GUI, by itself, creates a new BEISPIEL1_GUI or raises the
existing
%      singleton*.
%
%      H = BEISPIEL1_GUI returns the handle to a new BEISPIEL1_GUI or the
handle to
%      the existing singleton*.
%
%      BEISPIEL1_GUI('CALLBACK',hObject,eventData,handles,...) calls the
local
%      function named CALLBACK in BEISPIEL1_GUI.M with the given input
arguments.
%
%      BEISPIEL1_GUI('Property','Value',...) creates a new BEISPIEL1_GUI or
raises the
%      existing singleton*.  Starting from the left, property value pairs
are
%      applied to the GUI before Beispiel1_gui_OpeningFcn gets called.  An
%      unrecognized property name or invalid value makes property
application
%      stop.  All inputs are passed to Beispiel1_gui_OpeningFcn via
varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help Beispiel1_gui

% Last Modified by GUIDE v2.5 23-Dec-2012 06:30:52

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @Beispiel1_gui_OpeningFcn, ...
                   'gui_OutputFcn',  @Beispiel1_gui_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before Beispiel1_gui is made visible.
function Beispiel1_gui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
```

```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to Beispiel1_gui (see VARARGIN)

% Choose default command line output for Beispiel1_gui
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes Beispiel1_gui wait for user response (see UIRESUME)
% uiwait(handles.Beispiel1_gui);


% --- Outputs from this function are returned to the command line.
function varargout = Beispiel1_gui_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
close Beispiel1_gui
```

1.5 Quelltext „Beispiel2_gui.m“

```
function varargout = Beispiel2_gui(varargin)
% BEISPIEL2_GUI MATLAB code for Beispiel2_gui.fig
%      BEISPIEL2_GUI, by itself, creates a new BEISPIEL2_GUI or raises the
existing
%      singleton*.
%
%      H = BEISPIEL2_GUI returns the handle to a new BEISPIEL2_GUI or the
handle to
%      the existing singleton*.
%
%      BEISPIEL2_GUI('CALLBACK',hObject,eventData,handles,...) calls the
local
%      function named CALLBACK in BEISPIEL2_GUI.M with the given input
arguments.
%
%      BEISPIEL2_GUI('Property','Value',...) creates a new BEISPIEL2_GUI or
raises the
%      existing singleton*.  Starting from the left, property value pairs
are
%      applied to the GUI before Beispiel2_gui_OpeningFcn gets called.  An
%      unrecognized property name or invalid value makes property
application
%      stop.  All inputs are passed to Beispiel2_gui_OpeningFcn via
varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help Beispiel2_gui

% Last Modified by GUIDE v2.5 22-Apr-2013 13:49:08

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @Beispiel2_gui_OpeningFcn, ...
                   'gui_OutputFcn',  @Beispiel2_gui_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before Beispiel2_gui is made visible.
function Beispiel2_gui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
```

```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to Beispiel2_gui (see VARARGIN)

% Choose default command line output for Beispiel2_gui
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes Beispiel2_gui wait for user response (see UIRESUME)
% uiwait(handles.Beispiel2_gui);

open_system('Beispiel2_sim')

% Get the value of the gain edit box and synch the GUI
value = get_param(['Beispiel2_sim' '/Gain'],'Gain');
set(handles.edit_Gain,'String',value);


% --- Outputs from this function are returned to the command line.
function varargout = Beispiel2_gui_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes on button press in Startbutton.
function Startbutton_Callback(hObject, eventdata, handles)
% hObject    handle to Startbutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Turn off the Startbutton
set(handles.Startbutton,'Enable','off');
% Turn on the Stopbutton
set(handles.Stopbutton,'Enable','on');
% Start the model
set_param('Beispiel2_sim','SimulationCommand','start');


% --- Executes on button press in Stopbutton.
function Stopbutton_Callback(hObject, eventdata, handles)
% hObject    handle to Stopbutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Turn on the Startbutton
set(handles.Startbutton,'Enable','on');
% Turn off the Stopbutton
set(handles.Stopbutton,'Enable','off');
% Start the model
set_param('Beispiel2_sim','SimulationCommand','stop');
```

```
% Load and plot results after simulation
SineWave = evalin('base','SineWave');
pHandles = plot(SineWave(:,1),SineWave(:,2),'Parent',handles.axes);


% --- Executes on button press in Closebutton.
function Closebutton_Callback(hObject, eventdata, handles)
% hObject    handle to Closebutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Closes Simulink model
%close_system('Beispiel2_sim')
% Closes Simulink model unconditionally
bdclose('Beispiel2_sim')

% Closes GUI
close 'Beispiel2_gui'


function edit_Gain_Callback(hObject, eventdata, handles)
% hObject    handle to edit_Gain (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit_Gain as text
%        str2double(get(hObject,'String')) returns contents of edit_Gain as
a double

value = get(hObject,'String');

% Update the model's gain value
set_param(['Beispiel2_sim' '/Gain'],'Gain',value)

% Update simulation if the model is running
status = get_param('Beispiel2_sim','SimulationStatus');
if strcmp(status,'running')
    set_param('Beispiel2_sim', 'SimulationCommand', 'Update')
end


% --- Executes during object creation, after setting all properties.
function edit_Gain_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit_Gain (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

1.6 Quelltext „Beispiel3_gui.m“

```
function varargout = Beispiel3_gui(varargin)
% BEISPIEL3_GUI MATLAB code for Beispiel3_gui.fig
%      BEISPIEL3_GUI, by itself, creates a new BEISPIEL3_GUI or raises the
existing
%      singleton*.
%
%      H = BEISPIEL3_GUI returns the handle to a new BEISPIEL3_GUI or the
handle to
%      the existing singleton*.
%
%      BEISPIEL3_GUI('CALLBACK',hObject,eventData,handles,...) calls the
local
%      function named CALLBACK in BEISPIEL3_GUI.M with the given input
arguments.
%
%      BEISPIEL3_GUI('Property','Value',...) creates a new BEISPIEL3_GUI or
raises the
%      existing singleton*.  Starting from the left, property value pairs
are
%      applied to the GUI before Beispiel3_gui_OpeningFcn gets called.  An
%      unrecognized property name or invalid value makes property
application
%      stop.  All inputs are passed to Beispiel3_gui_OpeningFcn via
varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help Beispiel3_gui

% Last Modified by GUIDE v2.5 15-Jan-2013 15:24:37

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @Beispiel3_gui_OpeningFcn, ...
                   'gui_OutputFcn',  @Beispiel3_gui_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before Beispiel3_gui is made visible.
function Beispiel3_gui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
```

```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to Beispiel3_gui (see VARARGIN)

% Choose default command line output for Beispiel3_gui
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes Beispiel3_gui wait for user response (see UIRESUME)
% uiwait(handles.Beispiel3_gui);

open_system('Beispiel3_sim')

% Get the value of the gain edit box and synch the GUI
value = get_param(['Beispiel3_sim' '/Gain'],'Gain');
set(handles.edit_Gain,'String',value);


% --- Outputs from this function are returned to the command line.
function varargout = Beispiel3_gui_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes on button press in Startstopbutton.
function Startstopbutton_Callback(hObject, eventdata, handles)
% hObject    handle to Startstopbutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

mystring = get(hObject,'String');
status = get_param('Beispiel3_sim','SimulationStatus');

if strcmp(mystring,'Start')

    % Check the status of the simulation and start it if it's stopped
    if strcmp(status,'stopped')
        set_param('Beispiel3_sim','SimulationCommand','start')
    end

    % Update the string on the pushbutton
    set(handles.Startstopbutton,'String','Stop')

elseif strcmp(mystring,'Stop')

    % Check the status of the simulation and stop it if it's running
    if strcmp(status,'running')
        set_param('Beispiel3_sim', 'SimulationCommand', 'Stop')
        % Load and plot results after simulation
        SineWave = evalin('base','SineWave');
        pHandles = plot(SineWave(:,1),SineWave(:,2),'Parent',handles.axes);
    end
```

```
    % Update the string on the pushbutton
    set(handles.Startstopbutton,'String','Start')

end


% --- Executes on button press in Closebutton.
function Closebutton_Callback(hObject, eventdata, handles)
% hObject    handle to Closebutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Closes Simulink model
%close_system('Beispiel3_sim')
% Closes Simulink model unconditionally
bdclose('Beispiel3_sim')

% Closes GUI
close 'Beispiel3_gui'


function edit_Gain_Callback(hObject, eventdata, handles)
% hObject    handle to edit_Gain (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit_Gain as text
%        str2double(get(hObject,'String')) returns contents of edit_Gain as
a double

value = get(hObject,'String');

% Update the model's gain value
set_param(['Beispiel3_sim' '/Gain'],'Gain',value)

% Update simulation if the model is running
status = get_param('Beispiel3_sim','SimulationStatus');
if strcmp(status,'running')
    set_param('Beispiel3_sim', 'SimulationCommand', 'Update')
end


% --- Executes during object creation, after setting all properties.
function edit_Gain_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit_Gain (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

1.7 Quelltext „Beispiel4_gui.m“

```
function varargout = Beispiel4_gui(varargin)
% BEISPIEL4_GUI MATLAB code for Beispiel4_gui.fig
%      BEISPIEL4_GUI, by itself, creates a new BEISPIEL4_GUI or raises the
existing
%      singleton*.
%
%      H = BEISPIEL4_GUI returns the handle to a new BEISPIEL4_GUI or the
handle to
%      the existing singleton*.
%
%      BEISPIEL4_GUI('CALLBACK',hObject,eventData,handles,...) calls the
local
%      function named CALLBACK in BEISPIEL4_GUI.M with the given input
arguments.
%
%      BEISPIEL4_GUI('Property','Value',...) creates a new BEISPIEL4_GUI or
raises the
%      existing singleton*.  Starting from the left, property value pairs
are
%      applied to the GUI before Beispiel4_gui_OpeningFcn gets called.  An
%      unrecognized property name or invalid value makes property
application
%      stop.  All inputs are passed to Beispiel4_gui_OpeningFcn via
varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help Beispiel4_gui

% Last Modified by GUIDE v2.5 18-Jan-2013 09:19:14

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @Beispiel4_gui_OpeningFcn, ...
                   'gui_OutputFcn',  @Beispiel4_gui_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before Beispiel4_gui is made visible.
function Beispiel4_gui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
```

```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to Beispiel4_gui (see VARARGIN)

% Choose default command line output for Beispiel4_gui
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes Beispiel4_gui wait for user response (see UIRESUME)
% uiwait(handles.Beispiel4_gui);


% Open Simulation
open_system('Beispiel4_sim')

% Get the value of the gain edit box and synch the GUI
value = get_param(['Beispiel4_sim' '/Gain'],'Gain');
set(handles.edit_Gain,'String',value);


% --- Outputs from this function are returned to the command line.
function varargout = Beispiel4_gui_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes on button press in Startstopbutton.
function Startstopbutton_Callback(hObject, eventdata, handles)
% hObject    handle to Startstopbutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

mystring = get(hObject,'String');
status = get_param('Beispiel4_sim','simulationstatus');

if strcmp(mystring,'Start Simulation')

    % Check the status of the simulation and start it if it's stopped
    if strcmp(status,'stopped')
        set_param('Beispiel4_sim','simulationcommand','Start')
    end

    % Update the string on the pushbutton
    set(handles.Startstopbutton,'String','Stop Simulation')

    cla
    %cla (handles.axesplot,'reset')


elseif strcmp(mystring,'Stop Simulation')
```

```
    % Check the status of the simulation and stop it if it's running
    if strcmp(status,'running')
        set_param('Beispiel4_sim', 'SimulationCommand', 'Stop')
    end

    % Update the string on the pushbutton
    set(handles.Startstopbutton,'String','Start Simulation')

    % Plot results from workspace after simulation
    %x = evalin('base','SineWave(:,1)');
    %y = evalin('base','SineWave(:,2)');
    %pHandles = plot(x,y,'Parent',handles.axesplot);

end


function edit_Gain_Callback(hObject, eventdata, handles)
% hObject    handle to edit_gain (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit_gain as text
%        str2double(get(hObject,'String')) returns contents of edit_gain as
a double

value = get(hObject,'String');

% Update the model's gain value
set_param(['Beispiel4_sim' '/Gain'],'Gain',value)

% Update simulation if the model is running
status = get_param('Beispiel4_sim','simulationstatus');
if strcmp(status,'running')
    set_param('Beispiel4_sim', 'SimulationCommand', 'Update')
end


% --- Executes during object creation, after setting all properties.
function edit_Gain_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit_gain (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end


% --- Executes on button press in closebutton.
function closebutton_Callback(hObject, eventdata, handles)
% hObject    handle to closebutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% delete(handles.Beispiel4_gui)

% Closes Simulink model
%close_system('Beispiel4_sim')
```

```
clear all
% clear Workspace
evalin('base','clear');
% Closes Simulink model unconditionally
bdclose('Beispiel4_sim')
% Closes GUI
close Beispiel4_gui


function edit_display_Callback(hObject, eventdata, handles)
% hObject    handle to edit_display (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit_display as text
%        str2double(get(hObject,'String')) returns contents of edit_display
as a double


% --- Executes during object creation, after setting all properties.
function edit_display_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit_display (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

1.8 Quelltext „update_gui.m“ von Beispiel4

```
function varargout = updategui(varargin)

%create a run time object that can return the value of the outport block's
%input and then put the value in a string

rto1 = get_param('Beispiel4_sim/SinusOut','RuntimeObject');

% create str from rto1
str = num2str(rto1.InputPort(1).Data);

% get a handle to the GUI's 'edit_display' window
display = findobj('Tag','edit_display');

% update the gui
set(display,'string',str);


% get Data from Simulation
XData = get_param('Beispiel4_sim','SimulationTime');
YData = rto1.InputPort(1).Data;

% save Data to workspace
assignin('base','XData',XData)
assignin('base','YData',YData)


% real-time plot
persistent guiplot

if isempty(guiplot)
    guiplot=findobj(0, 'Tag','axesplot');
end
plot(guiplot,XData,YData)
hold on
grid on
```

1.9 Quelltext „Sinus_gui.m“

```
function varargout = sinus_gui(varargin)
% SINUS_GUI MATLAB code for sinus_gui.fig
%      SINUS_GUI, by itself, creates a new SINUS_GUI or raises the existing
%      singleton*.
%
%      H = SINUS_GUI returns the handle to a new SINUS_GUI or the handle to
%      the existing singleton*.
%
%      SINUS_GUI('CALLBACK',hObject,eventData,handles,...) calls the local
%      function named CALLBACK in SINUS_GUI.M with the given input
arguments.
%
%      SINUS_GUI('Property','Value',...) creates a new SINUS_GUI or raises
the
%      existing singleton*.  Starting from the left, property value pairs
are
%      applied to the GUI before sinus_gui_OpeningFcn gets called.  An
%      unrecognized property name or invalid value makes property
application
%      stop.  All inputs are passed to sinus_gui_OpeningFcn via varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help sinus_gui

% Last Modified by GUIDE v2.5 21-Jan-2013 09:15:00

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @sinus_gui_OpeningFcn, ...
                   'gui_OutputFcn',  @sinus_gui_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before sinus_gui is made visible.
function sinus_gui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to sinus_gui (see VARARGIN)
```

```
% Choose default command line output for sinus_gui
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% open simulink model
open_system('sinus_sim')

% UIWAIT makes sinus_gui wait for user response (see UIRESUME)
% uiwait(handles.sinus_gui);


% --- Outputs from this function are returned to the command line.
function varargout = sinus_gui_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes on button press in pushbutton_startstop.
function pushbutton_startstop_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton_startstop (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

mystring = get(hObject,'String');
status = get_param('sinus_sim','SimulationStatus');

if strcmp(mystring,'Messen')

    % Check the status of the simulation and start it if it's stopped
    if strcmp(status,'stopped')
        set_param('sinus_sim','simulationcommand','Start')
    end

    % Update the string on the pushbutton
    set(handles.pushbutton_startstop,'String','Stop')

elseif strcmp(mystring,'Stop')

    % Check the status of the simulation and stop it if it's running
    if strcmp(status,'running')
        set_param('sinus_sim', 'SimulationCommand', 'Stop')
    end

    % Update the string on the pushbutton
    set(handles.pushbutton_startstop,'String','Messen')

    % Plot results from file after simulation
    load('sinus_export')
    x = sinus_export(1,:);
    y = sinus_export(2,:);
    pHandles = plot(x,y,'Parent',handles.sinus_export);
```

```
    load('sinus_import')
    x = sinus_import(1,:);
    y = sinus_import(2,:);
    pHandles = plot(x,y,'Parent',handles.sinus_import);

end


% --- Executes on button press in pushbutton3.
function pushbutton3_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Closes Simulink model unconditionally
bdclose('sinus_sim')
% Closes GUI
close 'sinus_gui'
```

1.10 Kanalbelegung Data Acquisition-Board „Sinus_GUI“

Karten Nr. 1

Kartentyp	Hersteller
PCI-MIO-16E-4	National-Instruments

Betriebsart: 0-10V single-ended Projekt: **Sinus_GUI**
Windows 7, MATLAB 2012b

Kanal	Signal	Adapterboard	Bemerkung
CH0	sinus_import	68	Signal von Oszilloskop
CH1		33	
CH2		65	
CH3		30	
CH4		28	
CH5		60	
CH6		25	
CH7		57	
CH8		34	
CH9		66	
CH10		31	
CH11		63	
CH12		61	
CH13		26	
CH14		58	
CH15		23	
AIGND		32-67-29-64-27-59-24-56	
AISENSE		62	
DAC0	sinus_export	22	Signal von Simulink
DAC1		21	
AOGND		54-55	
DIO0		52	
DIO1		17	
DIO2		49	
DIO3		47	
DIO4		19	
DIO5		51	
DIO6		16	
DIO7		48	
DGND		53-18-50-15-12-13-7-39-4-36-35	
GPCTR0_OUT		2	
GPCTR1_OUT		40	

Pinbelegung S 4-2 PCI E Series User Manual

1.11 Quelltext „Q_Messen_gui.m“

```
function varargout = Q_Messen_gui(varargin)
% Q_MESSEN_GUI MATLAB code for Q_Messen_gui.fig
%      Q_MESSEN_GUI, by itself, creates a new Q_MESSEN_GUI or raises the
existing
%      singleton*.
%
%      H = Q_MESSEN_GUI returns the handle to a new Q_MESSEN_GUI or the
handle to
%      the existing singleton*.
%
%      Q_MESSEN_GUI('CALLBACK',hObject,eventData,handles,...) calls the
local
%      function named CALLBACK in Q_MESSEN_GUI.M with the given input
arguments.
%
%      Q_MESSEN_GUI('Property','Value',...) creates a new Q_MESSEN_GUI or
raises the
%      existing singleton*.  Starting from the left, property value pairs
are
%      applied to the GUI before Q_Messen_OpeningFcn gets called.  An
%      unrecognized property name or invalid value makes property
application
%      stop.  All inputs are passed to Q_Messen_gui_OpeningFcn via
varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help Q_Messen_gui

% Last Modified by GUIDE v2.5 01-Jun-2013 17:59:25

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @Q_Messen_gui_OpeningFcn, ...
                   'gui_OutputFcn',  @Q_Messen_gui_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT



% --- Executes just before Q_Messen_gui is made visible.
function Q_Messen_gui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
```

```
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to Q_Messen_gui (see VARARGIN)

% Choose default command line output for Q_Messen_gui
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes Q_Messen_gui wait for user response (see UIRESUME)
% uiwait(handles.Q_Messen_gui);


% Open Simulation
open_system('Q_Messen_sim')


% --- Outputs from this function are returned to the command line.
function varargout = Q_Messen_gui_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;


% --- Executes on button press in Startstopbutton.
function Startstopbutton_Callback(hObject, eventdata, handles)
% hObject    handle to Startstopbutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

mystring = get(hObject,'String');
status = get_param('Q_Messen_sim','SimulationStatus');

if strcmp(mystring,'Start')

    % clear Workspace
    evalin('base','clear');

    % Check the status of the simulation and start it if it's stopped
    if strcmp(status,'stopped')
        set_param('Q_Messen_sim','SimulationCommand','start')
    end

    % Update the string on the pushbutton
    set(handles.Startstopbutton,'String','Stop')

    cla
    %cla (handles.axesplot,'reset')


elseif strcmp(mystring,'Stop')
```

```
 % Check the status of the simulation and stop it if it's running
    if strcmp(status,'running')
        set_param('Q_Messen_sim','SimulationCommand', 'stop')
    end

    % Update the string on the pushbutton
    set(handles.Startstopbutton,'String','Start')

end


% --- Executes on button press in closebutton.
function closebutton_Callback(hObject, eventdata, handles)
% hObject    handle to closebutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% delete(handles.Q_Messen_gui)

% Closes Simulink model
%close_system('Q_Messen_sim')

clear all
% clear Workspace
evalin('base','clear');
% Closes Simulink model unconditionally
bdclose('Q_Messen_sim')
% Closes GUI
close Q_Messen_gui


% --- Executes on button press in save_button.
function save_button_Callback(hObject, eventdata, handles)
% hObject    handle to save_button (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

Bezeichnung = get(handles.edit_Bezeichnung,'String');
Durchfluss = evalin('base','Durchfluss');
STIME = evalin('base','STIME');

Durchfluss=Durchfluss(1,:);
Durchfluss=Durchfluss';

Ergebnis=[STIME,Durchfluss];
save(Bezeichnung,'Durchfluss','STIME')
dlmwrite(Bezeichnung,Ergebnis,';')


function edit_display_Callback(hObject, eventdata, handles)
% hObject    handle to edit_display (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit_display as text
%        str2double(get(hObject,'String')) returns contents of edit_display
as a double
```

```
% --- Executes during object creation, after setting all properties.
function edit_display_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit_display (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end



function edit_Bezeichnung_Callback(hObject, eventdata, handles)
% hObject    handle to edit_Bezeichnung (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit_Bezeichnung as text
%        str2double(get(hObject,'String')) returns contents of
edit_Bezeichnung as a double


% --- Executes during object creation, after setting all properties.
function edit_Bezeichnung_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit_Bezeichnung (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

1.12 Quelltext „update_gui.m“ von „Q_Messen“

```
function varargout = updategui(varargin)

%create a run time object that can return the value of the outport block's
%input and then put the value in a string

rto1 = get_param('Q_Messen_sim/Q_Out','RuntimeObject');

% create str from rto1
str = num2str(rto1.InputPort(1).Data);

% get a handle to the GUI's 'edit_display' window
display = findobj('Tag','edit_display');

% update the gui
set(display,'string',str);


% get Data from Simulation
XData = get_param('Q_Messen_sim','SimulationTime');
YData = rto1.InputPort(1).Data;

% save Data to workspace
assignin('base','XData',XData)
assignin('base','YData',YData)


% real-time plot
persistent guiplot

if isempty(guiplot)
    guiplot=findobj(0, 'Tag','axesplot');
end
plot(guiplot,XData,YData)
hold on
grid on
```

1.13 Kanalbelegung Data Acquisition-Board „Q_Messen"

Karten Nr. 1

Kartentyp	Hersteller
PCI-MIO-16E-4	National-Instruments

Betriebsart: 0-10V single-ended Projekt: **Q_Messen**
Windows 7, MATLAB 2012b

Kanal	Signal	Adapterboard	Bemerkung
CH0		68	
CH1		33	
CH2		65	
CH3		30	
CH4		28	
CH5		60	
CH6		25	
CH7		57	
CH8	Durchfluss 6"	34	V (100l = 10V)
CH9		66	
CH10		31	
CH11		63	
CH12		61	
CH13		26	
CH14		58	
CH15		23	
AIGND		32-67-29-64-27-59-24-56	
AISENSE		62	
DAC0		22	
DAC1		21	
AOGND		54-55	
DIO0		52	
DIO1		17	
DIO2		49	
DIO3		47	
DIO4		19	
DIO5		51	
DIO6		16	
DIO7		48	
DGND		53-18-50-15-12-13-7-39-4-36-35	
GPCTR0_OUT		2	
GPCTR1_OUT		40	

Pinbelegung S 4-2 PCI E Series User Manual

Bemerkung: IDM Einstellung Qmax = 100 l/s

1.14 Quelltext „Q_Regelung_gui.m“

```
function varargout = Q_Regelung_gui(varargin)
% Q_Regelung_gui MATLAB code for Q_Regelung_gui.fig
%      Q_Regelung_gui, by itself, creates a new Q_Regelung_gui or raises 
the existing
%      singleton*.
%
%      H = Q_Regelung_gui returns the handle to a new Q_Regelung_gui or the 
handle to
%      the existing singleton*.
%
%      Q_Regelung_gui('CALLBACK',hObject,eventData,handles,...) calls the 
local
%      function named CALLBACK in Q_Regelung_gui.M with the given input 
arguments.
%
%      Q_Regelung_gui('Property','Value',...) creates a new Q_Regelung_gui 
or raises the
%      existing singleton*.  Starting from the left, property value pairs 
are
%      applied to the GUI before Q_Regelung_gui_OpeningFcn gets called.  An
%      unrecognized property name or invalid value makes property 
application
%      stop.  All inputs are passed to Q_Regelung_gui_OpeningFcn via 
varargin.
%
%      *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
%      instance to run (singleton)".
%
% See also: GUIDE, GUIDATA, GUIHANDLES

% Edit the above text to modify the response to help Q_Regelung_gui

% Last Modified by GUIDE v2.5 06-Nov-2012 10:59:50

% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @Q_Regelung_gui_OpeningFcn, ...
                   'gui_OutputFcn',  @Q_Regelung_gui_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before Q_Regelung_gui is made visible.
function Q_Regelung_gui_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
```

```
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to Q_Regelung_gui (see VARARGIN)

% Choose default command line output for Q_Regelung_gui
handles.output = hObject;

% Update handles structure
guidata(hObject, handles);

% UIWAIT makes Q_Regelung_gui wait for user response (see UIRESUME)
% uiwait(handles.Q_Regelung_gui);

open Q_Regelung_sim


% Get the value of Q_SOLL and synch the GUI
value = get_param(['Q_Regelung_sim' '/Q_SOLL'],'value');
set(handles.edit_QSOLL,'String',value);

% Set the value of the Constant slider, with max/min of +10/-10
%slider_position = max(0,min(1,(str2double(value) + 10)/20));

% Set the value of the slider, with max/min
slider_position = max(0,min(100,(str2double(value))));
set(handles.slider_QSOLL,'Value',slider_position);


% Assign gui, startstop and constant handles to the base workspace
%assignin('base','Q_Regelung_handles',handles)
%assignin('base','startstop_hObject',handles.pushbutton_startstop)
%assignin('base','Constant_hObject',handles.edit_QSOLL)



% --- Outputs from this function are returned to the command line.
function varargout = Q_Regelung_gui_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;



% --- Executes on button press in pushbutton_startstop.
function pushbutton_startstop_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton_startstop (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

mystring = get(hObject,'String');
status = get_param('Q_Regelung_sim','SimulationStatus');
%manswitch = get_param(['Q_Regelung_sim' '/Manual Switch'],'sw');
```

```
if strcmp(mystring,'Start')

    % clear Workspace
    evalin('base','clear');

    % Check the status of the simulation and start it if it's stopped
    if strcmp(status,'stopped')
        set_param(['Q_Regelung_sim' '/Manual Switch'],'sw','1')
        set_param('Q_Regelung_sim','SimulationCommand','start')
    end

    % Update the string on the pushbutton
    set(handles.pushbutton_startstop,'String','Stop')

    %cla (handles.axesplot,'reset')
    cla


elseif strcmp(mystring,'Stop')


    % Check the status of the simulation and stop it if it's running
    if strcmp(status,'running')

        set_param(['Q_Regelung_sim' '/Manual Switch'],'sw','0')

        % pause sim 1 sec before closing to update manual switch
        pause(1); % n in Sekunden
        set_param('Q_Regelung_sim', 'SimulationCommand', 'stop')

    end

    % Update the string on the pushbutton
    set(handles.pushbutton_startstop,'String','Start')


    % Plot results from workspace after simulation
    Q_SOLL_IST = evalin('base','Q_SOLL_IST');
    STIME = Q_SOLL_IST(:,1);
    Q_SOLL = Q_SOLL_IST(:,2);
    Q_IST = Q_SOLL_IST(:,3);

    pHandles = plot(STIME,Q_IST,'gr','Parent',handles.axesplot);
    hold on
    pHandles = plot(STIME,Q_SOLL,'red','Parent',handles.axesplot);
    hold off

else
    warning('Unrecognized string for pushbutton_startstop') %#ok<WNTAG>
end

% Assign handles and the startstop object to the base workspace
%assignin('base','Q_Regelung_handles',handles)
%assignin('base','startstop_hObject',handles.pushbutton_startstop)
```

```
% --- Executes on button press in closebutton.
function closebutton_Callback(hObject, eventdata, handles)
% hObject    handle to closebutton (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

set_param(['Q_Regelung_sim' '/Manual Switch'],'sw','0')
% pause sim 1 sec before closing to update manual switch
pause(1); % n in Sekunden
set_param('Q_Regelung_sim', 'SimulationCommand', 'stop')

%delete(handles.Q_Regelung_gui)
%close Q_Regelung_sim

clear all
% clear Workspace
evalin('base','clear');
% Closes Simulink model unconditionally
bdclose('Q_Regelung_sim')
% Closes GUI
close Q_Regelung_gui


function edit_QSOLL_Callback(hObject, eventdata, handles)
% hObject    handle to edit_QSOLL (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit_QSOLL as text
%        str2double(get(hObject,'String')) returns contents of edit_QSOLL
as a double

value = get(hObject,'String');

% Update the model's Constant value
set_param(['Q_Regelung_sim' '/Q_SOLL'],'value',value)

% Set the value of the slider, with max/min
slider_position = max(0,min(100,(str2double(value))));
set(handles.slider_QSOLL,'Value',slider_position);

% Update simulation if the model is running
status = get_param('Q_Regelung_sim','SimulationStatus');
if strcmp(status,'running')
    set_param('Q_Regelung_sim', 'SimulationCommand', 'Update')
end


% --- Executes during object creation, after setting all properties.
function edit_QSOLL_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit_QSOLL (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

```
% --- Executes on slider movement.
function slider_QSOLL_Callback(hObject, eventdata, handles)
% hObject    handle to slider_QSOLL (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'Value') returns position of slider
%        get(hObject,'Min') and get(hObject,'Max') to determine range of
slider

% Get the value of the Constant slider and determine what the Constant
value should be
slider_position = get(hObject,'Value');
value = num2str(slider_position);

% Update the model's Constant value
set_param(['Q_Regelung_sim' '/Q_SOLL'],'value',value)

% Set the value of the Constant edit box
set(handles.edit_QSOLL,'String',value);

% Update simulation if the model is running
status = get_param('Q_Regelung_sim','SimulationStatus');
if strcmp(status,'running')
    set_param('Q_Regelung_sim', 'SimulationCommand', 'Update')
end


% --- Executes during object creation, after setting all properties.
function slider_QSOLL_CreateFcn(hObject, eventdata, handles)
% hObject    handle to slider_QSOLL (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: slider controls usually have a light gray background.
if isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor',[.9 .9 .9]);
end


% --- Executes on button press in save_button.
function save_button_Callback(hObject, eventdata, handles)
% hObject    handle to save_button (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

Bezeichnung = get(handles.edit_Bezeichnung,'String');
Q_SOLL_IST = evalin('base','Q_SOLL_IST');
STIME = Q_SOLL_IST(:,1);
Q_SOLL = Q_SOLL_IST(:,2);
Q_IST = Q_SOLL_IST(:,3);

save(Bezeichnung,'Q_SOLL_IST','STIME','Q_SOLL','Q_IST')
dlmwrite(Bezeichnung,Q_SOLL_IST,';')
```

```
function edit_Bezeichnung_Callback(hObject, eventdata, handles)
% hObject    handle to edit_Bezeichnung (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of edit_Bezeichnung as text
%        str2double(get(hObject,'String')) returns contents of
edit_Bezeichnung as a double


% --- Executes during object creation, after setting all properties.
function edit_Bezeichnung_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit_Bezeichnung (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc && isequal(get(hObject,'BackgroundColor'),
get(0,'defaultUicontrolBackgroundColor'))
    set(hObject,'BackgroundColor','white');
end
```

1.15 Quelltext „update_gui.m“ von „Q_Regelung“

```
function varargout = updategui(varargin)

%create a run time object that can return the value of the outport block's
%input and then put the value in a string

rto1 = get_param('Q_Regelung_sim/Q_Out','RuntimeObject');
rto2 = get_param('Q_Regelung_sim/QSOLL_Out','RuntimeObject');
rto3 = get_param('Q_Regelung_sim/Ventil_Out','RuntimeObject');

% create str from rto1 and rto2
str = num2str(rto1.InputPort(1).Data);
str3 = num2str(rto3.InputPort(1).Data);

% get a handle to the GUI's text fields
display = findobj('Tag','edit_display');
display3 = findobj('Tag','edit_ventil');


% update the gui
set(display,'string',str);

if strcmp (str3,'1')
    set(display3,'string','Öfnnet');
    set(display3,'BackgroundColor','gr');
elseif strcmp (str3,'-1')
    set(display3,'string','Schließt');
    set(display3,'BackgroundColor','red');
else
    set(display3,'string','-');
    set(display3,'BackgroundColor','white');
end


% get Data from Simulation
STime = get_param('Q_Regelung_sim','SimulationTime');
Q_IST = rto1.InputPort(1).Data;
Q_SOLL = rto2.InputPort(1).Data;

% save Data to workspace
assignin('base','STime',STime)
assignin('base','Q_IST',Q_IST)
assignin('base','Q_SOLL',Q_SOLL)

% real-time plot
persistent guiplot

if isempty(guiplot)
    guiplot=findobj(0, 'Tag','axesplot');
end

%plot(guiplot,STime,Q_SOLL,'red')
%set(guiplot,'Nextplot','add')
plot(guiplot,STime,Q_IST,'gr')

hold on
grid on
```

1.16 Kanalbelegung Data Acquisition-Board „Q_Regelung“

Karten Nr. 1

Kartentyp	Hersteller
PCI-MIO-16E-4	National-Instruments

Betriebsart: 0-10V single-ended Projekt: **Q_Regelung**
Windows 7, MATLAB 2012b

Kanal	Signal	Adapterboard	Bemerkung
CH0		68	
CH1		33	
CH2		65	
CH3		30	
CH4		28	
CH5		60	
CH6		25	
CH7		57	
CH8	Durchfluss 6“	34	V (100l = 10V)
CH9		66	
CH10		31	
CH11		63	
CH12		61	
CH13		26	
CH14		58	
CH15		23	
AIGND		32-67-29-64-27-59-24-56	
AISENSE		62	
DAC0		22	
DAC1		21	
AOGND		54-55	
DIO0	Schieber öffnet	52	boolean
DIO1	Schieber schließt	17	boolean
DIO2		49	
DIO3		47	
DIO4		19	
DIO5		51	
DIO6		16	
DIO7		48	
DGND		53-18-50-15-12-13-7-39-4-36-35	
GPCTR0_OUT		2	
GPCTR1_OUT		40	

Pinbelegung S 4-2 PCI E Series User Manual

Bemerkung: IDM Einstellung Qmax = 100 l/s

1.17 Benutzeroberfläche vom Analyseprogramm Steffturbine

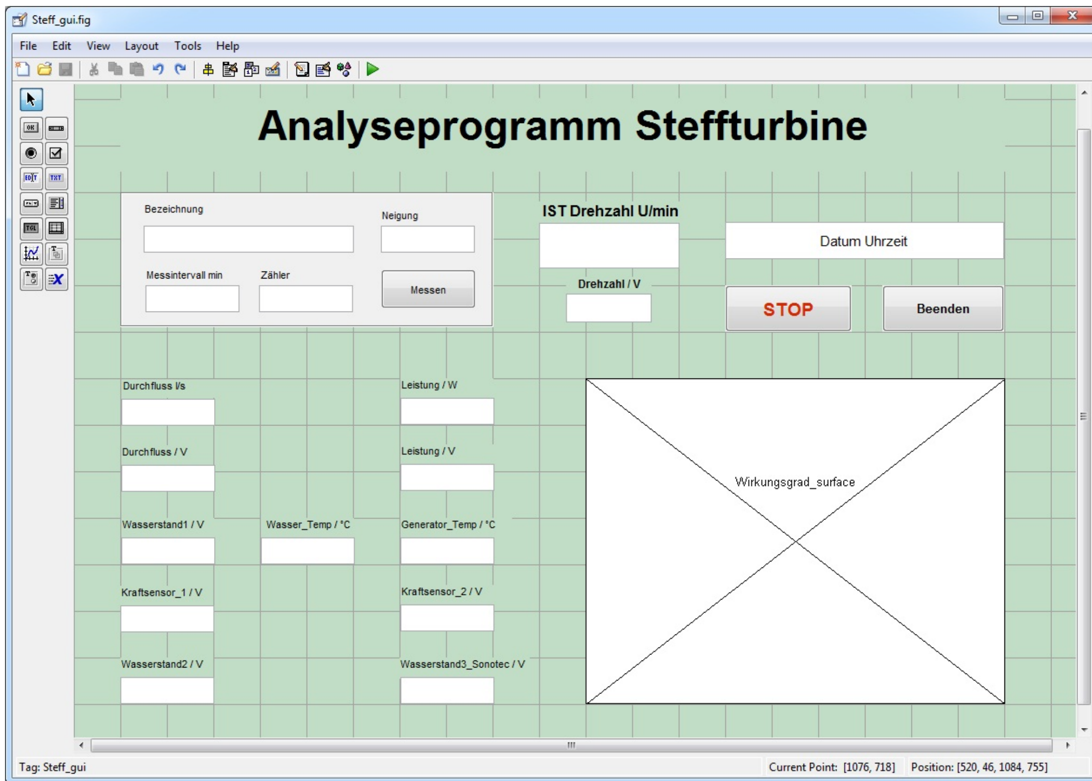

Abb. 1.17.1: Darstellung der Benutzeroberfläche "Steff_gui" im Dialog Editor

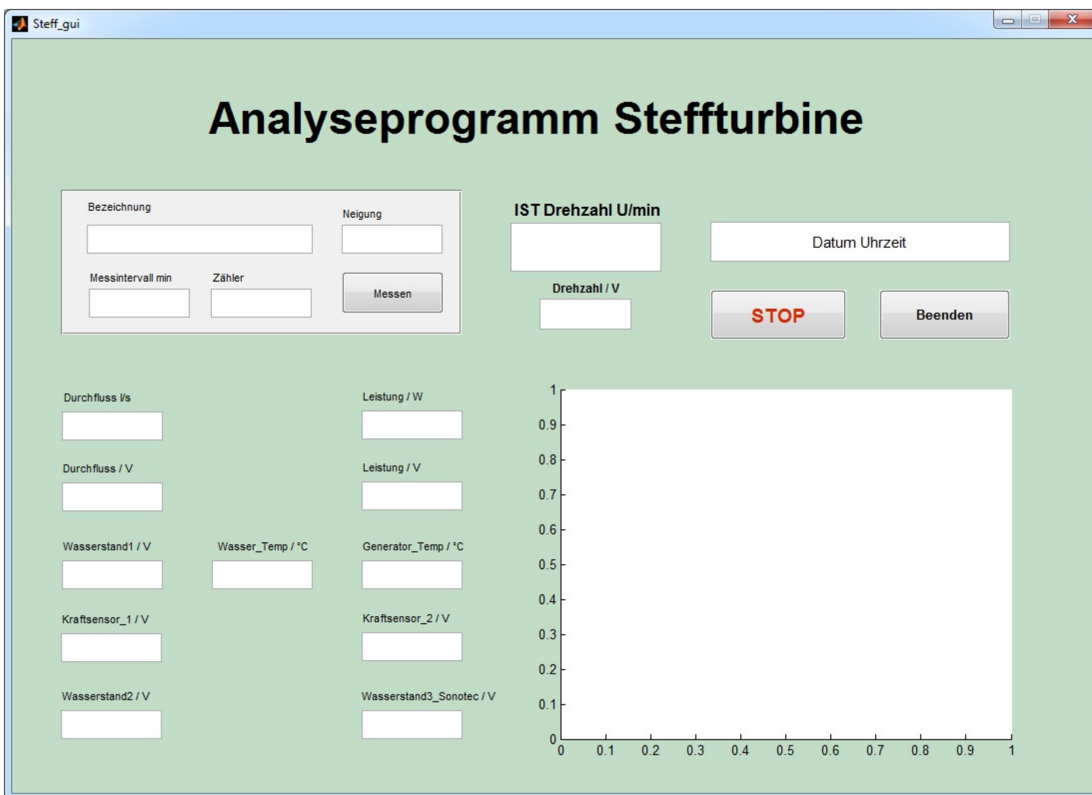

Abb. 1.17.2: Darstellung der Benutzeroberfläche "Steff_gui"

1.18 Quelltext „update_gui.m“ von „Steff_gui“

```
function varargout = updategui(varargin)
%create a run time object that can return the value of the outport block's
rto1 = get_param('Steff_sim/Out3','RuntimeObject');
rto2 = get_param('Steff_sim/Out3','RuntimeObject');
rto3 = get_param('Steff_sim/Out3','RuntimeObject');
rto4 = get_param('Steff_sim/Out4','RuntimeObject');
rto5 = get_param('Steff_sim/Out5','RuntimeObject');
rto6 = get_param('Steff_sim/Out6','RuntimeObject');
rto7 = get_param('Steff_sim/Out7','RuntimeObject');
rto8 = get_param('Steff_sim/Out8','RuntimeObject');
rto9 = get_param('Steff_sim/Out9','RuntimeObject');
rto10 = get_param('Steff_sim/Out10','RuntimeObject');
rto11 = get_param('Steff_sim/Out11','RuntimeObject');
rto12 = get_param('Steff_sim/Out12','RuntimeObject');
rto13 = get_param('Steff_sim/Out13','RuntimeObject');

% create str from rto's
str1 = num2str(rto1.InputPort(1).Data);
str2 = num2str(rto2.InputPort(1).Data);
str3 = num2str(rto3.InputPort(1).Data);
str4 = num2str(rto4.InputPort(1).Data);
str5 = num2str(rto5.InputPort(1).Data);
str6 = num2str(rto6.InputPort(1).Data);
str7 = num2str(rto7.InputPort(1).Data);
str8 = num2str(rto8.InputPort(1).Data);
str9 = num2str(rto9.InputPort(1).Data);
str10 = num2str(rto10.InputPort(1).Data);
str11 = num2str(rto11.InputPort(1).Data);
str12 = num2str(rto12.InputPort(1).Data);
str13 = num2str(rto13.InputPort(1).Data);

% get a handle to the GUI's edit text fields
display2 = findobj('Tag','edit_DrehzahlV');
display3 = findobj('Tag','edit_Durchfluss');
display4 = findobj('Tag','edit_DurchflussV');
display5 = findobj('Tag','edit_Wasser1');
display6 = findobj('Tag','edit_Kraft1');
display7 = findobj('Tag','edit_Wasser2');
display8 = findobj('Tag','edit_WasserT');
display9 = findobj('Tag','edit_Leistung');
display10 = findobj('Tag','edit_LeistungV');
display11 = findobj('Tag','edit_Temp');
display12 = findobj('Tag','edit_Kraft2');
display13 = findobj('Tag','edit_Wasser3');

% update the gui
set(display1,'string',str1);
set(display2,'string',str2);
set(display3,'string',str3);
set(display4,'string',str4);
set(display5,'string',str5);
set(display6,'string',str6);
set(display7,'string',str7);
set(display8,'string',str8);
set(display9,'string',str9);
set(display10,'string',str10);
set(display11,'string',str11);
set(display12,'string',str12);
set(display13,'string',str13);
```

Anlage 2: CD mit Beispiel- und Testprogrammen

Auf der beiliegenden CD befinden sich die Beispiel- und Testprogramme, sowie Vorarbeiten und Quelltexte für ein mögliches Analyseprogramm einer Steffturbine. Zudem sind die Auswertungen zu den Testprogrammen in Form von Excell-Tabellen, sowie ein Video zur Durchflussregelung auf ein Q von 25 l/s enthalten, um den Regelungsprozess über die MATLAB Software besser zu veranschaulichen.

Die Beispielprogramme über die Erstellung von GUI’s sind in der MATLAB Umgebung ohne weitere Toolboxen lauffähig. Für die Testprogramme mit denen eine Kopplung zwischen Messhardware des Wasserbaulabors und der MATLAB Software hergestellt wurde, benötigen zur Messwerterfassung und Datenverarbeitung die „Data Acquisition Toolbox“, sowie eine angeschlossene Messkarte, welche die Messdaten empfängt.

Erklärung

„Hiermit versichere ich, die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, die Zitate ordnungsgemäß gekennzeichnet und keine anderen, als die im Literatur/ Schriftenverzeichnis angegebenen Quellen und Hilfsmittel benutzt zu haben. Ferner habe ich vom Merkblatt über die Verwendung von Bachelor- und Abschlussarbeiten Kenntnis genommen und räume das einfache Nutzungsrecht an meiner Masterarbeit der Universität der Bundeswehr München ein.“

____

(Ort, Datum, Unterschrift)