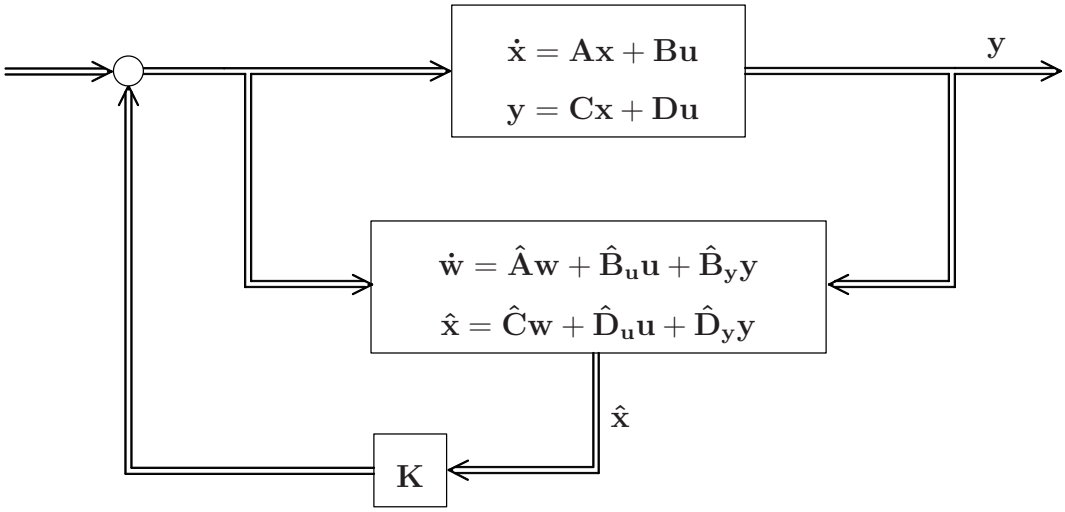
Johannes Kepler Universität Linz
Institut für Regelungstechnik
und Prozessautomatisierung

Praktikum
Automatisierungstechnik Vertiefung



Stand: WS 2011/12

O.Univ.Prof. Kurt Schlacher

Inhaltsverzeichnis

1 Organisation **1**

2 Maxima **4**
2.1 Maxima Einführung . . . 4
2.1.1 Entstehung von Maxima . . . 4
2.1.2 Benutzeroberflächen . . . 5
2.1.3 Installation . . . 5
2.1.4 Dokumentation . . . 6
2.2 Grundzüge der Syntax . . . 6
2.2.1 Grundlagen . . . 7
2.2.2 Grenzwerte, Differenziation und Integration . . . 8
2.2.3 Lineare Algebra . . . 11
2.2.4 Symbolische Manipulationen . . . 12
2.2.5 Gleichungen . . . 15
2.2.6 Graphische Darstellung . . . 17
2.3 Beispiel: Pendel mit Feder und Dämpfer . . . 19
2.3.1 Modellbildung in Maxima . . . 19
2.3.2 Linearisierung . . . 21

3 Scilab/Scicos **22**
3.1 Scilab Grundbegriffe . . . 22
3.1.1 Installation . . . 22
3.1.2 Dokumentation . . . 22
3.1.3 Grundbegriffe der Syntax . . . 23
3.1.3.1 Matrizen und Vektoren . . . 23
3.1.3.2 Polynome . . . 28
3.1.3.3 Grafische Darstellung . . . 28
3.1.3.4 Programmieren in Scilab . . . 30
3.2 Scicos Grundbegriffe . . . 33
3.2.1 Installation . . . 33
3.2.2 Dokumentation . . . 33
3.2.3 Grundbegriffe von Scicos . . . 33
3.2.3.1 Aufbau eines Modells . . . 34
3.2.3.2 Block Parameter . . . 36
3.2.3.3 Simulation und Datenaustausch . . . 36

3.3 Simulation von LTI Systemen . . . 38
3.3.1 Aufbau von LTI Modellen . . . 38
3.3.2 Kontinuierliche LTI Systeme . . . 40
3.3.2.1 Zustandsraummodell . . . 40
3.3.2.2 Übertragungsfunktion . . . 42
3.3.3 Diskrete LTI Systeme . . . 44
3.3.3.1 Zustandsraummodell . . . 44
3.3.3.2 Übertragungsfunktion . . . 45
3.4 Simulation mit Hilfe des GENERIC Blocks . . . 46
3.4.1 Einleitung . . . 46
3.4.2 Computational Function . . . 46
3.4.2.1 C Funktion . . . 47
3.4.2.2 Scilab Computational Functions . . . 48
3.4.3 Interfacing function . . . 49
3.5 Theoretische Einführung in Basisblöcke . . . 51
3.5.1 Super Blöcke . . . 51
3.5.2 Basisblöcke . . . 52
3.5.2.1 Einführung . . . 52
3.5.2.2 Interfacing Function . . . 52
3.5.2.3 Computational Function . . . 56
3.5.2.4 Kompilierung . . . 61
3.5.2.5 Simulation . . . 63

4 Praktikum 1 **64**
4.1 Mathematisches Pendel . . . 64
4.2 Ein elektrisches System . . . 65
4.3 Wagen & Pendel . . . 66

5 Praktikum 2 **68**
5.1 Ein elektrisches System . . . 68

6 Praktikum 3 **70**
6.1 Das Furuta Pendel . . . 70

Literaturverzeichnis **73**

Kapitel 1

Organisation

Das Praktikum bietet den Studierenden die Möglichkeit, die Open-Source Programme Maxima und Scilab/Scicos sowie ein Linux-basiertes Echtzeitsystem kennen zu lernen. Im Vordergrund stehen regelungstechnische Anwendungen. Für die Einarbeitung in die Programmpakete werden entsprechende Unterlagen bereitgestellt, welche z.T. eine Überarbeitung von [6] darstellen. Zudem werden die theoretischen Erklärungen durch Beispiele begleitet. Den Abschluss des Praktikums bildet die Scicos gestützte Implementierung eines Regelalgorithmus auf einem Linux-basierten Echtzeitsystem.

Ansprechpersonen

Bei Fragen zu den Aufgaben wenden Sie sich an den Praktikumsleiter Prof. Kurt Schlacher.

Praktikumsvorbereitung und -durchführung

Das Praktikum wird in Zweiergruppen absolviert. Die Gruppeneinteilung erfolgt in der Vorbesprechung. Die Aufgabenstellungen werden zeitgerecht auf der Institutshomepage im Downloadbereich zur Verfügung gestellt. Jeder Studierende muss mit der Lösung der Aufgabenstellungen vertraut sein. Um dies zu überprüfen werden in unregelmäßigen Abständen Einzelaufgaben gestellt.

Rechnerraum

Der Rechnerraum steht allen Praktikumsteilnehmern zur Verfügung. Folgende Punkte sind zu beachten:

- Der Rechnerraum muss nach dem Benützen wieder ordnungsgemäß verlassen werden.
- Kopieren von Software ist Diebstahl.
- Zum Ablegen von Daten steht Ihnen nur das Laufwerk D zur Verfügung. Bitte legen Sie keine Dateien auf dem Desktop, auf dem Laufwerk C oder in den Eigenen Dateien

ab. Für den Erhalt der lokal abgespeicherten Daten können wir keine Garantie abgeben. Die Laufwerke werden gelegentlich von den Studentendaten gesäubert, d.h. bringen Sie ins Praktikum Ihre Daten auf einem Datenträger (USB-Stick, externe Festplatte, CD,...) mit.

- Stecken Sie keine Verbindungskabel um! Bei unsachgemäßer Behandlung kann es zur Beschädigung der Hardware kommen. Bei Fragen wenden Sie sich bitte an einen Institutsangestellten.
- Sollten bei den Rechnern technische Gebrechen auftreten, so ist dies ebenfalls am Institut zu melden!
- Es ist nicht möglich mit den Laborrechnern ins Internet zu gelangen.
- Die Software Maxima und Scilab/Scicos sind auf den Laborechnern installiert. Kontrollieren Sie, ob ihre Dateien kompatibel sind. Im Praktikum kann keine Rücksicht auf nicht funktionierende Ausarbeitungen genommen werden!

Maxima

Maxima ist ein symbolisches Rechenprogramm, das ähnlich zu Scilab für Privatnutzer kostenlos zur Verfügung steht. Den Ursprung hat Maxima im Computer-Algebra-System Macsyma, das ursprünglich 1967 vom MIT (*Massachusetts Institute of Technology*) in Boston entwickelt wurde. Seit 1998 ist Macsyma Dank der Erlaubnis des amerikanischen Energieministeriums öffentlich zugänglich. Im Jahre 2000 wurde Maxima an SourceForge übergeben und wird seit dem über dieses Portal gewartet und verbessert.

Maxima ist in der Programmiersprache Common LISP implementiert. Inzwischen gibt es einige Benutzeroberflächen, allen voran wxMaxima, das auf wxWidgets basiert. Weiters gibt es noch Imaxima, das sowohl von Emacs als auch von TEXmacs einbunden werden kann.

Maxima wurde für symbolisches Rechnen programmiert und hat seine Stärken in der Linearen Algebra. Die graphische Darstellung von Funktionen beruht auf Gnuplot.

Scilab

Scilab ist ein numerisches Rechenprogramm, entwickelt für die Regelungstechnik und Signalverarbeitung. Als Alternative zu Matlab wurde es 1990 am *Institut national de recherche en informatique et en automatique* (INRIA) in Frankreich entwickelt und die Weiterentwicklung wird seit 2004 von einem Konsortium unter der Leitung des INRIA koordiniert. Dieses in C und Fortran programmierte Softwarepaket steht für Nutzung jeglicher Art kostenlos zur Verfügung. Ursprünglich wurde Scilab für Unix Workstations entwickelt, ist jedoch inzwischen auch für fast alle anderen Betriebssysteme (z.B. Windows, MacOSX) erhältlich.

Die Funktionalität und die Syntax ist stark an das weit verbreitete numerische Rechenprogramm Matlab angelehnt. Zusätzlich gibt es integrierte Skriptkonverter von Matlab

nach SCILAB. Ähnlich zu MATLAB, basiert SCILAB auf Matrizenrechnung, wofür etliche implementierte SCILAB Funktionen gibt, die das wissenschaftliche Rechnen mit Matrizen vereinfachen. Die Matrizen können aus verschiedenen Datentypen bestehen inklusive reellen und komplexen Zahlen, Funktionen, Polynomen, Zeichenketten und vielen mehr.

Seine Stärke erhält SCILAB erst durch die große Anzahl von Bibliotheken. Diese bestehen aus Funktionen, die in der SCILAB eigenen Skriptsprache geschrieben sind. Viele der Bibliotheken sind auf die lineare Algebra, numerische Integration und Optimierung spezialisiert und enthalten auch Funktionen für die Analyse von nichtlinearen Funktionen und (impliziten) dynamischen Systemen. Durch die immer größere Anzahl von Programmpaketen, die zum Großteil auf der offiziellen Homepage `http://www.scilab.org` veröffentlicht werden, ergibt sich eine Vielzahl von Anwendungsgebieten.

Es gibt zusätzlich einige graphische Möglichkeiten um 2D-, 3D- oder parametrische Plots und Animationen zu erzeugen. Diese können in verschiedensten Formaten (z.B. Postscripts, Gif, Postscript-Latex, Xfig) exportiert werden.

Scicos

SCICOS (*Scilab Connected Object Simulator*) ist eine der wichtigsten Programmpakete aus SCILAB. Diese enthält die Möglichkeit zur Modellbildung und zur Simulation von dynamischen Systemen und entspricht dem MATLAB Paket SIMULINK. Mittels verschiedener Blöcke, die entweder aus Bibliotheken gewählt oder selbst implementiert werden, kann man zeitkontinuierliche oder zeitdiskrete Systeme simulieren. So ist es möglich komplexe Systeme mittels Knopfdruck auf Plausibilität zu prüfen.

Der große Vorteil liegt, ähnlich zu SIMULINK, im modularen Aufbau einer Scicosdatei. Funktionen, Systeme, Subsysteme und vieles mehr werden in SCICOS durch I/O Blöcke charakterisiert. Damit können Parameter des Systems meist mit einem Klick verändert werden. Ein Großteil der Standardblöcke, die häufig gebraucht werden, sind bereits in SCICOS enthalten. Falls diese nicht ausreichen, können benutzerdefinierte Blöcke implementiert werden. Für diese selbst erstellten Blöcke gibt es verschiedene Möglichkeiten der Implementierung. Einerseits kann man aus schon bestehenden Standardblöcken ein Subsystem zusammenfügen und dieses Subsystem als Block speichern. Weiters gibt es die Möglichkeit eine eigene Funktion mit Hilfe der flag-Schreibweise zu verfassen. Dies kann u.a. in der SCILAB Skriptsprache, in C oder in FORTRAN erfolgen. Für die Kompilierung der benutzerdefinierten Blöcke ist ein C Compiler erforderlich.

Kapitel 2

Maxima

Dieses Kapitel gibt dem Leser eine grobe Übersicht über die Syntax und Anwendung von Maxima. Bevor jedoch auf diese näher eingegangen wird, steht die Installation und eine Übersicht verschiedener Benutzeroberflächen im Vordergrund. Zur Veranschaulichung der Befehle und Vorgehensweisen dient die Modellbildung des mathematischen Pendels im Kapitel 2.3.

2.1 Maxima Einführung

Neben der Entstehungsgeschichte handelt dieser Abschnitt von den verschiedenen Benutzeroberflächen, Betriebssystemen und verfügbaren Dokumentationen.

2.1.1 Entstehung von Maxima

Das Computer-Algebraprogramm Maxima ist eine Weiterentwicklung des Systems Macsyma, welches vom MIT als Teil des MAC Projekts[1] in den 60er Jahren des letzten Jahrhunderts entwickelt wurde. Dieses Projekt wurde hauptsächlich vom amerikanischen Verteidigungsministerium und Energieministerium unterstützt. Bis Mitte der 80er Jahre entwickelten sich mehr als sechs verschiedene Versionen (*Macsyma*, *Paramax*, *Punimax*, *Aljbar* und *Vmaxima*), wobei klar war, dass nicht alle am Markt überleben können. Die Einführung von Maple und Mathematica Ende der 80er Jahre verdrängte beinahe alle Versionen von *Macsyma* vom Markt. Erst durch die Veröffentlichung der Version von Prof. William Schelter, der *Macsyma* über die Jahre immer weiter entwickelt hatte und diese *Maxima* nannte, brachte den Durchbruch im Jahre 1998. Auf einen Schlag verschwanden *Macsyma* und alle andere Variationen, da Maxima ab sofort als kostenloses Programm zur Verfügung stand. Bis zu seinem Tod im Jahre 2001 wartete Prof. Schelter das System und übergab es im Jahre 2000 an Sourceforge.

[1] Das MAC Projekt steht für Multiple Access Computer, Machine Aided Cognition und Man And Computer und wurde am MIT (Massachusetts Institute of Technology) 1963 gegründet und war in den 60er und 70er Jahren am MIT eines der bedeutendsten IT Projekte. Der Schwerpunkt dieses Projekts lag bei der Entwicklung neuer Betriebssysteme, künstlicher Intelligenz und Rechnertheorie.

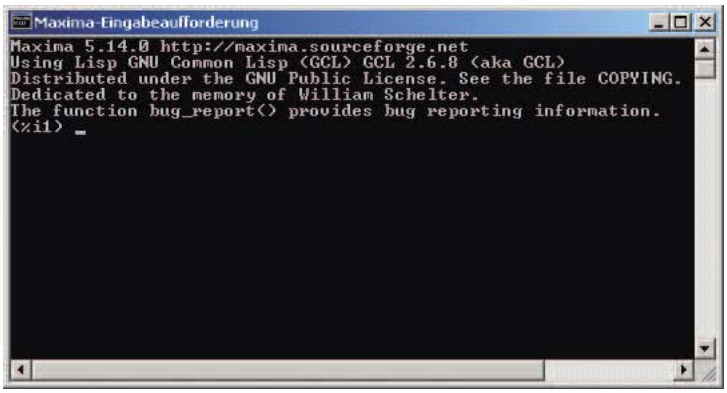


Abbildung 2.1: Maxima Terminal

2.1.2 Benutzeroberflächen

Die ursprüngliche Benutzeroberfläche von Maxima ist der Terminal. Trotz neuer Benutzeroberflächen kann Maxima noch mit dem Terminal (Konsole) gestartet und verwendet werden. Für Linuxuser gibt es mit *rmaxima* eine gute Alternative für eine erweiterte Konsole, die die Eingabe durch erweiterte Features etwas erleichtert.

Für eine Einbindung in LaTeX empfiehlt sich die Benutzeroberfläche *Emacs*. Die Entwicklung einer neueren Benutzeroberfläche und Einbindung von Graphiken durch *Xmaxima* brachte Maxima einen großen Schub. Die aktuelle Version *wxMaxima* wurde mit Hilfe von wxWidgets, eine Framework für eine plattformunabhängige Entwicklung von Benutzeroberflächen, implementiert. Dadurch erhöhte sich die Benutzerfreundlichkeit und die Kompatibilität zu unterschiedlichen Betriebssystemen, sodass die aktuelle Version in allen gängigen Betriebssystemen erhältlich ist. In dieser Version können, ähnlich zu *Emacs*, die einzelnen Ergebnisse in LaTeX Code ausgeben werden.

Die vorliegende Dokumentation verwendet die Version 0.8.4 von *wxMaxima* und Maxima 5.20.1 unter Windows XP Professional.

2.1.3 Installation

Maxima ist in den Paketverwaltungssystemen aller gängigen Linuxdistributionen zu finden und kann über deren Paketverwaltungssysteme (z.B.: Ubuntu: `apt`, Red Hat: `rpm`) installiert werden. Es ist jedoch zu beachten, dass die gewünschte Benutzeroberfläche eigens zu installieren ist. Für Windows und MacOSX findet man die Installationspakete unter den unten angegeben Links. Die Pakete für die Installation der Benutzeroberflächen beinhalten meist auch das Grundprogramm Maxima.

wxMaxima:
http://wxmaxima.sourceforge.net
Xmaxima:
http://sourceforge.net/project/showfiles.php?group_id=4933
Maxima:
http://maxima.sourceforge.net/
wxMaxima für MacOS X:

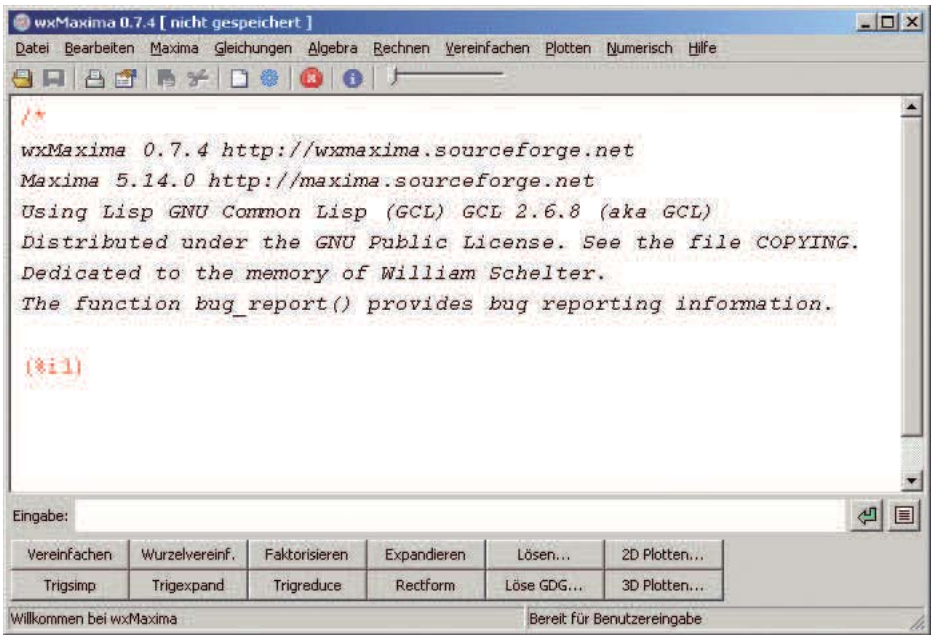


Abbildung 2.2: Benutzeroberfläche wxMaxima in Windows

http://wxmaxima.sourceforge.net/wiki/index.php/MacOSX

Xmaxima für MacOS X:

http://download.4pple.de/maxima-5.12.0-macosx-intel.dmg

2.1.4 Dokumentation

Über Maxima gibt es Dokumentationen in vielen verschiedenen Sprachen. Dank Moses Glasner vom Departement of Mathematics an der Universität von Pennsylvania gibt es eine Einführung für Studierende in Maxima und eine Sammlung von Dokumentationen und Tutorials.

http://www.math.psu.edu/glasner/

Zusätzlich zu dieser Dokumentation werden laufend auf der Maxima - Sourceforge - Homepage die neuesten Versionen und Dokumentationen zur Verfügung gestellt.

http://maxima.sourceforge.net
http://maxima.sourceforge.net/wiki/

Paulo Ney de Souza, ein Mitentwickler von Maxima schrieb 2004 ein Buch The Maxima Book [14], das er jedoch leider noch nicht vollendete. Trotz allem bildet dieses mit [1] die ausführlichste Quelle.

2.2 Grundzüge der Syntax

Die folgenden Seiten stehen ganz im Zeichen der Syntax von Maxima. Dabei werden sowohl Ähnlichkeiten als auch Unterschiede zu den weit verbreiteten Computeralgebrasystemen

wie *Maple* und *Mathematica* erläutert. Neben den Grundlagen stehen die lineare Algebra, Differenziation, Integration, die Lösung von Differenzialgleichungssystemen und die für die Regelungstechnik relevanten Befehle im Vordergrund.

2.2.1 Grundlagen

Maxima ist ein symbolisches Rechenprogramm mit dem Schwerpunkt auf symbolischer Algebra. Rationale oder irrationale Zahlen werden dabei symbolisch gespeichert. Jede Eingabe in Maxima muss mit einem Strichpunkt oder einem Dollarzeichen beendet werden, wobei mit Letzterem die Ausgabe unterdrückt wird. Im Gegensatz zu *Maple* wird durch Drücken der Tasten Strg+Enter ein Befehl ausgeführt. Seit Maxima 5.9.1 muss zwischen Groß- und Kleinschreibung unterschieden werden.

Zuweisungen In Maxima muss zwischen den verschiedenen Arten von Zuweisungen unterschieden werden. Für Funktionen, Gleichungen und Variablen müssen verschiedene Operatoren verwendet werden, die in Tabelle 2.1 angegeben sind. Mit dem Befehl `kill(var)` kann man die Zuweisung auf die Variable `var` löschen.

Typ	Operator	Beispiel
Variablen	`:`	`a:5`
Variablen	`::`	`B:3; C::B` (weiter einsetzen)
Funktionen	`:=`	`f(x) := x^2`
Gleichungen	`=`	`eq1: a+3=b`

Tabelle 2.1: Zuweisungsoperatoren Maxima

Ausgabe und Labels Jede Zeile in Maxima wird mit einem so genannten Label (`%i..`) begonnen, wobei `i` für Input und `o` für Output steht. Die Zahl nach dem Buchstaben kennzeichnet die Nummer der Ein- bzw. Ausgabe. Mit Hilfe der Labels kann auf die einzelnen Ein- und Ausgaben zugegriffen werden. Das letzte Ergebnis ist mit dem Zeichen `%` aufgreifbar. Jede Eingabezeile wird entweder mit einem Strichpunkt `;` oder einem Dollarzeichen `$` beendet. Im Falle des Dollarzeichens wird die Ausgabe unterdrückt. Um den Wert einer Variable auszugeben, kann man den Befehl `numer` an das Ende der Eingabe hängen. Eine weitere Möglichkeit besteht darin, die auszugebende Variable mit dem Befehl `bfloat(x)` aufzurufen. Dabei kann die Anzahl der eingeblendeten Stellen durch die globale Variable `fpprec` eingestellt werden. Standardmäßig ist diese auf den Wert 16 festgelegt. Die mathematischen Konstanten, die in Maxima implementiert sind, können mit einem bevorstehenden Prozentzeichen aufgerufen werden (z.B. `%pi`, `%e`, ...).

```
(%i1) 3*5;
(%o1)
```

$$15$$

```
(%i2) 3*5$
(%i3) 9/4*%pi, numer;
(%o3)
```

$$7.068583470577035$$

```
(%i4) var:2$
(%i5) var1:rhs(%i4)*3;
(%o5)
```

$$6$$

```
(%i6) var2:%*3;
(%o6)
```

$$18$$

```
(%i7) Var;
(%o7)
```

$$Var$$

```
(%i8) 2*x+1=7$
(%i9) solve(%,x);
(%o9)
```

$$[x = 3]$$

```
(%i10) fpprec:60$
(%i11) bfloat(%pi);
(%o11)
```

$$3.14159265358979323846264338327950288419716939937510b0$$

2.2.2 Grenzwerte, Differenziation und Integration

Grenzwerte und Beschränkungen Oft ist es notwendig für gewisse Variablen Einschränkungen durchzuführen. Dies unterstützt Maxima, äquivalent zu Maple, mit den Befehl `assume`. Um die Einschränkung durch `assume` rückgängig zu machen, ist der Befehl `forget` anzuwenden.

Mit den Funktionen `limit` und `tlimit` können Grenzwerte einer Funktion an einer gewissen Stelle berechnet werden. `limit` benützt hierfür die Regel von L'Hospital und

`tlimit` die Taylorreihenentwicklung. Zusätzlich kann man bei unstetigen Funktionen angeben ob der links- oder der rechtsseitige Grenzwert berechnet werden soll. Dafür muss man im Funktionsaufruf die Option `plus` für den rechtsseitigen und `minus` für den linksseitigen Grenzwert angeben. Für die Grenzwertsuche mittels L'Hospital kann man darüber hinaus festlegen, wie oft die Regel von L'Hospital angewendet werden soll. Dies geschieht durch die globale Variable `lhospitallim`, die standardmäßig auf 4 gesetzt ist. Die Zeichen größer als `>` und kleiner als `<` bedeuten, dass die Eingabe des Werts zwischen den Klammern optional ist.

```
limit(ausdruck, var, wert <, richtung>)
```

Falls kein Grenzwert existiert und der Ausdruck gegen positiv oder negativ unendlich geht, liefert `limit` den Wert `inf`, bzw `minf` für positiv bzw. negativ unendlich. Falls Maxima den Grenzwert nicht findet, die Funktion jedoch konvergiert, liefert es das Ergebnis `ind` (indefined) oder `und` (undefined). Bei Grenzwerten im komplexen Unendlichen gibt `limit` den Wert `infinity` zurück.

```
(%i1) limit((sin(x)^10/x^9),x,0)$
sin(x)^10
Assumed to be zero in 'taylor'
(%i2) lhospitallim:10$
(%i3) limit((sin(x)^10/x^9),x,0)
(%o3)
```

$$0$$

```
(%i4) limit(diff(abs(x),x),x,0,minus);
(%o4)
```

$$-1$$

Differenziation Ähnlich wie in Maple muss man vor der Differenzierung die Abhängigkeiten der Variablen untereinander definieren. Dazu dient der Befehl `depends`. Die Differenzierung erfolgt mit der Syntax

```
diff(f,x,n),
```

wobei `f` die Funktion ist, die differenziert wird, `x` die Variable ist, nach der differenziert wird, und `n` die Anzahl ist, wie oft differenziert wird. Der Befehl des Hochkommas `'` bedeutet allgemein, dass der folgende Ausdruck noch nicht berechnet werden soll.

```
(%i1) 'diff(f,x);
(%o1)
```

$$\frac{d}{d\,x}\,f$$

```
(%i2) depends(f,x);
(%o2)
```

$$[f\,(x)]$$

```
(%i3) diff(f,x);
(%o3)
```

$$\frac{d}{d\,x}\,f$$

```
(%i4) diff(g(x),x);
(%o4)
```

$$\frac{d}{d\,x}\,g\,(x)$$

```
(%i5) diff(g,x);
(%o5)
```

$$0$$

Alle Abhängigkeiten können mit `dependencies` aufgelistet werden. Gegebenenfalls können diese Abhängigkeiten einer Variable `var` auch mit

`remove(var,dependency)`

gelöscht werden. Funktionen die mit Hilfe der passiven Form, also mit dem Hochkomma `'`, differenziert wurden, können mit Hilfe des Befehls `ev()` und dem Zusatz `diff` berechnet werden.

```
(%i6) f(x) := 'diff(y,x)$
(%i7) ev(f(x),y = x^2,diff);
(%o7)
```

$$2\,x$$

Natürlich sind auch Ableitung mehrerer Veränderlicher mit `diff` möglich. Die Syntax lautet dafür

`diff(f(x1,x2,...),x1,n1,x2,n2,...)`,

wobei `n1` und `n2` die Anzahl der Ableitungen nach `x1` bzw. `x2` sind.

```
(%i8) f(x,y):= sin(x*y)$
(%i9) diff(f(x,y),x,1,y,1);
(%o9)
```

$$\cos\,(x\,y)-x\,y\,\sin\,(x\,y)$$

Matrixdefinition	`matrix([a11,a12,...],[a21,a22,...],...);`
Einheitsmatrix	`ident(n)`
Nullmatrix	`zeromatrix(m,n)`
Zeilen extrahieren	`row(Matrix, m)`
Spalte extrahieren	`col(Matrix, n)`
Teilmatrix extrahieren	`submatrix(m1,m2,...,Matrix,n1,n2,...)`
Zeile hinzufügen	`addrow(Matrix, v1, v2,...)`
Spalte hinzufügen	`addcol(Matrix, v1, v2,...)`

Tabelle 2.2: Grundbefehle Linearer Algebra in Maxima

Integration Unbestimmte Einfachintegrale können mit

```
integrate(ausdruck, var)
```

bestimmt werden. Maxima gibt die Lösungen ohne der immer auftretenden Integrationskonstante an. Für die Berechnung bestimmter Integrale müssen zusätzlich zur unabhängigen Variable `var` die obere und die untere Integrationsgrenze angegeben werden. Damit ergibt sich folgende Syntax für die Berechnung bestimmter Integrale

```
integrate(ausdruck, var, untereGrenze, obereGrenze)
```

```
(%i10) integrate(exp(a*x),x,1,inf);
Is a positive, negative, or zero?
negativ;
(%o10)
```

$$-\frac{e^a}{a}$$

```
(%i11) assume(a<0)$
(%i12) integrate(exp(a*x),x,1,inf);
(%o12)
```

$$-\frac{e^a}{a}$$

2.2.3 Lineare Algebra

Ebenso wie in Scilab kann auch in Maxima mit Matrizen und Vektoren gerechnet werden. Bis zur vollen Integration des LAPACK in Maxima ist zum Beispiel die Größe der Matrizen für die Berechnung der Eigenwerte begrenzt.

Vektoren sind wie auch in Scilab eine Sonderform von einspaltigen Matrizen und können auch so behandelt werden. Auf einzelne Elemente einer Matrix kann mit `matrix(m,n)` (oder in neueren Versionen mit `matrix[m,n]`) zugegriffen und diese gegebenenfalls verändert werden. Für einige Befehle (z.B. für das Vektorprodukt) muss das Paket `vect` geladen werden. Um das Ergebnis der Vektorberechnung anzuzeigen, muss der Befehl `express` ausgeführt werden.

Matrixinverse	`invert(Matrix)`
char. Polynom	`charpoly(Matrix, variable)`
Eigenwerte und -vektoren	`eigenvalues(Matrix)`
Eigenwerte	`allroots(charpoly(A,s))`
Determinante	`determinant(Matrix)`
Kreuzprodukt zweier Vektoren	`vect1~vect2`

Tabelle 2.3: Matrixfunktionen in Maxima

```
(%i1) load(vect)$
(%i2) a:[1,1,0]$
(%i3) b:[0,1,1]$
(%i4) express(a~b);
(%o4)
```

$$[1, -1, 1]$$

Mit Hilfe des Befehls `genmatrix()` können Matrizen mit bestimmter Systematik bzw. mit einer Funktion erzeugt werden. Es gilt folgende Sytnax, wobei die Startindizes weggelassen werden können, wenn sie 1 sind.

`genmatrix(feld, anzahlzeilen, anzahlspalten, zeilenstartindex, spaltenstartindex)`.

```
(%i1) f[m,n]:=sin(m)*n$
(%i2) genmatrix(f,3,3);
(%o2)
```

$$\begin{bmatrix} sin\,(t) & 2\,sin\,(t) & 3\,sin\,(t) \\ sin\,(2\,t) & 2\,sin\,(2\,t) & 3\,sin\,(2\,t) \\ sin\,(3\,t) & 2\,sin\,(3\,t) & 3\,sin\,(3\,t) \end{bmatrix}$$

Weiters berechnet der Befehl

`jacobian(Liste der Funktionen,Liste der Variablen)`

die Jacobische Funktionalmatrix .

```
(%i3) jacobian([sin(x)*y,cos(y)],[x,y]);
(%o3)
```

$$\begin{bmatrix} cos\,(x)\;y & sin\,(x) \\ 0 & -sin\,(y) \end{bmatrix}$$

2.2.4 Symbolische Manipulationen

Im Vergleich zu Maple ist die Definition von Funktionen in Maxima sehr einfach, da die Syntax sehr intuitiv erfolgt. Die Syntax wird mit Hilfe eines Beispiels erklärt.

```
(%i1) f(x,y):= x^2 + 2*y^2$
(%i2) a:f(2,3);
(%o2)
```

$$22$$

Faktorisieren In Maxima kann man einen rationalen Ausdruck sowohl automatisch als auch gesteuert mittels bestimmter Optionen faktorisieren. Die Funktion `factor` faktorisiert rationale Ausdrücke vollständig in reell irreduzierbare Faktoren. Dem gegenübergestellt ist die Funktion `gfactor`, die auch komplexe, irreduzierbare Faktoren liefert.

```
(%i1) gfactor(x^2+1);
(%o1)
```

$$(x-i)\,(x+i)$$

```
(%i2) factor(x^2+1);
(%o2)
```

$$x^2+1$$

Expandieren Ähnlich zum Faktorisieren, gibt es beim Expandieren einige verschiedene Befehle mit entsprechenden Optionen. Die wichtigsten sind `distrib` und `expand`. `distrib` multipliziert alle Produkte von Summen auf erster Ebene aus. Der Befehl `expand` expandiert dafür auf allen Ebenen, sodass bei Brüchen Zähler und Nenner expandiert werden. Die Syntax dafür lautet

```
expand(ausdruck <, pos_exponent_max, neg_exponent_min>)
```

Die beiden letzten Parameter des Aufrufs sind optional und legen den maximalen positiven und negativen Exponenten fest, der noch ausmultipliziert werden soll.

```
(%i3) distrib((a+b)*(c+d)/((e+f)*(g+h)));
(%o3)
```

$$\frac{b\,d}{(f+e)\,(h+g)}+\frac{a\,d}{(f+e)\,(h+g)}+\frac{b\,c}{(f+e)\,(h+g)}+\frac{a\,c}{(f+e)\,(h+g)}$$

```
(%i4) expand((a+b)*(c+d)/((e+f)*(g+h)));
(%o4)
```

$$\frac{b\,d}{f\,h+e\,h+f\,g+e\,g}+\frac{a\,d}{f\,h+e\,h+f\,g+e\,g}+\frac{b\,c}{f\,h+e\,h+f\,g+e\,g}+\frac{a\,c}{f\,h+e\,h+f\,g+e\,g}$$

Die Funktion `trigexpand` expandiert trigonometrische und hyperbolische Ausdrücke durch Anwendung von Additionstheoremen und Winkelvielfachen, wobei pro Aufruf dieses Befehls nur ein einziges Mal expandiert wird.

```
(%i5) trigexpand(sin(3*y+2*x));
(%o5)
```

$$\cos(2x)\,\sin(3y) + \sin(2x)\,\cos(3y)$$

```
(%i6) trigexpand(%);
(%o6)
```

$$\left(\cos^2 x - \sin^2 x\right)\,\left(3\,\cos^2 y\,\sin y - \sin^3 y\right)$$
$$+2\,\cos x\,\sin x\,\left(\cos^3 y - 3\,\cos y\,\sin^2 y\right)$$

Simplifizieren Maxima unterscheidet beim Faktorisieren und Simplifizieren zwischen rationale und nicht rationale Ausdrücke. Die Funktion `factor` ist nur auf rationale Funktionen beschränkt. `ratsimp` und `fullratsimp` sind die äquivalenten Befehle beim Vereinfachen von Ausdrücken. Sie multiplizieren Produkte von Summen aus und fassen gemeinsame Terme zusammen. Zusätzlich wird bei Brüchen der größte gemeinsame Teiler ermittelt und gekürzt. Der Unterschied zwischen `ratsimp` und `fullratsimp` ist die Anzahl der Vereinfachungen. Während man mit `ratsimp` nur eine Vereinfachung pro Aufruf erreicht, vereinfacht `fullratsimp` den Ausdruck bis keine Änderung mehr möglich ist.
Für trigonometrische Funktionen gibt es neben dem Befehl `trigexpand`, der im vorigen Absatz diskutiert wurde, die Befehle `trigreduce` und `trigsimp`. `trigreduce` entspricht ungefähr dem Gegenteil von `trigexpand`. Der Aufruf versucht den Ausdruck mit Hilfe der Additionstheoreme und Winkelvielfache zu verkürzen. `trigsimp` hingegen versucht durch die Beziehungen $\sin^2 x + \cos^2 x = 1$ und $\sinh^2 x - \coth^2 x = 1$ den Ausdruck so weit wie möglich zu vereinfachen. Dabei gilt wieder, dass diese Befehle nur eine Vereinfachung pro Aufruf vornehmen.

Partialbruchzerlegung Mit Hilfe des Befehls `partfrac` kann eine Partialbruchzerlegung bezüglich einer Variable durchgeführt werden. Die Syntax für diese Operation lautet

```
partfrac(ausdruck, var).
```

```
(%i1) f:(x^2+2)/2+2/(x^2-2);
(%o1)
```

$$\frac{x^2+2}{2}+\frac{2}{x^2-2}$$

```
(%i2) fullratsimp(f);
(%o2)
```

$$\frac{x^4}{2\,x^2-4}$$

```
(%i3) partfrac(%,x);
(%o3)
```

$$\frac{x^2+2}{2}+\frac{2}{x^2-2}$$

2.2.5 Gleichungen

Mit dem Befehl `solve` können in Maxima sowohl Nullstellen von Polynomen berechnet als auch trigonometrische, hyperbolische, gemischte oder viele weitere Gleichungen gelöst werden. Dabei muss man zwischen der symbolischen und der numerischen Lösung von Gleichungen unterscheiden. Während numerische Lösungen nur Näherungslösungen sind, bilden symbolische eine exakte Lösung der angegebenen Gleichungen. Die Syntax für das symbolische Lösen von Gleichungen lautet

```
solve(ausdruck, var).
```

Falls der Ausdruck nicht als Gleichung vorliegt, geht Maxima davon aus, dass die Gleichung `ausdruck=0` lautet. `var` ist die unabhängige Variable des Ausdrucks, dessen Nullstellen bestimmt werden sollen. Bei Polynomen ist zu beachten, dass eine symbolische Lösung im allgemeinen Fall nur bis zur vierten Ordnung möglich ist.

Symbolische Lösung von Gleichungssystemen Die Funktion `solve` löst ein System von linearen oder nichtlinearen polynomialen Gleichungen durch impliziten Aufruf der Funktionen `linsolve` oder `algsys`. Ein direkter Aufruf der letztgenannten Befehle ist nicht empfehlenswert, da damit auf einige nützliche interne Zusatzprüfungen verzichtet wird.

```
solve([ausdruck1, ausdruck2,...],[var1, var2,...])
```

Numerische Lösung von Gleichungen Numerische Lösungen sind für mathematische Probleme wichtig und oft notwendig, da eine symbolische Lösung in manchen Fällen gar nicht existiert. Die wichtigsten numerischen Lösungsverfahren in Maxima sind in Tabelle 2.4 angegeben.

`allroots(poly)`	findet alle Nullstellen
`realroots(poly)`	findet alle reellen Nullstellen
`nroots(poly,u_Gr, o_Gr)`	sucht die Anzahl der Nullstellen zwischen `u_Gr` und `o_Gr`
`newton(a1,x,x0,eps)`	sucht die Nullstellen von a1 der Variable x mit dem Startwert `x0` mit der Lösung `a1`(x) < `eps`. vorheriges Laden von `newton1`

Tabelle 2.4: numerische Lösungsverfahren

Für die numerische Lösung von Gleichungssystemen eignet sich das in Maxima implementierte Newtonverfahren. Die notwendige Syntax für die Gleichungen `a1=0`, `a2=0`, ... lautet:

```
newton([a1,a2,...],[var1,var2,...],[s1,s2,...], fehler)
```

Vor dem Ausführen des Befehls muss je nach Maxima Version das Paket `newton1` mit Hilfe von `load(newton1)` geladen werden. Der Faktor `eps` steht für das Abbruchkriterium des Newton Algorithmus. Die Berechnung der Lösung wird so lange fortgesetzt, bis der Ausdruck `a1(x)`, `a2(x)`, ... kleiner ist als der Wert `eps`. Für die Gleichung

$$0 = -x + sin\,(x) + 0.5$$

muss folgender Maximacode implementiert werden.

```
(%i1) load(newton1)$
(%i2) newton(x-sin(x)-0.5,x,0.5,0.00001);
(%o2)
                          1.497300699458674
```

Gewöhnliche Differenzialgleichungen Die wichtigste Funktion für die allgemeine Lösung einer gewöhnlichen Differenzialgleichung ist `ode2` (ordinary differential equation). `ode2` ist in Maxima eine Sammlung verschiedener Solver zur Lösung gewöhnlicher Differenzialgleichungen maximal zweiter Ordnung, die Maxima in einer fixen Reihenfolge nach dem Aufruf von `ode2` durchläuft. Die dazugehörige Syntax lautet:

```
ode2(dgl, abhängige_var, unabhängige_var)
```

Für die Lösung speziellerer Differenzialgleichungen gibt es in Maxima das Paket `contrib_ode`, das mit dem Befehl `load('contrib_ode)` geladen werden kann. Mit Hilfe dieses Pakets können sowohl implizite als auch explizite Differenzialgleichungen mit unterschiedlichen Methoden gelöst werden. Zusätzlich gibt es Funktionen wie `ode_lin`, die die Lösung von Differenzialgleichungen in bestimmter Form durchaus schneller und effizienter als `ode2` berechnen.

Für die Mitberücksichtigung der Anfangsbedingungen oder der Randbedingungen gibt es in Maxima die Befehle `ic1` bzw. `ic2` und `bc2`. Der Befehl `ic1` berechnet die Konstanten der Differenzialgleichung erster Ordnung mittels einer Anfangsbedingung. `ic2` ist der äquivalente Befehl für Differenzialgleichungen zweiter Ordnung. `bc2` berechnet die Konstanten eines Randwertproblems auf Grund von zwei Randbedingungen. Für diese drei Befehle gilt folgende Syntax:

```
ic1(loesung, xval, yval)

ic2(loesung, xval, yval, dval)

bc2(loesung, xval1, yval1,xval2, yval2)
```

Für die Lösung gewöhnlicher Differenzialgleichungen mittels der Laplacetransformation existiert `desolve`. Dieser Befehl kann mit folgender Syntax sowohl für einzelne Differenzialgleichungen als auch für Differenzialgleichungssysteme verwendet werden.

```
desolve([eqn_1, ..., eqn_n], [x_1, ..., x_n])
```

```
(%i1) depends(x,t)
(%i2) eq:1+diff(x,t)+diff(x,t,2)=0$
(%i3) loes:ode2(eq,x,t);
(%o3)
```

$$x = \%k2\, e^{-t} - t + \%k1 + 1$$

```
(%i4) loes_ges:ic2(loes,t=0,x=0,diff(x,t)=0);
(%o4)
```

$$x = -e^{-t} - t + 1$$

Da eine ausführliche Behandlung der weiteren Berechnungsmöglichkeiten den Rahmen dieser Einführung in Maxima sprengen würde, muss an dieser Stelle auf die Literatur [15] und [14] verwiesen werden.

2.2.6 Graphische Darstellung

Mit Hilfe der Einbindung verschiedener Benutzeroberflächen besteht die Möglichkeit in Maxima zwei- oder dreidimensionale graphische Darstellungen zu erzeugen. Die Syntax für zweidimensionale Darstellung ist

```
plot2d([funkt1, funkt2, ...],[x_Bereich], [y_Bereich] <, Optionen>)
```

Der Aufruf eines dreidimensionalen Plots ist abgesehen vom Schlüsselwort (`plot3d`) völlig identisch. Entscheidend für die graphische Darstellung von Funktionen sind die Darstellungsoptionen, die entweder bei Bedarf global mit dem Befehl `set_plot_option` oder durch die Mitübergabe entsprechender Parameter beim plot-Aufruf verändert werden

`[var,lim_neg,lim_pos]`	Wertebereich für `var` von `lim_neg` bis `lim_pos`
`[grid,points_x,points_y]`	setzt Anzahl der Punkt in x und y Richtung fest
`[view_direction,x,y,z]`	definiert für 3D Grafiken den Blickpunkt
`[xlabel, name]`	setzt den Namen der x-Achse auf `name`
`[ylabel, name]`	setzt den Namen der y-Achse auf `name`
`logx` bzw. `logy`	skaliert die x bzw. y Achse logarithmisch
`[legend, name1,name2,..]`	erzeugt eine Legende mit `name1`, `name2`, ...

Tabelle 2.5: Grafikoptionen Maxima

können. Die Grundoptionen sind in Tabelle 2.5 angegeben. Für weitere Details sei an dieser Stelle auf [3] und [15] verwiesen.

Der in Abbildung 2.3 gezeigte dreidimensionale Plot wurde mit folgendem Befehl erzeugt:

```
plot3d(2^(-u^2 + v^2), [u, -3, 3], [v, -2, 2],[grid,30,30]);
```

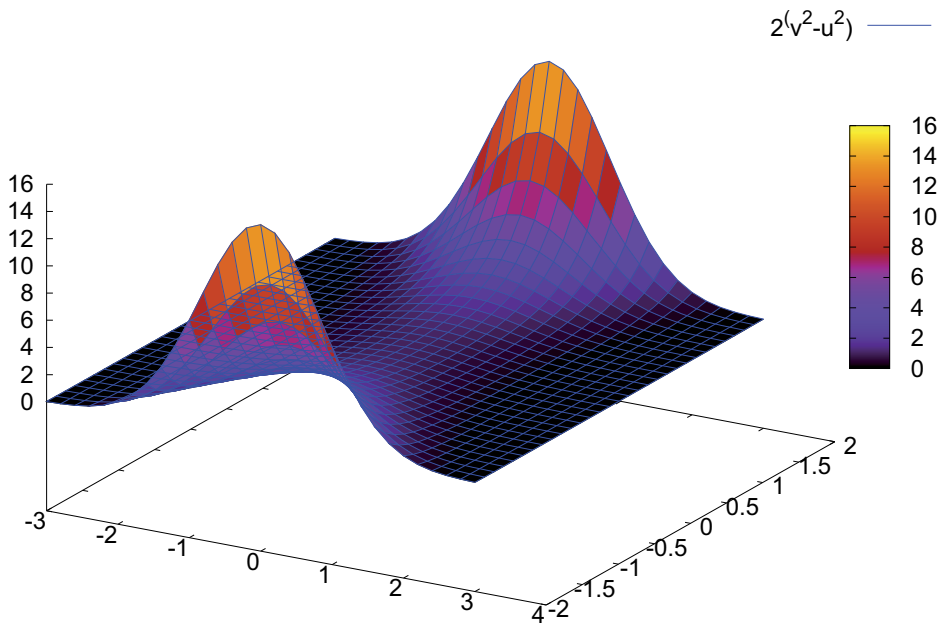


Abbildung 2.3: Maxima 3D Plot

2.3 Beispiel: Pendel mit Feder und Dämpfer

Im folgenden Abschnitt wird mit Hilfe von Maxima die Modellbildung des mathematischen Pendels mit Feder und Dämpfer nach Abbildung 2.4 auf Basis des Drallsatzes und der Lagrange Gleichung zweiter Art durchgeführt. Der Aufstellung der nichtlinearen Bewegungsgleichungen folgt die Linearisierung des Systems.

Ein mathematisches Pendel (masseloser Stab der Länge l und der Punktmasse m) ist über eine lineare Drehfeder, die bei $\varphi = 0$ entspannt ist, und einem winkelgeschwindigkeitsproportionalen Drehdämpfer im Drehgelenk mit einem Rahmen verbunden. Der Winkel φ des Pendels wird von der Horizontalen durch den Drehpunkt im Uhrzeigersinn gemessen. Auf das System wirkt ein Moment $M(t)$ und die Schwerkraft g (siehe Abbildung 2.4). Mit Hilfe des Drallsatzes um den Lagerpunkt des Pendels erhält man die Gleichung 2.1

$$ml^2\ddot{\varphi} = mgl\cos(\varphi) - c\varphi - d\dot{\varphi} - M(t)\,. \tag{2.1}$$

Daraus folgt das System nichtlinearer Differenzialgleichungen erster Ordnung

$$\begin{aligned}\dot{\varphi} &= \omega \\ \dot{\omega} &= \frac{1}{ml^2}(mgl\cos(\varphi) - c\varphi - d\omega - M(t))\,.\end{aligned} \tag{2.2}$$

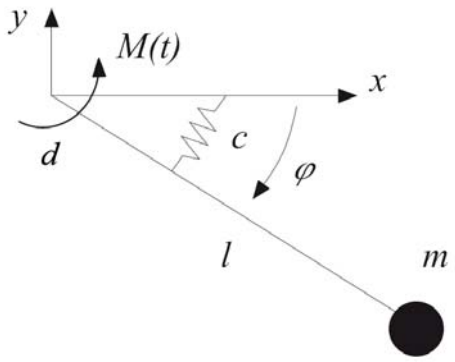


Abbildung 2.4: Modell Pendel

2.3.1 Modellbildung in Maxima

Die Modellbildung wurde mit Hilfe der Lagrange Gleichung zweiter Art durchgeführt. Diese lautet

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial L}{\partial \dot{q}}(q,\dot{q})\right) - \frac{\partial L}{\partial q}(q,\dot{q}) + \frac{\partial R}{\partial \dot{q}}(q,\dot{q}) = Q$$

mit

$$L = T - V\,,$$

wobei T für die kinetische und V für die potentielle Energie, R für die Rayleigh-Dissipationsfunktion, q für die Minimalkoordinaten und Q für den Vektor der verallgemeinerten, eingeprägten Kräfte steht.

Im folgenden Abschnitt erkennt man den Maxima-Code für die Modellbildung und die Berechnung der Ruhelagen des mathematischen Pendels mit Feder und Dämpfer.

```
(%i1) kill(All)$
(%i2) var:[phi,omega]$
      inp:[M]$
      depends(var,t)$
(%i5) const:[g=10,m=2*%pi,l=1,c=60*sqrt(3)]$
/*Energiegleichungen
(%i6) T:1/2*m*l^2*diff(phi,t)^2$
(%i7) V:1/2*c*phi^2-m*g*l*sin(phi)$
(%i8) L:T-V$
/*Lagrangegleichung
(%i9) R:d/2*diff(phi,t)^2$
(%i10) eq_L:diff(diff(L,diff(phi,t)),t)-diff(L,phi)+diff(R,diff(phi,t))=-
M$
(%i11) eq_omega_p:solve(eq_L,diff(phi,t,2));
(%o11)
```

$$[\frac{d^2}{d\,t^2}\,\phi = -\frac{M + d\,\left(\frac{d}{d\,t}\,\phi\right) - g\,l\,m\,cos\,(\phi) + c\,\phi}{l^2\,m}]$$

```
(%i12) sys_eq:[diff(phi,t)=omega,
       diff(omega,t)=rhs(subst(omega,diff(phi,t),eq_omega_p[1]))];
(%o12)
```

$$\left[\begin{array}{c}\frac{d}{d\,t}\,\phi = \omega \\ \frac{d}{d\,t}\,\omega = -\frac{M - g\,l\,m\,cos(\phi) + c\,\phi + d\,\omega}{l^2\,m}\end{array}\right]$$

```
/*Berechnung der Ruhelagen
(%i13) load(newton1)$
(%i14) sys_ruhe:solve(subst([M=0,omega=0],rhs(sys_eq[2]))=0,phi);
(%o14)
```

$$[\phi = \frac{g\,l\,m\,cos\,(\phi)}{c}]$$

```
(%i15) sys_ruhe:subst(const,sys_ruhe)$
       phi_s:bfloat(newton(rhs(sys_ruhe[1])-phi,phi,1,1/1000));
(%o16)
```

$$5.237768188747123b - 1$$

```
(%i17) ruhelage:[phi=phi_s, omega=0]$
```

Durch die Definition `depends` am Beginn des Maxima Skripts werden die Abhängigkeiten der Variablen untereinander definiert. Dies erleichtert besonders das Differenzieren

von Ausdrücken, da dadurch das Ableiten von $L(\varphi(t))$ nach t möglich wird.

2.3.2 Linearisierung

Die Zustandsraummatrizen des linearisierten Systems (2.4) ergeben sich aus den Jacobimatrizen der nichtlinearen Funktionen (2.3). In diese Matrizen müssen die Ruhelagen, um welche linearisiert werden soll, eingesetzt werden. Im Falle dieses Systems erkennt man, dass die obere Ruhelage bei $\varphi = (2k-1)\pi$ instabil und die untere bei $\varphi = 2k\pi$ stabil ist. Die Berechnungsvorschrift der Systemmatrizen erhält man aus der Taylorreihenentwicklung von

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{u}), \;\; \mathbf{y} = \mathbf{g}(\mathbf{x}, \mathbf{u}) \tag{2.3}$$

um eine Ruhelage

$$\mathbf{f}(\mathbf{x}_s, \mathbf{u}_s) = 0\,.$$

Daraus folgt

$$\begin{aligned} \Delta\dot{\mathbf{x}} &= \mathbf{A}\,\Delta\mathbf{x} + \mathbf{B}\,\Delta\mathbf{u}\,, \\ \Delta\mathbf{y} &= \mathbf{C}\,\Delta\mathbf{x} + \mathbf{D}\,\Delta\mathbf{u} \end{aligned} \tag{2.4}$$

mit den Matrizen

$$\begin{aligned} \mathbf{A} &= \frac{\partial}{\partial\mathbf{x}}\mathbf{f}(\mathbf{x},\mathbf{u})\,|_{\mathbf{x}_s,\mathbf{u}_s}\,, \;\; \mathbf{B} = \frac{\partial}{\partial\mathbf{u}}\mathbf{f}(\mathbf{x},\mathbf{u})\,|_{\mathbf{x}_s,\mathbf{u}_s}\,, \\ \mathbf{C} &= \frac{\partial}{\partial\mathbf{x}}\mathbf{g}(\mathbf{x},\mathbf{u})\,|_{\mathbf{x}_s,\mathbf{u}_s}\,, \;\; \mathbf{D} = \frac{\partial}{\partial\mathbf{u}}\mathbf{g}(\mathbf{x},\mathbf{u})\,|_{\mathbf{x}_s,\mathbf{u}_s}\,. \end{aligned}$$

Die Linearisierung wird mit folgendem Maximacode durchgeführt, wobei die Matrix `sys_eq` für die nichtlinearen Systemgleichungen $\mathbf{f}(\mathbf{x}, \mathbf{u})$ steht und der Befehl `jacobian`, dem Namen entsprechend, die Jacobimatrix berechnet.

```
/*Linearisierung
(%i17) A:jacobian(makelist(rhs(sys_eq[i]),i,1,length(sys_eq)),states)$
(%i18) B:jacobian(makelist(rhs(sys_eq[i]),i,1,length(sys_eq)),inp)$
(%i19) cT:matrix([1.,0])$
```

Kapitel 3

Scilab/Scicos

Das folgenden Kapitel steht ganz im Zeichen der numerischen Berechnung und Simulation mit Hilfe von Scilab/Scicos. Der Leser soll eine Übersicht von der Syntax und Anwendung bis hin zur eigenen Erstellung von Simulationsblöcken erhalten. Zur Veranschaulichung dient auch in diesem Kapitel das Beispiel des mathematischen Pendels.

3.1 Scilab Grundbegriffe

Dieser Abschnitt handelt von der Installation, Dokumentation und der grundlegenden Syntax von Scilab.

3.1.1 Installation

Die aktuellste Scilabversion kann von Homepage `www.scilab.org` heruntergeladen werden. Im Praktikum wird die (ältere) Version *Scilab 4.1.2* verwendet. Für eine einwandfreie Installation der Binaryversionen sind minimal 40Mb notwendig. Die Sourcecodeversion verlangt ungefähr 130 MB Festplattenspeicher für das Entpacken und Installieren. Für einige Applikationen und Funktionen aus Scilab benötigt man einen C und Fortran Compiler (empfehlenswert ist Visual C++ oder mingw 5.1.4. für Windows).

3.1.2 Dokumentation

Über Scilab gibt es sehr ausführliche Dokumentationen und Tutorials, von denen einige auf Grund der Herkunft des Programms auf französisch sind. Das offizielle Handbuch mit der Beschreibung aller Scicos Funktionen kann von der Scilab Homepage heruntergeladen werden.

```
www.scilab.org/publications/index_publications.php?page=freebooks
```

Weiters sei noch auf das Buch *Modeling and Simulation in Scilab/Scicos* [2] verwiesen, wobei die Kapitel 1 und 2 direkt auf der Homepage des Springerverlags verfügbar sind und die Kapitel 6 und 7, eine Einführung zum Zusatzpaket Scicos, von der offiziellen Scicos Homepage heruntergeladen werden können. Weitere Einführungen können unter folgenden Links gefunden werden:

wiki.scilab.org/
www.nt.fh-koeln.de/fachgebiete/mathe/rechlehr.html
kiwi.emse.fr/SCILAB/(französisch)
www.fh-htwchur.ch/Arbeiten-mit-Scilab-und-Scicos.1100.0.html

Abgesehen von diesen Dokumentationen und Tutorials beinhaltet Scilab eine sehr ausführliche Hilfefunktion, welche über die Befehle `help` und `apropos` mit anschließendem Funktionsnamen aufgerufen werden kann. In diesem Fall öffnet ein Fenster mit den Definitionsdaten dieser Funktion. Als Beispiel dient hier die Funktion zur Definition der Einheitsmatrix

```
-->help eye;
```

Neben der Hilfefunktion liefert Scilab eine Vielzahl von programminternen Demos, die unter anderem praktische Beispiele graphischer Darstellungen und Anwendungen aus der Regelungstechnik beinhalten.

3.1.3 Grundbegriffe der Syntax

Da dieses Projektseminar weder eine Auflistung der exakten Syntax noch Einführung in die Programmierung von Scilab sein soll, sei an dieser Stelle auf die Literatur [2] und besonders auf die oben genannten Links verwiesen. Jedoch sollen hier in aller Kürze die Grundbegriffe erwähnt werden.

In Scilab existieren, ähnlich zu MATLAB, zwei verschiedene Grundfunktionen. Einerseits ist Scilab ein Interpreter und andererseits eine skriptbasierte Programmiersprache. Beim Starten von Scilab erscheint ein Fenster ähnlich der Abbildung 3.1, wobei die Darstellung abhängig vom jeweiligen Betriebssystem variieren kann. In diesem Fall werden die einzelnen Befehle durch Drücken der Return-Taste ausgeführt. Falls die Eingabezeile mit einem Strichpunkt beendet wird, unterdrückt Scilab die Ausgabe des Ergebnisses.

Im Gegensatz dazu gibt es einen Scilab Editor mit inkludiertem Debugger, mit dessen Hilfe kleine Skriptprogramme erstellt werden können. Dieser Editor wird mit Hilfe des Befehls `scipad()` gestartet. Alle eigens erstellten Funktionen müssen bei Verwendung im Arbeitsverzeichnis liegen, welches entweder mit Hilfe der Menüleiste `File -> Change Directory` oder mit dem Befehl `chdir("Pfadname")` verändert werden kann.

Grundelemente, wie Konstanten oder die Boolschen Operatoren werden mit der Präfix % gekennzeichnet. Mit dem Befehl `who()` gibt Scilab alle deklarierten Variablen aus.

3.1.3.1 Matrizen und Vektoren

Matrizenerstellung Das Grundelement von Scilab sind Matrizen, die mit allen verschiedenen Typen von Elementen gefüllt werden können. So sind Skalare, Vektoren oder Listen nur spezielle Strukturen von allgemeinen Matrizen. Die Eingabe einer Matrix erfolgt durch eckige Klammern, wobei diese im Falle eines Skalars entfallen können. Die Elemente der Matrix innerhalb einer Zeile werden durch Leerzeichen oder Kommata getrennt. Ein Zeilenumbruch wird durch ein Strichpunkt signalisiert.

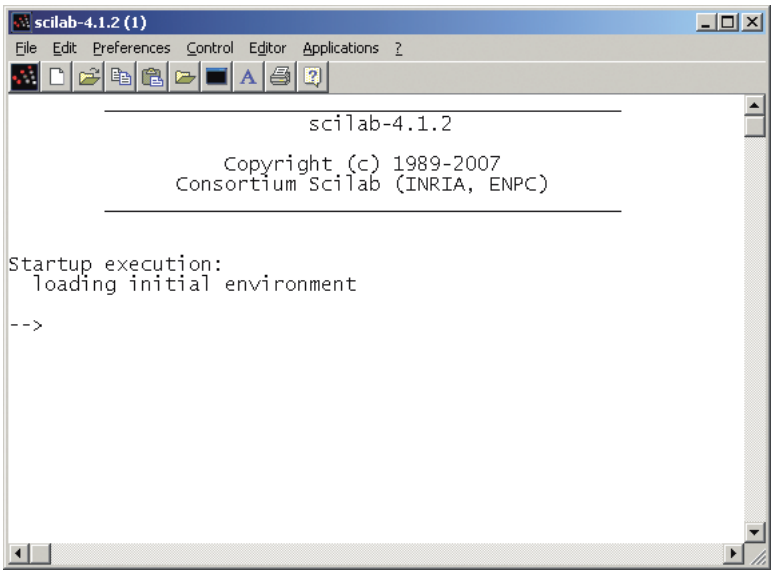


Abbildung 3.1: Scilab Hauptfenster

`scipad()`	öffnet den Scilab Editor
`who()`	zeigt alle verwendeten Variablen an
`stacksize()`	zeigt die maximale und verwendete Speichergröße durch Anzahl von double precision words an
`stacksize(n)`	setzt die maximal verwendbare Speichergröße auf `n` double precision words
`chdir('Pfadname')`	setzt das Arbeitsverzeichnis auf das angegebene Verzeichnis
`typeof(var)`	gibt den Variablen Typ von `var` zurück
`clear('var')`	löscht die Variable `var`
`exists('var')`	gibt 0 oder 1 für die Existenz der Variable zurück

Tabelle 3.1: Grundbefehle Scilab

```
-->A=[1 1 2; 4 5, 7; 5 7 8]
 A  =
    1. 1. 2.
    4. 5. 7.
    5. 7. 8.
```

Falls sich die Eingabe über mehrere Zeilen erstrecken soll, um zum Beispiel eine Matrixdeklaration übersichtlicher zu gestalten, müssen vor jedem Zeilenwechsel drei Punkte eingegeben werden.

`%pi`	3.141...
`%e`	2.718...
`%i`	imaginäre Zahl
`%t`	true/wahr
`%f`	false/falsch
`%s`	kontinuierliche, komplexe Frequenzvariable
`%z`	diskrete, komplexe Frequenzvariable

Tabelle 3.2: Grundelemente

```
-->A=[1 1 2;...
      4 5 7;...
      5 7 8]
 A  =
    1. 1. 2.
    4. 5. 7.
    5. 7. 8.
```

Für die einfache Erstellung von Matrizen sind in Tabelle 3.3 einige Funktionen angegeben.

' und .'	transponiert (konjugiert und nicht konjugiert)
`diag(n)`	Diagonalmatrix aus den Elementen der Liste `n`
`eye(m,n)`	`(m,n)` Matrix mit Einsern auf der Hauptdiagonalen
`rand(m,n)`	`(m,n)` Zufallsmatrix mit Werten zwischen 0 und 1
`zeros(m,n)`	`(m,n)` Nullmatrix
`ones(m,n)`	`(m,n)` Matrix bestehend aus Einsern
`linspace(a,e,j)`	Vektor beginnend mit `a` und dem Ende `e` mit `j` Elem.
`a:i:e`	Vektor beginnend mit `a`, dem Inkrement `i` und dem Ende `e`
`logspace(a,e,j)`	Vektor beginnend mit `a`, dem Ende `e` mit `j` Elementen im logarithmischen Inkrement
`matrix(v,n,m)`	`(m,n)` Matrix aus dem Vektor `v` (Länge `n*m`)

Tabelle 3.3: Funktionen zur Matrixerstellung

Operatoren Matrixoperationen können auf Grund des matrixbasierten Aufbaus von Scilab sowohl für Skalare als auch für Matrizen oder Listen geeigneter Dimension verwendet werden. Es gibt jedoch einige Unterschiede zwischen komponentenweisen Operationen und Matrixoperationen. Grundsätzlich weisen Operatoren, die mit einem dot Symbol "." beginnen, auf eine komponentenweise Operation hin. Der Unterschied wird im folgenden Beispiel erläutert.

```
-->A = [1, 2; 3 4]
 A =
    1. 2.
    3. 4.
-->A*ones(2,2)
 ans =
    3. 3.
    7. 7.
-->A.*ones(2,2)
 ans =
    1. 2.
    3. 4.
```

Im zweiten Fall wird mit dem Operator ".*" jede einzelne Komponente der Matrix A mit der entsprechenden Komponente der Einsermatrix multipliziert. Damit wird jede Komponente mit eins multipliziert. Im Falle der Matrixmultiplikation sieht die Rechenregel "Zeile mal Spalte" ein anderes Ergebnis vor. In der Tabelle 3.4 sind mögliche Operatoren zusammengefasst.

|	logisches ODER
&	logisches UND
~	logisches nicht
==, <=, >=, >, <, <>, ~=	Vergleichsoperatoren
+, -	unäre Operatoren
+, -	binäre Operatoren
.*,./,.\,.*.,.\.,./.,*,/,\.,/.	Multiplikationen und Divisionen
^,**,.^, .**	exponential Operatoren
', .'	transponiert Operatoren

Tabelle 3.4: Operatoren

Der Unterschied zwischen komponentenweisen Operationen und Matrixoperationen ist wichtig für Funktionen, wie etwa die Exponentialfunktion, die sowohl auf Skalare als auch auf Matrizen angewandt werden kann. Im Falle der Regelungstechnik, in der die homogene Lösung eines linearen, zeitinvarianten und autonomen Systems $\dot{\mathbf{x}}(\mathrm{t}) = \mathbf{A}\,\mathbf{x}(\mathrm{t})$ durch $\mathbf{x}(\mathrm{t}) = \boldsymbol{\phi_t}(\mathbf{x_0}) = e^{\mathbf{A}t}\mathbf{x_0}$ berechnet wird, spielt dies eine wichtige Rolle. Aus diesem Grund gibt es in Scilab einerseits die Funktion `exp`, welche den Operator komponentenweise ausführt und andererseits die Funktion `expm`. Diese führt die Exponentialfunktion, wie im Falle von $\boldsymbol{\phi_t}(\mathbf{x_0})$ als numerische Matrixoperation aus.

Einfügen, Löschen und Extrahieren Für die Matrizenrechnung ist es immer notwendig einzelne Spalten oder Zeilen zu löschen, einzufügen oder aus einer Matrix heraus zu lösen. Dazu gibt es in Scilab einige Möglichkeiten, wobei hier nur die einfachsten erklärt werden sollen.
Um Elemente einer Matrix zu löschen, genügt es ihnen die leere Matrix zuzuweisen. Um eine Matrix zu erweitern, genügt es ebenfalls neuen Indizes einen Wert zuzuordnen. Zu-

=	Zuweisungsoperator
$	letzter Index
:	alle Elemente
[]	leere Matrix

Tabelle 3.5: Matrixmanipulationen

sätzlich kann man einer Matrix eine gesamte Matrix anhängen.

```
-->A = [1, 2; 3 4]
 A =
    1. 2.
    3. 4.
-->A(:,$+1) = [1;1]
 A =
    1. 2. 1.
    3. 4. 1.
-->A(:,$) = []
 A =
    1. 2.
    3. 4.
-->B = [1;4];
-->A = [A,B]
 A =
    1. 2. 1.
    3. 4. 4.
```

Lineare Gleichungssysteme Eine wichtige Aufgabe für numerische Rechenprogramme ist das Lösen linearer Gleichungssysteme (`Ax = b`). Damit dieses Gleichungssystem eindeutig lösbar ist, muss die Matrix A quadratisch und invertierbar sein. Scilab nützt dafür, ähnlich wie MATLAB, eine numerisch LU-Zerlegung mit partieller Pivotzerlegung.

```
-->A = [1, 2; 3 4];
-->b = [1;1];
-->x = A\b     // Lösung des Gleichungssystems A*x = b
 x =
  - 1.
    1.
```

Zusätzliche Matrixbefehle In der Tabelle 3.6 sind einige zusätzliche Matrixbefehle angegeben, die öfters von Nutzen sind. Diese Befehle kann man teilweise mit Optionen versehen. Zum Beispiel berechnet man mit der Option `'r'` im Befehl `sum(A,'r')` die zeilenweise Summe der Matrix (äquivalent bei `prod(A)`). Anstatt `'r'` kann auch `'c'` eingesetzt werden, wobei dann die Summen- bzw. Produktbildung über der Matrix spaltenweise durchgeführt wird. Die genauen Ein- und Ausgabe Parameter können im Handbuch und in der Hilfe nachgelesen werden.

`sum(A)`	berechnet die Summe Elemente von A
`prod(A)`	berechnet das Produkt der Elemente von A
`min(A)`	gibt minimalen Wert und Position von A an
`max(A)`	gibt maximalen Wert und Position von A an
`mean(A)`	berechnet Mittelwert der Elemente von A
`st_deviation(A)`	berechnet Standardabweichung der Elemente von A
`size(A)`	gibt Dimensionen von A an
`length(A)`	gibt Anzahl der Elemente in A an
`spec(A)`	berechnet die Eigenwerte und Eigenvektoren von A

Tabelle 3.6: Zusätzliche Matrixbefehle

3.1.3.2 Polynome

In Scilab gibt es ein eigenes Polynomobjekt, das sich jedoch bei genauerem Hinsehen als Spezialart einer Liste und damit eines Matrixobjekts herausstellt. Polynome können in Scilab durch den Befehl `poly` definiert werden. Als Argumente werden entweder die Nullstellen oder die Koeffizienten als Liste übergeben. Die Polynomvariable muss bei der Definition als Option beigefügt werden (zum Beispiel `'s'`). Falls die Koeffizienten und nicht die Nullstellen des Polynoms bekannt sind, ist die Option `'c'` von Nöten.

```
-->p = poly([1,2],'s')
 p =
                2
    2 - 3s + s
-->p = poly([1,2],'s','c')
 p =
    1 + 2s
```

Additionen, Multiplikationen und Divisionen sind Dank der in Scilab hinterlegten Polynomoperatoren ohne weiters möglich.

3.1.3.3 Grafische Darstellung

Scilab beinhaltet sehr vielfältige Grafikfunktionen, die teilweise eine große Ähnlichkeit zu MATLAB aufweisen.

Grafikfenster Die Befehle `plot2d`, `plot3d` bilden die Basis grafischer Darstellungen von zweidimensionalen Funktionen, Vektorfeldern oder Flächen. Mit diesem Befehl wird ein Scilab Fenster - meist Nr. 0 - geöffnet und für weitere Aktionen aktiviert. Falls mehrere Funktionseingaben erfolgen, werden diese Graphen im Allgemeinen überlagert. Für den Umgang mit den Grafikfenstern benötigt man die in Tabelle 3.7 angegebenen Befehle
Weitere Einstellungen über Farben, Dicke der Linien und Zeichensatz des gesamten Fensters kann man entweder interaktiv im Grafikfenster oder direkt durch den Befehls `xset('name', a1, a2)` einstellen. (`'name'` steht für den Parametertyp und `a1` bzw. `a2` für die Parameter). Durch die Eingabe des Befehl `xget()` erscheinen alle eingestellten und standardmäßigen Grafikparameter.

`clf(num)` oder `xbasc(num)`	löscht den Inhalt des Fensters `num`
`xdel(num)`	schließt das Fenster Nummer `num`
`xset("window", num)`	öffnet oder aktiviert das Fenster `num`
`xselect(num)`	bringt das Fenster `num` in den Vordergrund

Tabelle 3.7: Befehle Grafikfenster

`'backround', color`	Hintergrundfarbe
`'color', value`	Farbe der Linien, Flächen und Texte
`'font', fontid, fontsize`	Zeichensatz
`'foreground', color`	Vordergrundfarbe
`'line style', value`	Linientyp (durchgehend, gestrichelt)
`'mark', markid, marksize`	Markierungssymbol
`'thickness', value`	Dicke der Linien
`'wdim', width, height`	Höhe und Breite des Fensters
`'window', number`	öffnet oder aktiviert Fenster `num`
`'wpos', x, y`	linkes, oberes Eck des Fensters auf x,y

Tabelle 3.8: `xset` Parameter

plot2d heißt die Anweisung für einen zweidimensionalen Plot in Scilab. Dabei gilt folgende Syntax `plot2d(Abszisse, Ordinate <,optionen>)`. Die Matrix `Abszisse` muss natürlich dieselben Dimensionen wie die Matrix `Ordinate` haben, wobei pro Spalte ein Graph gezeichnet wird. Somit können auch mehrere Graphen gleichzeitig gezeichnet werden. Zusätzlich zu dem Befehl `xtitle(String)`, mit dessen Hilfe der Titel des Plot definiert wird, können dem Aufruf `plot2d` verschiedene Optionen beigefügt werden. Dazu gibt es folgende Syntax (wobei die Reihenfolge der Schlüsselwörter keine Rolle spielt)

Schlüsselwort = Wert

Die verschiedenen Parameterwerte sind der Hilfe zu entnehmen.

Option	Syntax	Beispiel
Farbe	*style = vector*	`style = [1 4 8]`
Legende	*leg = string*	`leg = 'G1@G2@G3'`
Markierungssymbole	*style = vector*	`style = [-1 -5 -7]`
Maßstab	*frameflag = int*	`frameflag = 3`
Lage der Achsen	*axesflag = int*	`axesflag = 5`
log. Maßstab	*logflag = String*	`logflag = 'ln'`

Tabelle 3.9: Grafikoptionen

Ähnlich wie in MATLAB gibt es in Scilab den Befehl `subplot(m,n,num)`, wobei m die Anzahl der Zeilen, `n` die Anzahl der Spalten und `num` die Zahl des Plots ist (es wird zeilenweise von links oben nach rechts unten gezählt). Für eine Treppenfunktion muss anstatt `plot2d`, der Befehl `plot2d2` und äquivalent dazu bei Balkendiagrammen `plot2d3` verwendet werden.

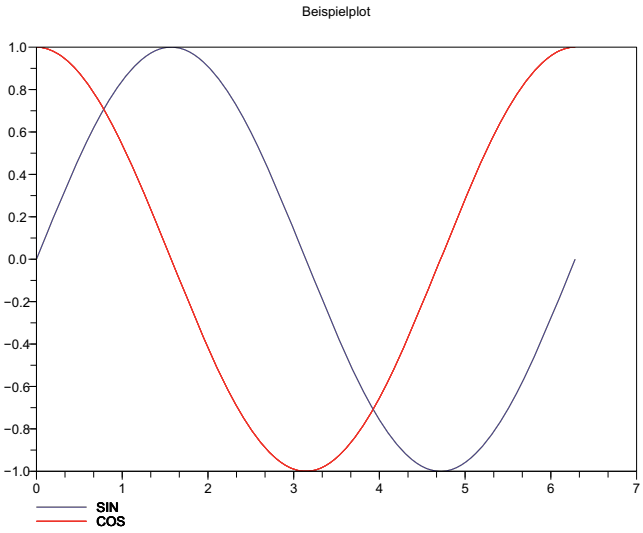


Abbildung 3.2: Beispielplot

Der Scilab Code zu Abbildung 3.2:

```
--> x1 = linspace(0,2*%pi,100)';
--> xbasc()
--> plot2d(x1, [sin(x1), cos(x1)], style=[9,5], leg='SIN@COS');
--> xtitle('Beispielplot')
```

3.1.3.4 Programmieren in Scilab

Ein Scilab Programm ist, ähnlich zu einem MATLAB Programm, eine Gruppe von Scilabbefehlen, die in einer bestimmten Reihenfolge ausgeführt werden. Diese Befehle können einerseits in die Konsole geschrieben werden, oder mit der Hilfe eines Editors, zum Beispiel dem `Scipad`, in einer ASCII Datei gespeichert werden. Diese Skripts können, nachdem sie in die Scilabumgebung mit dem Befehl `exec(name)` (ggf. kann auch der Befehl `getf(name)` verwendet werden). Es gibt zwei verschiedene Arten von Endungen der abgespeicherten Scilabdateien. Wenn die Datei nur Scilabfunktionen beinhaltet, wird die Endung `.sci` verwendet, in allen anderen Fällen `*.sce`.

Da Scilab eine typische Skriptsprache ist, besteht der große Unterschied zu herkömmlichen Programmiersprachen wie C oder Java dadurch, dass Variablen vor dem Gebrauch nicht deklariert werden müssen. Somit muss zum Beispiel die Größe und der Typ von Matrizen im Vorhinein nicht bekannt sein. Scilab besitzt direkte Grafikbefehle, die nicht über eine Bibliotheksfunktion aufgerufen werden müssen. Der Preis für den Komfort der Bedienerfreundlichkeit ist die reduzierte Geschwindigkeit der Programme. Skriptbasierte Programme sind meist wesentlich langsamer als äquivalente in C implementierte Programme.

Funktionen sind eine Gruppe von Befehlen, die mit einem Aufruf und der Übergabe von Parametern in Scilab ausgeführt werden können. Sie werden im Unterschied von `.sce` Dateien, die nur eine bestimmte Reihenfolge von Befehlen ohne einer Parameterübergabe abarbeiten, in `.sci` Dateien abgespeichert. Für die Erstellung von Scilab Funktionen gilt folgende Syntax

```
function [y1,y2,...,yn] = Funktionsname(x1,x2,...,xm)
```

Diese Funktion wird mit `m` Eingangsparameter aufgerufen und `n` Parameter werden in Form eines Vektors zurückgegeben. Die übliche Methode, eine Funktion in Scilab zu definieren, besteht darin, sie und möglicherweise weitere in einer Datei zu speichern, wobei jede einzelne Funktion mit der oben genannten Syntax beginnen muss. Zusätzlich zu der Syntax am Beginn einer Funktion muss diese mit `endfunction` beendet werden. Damit Scilab diese Funktion aufrufen kann, muss die Datei, die die Funktion beinhaltet, mit dem Befehl `getf(Dateiname)` geladen werden. Zu beachten ist, dass bei einer Änderung der Funktion diese neu geladen werden muss.

```
function [y1,n] = facultaet(n)
  if(n<0)
    error('Eingangsparameter ist kleiner eins!',1001)
  end
  y1 = prod(1:n)
endfunction
```

Als Beispiel sieht man im Absatz oberhalb eine selbst implementierte Funktion, die die Fakultät einer Zahl berechnet, indem sie das Produkt des Vektors von 1 bis `n` ausgibt. Bei einem negativen Eingangsparameter bricht das Programm ab und liefert eine Fehlermeldung. Falls der Aufruf dieser Funktion ohne Zuweisung der Ausgangsparameter in eine Variable erfolgt, wird nur die erste Ausgangsvariable, also in diesem Fall y1, übergeben. Dies soll an Hand eines Beispiels verdeutlicht werden.

```
-->exec('facultaet.sci');
-->facultaet(5)
 ans =
    120.
-->[a b] = facultaet(5)
 b =
    5.
 a =
    120.
--> facultaet(-2)
!--error 1001 Eingangsparameter ist kleiner eins!
at line 3 of function facultaet called by :
facultaet(-2)
```

Schleifen und Verzweigungen Ähnlich zum Großteil der Programmiersprachen gibt es in Scilab Schleifen und Verzweigungen. Die Syntax ist in diesem Fall sehr stark an kon-

ventionelle Sprachen wie C oder Java angelehnt. Da diese Programmierelemente selbsterklärend sind, wird an dieser Stelle nur die Syntax angegeben.

for - Schleife

```
for <Name>=<exp> // wobei exp ein Vektor der Länge n ist,
  <Befehle>      // und die Schleife n mal durchlaufen wird
end
```

andere Möglichkeit:

```
for var=n1:step:n2
  <Befehle>
end
```

while - Schleife

```
while <Bedingung> //vor jedem Durchlauf wird die Bedingung geprüft
  <Befehle>
end
```

if - then - else Konstruktion

```
if Bedingung_1 then
  <Befehle>
elseif Bedingung_2 then
  <Befehle>
.....
else
  <Befehle>
end
```

select-case Konstruktion

```
num = var
select num
case Ausdruck_1
  <Befehle>
case Ausdruck 2
  <Befehle>
.....
end
```

Diverses Weiters sei noch auf ein paar weitere praktische Befehle hingewiesen: Mit dem Wort `break` und einem nachfolgenden `end` besteht die Möglichkeit `for` oder `while` Schleifen vorzeitig abzubrechen. Somit kann man mit geringem Aufwand Schleifen mit dem Charakter repeat-until implementieren. Aus den Fehlermeldungen verschiedener Programme nützliche Informationen herauszulesen ist oft schwierig, deshalb gibt es eine Scilabfunktion `error(Fehlermeldung)`. Falls das Programm auf diesen Befehl stößt, bricht es die Funktion ab und gibt die Fehlermeldung auf der Konsole aus. Wenn man den Ablauf des Programms jedoch nicht abbrechen möchte, ist man mit dem Befehl `warning(Meldung)`

gut beraten.

3.2 Scicos Grundbegriffe

Scicos ist eines der wichtigsten Programmpakete innerhalb von Scilab. Sie dient der numerischen Simulation von linearen und nichtlinearen Systemen. In diesem Kapitel soll die Vorgangsweise zur Erstellung und Durchführung einer Simulation in Scicos näher erläutert werden.

3.2.1 Installation

Das Programmpaket Scicos wird standardmäßig bei der Installation von Scilab mitinstalliert. Um schon gefundene Bugs zu beheben, sollte die Version mit dem aktuellen Update-Patch der offiziellen Scicos Homepage

`www.scicos.org`

aktualisiert werden. Dokumentationen und Mitteilungen über mögliche Bugs und Updates sind ebenfalls auf dieser Homepage verfügbar.

3.2.2 Dokumentation

Besonders auf der Suche nach Dokumentationen über Scicos erkennt man die französischen Wurzeln dieses Programms. Viele Dokumentationen und Tutorials sind nur auf französisch erhältlich. Trotz alledem sind hier einige Links zum Thema Scicos angegeben. Die Kapitel 6 und 7 des Buchs *Modeling and Simulation in Scilab/Scicos* [2], welches schon in der Scilab Einführung mehrere Male erwähnt wurde, kann man als pdf Datei von der Homepage des Springerverlags oder von `www.scicos.org/documentations.html` herunterladen. Diese Kapitel enthalten eine Zusammenfassung der wichtigsten Scicos Funktionen.

`www.fh-htwchur.ch/uploads/media/Arbeiten_mit_Scilab_und_Scicos_v1.pdf`
`norma.mas.ecp.fr/wikimas/Scicos` (französisch)
`www.scicos.org/TUTORIAL/tutorial.html`
`www.scicos.org/p.pdf`

Das offizielle Benutzerhandbuch zum Programm ist in der Scilab Hilfe integriert.

3.2.3 Grundbegriffe von Scicos

Scicos kann mit dem Befehl `scicos` oder `scicos('dateiname')` von der Scilab Konsole gestartet werden. Daraufhin wird ein graphischer Editor für die Konstruktion von Modellen durch interagierende Blöcke geöffnet. Die erstellten Modelle können mit der Dateiendung `*.cos` gespeichert werden. Blöcke für die Konstruktion eines Modells können entweder aus den bestehenden Paletten kopiert oder selber erstellt werden. Sofern möglich, sollte man auf bestehende Blöcke zurückgreifen, oder diese bei Bedarf in Superblöcke zusammenfassen. Falls man während einer Scicos Sitzung auf Scilab zugreifen möchte,

muss man das Scilabfenster reaktivieren. Dies ist durch den Button `Activate Scilab Window` im Scicos Menu `Tools` möglich.

3.2.3.1 Aufbau eines Modells

Für den Aufbau eines Modells in Scicos stehen folgende Basispaletten aus Tabelle 3.10 zur Verfügung. Diese können entweder einzeln oder gesammelt in Baumstruktur unter `Palette --> PalTree` aufgelistet werden:

Sources	Sinks
Linear	Non_linear
Matrix	Integer
Events	Threshold
Others	Branching
Elektrical	ThermoHydraulics
OldBlocks	DemoBlocks

Tabelle 3.10: Scicos Paletten

Im Gegensatz zu Simulink wird in Scicos zwischen zwei Arten von Signalen und Leitungen unterschieden. Einerseits gibt es Signale im herkömmlichen Sinn und andererseits Steuersignale für die zeitliche Ansteuerung von Blöcken. Die verschiedenen Leitungen, die die unterschiedlichen Signaltypen transmittieren, sind durch Farbe und Position der Anschlüsse an den Blöcken unterscheidbar. Rote Signale deuten auf ein Steuerungssignal hin, wobei sie ihre Anschlüsse entweder auf der oberen oder auf der unteren Seite des Blocks haben, je nachdem ob sie der Ansteuerung des Eingangssignals (oben) oder des Ausgangssignals (unten) dienen. Falls ein Block kein zeitliches Steuerungssignal benötigt, "erbt" dieser es von dem Block, von dem das Signal zuvor stammt. Als Ansteuerungsquelle dient meist eine `Event Clock` mit einer bestimmten Taktzeit, welche in der Palette `Sources` gefunden werden kann.

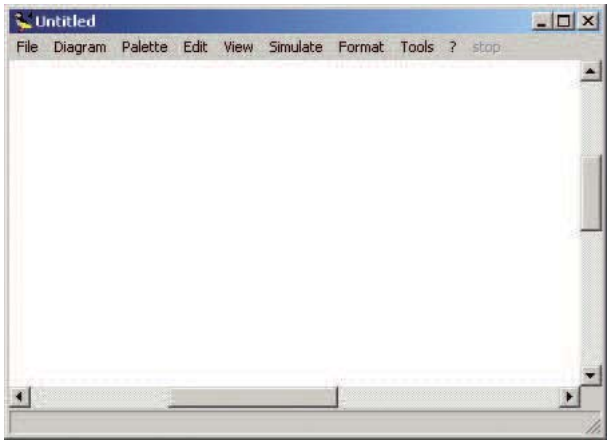


Abbildung 3.3: Scicos Editor Hauptfenster

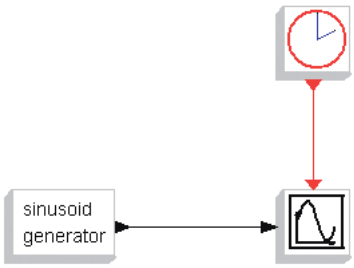


Abbildung 3.4: Scicos Testmodell

Wie man in Abbildung 3.4 erkennen kann, dient der Block `Event Clock` als Steuerungssignal für den Plot. Sobald ein Signal von diesem Steuerungsblock am Scope eintrifft, wird dieser aktiviert um einen Wert von seinem Eingangsport einzulesen. Aus diesem Grund ist die Leitung zwischen `Event Clock` und dem Scope als Zeichen für ein Steuerungssignal rot und an der Oberseite des Blocks. Die Leitung des Sinusgenerators ist schwarz, da es sich um eine reine Signalleitung handelt.

Synchronisation Wenn bei einem größeren Modell mit mehreren Blöcken Synchronisation zwischen zwei Blöcken erforderlich ist, müssen diese Blöcke aus derselben `Event Clock` angesteuert werden, da der Compiler ansonsten eine Berechnungsreihenfolge erstellt, die Asynchronität zur Folge haben kann. Bei der Ansteuerung von zwei Blöcken mit zwei `Event Clocks` und derselben `step time` ist Synchronität nicht gewährleistet.

Super Blocks In Scicos gibt es die Möglichkeit mit Hilfe von `Super Blocks`, ähnlich zu den Subsystemen in MATLAB, einen Teil eines Modells in ein weiteres Modell auszulagern. Die einfachste Möglichkeit einen Super Block zu schaffen, ist die auszulagernden Blöcke zu markieren und mit dem Befehl `Region-to-Super-block` auszulagern. Der Super Block kann wie ein eigenes Modell in einem eigenen Scicos Editor angesehen werden. Bei Super Blocks mit mehreren Ein/Ausgängen, müssen diese einzeln nummeriert sein. Im Ersatzschaltbild des Super Blocks sind seine Ausgänge bzw. Eingänge aufsteigend gemäß der internen Nummernvergabe von oben nach unten sortiert.

Super Blocks können auf zwei verschiedene Arten gespeichert werden. Die erste Möglichkeit besteht darin einen Super Block als Super Block abzuspeichern. Damit kann er in jedes Modell importiert und sein Schaltbild angepasst werden. Wenn man ihn jedoch als Palette speichert, kann man zwar die Blöcke aus ihm hinaus kopieren, sein Diagramm aber nicht verändern. Zusätzlich besteht die Möglichkeit einen Super Block in eine schon bestehende Palette mit Hilfe des `Pal Editors` aufzunehmen. Eine genaue Erörterung dieser Vorgehensweise würde jedoch den Rahmen sprengen und ist der Scicos Hilfe zu entnehmen.

Lineare und Nichtlineare Blöcke Für die Simulation linearer Systeme gibt es in Scicos kontinuierliche bzw. diskrete Übertragungsfunktionen und Zustandsraummodelle, sodass nur noch die Matrizen oder Koeffizienten als Blockparameter eingegeben werden

müssen. Für nichtlineare Blöcke gibt es einige mehr Möglichkeiten. Die einfachste, jedoch aufwändigste Alternative ist der Modellaufbau mittels vorhandener Blockdiagramme. Durch die Vielzahl der schon vorhandenen Blöcke ist dies zwar zum Teil möglich, jedoch mit Unübersichtlichkeit verbunden. Äquivalent zur CS-Funktion in Simulink gibt es die C-Funktion in Scilab. Diese ist jedoch nicht auf die Programmiersprache C beschränkt. Eine Implementierung in Fortran (F-Funktion) und in Scilab Skript Code (Scifunction) ist auch möglich. Weitere Informationen zur Erstellung eines eigenen Blocks können in Kapitel 3.5.2 und 3.4 gefunden werden.

3.2.3.2 Block Parameter

Es besteht die Möglichkeit numerische oder symbolische Werte für Blockparameter zu verwenden. Falls symbolische Werte verwendet werden, müssen diese, falls sie in Scilab noch nicht definiert wurden, im `Context` Menu (`Diagramm --> Context`) deklariert werden. In diesem Editor kann man sowohl ein Skript implementieren als auch eine externe Funktion aus Scilab mit vorigem Laden aufrufen. Um das `Context` Skript auszuführen, muss der Befehl `Eval` im Menu `Simulate` ausgeführt werden. Dies muss nach jeder Änderung des `Context` wiederholt werden, um die neuen Werte in die Simulation zu laden.

3.2.3.3 Simulation und Datenaustausch

Simulation in Scicos ist die einfachste aller Simulationen in Scilab. Ein durch Blöcke aufgebautes Modell wird durch eine Quelle innerhalb von Scicos angeregt. Das Ergebnis kann anhand eines Scopes betrachtet werden. Die Simulationsparameter können unter `Simulate --> Setup` eingestellt werden. Das Fenster mit den Simulationseigenschaften kann man in Abbildung 3.5 erkennen. Um an den Scopes ein stehendes Bild zu erhalten, sollte die `Final integration time` den selben Wert wie die `Refresh period` des entsprechendes Scopes haben. Um die Simulation zu starten, muss der Befehl `run` aus dem `Simulate` Menu ausgeführt werden.

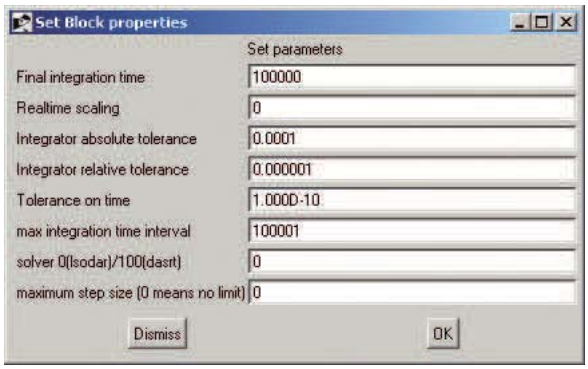


Abbildung 3.5: Scicos Simulationseigenschaften

Datenaustausch über Dateien Es besteht die Möglichkeit von Scilab aus direkt eine Simulation in Scicos zu starten und die Ergebnisse automatisch auszulesen. Als Schnittstelle zwischen den Eingangsdaten, die in Scilab generiert werden, und den Ausgangsdaten,

die in Scicos simuliert werden, dienen entweder ASCII Dateien, in denen die Werte gespeichert werden, der Scilabworkspace oder globale Variablen. Da der Datenaustausch über globale Variablen sehr ineffizient und langsam ist, wird an dieser Stelle auf das Kapitel *Batch Processing in Scilab* in [2] verwiesen. Schreiben und Lesen von Dateien in Scilab beruht auf den Befehlen `write('Dateiname')` und `read('Dateiname')`. Die Werte werden in Spalten gespeichert, wobei zu beachten ist, dass die entsprechenden Blockparameter in Scicos auf die Anzahl der Spalten angepasst sind. Weiters muss berücksichtigt werden, dass man Scilab Dateien nicht überschreiben kann. Aus diesem Grund müssen die ASCII Dateien vor dem Überschreiben mit dem Befehl `mdelete('Dateiname')` gelöscht werden. Zusätzlich ist wichtig, dass sich alle verwendeten Dateien im Arbeitsverzeichnis befinden.

```
chdir('D:\Projektseminar\Scilab');
mdelete('inputfile')
u = [zeros(0:.1:1) ones(1.1:0.1:2) zeros(2.1:0.1:3)]';
write('inputfile',u);

load inputtest.cos
%scicos_context.k = 2;
scicos_simulate(scs_m, list(), %scicos_context);

data=read('outputfile',-1,2)
t = data(:,1);
y = data(:,2);
plot2d(t,y);
```

Vor der externen Verwendung des Scicos Modells in Scilab muss man die Scicosdatei mit dem Befehl `load` laden. Mit dem Ausdruck `%scicos_context.var` kann man auf die Variable `var` im Contextmenu des Scicos Modells zugreifen und diese verändern. Um die Simulation auszuführen muss folgende Syntax eingehalten werden

```
scicos_simulate(scs_m, list(), %scicos_context) .
```

Mit Hilfe der Scicosblöcke `read from input file` und `write to output file` werden Daten importiert oder exportiert.

Datenaustausch über Workspace Diese Möglichkeit des Datenaus- tauschs über Workspacevariablen ist erst seit der Scicosversion 4.2, bzw. der Scilabversion 4.1.2 möglich. Dementsprechend finden sich noch einige Bugs im Programm. Die Simulation und das Auslesen der Werte in eine Variable des Workspace von einer `*.sce` Funktion aus ist noch nicht möglich. Dieses Problem soll jedoch laut offiziellen Angaben der Scilab Homepage in der nächsten Version behoben sein. Aus diesem Grund kann man die Simulation mit den Blöcken `to Workspace` und `from Workspace` nur direkt in Scicos starten. In diesen beiden Blöcken müssen die Namen der Input- bzw. Outputvariablen angegeben sein, wobei das Eingangssignal des Blocks `from Workspace` sowohl ein `time` Feld als auch ein `values` Feld benötigt. In den Blockparametern des Ausgangsblocks muss die Anzahl der auszulesenden Werte definiert werden.

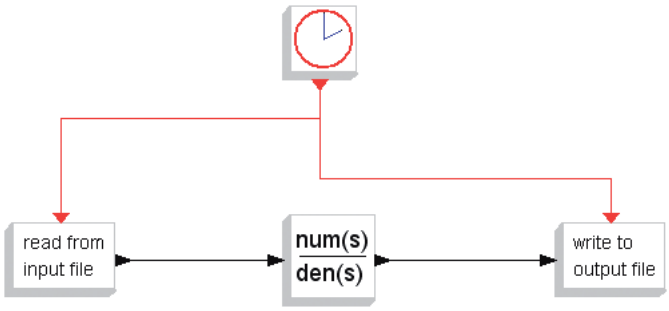


Abbildung 3.6: Scicos Simulation mit Datenaustausch über Dateien

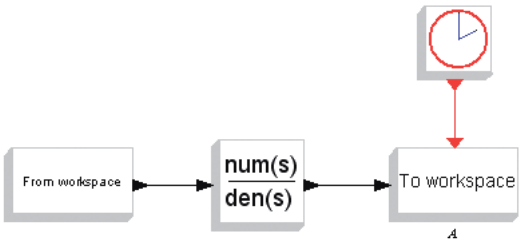


Abbildung 3.7: Scicos Simulation mit Datenaustausch über Workspace

In Abbildung 3.7 wurde die Eingangsvariable `V` und die Ausgangsvariable `A` benannt. Mit folgenden Befehlen wurde das Eingangssignal initialisiert.

```
V.time=[0:.1:3]';
V.values=[zeros(0:.1:1) ones(1.1:0.1:2) zeros(2.1:0.1:3)]';
```

3.3 Simulation von LTI Systemen

In diesem Abschnitt wird die Simulation von zeitkontinuierlichen und zeitdiskreten LTI Systemen behandelt.

3.3.1 Aufbau von LTI Modellen

Ähnlich zu MATLAB gibt es in Scilab zwei verschiedene Methoden ein lineares, zeitinvariantes Eingrößensystem

$$\begin{aligned} \dot{\mathbf{x}} &= \mathbf{A}\mathbf{x} + \mathbf{b}u \\ y &= \mathbf{c}^T\mathbf{x} + du \end{aligned} \tag{3.1}$$

bzw.

$$\begin{aligned} \mathbf{x}_{k+1} &= \mathbf{A}\mathbf{x}_k + \mathbf{b}u_k \\ y_k &= \mathbf{c}^T\mathbf{x}_k + du_k \end{aligned} \tag{3.2}$$

zu beschreiben und zu simulieren. Dieses System kann einerseits mit Hilfe der Matrizen des Zustandsraummodells oder andererseits mit einer Übertragungsfunktion beschrieben werden. Diese beiden Vorgehensweisen sind sowohl im Zeitkontinuierlichen als auch im Zeitdiskreten möglich und mit den Formeln

$$G(s) = \mathbf{c}^T\left(s\mathbf{E} - \mathbf{A}\right)^{-1}\mathbf{b} + d \tag{3.3}$$

bzw.

$$G(z) = \mathbf{c}^T\left(z\mathbf{E} - \mathbf{A}\right)^{-1}\mathbf{b} + d \tag{3.4}$$

in einander überführbar.

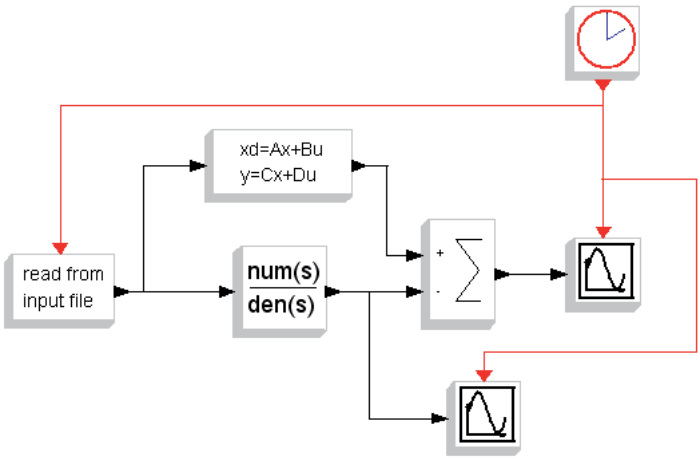


Abbildung 3.8: Simulation von LTI Systemen in Scilab

Für die Simulation von LTI Systemen in Scicos stehen zwei Blöcke in der `Linear` Palette (`transfer function` und `state-space system`) zur Verfügung. Die Übertragungsfunktion wird programmintern in ein Zustandsraummodell transformiert und daraufhin (für den zeitkontinuierlichen Fall) wie das Zustandsraummodell mit der Funktion `csslti` simuliert.

In Abbildung 3.8 kann man einen Modellaufbau mit einer Übertragungsfunktion und einem äquivalenten Zustandsraummodell erkennen. Die aus dem Schaltbild errechnete Differenz zwischen den beiden Ausgangssignalen wird zu Null, weil in diesem Fall nicht einmal numerische Ungenauigkeiten eine Rolle spielen, da es sich um den exakt selben Funktionsaufruf handelt.

In Tabelle 3.11 sind nützliche Befehle für den Umgang mit LTI Systemen zusammengefasst. Die in MATLAB üblichen Funktionen wie `tf()` oder `pzmap()` sind in Scilab nicht implementiert, können aber ohne viel Aufwand selbst implementiert werden. Mit Hilfe

Scilab	MATLAB	Erklärung
`syslin('c',A,B,cT,<D>,<x0>)`	`ss(A,B,cT,D)`	Definition eines kont. Systems
`syslin('d',A,B,cT,<D>,<x0>)`	`ss(A,B,cT,D,Ta)`	Definition eines diskr. Systems
`s = poly(0,'s')`	`s = tf('s')`	Definition der Variable der ÜF
`s = %s`	`s = tf('s')`	Definition der Variable der ÜF
`[Ds,NUM,chi]=ss2tf(sys)`	`[num,den]=ss2tf(sys)`	Transformation vom Zustandsraum zur ÜF
`tf2ss(uf)`	`tf2ss(uf)`	Umrechnen der ÜF in den Zustandsraum
`dscr(syslin,Ta)`	`c2d(sys,Ta)`	Diskretisiert das Zustandsraummodell
`bode(syslin)`	`bode(sys)`	zeichnet das Bodediagramm von syslin
`nyquist(syslin)`	`nyquist(sys)`	zeichnet die Nyquistortskurve
`horner(P,x)`		Bilineartransformation, x für s in P
`cls2dls(syslin,Ta)`	`c2d(sys, Ta)`	Transformiert nach Tustin von q nach z
`[A,B,C,D]=abcd(sl)`	`ssdata(sys)`	Ließt die Systemmatrizen aus dem Modell
`y=csim(u,t,syslin)`		Simulation des kont. Systems mit Eingang u
`y=dsimul(syslin,u)`		Simulation des diskr. Systems mit Eingang u

Tabelle 3.11: Scilab LTI Befehle

der Funktionen `dsimul` und `csim` können Sprung oder Impulsantworten generiert werden. Die dazugehörende Befehle sind unten angegeben. Zu beachten ist, dass der Zeitvektor `t` vorher definiert werden muss.

```
y = csim('step',t,sys)
```

```
y = csim('imp',t,sys)
```

In Abbildung 3.9 ist eine Simulation mit den verschiedenen Arten von linearen Systemen angegeben. Zu beachten ist, dass das Steuersignal des Plots völlig unabhängig von den Steuersignalen für das Abtasten des zeitkontinuierlichen Signals ist.

3.3.2 Kontinuierliche LTI Systeme

Für die Simulation müssen die berechneten Matrizen von Scilab in Scicos importiert und mit Hilfe des Zustandsraummodells oder der Übertragungsfunktion simuliert werden.

3.3.2.1 Zustandsraummodell

In Abbildung 3.10 wurden verschiedene Abtastzeiten für die `Clocks` für die Ein- und Ausgangssignale gewählt, da das Ausgangssignal für eine gute Darstellung im Plot schneller abgetastet wird.

Initialisierung eines Zustandsraummodells Um ein kontinuierliches Zustandsraummodell zu initialisieren werden die in Tabelle 3.11 angegebenen Befehle verwendet, wobei `A`, b, `cT` und `d` laut Formel 3.1 die Systemmatrizen des kontinuierlichen Systems sind. Bei Bedarf kann zusätzlich der Anfangszustand mitübergeben werden.

```
sys_cont = syslin('c',A,b,cT,d)
```

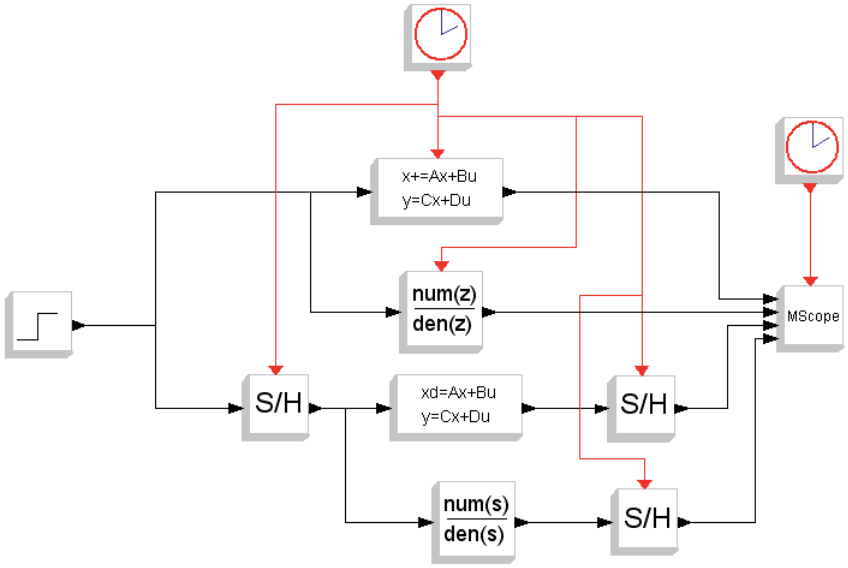


Abbildung 3.9: Aufbau verschiedener LTI Systeme

Batchsimulation Eine Batchsimulation, also eine aus Scilab gestartete Scicos Simulation, kann aus einer beliebigen `*.sce` Datei durchgeführt werden. Leider sind die Scicos Blöcke `From workspace` und `To workspace` noch nicht stabil und führen oft zu Fehlern. Aus diesem Grund wird von deren Gebrauch abgeraten und die Verwendung der Blöcke `write to output file` und `read from input file` empfohlen. An dieser Stelle sei jedoch erwähnt, dass nur das Schreiben von Spaltenvektoren (und nicht von Zeilenvektoren) in eine Datei einwandfrei funktioniert.

Zu beachten ist jedoch, dass die Variablen, die in Scicos benötigt werden, im `Context` definiert und mit dem Befehl `%scicos_context.var = var` an Scicos übergeben werden müssen. Zur Verdeutlichung ist eine `*.sce` Datei einer Batchsimulation angegeben.

```
chdir('D:\Projektseminar\...\Pendel');
//Laden der Init Datei
exec("D:\Projektseminar\Scilab\...\pendel_lin.sce");
// Löschen der vorherigen Datendatei
mdelete('input_pendel_lin')
// Erzeugen eines Eingangssignals
u = [zeros(0:.1:1) ones(1.1:0.1:10)]';
write('input_wp_lin',u);
x0 = [%pi/6,0];
// Laden der Pendel Simulation in Scicos
load pendel_lin_sim.cos
//Setzen der Contextvariablen in Scicos
%scicos_context.A = A;
%scicos_context.B = B;
%scicos_context.C = C;
%scicos_context.D = D;
%scicos_context.x0 = x0;
//Simulation des Pendelmodells in Scicos
scicos_simulate(scs_m, list(), %scicos_context);
```

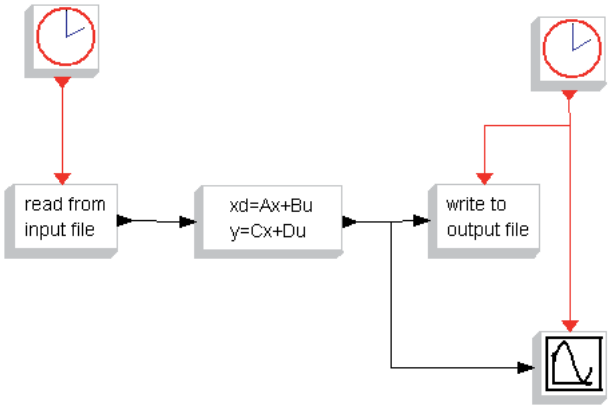


Abbildung 3.10: Modell des linearisierten Pendels

```
//Auslesen der Ergebnisdatei
data=read('output_pendel_lin',-1,2);
t = data(:,1);
y = data(:,2);
//Plotten der Ergebnisse
plot2d(t,y)
xtitle('Pendel Modell linearisiert um die untere Ruhelage');
xgrid(2)
```

3.3.2.2 Übertragungsfunktion

Im Gegensatz zu MATLAB gibt es in Scilab kein eigenes Transferfunktionsobjekt `tf`. Als Ersatz dafür dient ein Polynom der Variable `s` beziehungsweise `z`. Vor der Definition einer Übertragungsfunktion muss die Variable `s` bzw. `z` definiert werden. Dies geschieht mit einem der beiden folgenden Befehlen:

```
s=poly(0,'s')
```

```
s=%s
```

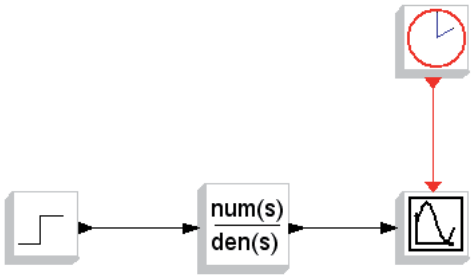


Abbildung 3.11: Zeitkontinuierliche Übertragungsfunktion

Mit dem Befehl `tf2ss` bzw. `ss2tf` können Zustandsraummodelle bzw. Übertragungsfunktionen in einander umgerechnet werden. Dabei muss jedoch beachten werden, dass bei Übertragungsfunktionen laut Definition kein Anfangszustand möglich ist. Die Syntax des Befehls `ss2tf` in Scilab unterscheidet sich zu MATLAB. In Scilab gibt der Befehl nicht Zähler und Nenner, sondern Werte in der Form `[Ds,NUM,chi]=ss2tf(sys)` für die Übertragungsfunktion `NUM/chi + Ds` zurück. Zusätzlich sind Zähler und Nenner im 2 bzw. 3 Element der Übertragungsfunktion gespeichert.

```
--> A = [0,1;0,1]; B = [0;2]; C = [1,0]; D = 0;
--> sys = syslin('c',A,B;C,D);
--> G=ss2tf(sys);
G =
     2
  -------
  -s + s^2
--> G(2)
  2
--> G(3)
  -s + s^2
```

q-Transformation Die q-Transformation ist in der Regelungstechnik mit T als Abtastzeit wie folgt definiert:

$$z = \frac{1 + q\frac{T}{2}}{1 - q\frac{T}{2}}$$
$$q = \frac{2}{T}\frac{z-1}{z+1} \tag{3.5}$$

Diese bilineare Transformation, ist in MATLAB mit dem Befehl `tustin` vorimplementiert. In Scilab ist die Transformation von `q` nach `z` mit dem Befehl `cls2dls` möglich. Für die Transformation von `z` nach `q` muss der Befehl `horner(Gz,x)` zu Hilfe genommen werden. Dieser ersetzt die Variable `z` in `Gz(z)` durch `x`. Damit ist die q Transformation von `z` nach `q` des Übertragungsfunktion P durch folgenden Befehl gegeben.

```
Gq = syslin('c',horner(Gz,(1+q*Ta/2)/(1-q*Ta/2)))
```

Als Beispiel dient im diesen Fall die Übertragungsfunktion

$$G(s) = \frac{1}{1+s}$$

Im folgenden Code wird aus Übertragungsfunktion im Laplacebereich die z-Übertragungsfunktion und anschließend mit Hilfe des `horner` Befehls und der Definition 3.5 die q-Übertragungsfunktion berechnet. Durch die anschließende Rücktransformation von $G^{\#}(q)$ in

den z-Bereich mit Hilfe des Befehls `cls2dls` kann die Transformation in den q-Bereich überprüft werden.

```
-->s = %s;
-->q = poly(0,'q');
-->z = %z;
-->Ta = 0.1;
-->Gs = syslin('c',1/(1+s));
-->Gz = ss2tf(dscr(Gs,Ta))
Gz =
      0.0951626
    -------------
   - 0.9048374 + z
-->Gq = syslin('c',horner(Gz,(1+q*Ta/2)/(1-q*Ta/2)))
Gq =
    0.0951626 - 0.0047581q
    ----------------------
    0.0951626 + 0.0952419q
-->Gz2 = ss2tf(cls2dls(tf2ss(Gq),Ta))
Gz2 =
    0.0951626 - 1.388D-17z
    ----------------------
       - 0.9048374 + z
```

Wie man an Hand der Variable `Gz2` im Vergleich zu `Gz` erkennt, erzeugt der Befehl `cls2dls` bis auf numerische Ungenauigkeiten die richtige Übertragungsfunktion. Zusätzlich stimmen die Ergebnisse mit den äquivalenten MATLAB Ergebnissen genau überein.

3.3.3 Diskrete LTI Systeme

In diesem Abschnitt wird speziell die Simulation von zeitdiskreten LTI Systemen behandelt.

3.3.3.1 Zustandsraummodell

Bei der Definition eines Zustandsraummodells mit Hilfe von `syslin` muss im Gegensatz zum zeitkontinuierlichen Modell der erste Parameter von `'c'` auf `'d'` oder auf die Abtastzeit `Ta` geändert werden. Im Beispiel unten erkennt man die Definition des Abtastsystems mit der Übertragungsfunktion

$$G(z) = \frac{z}{1+z}$$

mit einer Abtastzeit von 0.1 Sekunden.

```
-->z = %z;
-->Gz = z/(1+z);
-->Ta = 0.1;
-->sys = syslin(Ta,Gz);
```

Bei der Simulation von diskreten Systemen sind in Scicos einige Dinge zu berücksichtigen. Den Block *Discrete state-space system* findet man in der `Linear` Palette. Um Synchronität und ein korrektes Ergebnis zu gewährleisten müssen die Sample/Hold Blöcke und der Block des Zustandsraummodells mit der selben `Clock` und Abtastzeit angesteuert werden. Falls ein kontinuierliches Modell mit der Abtastzeit `Ta` diskretisiert oder ein diskretes Modell mit `Ta` initialisiert wurde, muss die `Clock` für die Ansteuerung der diskreten Blöcke eine Periodendauer von `Ta` besitzen. Die `Clock` für den Plot ist jedoch völlig unabhängig vom restlichen Modell.

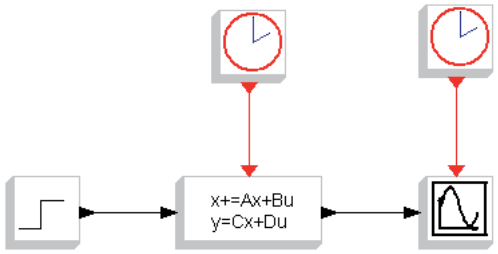


Abbildung 3.12: Zeitdiskretes Zustandsraummodell

Die Diskretisierung eines kontinuierlichen Modells ist in Scicos mit `dscr(syslin,Ta)` (Abtastzeit `Ta`) möglich. Damit können kontinuierliche Zustandsraummodelle in Diskrete umgewandelt werden.

```
-->sys= syslin('c',A,B,C,D);
-->Ta = 0.2;
-->sysd = dscr(sys,Ta);
```

3.3.3.2 Übertragungsfunktion

Äquivalent zum zeitkontinuierlichen Fall gibt es eine zeitdiskrete Übertragungsfunktion mit der Variable `z`. Die Ansteuerung des Blocks mit Hilfe der `Clock` erfolgt identisch zum diskreten Zustandsraummodell. Die Befehle zur Erzeugung von diskreten Übertragungsfunktionen entsprechen denen des zeitkontinuierlichen Falls.

```
-->Gz = ss2tf(sysd)
Gz =
      0.0457556 + 0.0428055z
    ----------------------------
    1.2214028 - 2.2214028z + z^2
-->z = %z;
-->Gz1 = z/(z+1);
```

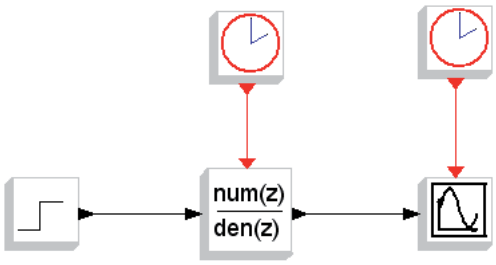


Abbildung 3.13: Zeitdiskrete Übertragungsfunktion

3.4 Simulation mit Hilfe des GENERIC Blocks

Im folgenden Kapitel steht die Erstellung von Blöcken mit Hilfe des GENERIC Blocks im Vordergrund. Dies ist die einfachste Methode zur schnellen Implementierung einzelner nichtlinearer Blöcke. Im Kapitel 3.5 wird in weiterer Folge auf die theoretischen Grundlagen von *interfacing* und *computational functions* eingegangen.

3.4.1 Einleitung

Für die Simulation nichtlinearer Systeme in Scicos gibt es verschiedene Möglichkeiten, die vom gewöhnlichen Blockschaltbild mit nichtlinearen Blöcken bis zur eigenen Implementierung von C oder Fortran Funktionen oder der automatisierten Generierung von C Funktionen aus Blockschaltbildern reichen. Die verschiedenen Möglichkeiten sind in Tabelle 3.12 zusammengefasst.

Jeder Basis Block in Scilab besteht aus zwei verschiedenen Funktionen, einer *computational function* und einer *interfacing function*. Erstere ist in C (Typ 4) oder in der Scilab eigenen Skriptsprache (Typ 5) programmiert und bestimmt das Verhalten des Blocks. Die *interfacing function* ist für die Interaktion zwischen Editor und *computational function* notwendig. Sie definiert den Blocktyp, die Anzahl der Ein- und Ausgänge und die Geometrie im Editor. Falls ein GENERIC Block, eine vorprogrammierte *interfacing function*, verwendet wird, entfällt jedoch die Implementierung einer eigenen *interfacing function*.

nichtlineare Blöcke	Modellaufbau mit nichtlinearen Blöcken
Scifunc	Programmierung von Scilab Funktionen
C Block	Programmierung von C Dateien
Fortran Block	Programmierung von Fortran Dateien

Tabelle 3.12: Möglichkeiten der Simulation nichtlinearer Systeme

3.4.2 Computational Function

Computational functions sind das Scilab-Pendant zu S-Functions in MATLAB und können u.a. in C oder in der Scilab eigenen Skriptsprache implementiert werden. Diese unterscheiden sich ausgenommen von der Syntax in der Rechengeschwindigkeit und der Kompilie-

rung. Eine *computational function* in der Scilab eigenen Skriptsprache muss, ähnlich zu einem m-File in MATLAB, nicht kompiliert werden, die C Funktion jedoch schon. Neben der Möglichkeit der Onlineprogrammierung in Scicos (siehe Kapitel 3.5.2.3), auf die hier nicht weiter eingegangen werden soll, wird die *computational function* in einem Editor oder im `Scipad` erstellt und in Scilab gegebenenfalls kompiliert und mit der Scilab Bibliothek verknüpft.

Computational functions werden ähnlich zu S-Functions mit Hilfe von Flags gesteuert. Die wichtigsten davon sind `flag=1` für die Berechnung des Ausgangs und `flag=0` für die Berechnung der Ableitung des Zustandes. Die wichtigsten Variablen der Funktion, die in der C Datei verwendet werden, sind in Tabelle 3.13 angegeben. In diesen werden die Zustände, Ausgänge, Parameter und vieles mehr des Basisblocks gespeichert. Für Details muss auf Kapitel 3.5 oder auf [2] verwiesen werden.

Name	Bedeutung
`int type`	Typ der *computational function*
`int nz`	Anzahl der diskreten Zustände
`int nx`	Anzahl der kontinuierlichen Zustände
`double *z`	Vektor der diskreten Zustände
`double *x`	Vektor der kontinuierlichen Zustände
`int nout`	Anzahl der Systemausgänge
`double **outptr`	Zeiger auf die Systemausgänge
`int *ipar`	Vektor der Integer Parameter
`double *nrpar`	reeller Parametervektor

Tabelle 3.13: Variablen der *computational function*

3.4.2.1 C Funktion

Für die Simulation in einer *computational function* in *Scilab/Scicos* sollte folgende Vorgehensweise eingehalten werden:

1. Erstellung der C Funktion (z.B. `progname`) in einem Editor und Abspeichern der Datei (z.B. `file.c`) mit Endung `'.c'` im Arbeitsverzeichnis von Scilab.

2. Kompilieren der C Datei innerhalb von Scilab mit Hilfe des Befehls

   ```
   -->ilib_for_link('progname','file.o',[],'c');
   ```

3. Verknüpfen der kompilierten Datei mit der Scilab Bibliothek.

   ```
   -->x=link('libprogname.dll', ['progname'],'c');
   ```

   oder

   ```
   -->exec('loader.sce');
   ```

4. Verknüpfen der *computational function* mit einer *interfacing function* oder einem GENERIC Block (siehe Kapitel 3.4.3).

5. Freigeben der kompilierten Datei von der Scilab Bibliothek (falls z.B. durch eine Änderung in der Funktion eine weitere Kompilierung der *computational function* notwendig ist).

```
-->ulink(x)
```

Als Beispiel dient an dieser Stelle die Funktion `pendel_c`, die das mathematische Pendel mit Feder und Dämpfer aus dem Kapitel 2.3 beschreibt. Zu beachten ist, dass die ersten beiden Zeilen der Funktion bei jeder C *computational function* benötigt werden.

```
#include <scicos/scicos_block.h>
#include <math.h>

void pendel_c(scicos_block *block, int flag) {
  if(flag==1){
    block->outptr[0][0]=block->x[0];
  }
  if(flag==0){
    double g=9.81;
    double m = block->rpar[0];
    double l = block->rpar[1];
    double c = block->rpar[2];
    double d = block->rpar[3];
    double phi=block->x[0]; //phi
    double omega=block->x[1]; //omega
    double M = block->inptr[0][0];

    block->xd[0]=block->x[1];
    block->xd[1]=1/m/l/l*(g*l*cos(phi)*m-c*phi-M-d*omega);
  }
}
```

3.4.2.2 Scilab Computational Functions

Für die in Scilab Skriptsprache programmierte *computational function* gilt im Vergleich zur C Funktion eine ähnliche Vorgehensweise. Auf die Kompilierung und die Verknüpfung mit der Bibliothek kann verzichtet werden, jedoch ist ein Laden der Funktion in Scilab notwendig. Folgende Reihenfolge ist bei der Erstellung einer Scilab *computational function* empfehlenswert:

1. Erstellung der Funktion im `Scipad` und Abspeichern mit Endung `'.sci'` im Arbeitsverzeichnis

2. Laden der Datei in Scilab, die die *computational function* beinhaltet

   ```
   --> exec('file.sci')
   ```

3. Verknüpfen der *computational function* mit einer *interfacing function* oder einem GENERIC Block (siehe Kapitel 3.4.3)

Auch hier dient das mathematische Pendel mit Feder und Dämpfer als Beispiel. Zu beachten ist, dass ähnlich zu m-Files in MATLAB die Indizes von Listen, Vektoren und Matrizen in der Scilab eigenen Skriptsprache im Gegensatz zu C mit `1` und nicht mit `0` beginnen.

```
function block=pendel_scilab(block, flag)
  if flag==1
    block.outptr(1)(1)=block.x(1)
  elseif flag==0
    g = 9.81;
    m = block.rpar(1);
    l = block.rpar(2);
    c = block.rpar(3);
    d = block.rpar(4);
    phi=block.x(1);
    omega=block.x(2);
    M = block.inptr(1)(1);
    block.xd(1)=omega;
    block.xd(2)=1/m/l/l*(g*l*cos(phi)*m-c*phi-M-d*omega)
  end
endfunction
```

3.4.3 Interfacing function

Die *interfacing function* dient der Darstellung im Scicos Editor und enthält wichtige Parameter, wie Anzahl und Größe der Ein- und Ausgänge. Zusätzlich beinhaltet sie den Namen, die Blockparameter und die Anfangsbedingungen der *computational function*. Für die Implementierung der *interfacing function* gibt es zwei grundlegende verschiedene Methoden.

- Programmierung einer *interfacing function*
- Verwendung des GENERIC Blocks

Im folgenden Abschnitt wird nur auf die Verwendung des GENERIC Blocks eingegangen. Für weitere Informationen zur *interfacing function* muss auf Kapitel 3.5 und auf [2] verwiesen werden.

GENERIC Block Der GENERIC Block ist ein in Scilabskriptsprache vorgefertigter Block, der die Implementierung einer *interfacing function* ersetzt und in der Scicos Palette `Others` zu finden ist. Damit zu Testzwecken nicht immer eine eigene *interfacing function* erzeugt werden muss, wird bei vielen Simulationen der GENERIC Block als *interfacing function* verwendet. Mit Hilfe des Benutzerinterfaces des GENERIC Blocks (siehe Abbildung 3.14) können die Block Parameter, die sonst innerhalb einer *interfacing function* definiert werden müssen, an die entsprechende *computational function* angepasst werden.

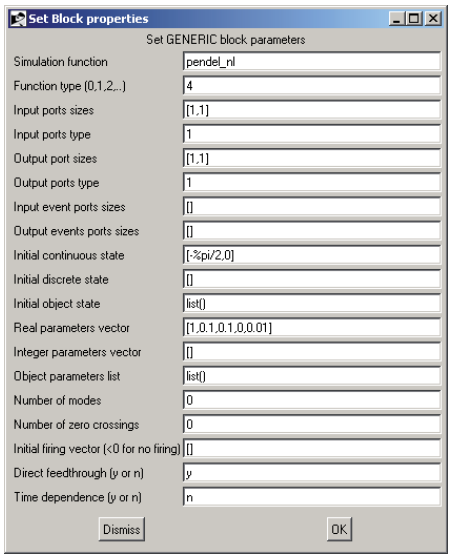


Abbildung 3.14: GENERIC Block Eigenschaften für die C Funktion des mathematischen Pendels

In Tabelle 3.14 sind die wichtigsten Parameter des GENERIC Blocks, die an die verwendete *computational function* angepasst werden müssen, erläutert. Die entsprechenden Blockparameter des GENERIC Blocks für das mathematische Pendel sind in Abbildung 3.14 ersichtlich.

Simulation Funktion	Name der C Funktion (nicht der Datei!)
Function type	4 für C Funktion, 5 für Scilab Funktion
Input ports size	Dimension des Eingangssignals
Input ports type	Typ des Eingangssignals (-1 für Vektoren)
Output ports sizes	Dimension des Ausgangssignals
Output ports type	Typ des Ausgangssignals (-1 für Vektoren)
Initial continuos state	Anfangsbedingung des kont. Zustands
Initial discrete state	Anfangsbedingung des diskreten Zustands
Real parameters vector	Vektor der reellen Parameter
Integer parameter vector	Vektor der ganzzahligen Parameter
Number of Modes	Anzahl der Simulationsmoden
Numbers of zero crossings	Anzahl der Nulldurchgänge

Tabelle 3.14: GENERIC Block Parameter

Abbildung 3.15 zeigt den beispielhaften Aufbau der Simulation des mathematischen Pendels mit einer C *computational function* und dem GENERIC Block.

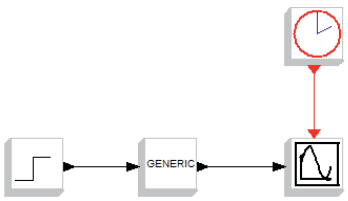


Abbildung 3.15: Aufbau der Simulation des mathematischen Pendels mit dem GENERIC Block

3.5 Theoretische Einführung in Basisblöcke

Im folgenden Kapitel sollen einige theoretische Grundlagen der Basisblöcke, die bereits im vorigen Kapitel erwähnt wurden, dargestellt werden. Bei Basisblöcken wird zwischen automatisiert und manuell erstellten Blöcken unterschieden. Dem Super Block, der durch eine automatisierte Codegenerierung eines Subsystems in eine C Datei oder standalone Funktion erzeugt wird, steht die On- oder Offline Programmierung einer C- oder Scilab Funktion gegenüber.

3.5.1 Super Blöcke

Wie schon in Kapitel 3.2.3 erwähnt, kann man lineare und nichtlineare Blöcke aus einem Scicos Modell durch den Befehl `Region-to-Super-block` in einen Super Block auslagern. Dieser kann in einem eigenen Scicos Editor geöffnet werden und sowohl als Super Block als auch als Palette gespeichert werden. Durch diese Vorgehensweise können Blöcke mit beliebigen Inhalten zusammengefasst werden.

In Scicos gibt es die Möglichkeit den C Code eines Super Blocks automatisiert zu generieren. Dafür genügt es, im Objekt Menü des Super Blocks den Befehl `Code Generation` auszuwählen. Daraufhin wird eine Dialog Box geöffnet um den Namen (z.B. SuperBlock) und den Pfad, in dem die C Datei gespeichert werden sollen, einzugeben. Die gesamte mathematische Beschreibung findet man in der Datei `Superblock.c`. Diese Datei ist bereits kompiliert und mit Scilab verknüpft. Des weiteren werden einige Dateien, die für die Erstellung einer standalone Anwendung benötigt werden, bei der automatisierten C Code Generierung erzeugt.

Bei dieser Generierung von C Dateien bestehen jedoch Grenzen hinsichtlich der Komplexität der Modelle. Sowohl das Zusammenspiel zwischen kontinuierlichen und diskreten Systemen als auch die Synchronität zwischen einzelnen Blöcken bereitet Scilab noch einige Probleme, die jedoch laut Scilab Homepage in den zukünftigen Versionen behoben werden sollen. Zusätzlich muss beachtet werden, dass im Betriebssystem ein C Compiler installiert sein muss, da sonst die Kompilierung von C Dateien nicht möglich ist. In Linux,

Unix und MacOSX ist dieser bereits vorinstalliert. Für weitere Informationen zu diesem Thema wird auf Kapitel 12 *Code Generation* in [2] verwiesen.

3.5.2 Basisblöcke

3.5.2.1 Einführung

Jeder Basis Block in Scilab besteht aus zwei verschiedenen Funktionen. Die erste nennt man *computational function* und ist meistens in C (Typ 4) oder in der Scilab eigenen Skriptsprache (Typ 5) programmiert. Diese Funktion bestimmt das Verhalten des Blocks und wird mit Hilfe von Eingangsflags gesteuert.

Die zweite Funktion, die jeder Basis Block beinhalten muss, ist die in Scilab Skriptsprache programmierten *interfacing function*. Diese ist für die Interaktion zwischen Editor und Funktion notwendig. Sie definiert den Blocktyp, die Anzahl der Ein- und Ausgänge und die Geometrie im Editor. Zusätzlich ist sie verantwortlich für die Kommunikation zwischen dem Editor und dem Benutzer (zum Beispiel eine Benutzeroberfläche für die Eingabe der Blockparameter oder Anfangsparameter).

interfacing function	Datei in Scilab Skriptsprache (Typ 5), muss mit einer
	computational function verknüpft werden
GENERIC Block	vorgefertigte *interfacing function* in Blockform für die
	direkte Verknüpfung mit einer *computational function*

Tabelle 3.15: Möglichkeiten der *interfacing function*

Die *computational function* kann auf verschiedene Arten (siehe Tabelle 3.16) erzeugt werden. Die Funktionen unterscheiden sich in Programmiersprache und Entwicklungsumgebung. Die Scilab *computational function* wird in der Scilab eigenen Skriptsprache erstellt und muss nicht kompiliert werden, sondern nur noch in Scicos geladen werden (`getf`). Die C *computational function* kann einerseits offline in einem Editor und andererseits direkt in Scicos (CBlock2) implementiert werden. Diese weisen im Gegensatz zur Scilab Funktion eine höhere Rechengeschwindigkeit auf, müssen jedoch kompiliert werden und die extern programmierte C Funktion mit der Scilab Bibliothek und einer *interfacing function* verknüpft werden. Die Grundstruktur dieser *computational functions* ist durch die Flagsteuerung vorgegeben.

Die *interfacing function* kann entweder selbst erstellt werden oder es kann auf eine vorhandene Vorlage (GENERIC Block) zurückgegriffen werden. Die zweite Alternative ist für das Testen von *computational functions* sehr empfehlenswert (siehe Kapitel 3.4). Die weiteren Kapitel behandeln die Vorgehensweise zur Erstellung von Basic Blocks (*interfacing function*, *computational function*, Kompilierung, Simulation)

3.5.2.2 Interfacing Function

Wie schon weiter oben erwähnt, gibt es die Möglichkeit die *interfacing function* selbst zu programmieren und mit einer *computational function* zu verknüpfen oder mit Hilfe

C	ausprogrammierte C Datei (Typ 4), muss kompiliert und
	mit Scicos und *interfacing function* verknüpft werden
Scilab	Datei in der Scilab Skriptsprache (Typ 5), muss mit `getf`
	geladen und einer *interfacing function* verknüpft werden
CBlock2	online programmierter C Code innerhalb Scicos,
	muss innerhalb Scicos kompiliert werden
Code Generation	automatisiertes Generieren eines Blocks inklusive
	eines C Codes aus einem Super Block

Tabelle 3.16: Möglichkeiten der *computational function*

eines GENERIC Blocks auf eine vorprogrammierte Vorlage einer *interfacing function* zurückzugreifen. Für weitere Informationen zur GENERIC Funktion wird auf Kapitel 3.4.3 verwiesen.

Die *interfacing function* an sich bestimmt die geometrische Darstellung des Blocks im Scicos Editor. In dieser Funktion werden Parameter, wie die Anzahl der Ein- und Ausgänge, die Größe, die äußere Erscheinung und einige mehr definiert. Zusätzlich beinhaltet sie den Namen der *computational function*, die Block Parameter und Anfangsbedingungen. Dies kann auch am Beispiel des `GENERIC` Blocks in Abbildung 3.14 erkannt werden. Der Eingangswert `job` bestimmt dabei die Aufgabe der *interfacing Function.*

```
[x,y,typ] = block(job,arg1,arg2)
```

Der Parameter `job` kann folgende Werte annehmen:

- '`plot`': Die Funktion zeichnet den Block und die Beschriftung. `arg1` ist die Datenstruktur des Blocks. Die weiteren Parameter sind unbenutzt. Für eine Standardzeichnung kann die Funktion `standard_inputs` verwendet werden.

- '`getinputs`': Die Funktion gibt die Position und den Typ der Eingänge zurück. `arg1` ist die Datenstruktur des Blocks. `x` und `y` sind die jeweiligen Koordinaten der Eingänge. `typ` ist ein Vektor der Eingangstypen. Für Standardeingänge (Signaleingänge links und Steuereingänge oben) kann die vordefinierte Funktion `standard_input` verwendet werden.

- '`getoutputs`': Die Funktion gibt die Position und den Typ der Ausgänge zurück. `arg1` ist die Datenstruktur des Blocks. `x` und `y` sind die jeweiligen Koordinaten der Ausgänge. `typ` ist ein Vektor der Ausgangstypen. Für Standardausgänge (Signalausgänge rechts und Steuerausgänge unten) kann die vordefinierte Funktion `standard_output` verwendet werden.

- '`getorigin`': Die Funktion gibt den Ursprung der linken unteren Ecke des Rechtecks, das die Blockform beinhaltet, zurück. `arg1` ist die Datenstruktur des Blocks und `x` und `y` sind die jeweiligen Koordinaten der Ausgänge. Für Standardanwendungen kann die vordefinierte Funktion `standard_origin` verwendet werden.

- 'set': Die Funktion öffnet ein Fenster für die Definition der Block Parameter. Hier kann die Programmierung einer graphischen Oberfläche für die Eingabe sämtlicher Blockparameter erfolgen. `arg1` ist die Datenstruktur und `x` ist die neue Datenstruktur des Blocks.

- 'define': Die Funktion initialisiert den Block. Sie setzt die Anzahl und Typen der Ein- und Ausgänge fest. Im Parameter `x` wird die Datenstruktur mitübergeben.

Für weitere und genauere Details zur Struktur und zu den Befehlen von *interfacing functions* ist im Anhang ein Beispiel angegeben.

Programmieren einer Interfacing Function Wie schon erwähnt, wird die *interfacing function* in der Scilabskriptsprache programmiert und mit den Eingangsparameter `job` gesteuert. Die wichtigsten Werte davon sind `'set'` und `'define'`. Bei `'set'` wird in der *interfacing function* eine Benutzeroberfläche für die Einstellung der Blockparameter definiert. Bei dem Wert `'define'` wird der Block initialisiert, die Anzahl der Ein- und Ausgänge festgelegt und die Anfangswerte definiert. Zusätzlich wird die graphische Darstellung des Blocks im Ausgangsparameter `x` definiert. Für eine genauere Erläuterung muss auf Kapitel 9.5 und A.1 von [2] verwiesen werden.

Als Beispiel werden Ausschnitte einer möglichen *interfacing function* für das Pendel Beispiel angegeben. Für eine Implementierung einer ähnlichen *interfacing function* sind folgende Werte zu ändern:

- für `'set'`

  – Anpassen der Benutzeroberfläche:

  ```
  [ok,m,l,c,d,exprs]=..
    getvalue('Set block parameters',..
         ['Masse des Pendels',..
         'Länge des Stabes',..
         'Federsteifigkeit',..
         'Reibung im Drehgelenk'],..
         list('vec',1,'vec',1,'vec',1,'vec',1),exprs);
  ```

  Dieser Block muss hinsichtlich der Anzahl und Namen der Parameter verändert werden. Die Länge des Vektors `list` muss ebenfalls auf die Anzahl der Parameter angepasst werden.

  – Übergabe der Blockparameter

  ```
  model.rpar=[m,l,c,d]
  model.ipar=[]
  ```

  Die Werte, die in der Benutzeroberfläche eingegeben wurden (siehe oben), müssen nun den reellen Parametern und den Integerparametern der *computational function* zugewiesen werden.

- für `'define'`
  - Zuweisung einer *computational function*

    ```
    model.sim=list('pendel_scilab',5);
    ```

    Dem `sim`-Objekt des Modells muss der Name und der Typ der *computational function* zugewiesen werden. Dabei ist zu beachten, dass nicht der Name der Datei, sondern der Name der Funktion verlangt wird. Zusätzlich muss bei einer C *computational function* (Typ 4) die Datei kompiliert und mit der Scilab Bibliothek verknüpft sein. Bei einer Scilab *computational function* (Typ 5) muss diese in Scilab mit `getf()` oder `exec()` geladen werden.
  - Bestimmung der Blockeigenschaften

    ```
    model.in=[1];          //Größe des Signalseingangs
    model.evtin=[];        //Größe des Steuerungseingangs
    model.out=[2,1];       //Größe des Ausgangs
    model.nzcross=0;       //Anzahl der Nulldurchgangsebenen
    model.nmode=0;         //Anzahl der Moden
    model.blocktype='c';   //kontinuierlich: 'c', diskret: 'd'
    model.dep_ut=[%t %t]; //stet. Abh. des Ausgangs vom
    //Eingang und Zeit (%t oder %f)
    model.rpar=[m,l,c,d]; //Übergabe der reellen Parameter
    model.ipar=[];         //Übergabe der Intergerparameter
    ```

  - Bestimmung des Anfangszustands $\mathbf{x}_0$

    ```
    model.state=[%pi/2;0];
    ```

  - Anpassung der Blockdarstellung

    ```
    gr_i=['txt=['math. Pendel'];';..
    'xstringb(orig(1),orig(2),txt,sz(1),sz(2),■fill■)'];
    ```

Um in Scicos einen neuen Block mit einer eigenen *interfacing* und *computational function* hinzuzufügen, muss man im Scicos Menü `'Edit'` den Button `'Add new block'` wählen (dazu Abbildung 3.16). Bevor der Block erscheint, muss der Funktionsname (nicht der Dateiname!) der *interfacing function* eingegeben werden. An dieser Stelle sollte nicht vergessen werden, dass zuvor die Datei in Scilab mit `getf()` geladen werden muss! Bei jeglichen Änderungen der *interfacing function* muss diese neu geladen und der zugehörige Block neu eingefügt werden.

Weitere Informationen zu Aufbau und Struktur des Scicos Editors gibt es im Anhang A von [2].

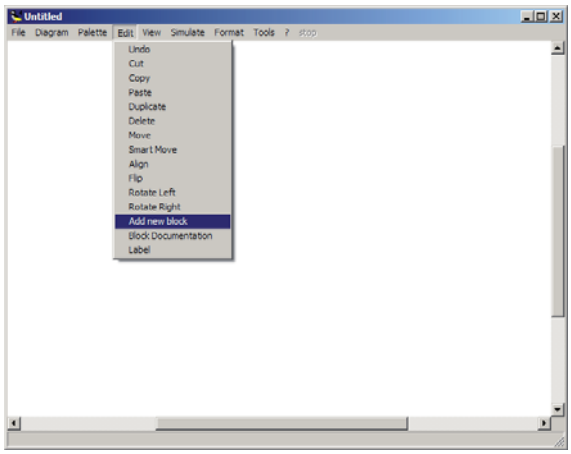


Abbildung 3.16: Hinzufügen eines neuen Blocks

3.5.2.3 Computational Function

Die *computational function* bildet das Herzstück einer Simulation. Sie wird immer wieder mit verschiedenen Parametern aufgerufen und bestimmt damit das Verhalten des Blocks. Sie kann sowohl in C als auch in Scilab programmiert sein. Um dies zu unterscheiden muss der `typ` der Funktion in den Parametern der *interfacing function* oder des `GENERIC` Blocks auf 4 für die C Funktion und 5 für die Scilab Funktion gesetzt werden. Die Steuerung der *computational function* erfolgt mittels Flags, die als Eingangsparameter übergeben werden. In Tabelle 3.17 sind die Bedeutungen der einzelnen Flags zusammengefasst.

flag	input	output	Beschreibung
0	`t,nervprt,x,z,inptr, mode,phase`	`xd`	berechnet die zeitliche Ableitung des Zustandes
1	`t,nervprt,x,z,inptr, mode,phase`	`outptr`	berechnet den Ausgang des Blocks
2	`t,nervprt>0,x,z,inptr`	`x, z`	Update der Zustände auf Grund externer Aktivierung
2	`t, nervprt=-1, x, z, inptr, jroot`	`x, z`	Update der Zustände auf Grund Nulldurchgänge
3	`t, x, z, inptr, jroot`	`evout`	Zeitverzögerung
4	`t, x, z`	`x,z,outptr`	Initialisierungen
5	`x, z, inptr`	`x,z,outptr`	Beenden der Simulation
6			nicht benötigt
7			für intern, implizite Blöcke
9	`t,phase=1,nevprt,x,z inptr`	`g, mode`	Berechnen von Nulldurchgängen und Festlegen von Moden
9	`t,phase=2,nevprt,x,z inptr`	`g`	Berechnen von Nulldurchgängen

Tabelle 3.17: Flags der *computational function*

Flags Der Eingangsparameter `flag` bestimmt welche Aktion die *computational function* durchzuführen hat. Es gibt Flags mit den Nummern 0 bis 9, wobei 6 unbenutzt ist. Im folgenden Absatz werden die wichtigsten Funktionen in chronologischer Reihenfolge genauer spezifiziert:

- **Initialisierung**: Um den Zustand eines Blocks zu initialisieren, wird die *computational function* mit dem `flag=4` aufgerufen. Daraufhin wird der diskrete und kontinuierliche Zustand initialisiert bzw. reinitialisiert, falls er schon in der *interfacing function* initialisiert wurde. Zusätzlich kann auch der Ausgang des Blocks initialisiert werden. Dieses ist zum Beispiel bei der Funktion `plot` (Öffnen des Grafikfensters) oder `read` bzw. `write` (Öffnen oder Erzeugen der Datei) notwendig.

- **Output Update**: Wenn die Funktion mit `flag=1` aufgerufen wird, gibt der Block den Ausgang zurück, welcher eine Funktion des Eingangs, Zustands und der Aktivierungsquelle sein kann.

- **Zustands Update**: Bei dem Funktionsaufruf mit `flag=2`, wird der Zustand aktualisiert. Diese Aktualisierung kann zur Folge von Nulldurchgangsebenen geschehen, die mittels `flag=9` berechnet werden. Bei einfachen Programmen ist dies, ähnlich zu den verschiedenen Moden, nicht notwendig. Für weitere Informationen sei hier auf das Kapitel 9.5.2 in [2] verwiesen.

- **Integration**: Bei `flag=0` wird die Ableitung des Zustands `x` berechnet. Dies ist natürlich nur für kontinuierliche Systeme möglich.

- **Moden und Nullstellen**: Der Scicos Solver verlangt stetig differenzierbare Modelle. Dies bedeutet, dass das Verhalten der Blöcke mit deren Zustände zumindest stückweise stetig differenzierbar sein muss. Mit der Einführung von Moden kann der Solver diese Unstetigkeiten lösen, wenn man damit die Funktion in stetig differenzierbare Abschnitte teilt. Als typisches Beispiel dient der Absolutbetrag, der im Punkt null nicht stetig differenzierbar ist. Um das Problem der Unstetigkeit zu lösen, werden in der *computational function* zwei Moden implementiert - eine für den positiven und eine für den negativen Teil.
  Mit Hilfe von `flag=9` werden Nulldurchgangsebenen berechnet. Bei Auftreten solcher Nulldurchgänge werden die Zustände mit Hilfe von `flag=2` aktualisiert, um zu vermeiden, dass der Solver möglicherweise in die falsche Richtung driftet.

- **Ende**: Am Ende der Simulation oder durch Betätigen der `End` Taste in Scicos, ruft die Simulation jede *computational function* innerhalb des Scicos Modells mit `flag=5` auf. Oft ist es nützlich die geöffneten Dateien zu schließen und eventuell benützten Speicher frei zu geben.

C Computational Functions Wie schon zu Beginn von Kapitel 3.5.2.3 erwähnt, ist es möglich *computational functions* in C oder in Scilab zu programmieren. Abgesehen von der Programmiersprache ist der große Unterschied zwischen den beiden Funktionen die

Rechenzeit. Die Geschwindigkeit der C Funktion liegt deutlich über der der Scilab Funktion. Die Flagstruktur mit deren Ein- und Ausgangsparameter, die im vorigen Abschnitt vorgestellt wurden, sind für die beiden Programmiersprachen identisch.

Abgesehen von den Ein- und Ausgangsparametern, mit denen die *computational functions* aufgerufen werden, stehen für die Programmierung weitere Variablen zur Verfügung. Diese sind einerseits die Variablen der Blockstruktur und andererseits spezielle externe Funktionen, die innerhalb einer *computational function* aufgerufen werden können (siehe Tabelle 3.18). Die Block- struktur des C Blocks ist wie folgt definiert:

```
typedef struct {
  int nevprt; /*Integer, der die Aktivation Port Nummer angibt, der den Block aktiviert*/
  voidg funpt; /*Pointer zur computational Function*/
  int type; /*Type der computational function*/
  int scsptr; /*für C interfacing functions nicht benötigt*/
  int nz; /*Anzahl der diskreten Zustände*/
  double *z; /*Vektor der diskreten Zustände*/
  int noz; /*Anzahl der diskreten Objekt Zustände*/
  int *ozsz; /*Array der Dim. noz,2 der die Dim. der diskreten Objektmatrizen beinhaltet */
  int *oztyp; /*Array der die Matrizentypen der diskr. Objekttypen beinhaltet*/
  void **ozptr; /*Zeigerarray, die auf die Objektzustände zeigen*/
  int nx; /*Anzahl der kontinuierlichen Zustände*/
  double *x; /*Vektor der kontinuierlichen Zustände*/
  double *xd; /*Vektor der zeitlichen Ableitung des kontinuierlichen Zustands*/
  double *res; /*für implizite Blockdarstellung*/
  int nin; /*Anzahl der Signaleingänge*/
  int *insz; /*Array der Länge 3*nin,1 der die Dim. und Typen der Eingänge angibt*/
  void **inptr; /*Zeiger auf die Eingänge*/
  int nout; /*Anzahl der Signalsausgänge*/
  int *outsz; /*Array der Länge 3*nout,1 der die Dim. und Typen der Ausgänge angibt*/
  void **outptr; /*Zeiger auf die Ausgänge*/
  int nevout; /*Anzahl der Steuerungsausgänge an*/
  double *evout; /*bestimmt die Zeitverzögerung der Steuerungsausgänge*/
  int nrpar; /*Anzahl der reellen Parameter an*/
  double *rpar; /*reeller Parametervektor*/
  int nipar; /*Anzahl der Integer Parameter*/
  int *ipar; /*Integer Parametervektor*/
  int nopar; /*Anzahl der Objektparameter*/
  int *oparsz; /*Array der die Dim. der Obejektparametermatrizen beinhaltet*/
  int *opartyp; /*Array der Länge nopar, der die Matrizentypen beinhaltet*/
  void **oparptr; /*Array von Pointern zu den Objektparametern*/
  int ng; /*Integer, der die Anzahl von Nulldurchgängen angibt*/
  double *g; /*Nulldurchgänge*/
  int ztyp; /*Bool'sches Objekt, ob der Block Nulldurchgänge enthaltet*/
  int *jroot; /*Vektor, der die Anwesendheit und Richtung der Nulldurchgänge bestimmt*/
  char *label; /*Block label*/
  void **work; /*Pointer zum Workspace*/
  int nmode; /*Anzahl der Moden*/
  int *mode; /*Modenvektor*/
} scicos_block;
```

Für weitere Informationen sei auf die Seite

`http://www.scicos.org/Help/eng/scicos/C_struct.htm`

aus [13] verwiesen. Auf dieser Seite sind im Detail die einzelnen Parameter und Bedeutungen der Typen und Nummern der Strukturelemente angegeben:

`double get_scicos_time()`	gibt die aktuelle Zeit zurück
`int get_phase_simulation()`	gibt die aktuelle Simulationsphse zurück
`int get_block_number()`	gibt die Blocknummer zurück
`void set_block_error()`	gibt dem Simulator eine Errormeldung
`void do_cold_restart()`	Iniziert einen Neustart des Solvers
`set *scicos_mallox(size_t)`	stellt Speicher für den Workspace bereit
`void scicos_free(void *p)`	gibt reservierten Speicher frei

Tabelle 3.18: Zusätzliche Variablen für C *computational functions*

Für das in Kapitel 2.3 vorgestellte Pendel mit Feder und Dämpfer ergibt dies folgende C *computational function*.

```
#include <scicos/scicos_block.h>
#include <math.h>

void pendel_c(scicos_block *block, int flag) {
  if(flag==1){
    block->outptr[0][0]=block->x[0];
  }
  if(flag==0){
    double g = 9.81;
    double m = block->rpar[0];
    double l = block->rpar[1];
    double c = block->rpar[2];
    double d = block->rpar[3];
    double phi=block->x[0]; //phi
    double omega=block->x[1]; //omega
    double M = block->inptr[0][0];

    block->xd[0]=block->x[1];
    block->xd[1]=1/m/l/l*(g*l*cos(phi)*m-c*phi-M-d*omega);
  }
}
```

Scilab Computational Functions In dieser Funktion kann man, ähnlich zur C *computational function*, sowohl die mitübergebenen Parameter als auch die externen Funktionen (siehe Tabelle 3.20) und die Variablen der Blockstruktur verwenden. Die Variablen der Blockstruktur sind in Tabelle 3.19 angegeben. Deren Bedeutungen entsprechen den äquivalenten Variablen der C Funktionen. Ein großer Unterschied zwischen den beiden Programmiersprachen sind die Indizes von Vektoren. In C beginnen diese mit 0 und in der Scilab Skriptsprache mit 1. Die Syntax der Scilab *computational function* lautet

```
function block=func_name(block,flag)
```

scicos_block	nevprt	funpt	type	scsptr	nz	z	nx
x	xd	res	nin	insz	inptr	nout	outsz
outptr	nevout	evout	nrpar	rpar	nipar	ipar	ng
g	ztyp	jroot	label	work	nmode	mode	

Tabelle 3.19: Block Struktur einer Scilab *computational function*

curblock()	gibt die aktuelle Blocknummer zurück
scicos_time()	gibt die aktuelle Uhrzeit zurück
phase_simulation()	gibt die aktuelle Simulationsphase zurück
set_blockerror(i)	setzt den Error Flag
pointer_xproperty	für interne implizite Blocks

Tabelle 3.20: Externe Funktionen der Scilab *computational function*

Zusätzlich ist zu beachten, dass der Typ der Scilab *computational function* 5 ist. Dies muss in der *interfacing function* bzw. in den Parametern des GENERIC Blocks angegeben werden. Während einer Simulation darf sich die Anzahl der Zustände eines Blocks nicht ändern. Die Flags haben dieselbe Bedeutung wie im Falle der C Programmierung.

Für weitere Informationen zur Struktur von Scilab *computational functions* sei auf die Scicos Hilfe [13] mit unten angeführtem Link verwiesen.

`http://www.scicos.org/Help/eng/scicos/sci_struct.htm`

Für das Beispiel Pendel mit Feder und Dämpfer ergibt dies folgende Scilab *computational function*:

```
function block=pendel_scilab(block, flag)
  if flag==1
    block.outptr(1)(1)=block.x(1)
  elseif flag==0
    g = 9.81;
    m = block.rpar(1);
    l = block.rpar(2);
    c = block.rpar(3);
    d = block.rpar(4);
    phi=block.x(1);
    omega=block.x(2);
    M = block.inptr(1)(1);
    block.xd(1)=omega;
    block.xd(2)=1/m/l/l*(g*l*cos(phi)*m-c*phi-M-d*omega)
  end
endfunction
```

Online Programmierung Neben der Offlineprogrammierung einer C *computational function* in einem Editor, ist es in Scicos möglich online eine Funktion zu programmieren. Hierfür stehen folgende Blöcke in Scicos zur Verfügung:

C Block	Fortran
Scifunc	C Block2

Wie man in Abbildung 3.17 erkennen kann, muss der Code, hier eines C Blocks, direkt in Scicos eingegeben werden. Damit wird der Programmcode innerhalb dieser Scicosdatei gespeichert. Bei allen Möglichkeiten der Onlineprogrammierung ist keine *interfacing function* notwendig, da die *computational function* direkt in einen Block in Scicos geschrieben wird. Vor der Simulation eines CBlocks, CBlocks2 oder eines Fortran Blocks ist eine Kompilierung notwendig. Dazu muss natürlich im Betriebssystem ein passender Compiler installiert sein. Die Kompilierung erfolgt durch den Button '`Compile`' im Menü '`Simulate`' direkt in Scicos. Das Anhängen an die Bibliothek entfällt, ebenso wie auch die *interfacing function* bei der Onlineprogrammierung.

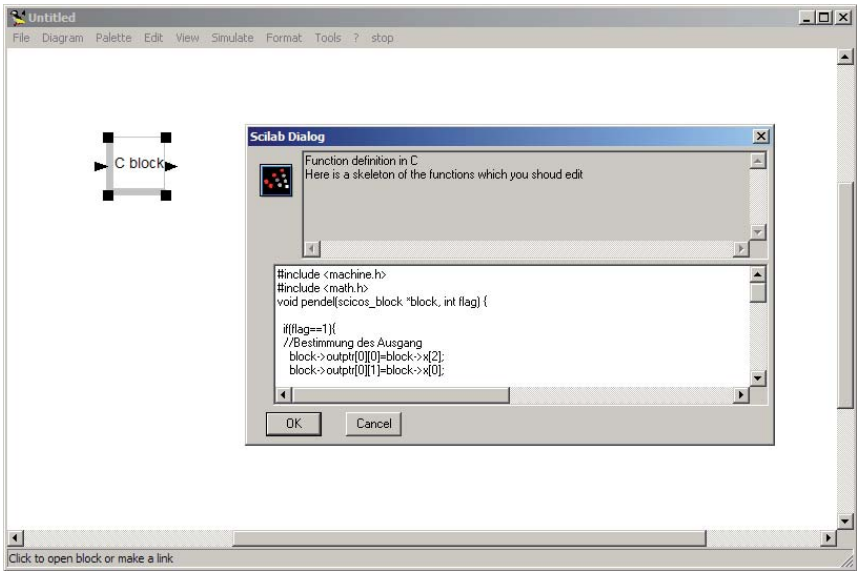


Abbildung 3.17: Programmierung eines C Blocks in Scicos

Im Falle des Pendel Beispiels wird für die Simulation der Inhalt einer *computational function* in das Scilab Dialog Fenster (siehe Abbildung 3.17) des Onlineprogrammierungs Blocks kopiert und daraufhin mit Hilfe des '`Compile`' Buttons im Simulationsmenü kompiliert. Davor müssen jedoch einige Parameter, wie in Abbildung 3.18 dargestellt, im C Block Fenster eingegeben werden. Darauf folgend sollte einer Simulation mit Hilfe des Buttons '`run`' funktionieren.

3.5.2.4 Kompilierung

Um einen neuen Block in eine Simulation einzufügen, ist sowohl eine *computational* als auch eine *interfacing function* notwendig. Im Falle einer in C programmierten Funktion ist

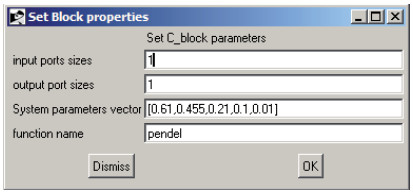


Abbildung 3.18: C Block Eigenschaften

jedoch eine Kompilierung der C Datei notwendig. Zusätzlich muss die Funktion mit Scilab verlinkt werden. Diese Kompilierung kann überaus zu Problemen führen und ist weder in Linux noch in Windows stabil. Windows benötigt einen eigenen C Compiler, wobei Visual Studio Express 2005 für Scilab 4.1.2 oder mingw 5.1.4 empfehlenswert ist. Zusätzlich können Fehler bei der Kompilierung auftreten, wenn die Windows Server Plattform nicht installiert ist (empfehlenswert ist Plattform SDK 2003 R2). In Linux, Unix oder MacOSX ist der Compiler schon inkludiert.

Für die Kompilierung der *computational function* `progname` in der C Datei `file.c` sind in Scilab folgende Befehle notwendig:

```
-->ilib_for_link('progname','file.o',[],'c') ;
-->exec loader.sce
```

Mehrmaliges Verknüpfen von Dateien zur Scilab Bibliothek führt zu Fehlern, da die von Scilab benützte `.dll` (oder dem Betriebssystem äquivalente) Datei vom Betriebssystem nicht überschrieben werden kann. Aus diesem Grund muss hier eine andere Vorgehensweise gewählt werden. Vor dem Verknüpfen der neuen Datei muss Scilab die alte Datei freigegeben, falls ein Neustart von Scilab vermieden werden möchte.

```
// Kompilieren von file.o inklusive dem Programm progname
-->ilib_for_link('progname','file.o',[],'c');

// Verknüpfen des Programms progname mit Scilab
-->x=link('libprogname.dll', ['progname'],'c');

// Freigeben der alten .dll Datei
-->ulink(x)
```

Für Scilab *computational functions* ist keine Kompilierung notwendig. Anstatt dessen muss jedoch die Funktion mit Scilab verlinkt werden. Dies geschieht mit dem üblichen Befehl

```
exec('file.sci')
```

Zusätzlich ist eine Änderung des Funktionstyps im GENERIC Block von 4 auf 5 erforderlich.

3.5.2.5 Simulation

In der Simulation von Abbildung 3.19 wird das nichtlineare Modell mit dem um die Ruhelage linearisierten Modell verglichen. Zu beachten ist, dass beim linearisierten Modell eine Arbeitspunktaufschaltung notwendig ist. Die nichtlineare *computational function* wurde sowohl in der Scilab Skriptsprache als auch in C implementiert.

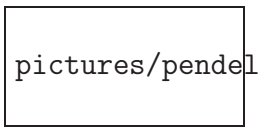


Abbildung 3.19: Schaltbild des nichtlinearen und linearisierten Pendels

In Abbildung 3.20 ist das Simulationsergebnis für eine Anfangsauslenkung aus der Ruhelage dargestellt. Der Unterschied zwischen dem linearisierten Modell und den nichtlinearen Modellen ist deutlich erkennbar, wobei die rote Kurve die Antwort der nichtlinearen Modelle zeigt, welche auf Strichstärke übereinstimmen.

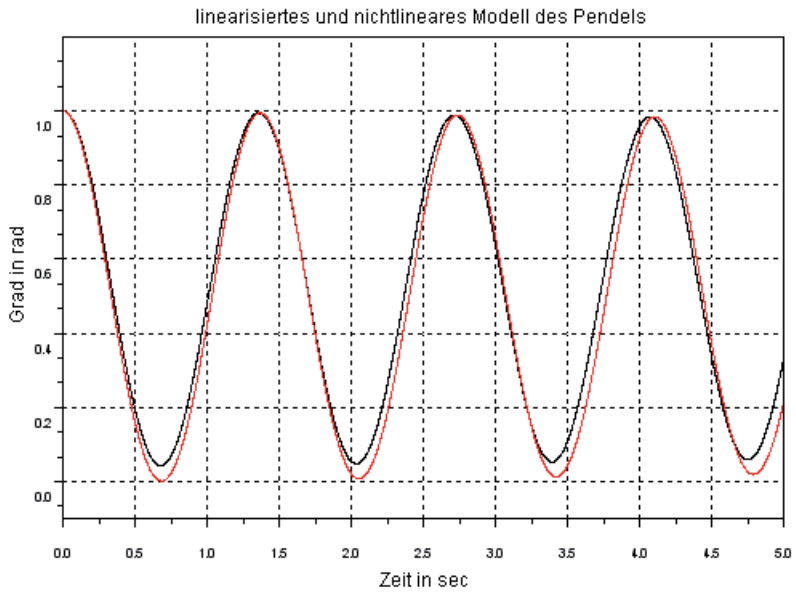


Abbildung 3.20: Simulationsergebnis des nichtlinearen und linearisierten Pendels

Kapitel 4

Praktikum 1

Das erste Praktikum widmet sich der Modellbildung und Simulation von dynamischen Systemen unter Verwendung der Open-Source Anwendungen MAXIMA und SCILAB/SCICOS.

4.1 Mathematisches Pendel

Aufgabe 4.1 *Lösen Sie die folgenden Aufgaben mit Hilfe des Computer-Algebra Programms* MAXIMA*:*

1. *Implementieren Sie die Modellbildung des mathematischen Pendels mit Feder und Dämpfer aus Abschnitt 2.3. Verwenden Sie im Gegensatz zu den Unterlagen eine nichtlineare Drehfeder mit der Federkraft* $F_F = -c\varphi^3\cosh(\varphi^2)$.

2. *Schreiben Sie eine* MAXIMA*-Prozedur* `Fibonacci` *zur Berechnung der ersten n Glieder der Fibonacci-Folge.*

3. *Schreiben Sie eine* MAXIMA*-Prozedur* `Jacobi` *zur Berechnung der Jacobi-Matrix, Eingangsparameter sind der Vektor der abzuleitenden Funktionen v und die Liste mit den Variablen vars, nach denen abgeleitet wird. Die vorgefertigte Prozedur in* MAXIMA *darf nicht verwendet werden.*

4. *Schreiben Sie eine* MAXIMA*-Prozedur* `Lagrange` *zur Berechnung der Bewegungsgleichungen auf Basis der Lagrange Gleichungen zweiter Art. Eingangsparameter ist die Lagrange Funktion, die Liste der generalisierten Koordinaten und die generalisierten Kräfte.*

Aufgabe 4.2 *Für das mathematische Modell des Pendels sollen folgende Aufgaben mit* SCILAB/SCICOS *gelöst werden:*

1. *Bauen Sie eine Simulation für das nichtlineare Modell mit den angegebenen Parametern auf. Verifizieren Sie durch Simulation die Ruhelage(n) des Systems. Verwenden Sie (a) eine Blockschaltstruktur, (b) eine* computational function *mit einem* GENERIC Block *sowie (c) mit einer computational* und *interfacing function. Die computational function kann wahlweise in C oder Scilab Skriptsprache erstellt werden.*

2. *Berechnen Sie jenes Moment $M(t)$, das notwendig ist, um $\varphi(t) = 0$, $\omega(t) = 0$ für die Anfangswerte $\varphi(0) = \omega(0) = 0$ zu erzeugen. Verifizieren Sie Ihr Resultat durch Simulation.*

3. *Simulieren Sie das linearisierte Modell des dynamischen Systems. Vergleichen Sie das Verhalten des nichtlinearen und des linearisierten Systems anhand der Sprungantwort im Arbeitspunkt.*

Aufgabe 4.3 *Simulieren Sie in* SCILAB/SCICOS *die Antwort des linearen Modells*

$$\dot{\mathbf{x}} = \begin{bmatrix} -2 & -2 \\ 1 & 0 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u \qquad \mathbf{x}_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$
$$y = \begin{bmatrix} 0 & 1 \end{bmatrix} \mathbf{x}$$

für die Eingangsgröße

$$u(t) = \sin\left(\sqrt{3}t\right) \sigma(t) \ .$$

Verifizieren Sie die Simulation durch Rechnung.

4.2 Ein elektrisches System

Gegeben ist ein aktiver Tiefpass zweiter Ordnung nach Bild 4.1.

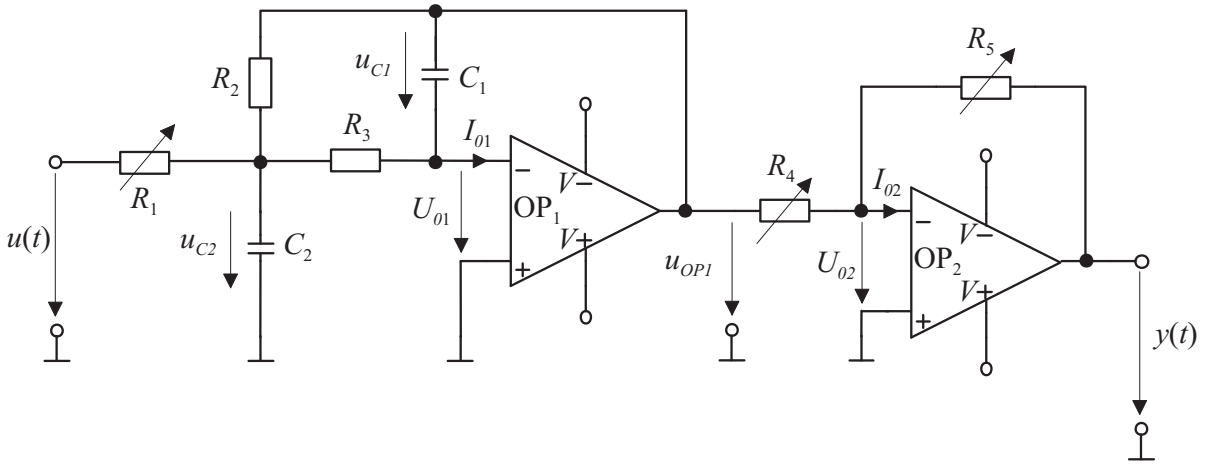


Abbildung 4.1: Ein einfaches elektrisches Netzwerk.

Aufgabe 4.4 *Für dieses elektrische Netzwerk sollen folgende Punkte untersucht werden:*

1. *Bestimmen Sie unter der Voraussetzung idealer Operationsverstärker das zugehörige mathematische Modell der Form*

$$\dot{\mathbf{x}} = \mathbf{A}\mathbf{x} + \mathbf{b}u$$
$$y = \mathbf{c}^T\mathbf{x} + du$$

mit der Eingangsspannung $u(t)$ und der Ausgangsspannung $y(t)$. Wählen Sie dazu geeignete Zustandsgrößen $\mathbf{x}$.

2. *Simulieren Sie die Sprungantwort des Systems unter Berücksichtigung, dass der Kondensator C_1 zum Zeitpunkt $t = 0$ auf* 5V *geladen ist. Verwenden Sie als Simulationsmodell sowohl eine* Blockschaltstruktur *als auch den* Continuous Linear System Block. *Für die Bauteilwerte gelte $C_1 = 10\text{nF}$, $C_2 = 6.8\mu F$, $R_1 = R_5 = 10\text{k}\Omega$, $R_2 = 20\text{k}\Omega$, $R_4 = 20\text{k}\Omega$ und $R_3 = 820\text{k}\Omega$.*

3. *Ist das System vollständig erreichbar bzw. beobachtbar?*

4.3 Wagen & Pendel

Auf einer Schiene gleitet ein Wagen, auf welchem ein drehbar gelagertes Pendel montiert ist, siehe Bild 4.2. Der Wagen hat die Masse m_1 und es wirkt darauf in horizontaler

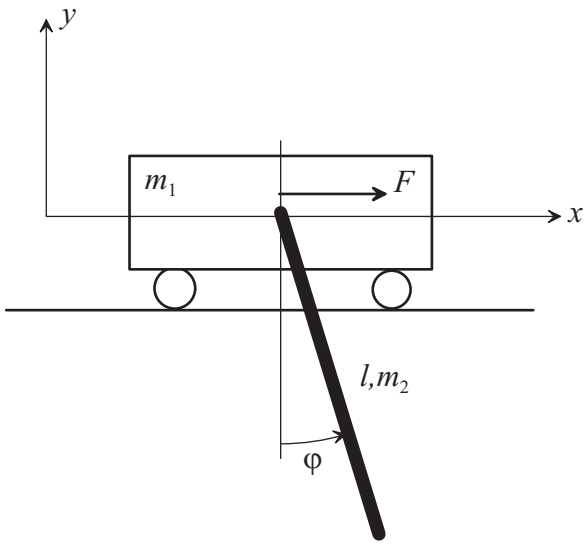


Abbildung 4.2: Wagen mit Pendel.

Richtung eine Kraft $F(t)$, welche die Eingangsgröße des Systems ist. Zwischen Wagen und Schiene wirkt eine geschwindigkeitsproportionale Reibung mit der Reibkonstante d_1. Das Pendel ist ein homogener Stab mit der Masse m_2 und der Länge l. Im Gelenk wirkt eine Reibung, welche proportional zur Winkelgeschwindigkeit ist (Reibkonstante d_2).

Aufgabe 4.5 *Untersuchen Sie die folgenden Punkte zum Wagen-Pendel Modell mit Hilfe von* Maxima.

1. *Bestimmen Sie ein mathematisches Modell der Form*

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, u)$$
$$y = \mathbf{g}(\mathbf{x}, u)$$

   mit Hilfe der Methode von Lagrange.

2. *Berechnen Sie alle Ruhelagen des Systems.*

3. *Linearisieren Sie das Modell um die Ruhelagen.*

4. *Untersuchen Sie das Modell ebenfalls für den Fall* $d_1 = d_2 = 0$.

5. *Nehmen Sie für diesen Punkt an, dass der Wagen an einer bestimmten Position befestigt ist. Zeigen Sie die Stabilität der unteren Ruhelage, indem Sie eine geeignete Liapunovfunktion finden.*

Aufgabe 4.6 *Für die nachfolgenden Aufgabenstellungen verwenden Sie die Parameter*

$$m_1 = 0.455\text{kg},\ m_2 = 0.21\text{kg},\ l = 0.61\text{m},\ d_1 = 0.1\text{kg/s},\ d_2 = 0.01\text{kgm}^2/\text{s}\ .$$

Untersuchen Sie die folgenden Punkte in SCILAB/SCICOS.

1. *Implementieren Sie das (nichtlineare) mathematische Modell des Systems. Achten Sie auf eine schnelle Änderbarkeit der Parameterwerte bzw. der Anfangswerte.*

2. *Vergleichen Sie das Verhalten des nichtlinearen und des linearisierten Modell in einer Umgebung um die Ruhelagen. Nehmen Sie z.B. verschiedene Anfangsauslenkungen für das Pendel und lassen es dann frei ausschwingen.*