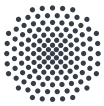
Institut für Systemdynamik
Univ.-Prof. Dr.-Ing. Dr. h. c. O. Sawodny



Universität Stuttgart

JuMP-Kurzbeschreibung

Dr.-Ing. Eckhard Arnold

Universität Stuttgart
Institut für Systemdynamik
Waldburgstr. 17/19
D-70563 Stuttgart

E-Mail: Eckhard.Arnold@isys.uni-stuttgart.de
Tel.: +49 (0)711/685-65928
Fax: +49 (0)711/685-66371

5. April 2019

Inhaltsverzeichnis

1 Übersicht

JuMP (siehe https://github.com/JuliaOpt/JuMP.jl und [6], [4]) ist eine algebraische Modellierungssprache für lineare, quadratische, gemischt-ganzzahlige und nichtlineare Optimierungsprobleme, die die Formulierung von Optimierungsproblemen („mathematische Modelle“) in einer abstrakten, der algebraischen Notation recht nahen Form ermöglicht. JuMP ist kein Solver, sondern eine Benutzer-Schnittstelle, die die Nutzung unterschiedlicher Solver vereinheitlicht. JuMP ist in die Programmiersprache Julia eingebettet.

Julia (https://julialang.org) ist eine relativ junge Programmiersprache (erste öffentliche Version 2012, aktuell Version 1.1.0) für numerisches und wissenschaftliches Rechnen und allgemeine Programmieraufgaben. Julia wurde und wird am MIT entwickelt und ist quelloffen mit MIT-Lizenz. Die wesentlichen Merkmale von Julia sind

- dynamisches Typsystem: Typprüfungen (z. B. Datentyp von Variablen) zur Laufzeit eines Programms
- Multimethoden („multiple dispatch“): Methodenauswahl anhand der dynamischen Typen mehrerer Objekte; damit kann das Verhalten von Funktionen durch Kombination der Argumenttypen bestimmt werden
- just-in-time Compiler (basierend auf LLVM) zur Übersetzung der Julia-Programme in Maschinencode zur Laufzeit
- sehr gute Performance (ähnlich C)
- effiziente Unterstützung von Unicode
- integrierte Paketverwaltung

Der Sprachumfang von Julia kann durch Zusatzpakete (derzeit ca. 2400, siehe https://pkg.julialang.org) erweitert werden.

Es gibt derzeit noch wenig Lehrbücher zu Julia, Ausnahmen sind [9], [2], [5] und [3]. Aktuelle Informationen finden sich im Internet, beispielsweise die Dokumentationen zu Julia https://docs.julialang.org/en/v1 und JuMP https://www.juliaopt.org/JuMP.jl/v0.19.0/. Unter https://julialang.org/learning und https://julialang.org/community finden sich weitere Informationsquellen.

Es soll zumindest erwähnt werden, dass die Verwendung von JuMP nicht die einzige Möglichkeit ist, Optimierungsprobleme in Julia zu lösen, siehe https://www.juliaopt.org. Mit `Convex.jl` gibt es eine weitere eingebettete algebraische Modellierungssprache, die speziell für konvexe Optimierungsprobleme geeignet ist. Im Paket `Optim.jl` sind Standardverfahren der unbeschränkten nichtlinearen Optimierung in Julia implementiert.

Im folgenden Kapitel wird eine kurze Beschreibung der wesentlichen Sprachelemente von JuMP gegeben. Kapitel 3 gibt eine Übersicht über Solver mit JuMP-Schnittstelle, und in Kapitel 4 folgen Installationshinweise für Julia. In den Kapiteln 5 bis 7 wird an drei Beispielen mit wachsender Komplexität die Lösung von Optimierungsaufgaben mit JuMP demonstriert.

2 Sprachbeschreibung

Die algebraische Modellierungssprache JuMP ist in Julia eingebettet, d. h. es steht der gesamte Sprachumfang von Julia zur Verfügung, der durch Komponenten zur Beschreibung von Optimierungsvariablen, Beschränkungen und Zielfunktionen erweitert ist.

Julia unterscheidet Groß- und Kleinschreibung. Die Indizes von Datenfeldern (Matrizen und Vektoren) beginnen wie in Matlab mit 1, werden jedoch in eckigen Klammern angegeben. Ein wesentlicher semantischer Unterschied besteht in der Übergabe von Datenfeldern bei Zuweisungen oder Funktionsaufrufen: Julia übergibt Referenzen an Objekte („call by reference“), während Matlab die Werte übergibt bzw. kopiert („call by value“). Im Unterschied zu Matlab werden 1×1-Matrizen nicht als Skalare behandelt. Kommentare werden in Julia durch `#` eingeleitet, mehrzeilige Kommentare in `#=` ... `=#` eingeschlossen. Fortsetzungszeilen (Matlab: `...`) werden nicht gesondert gekennzeichnet, müssen aber für den Compiler erkennbar sein, beispielsweise durch ensprechende Klammerung.

Nach Einbindung des Zusatzpakets `JuMP.jl` kann ein Modell (hier `m`) initialisiert werden[1]:

```
using JuMP, Ipopt, Cbc
m = Model(with_optimizer(Ipopt.Optimizer, print_level=0, tol=1e-6))
set_optimizer(m, with_optimizer(Cbc.Optimizer))
```

Die Auswahl des zu verwendenden Solvers kann bei der Initialisierung oder mit der Funktion `set_optimizer()` erfolgen, wobei zuvor die entsprechenden Zusatzpakete einzubinden sind. Solverspezifische Parameter werden in der Form `option1=value1,...` angegeben.

Optimierungsvariable werden mit dem Makro `@variable()` definiert. Mit Hilfe von Indexmengen (ganzzahlig oder Vektoren von Zeichenketten) können mehrdimensionale Optimierungsvariable definiert werden. Der Variablentyp kann mit `Int` als ganzzahlig oder `Bin` als binär (für Boolesche Variable) angegeben werden. Komponentenbeschränkungen sind in der Form `>=` bzw. `<=` vorzugeben. Startwerte werden mit dem Argument `start=` angegeben.

```
@variable(m, x)                          # skalare Optimierungsvariable
@variable(m, x1[-10:5:15,["A","B","C"]]) # Indexmengen
@variable(m, x2, Int)                    # ganzzahlig
@variable(m, x3, Bin)                    # binär
@variable(m, x4 >= 0)                    # Komponentenbeschränkung
@variable(m, -1.0 <= x5[i=2:4] <= 10*i)  # Schranke indexabhängig
@variable(m, x6, start=-1.0)             # Startwert
@variable(m, x7[1:3,1:3,-2:2])           # mehrdimensional
```

Komponentenbeschränkungen und Startwerte für Optimierungsvariable können auch mit Funktionsaufrufen angegeben werden.

```
set_start_value(x, -1.0)
set_lower_bound(x, -10.0)
set_upper_bound(x, 100.0)
```

Lineare Beschränkungen werden mit dem Makro `@constraint()`, nichtlineare Beschränkungen mit `@NLconstraint()` unter Angabe der Relation `<=`, `>=` oder `==` definiert.

[1] siehe Abschnitt „Mögliche Probleme“ auf Seite 8

```
@constraint(m, x+x3 == 5)
@constraint(m, sum(x1[i,j] for i=-10:5:15, j in ["A","B","C"]) <= 5.0)
@NLconstraint(m, x^2 >= 4.0)
```

Der Operator `sum()` erlaubt die Summation über Indexmengen.

Mehrdimensionale Beschränkungen können entweder mittels Schleifen oder durch Angabe einer zusätzlichen indizierten Variablen definiert werden.

```
for i = 1:10
    @constraint(m, d[i]-d[i+1] <= 0)
end
@constraint(m, cdd[i=1:10], d[i]-d[i+1] <= 0)
```

Mit Hilfe der Variablen `cdd` kann später die Beschränkung referenziert werden, beispielsweise zur Abfrage der LAGRANGE-Multiplikatoren mit der Funktion `dual()`.

Eine lineare oder quadratische Zielfunktion wird mit dem Makro `@objective()` definiert, für eine nichtlineare Zielfunktion ist `@NLobjective()` zu verwenden. Die Optimierungsrichtung ist durch `Min` oder `Max` auszuwählen. Es gilt immer die zuletzt angegebene Zielfunktion.

```
@objective(m, Min, 5*x*x+x2)
@NLobjective(m, Max, -(1-x1[-5,"B"]^2)^2)
```

Mit `optimize!()` wird der Optimierungslauf mit dem ausgewählten Solver gestartet. Rückgabewerte werden mit `termination_status()` abgefragt und sind beispielsweise `OPTIMAL`, `LOCALLY_SOLVED`, `ALMOST_LOCALLY_SOLVED`, `INFEASIBLE`, `LOCALLY_INFEASIBLE` oder `INVALID_MODEL`, siehe `JuMP.MathOptInterface`.

```
optimize!(m)
termination_status(m)
```

Wenn eine optimale Lösung gefunden wurde, dann liefert die Funktion `objective_value()` den Optimalwert der Zielfunktion. Die Optimalwerte von skalaren (der einzelne Komponenten von mehrdimensionalen) Optimierungsvariablen werden mit `value()` erhalten. Die Optimalwerte von Feldern (arrays) von Optimierungsvariablen werden mit `value.()` in ein `Array{Float64,...}` (einen „gewöhnlichen“ Vektor/Matrix) umgewandelt, falls die Indexmengen das zulassen. Ansonsten ergibt sich ein `DenseAxisArrayFloat64,...` mit den eigentlichen Zahlenwerten im Feld `data`.

```
objective_value(m)   # optimaler Zielfunktionswert
value(x1)            # optimale Lösung
value(x1[-5,"A"])    # optimale Lösung mit Indexauswahl
value.(x5.data)      # optimale Lösung als Vektor
```

Modellparameter, die nicht optimiert werden sollen, aber für weitere Optimierungsläufe ohne erneute Übersetzung des Modells geändert werden sollen, können mit dem Makro `@NLparameter()` definiert werden.

```
@NLparameter(m, p == 1.0)
```

Solver	Problemklasse				in	Bemerkung,
	LP	QP	MILP	NLP	Zusatzpaket	Lizenz
CBC			✔		`Cbc.jl`	EPL
CLP	✔				`Clp.jl`	EPL
GLPK	✔		✔		`GLPKMathProg Interface.jl`	GPL
OSQP	✔	✔			`OSQP.jl`	Apache
IPOPT	✔	✔		✔	`Ipopt.jl`	[10], EPL
GUROBI	✔	✔	✔		`Gurobi.jl`	kommerziell[a]
CPLEX	✔	✔	✔		`CPLEX.jl`	kommerziell[a]
NLOPT				✔	`NLopt.jl`	LGPL
KNITRO	✔	✔	✔	✔	`KNITRO.jl`	kommerziell
MINOS	✔	✔		✔	`AmplNLWriter.jl`	kommerziell[b]
SNOPT	✔	✔		✔	`AmplNLWriter.jl`	kommerziell[b]

Tabelle 1: Von JuMP unterstützte Solver (Auswahl).

In JuMP werden solche Parameter als „nichtlineare Parameter“ bezeichnet und können nur in `@NLconstraint()` und `@NLobjective()` verwendet werden.

Der Sprachumfang von JuMP kann durch benutzerdefinierte Julia-Funktionen mit skalaren Argumenten erweitert werden. Diese sind mittels `JuMP.register()` für JuMP verfügbar zu machen, dabei ist die Anzahl der Argumente und die Art der Bereitstellung der Ableitungen anzugeben.

```
mysquare(x) = x*x
JuMP.register(m, :mysquare, 1, mysquare, autodiff=true)
```

JuMP bietet noch viele weitere Funktionalitäten, auf die im Rahmen dieser Kurzanleitung nicht eingegangen werden kann. Hierzu wird auf die Dokumentation unter https://www.juliaopt.org/JuMP.jl/v0.19.0/ verwiesen. Es gibt zahlreiche Beispielsammlungen, siehe z. B. https://github.com/JuliaOpt/JuMP.jl/tree/master/examples.

Begrenzt wird die Leistungsfähigkeit von JuMP durch die Anforderung, dass die Optimierungsaufgabe komplett in JuMP formuliert werden muss. Damit ergibt sich eine Einschränkung auf den von der Sprache vorgegebenen Funktionsumfang, der beispielsweise keine numerische Lösung von Differentialgleichungen ermöglicht.

3 Solver

JuMP unterstützt zahlreiche Solver, siehe Tabelle 1. Die vollständige Liste ist unter https://www.juliaopt.org/JuMP.jl/v0.19.0/installation/#Getting-Solvers-1 zu finden.

Weitere Solver mit AMPL-Schnittstelle können mit `AmplNLWriter.jl` eingebunden werden.

Solver-spezifische Parameter werden bei der Solver-Auswahl in der Form `option1=value1,...` angegeben, siehe Abschnitt 2.

[1] kostenlose „Academic License“ verfügbar
[2] in AMPL frei verfügbar mit eingeschränkter Lizenz, siehe Abschnitt 4

Beschreibung	Option	Wertebereich	Standardwert
Ausgabe	`print_level`	$[0, 12]$	5
Abbruchbedingung	`tol`	$(0, +\infty)$	$1 \cdot 10^{-8}$
Max. Iterationen	`max_iter`	$[0, +\infty)$	3000
Max. Rechenzeit	`max_cpu_time`	$(0, +\infty)$	$1 \cdot 10^{6}$
Test Ableitungen	`derivative_test`	`{none, first-order, second-order}`	`none`

Tabelle 2: Wichtige Optionen für Ipopt, siehe https://www.coin-or.org/Ipopt/documentation/node40.html

Beschreibung	Option	Wertebereich	Standardwert
Ausgabe	`LogLevel`	$[0, 4]$	0
Abbruchbedingung	`PrimalTolerance`	$(0, +\infty)$	$1 \cdot 10^{-7}$
Abbruchbedingung	`DualTolerance`	$(0, +\infty)$	$1 \cdot 10^{-7}$
Max. Rechenzeit	`MaximumSeconds`	$(0, +\infty)$	
Algorithmus	`SolveType`	$[0, 5]$	
		0/1 – dualer/primaler Simplex	
		3/4 – Barrieremethode	
		5 – automatische Auswahl	

Tabelle 3: Wichtige Optionen für Clp, siehe https://github.com/JuliaOpt/Clp.jl

Beschreibung	Option	Wertebereich	Standardwert
Ausgabe	`logLevel`	$\{0, 1\}$	0
Max. Rechenzeit	`seconds`	$(0, +\infty)$	
Anzahl Prozesse	`threads`		

Tabelle 4: Wichtige Optionen für Cbc, siehe https://github.com/JuliaOpt/Cbc.jl

4 Installation

Im Folgenden wird die Standard-Installation unter Windows (Windows 10, 64 Bit) beschrieben.

Zunächst wird Julia von https://julialang-s3.julialang.org/bin/winnt/x64/1.1/julia-1.1.0-win64.exe heruntergeladen und in ein beliebiges Verzeichnis (im Beispiel `C:\Users\<user>\AppData\Local\Julia-1.1.0`) installiert.

Die Umgebungsvariable `JULIA_BINDIR` sollte den Verzeichnispfad zu `julia.exe` enthalten, im Beispiel `C:\Users\<user>\AppData\Local\Julia-1.1.0\bin`. Dieser ist auch zur Umgebungsvariablen `Path` hinzuzufügen: `%JULIA_BINDIR%;...`.

`julia.exe` startet ein interaktives Kommandozeilenprogramm (REPL: Read-Evaluate-Print-Loop), mit dem beliebige Julia-Anweisungen ausgeführt werden können, beispielsweise

```
julia> pwd()                          # Ausgabe des aktuellen Verzeichnisses
julia> cd("C:\\Users\\xyz\\HOME")     # Verzeichniswechsel
julia> ?                              # Hilfe zu Kommandos
```

```
help?> cd
julia> ;                          # Ausführung von Shell-Kommandos
shell> cmd /C dir
julia> using LinearAlgebra        # Einbinden des Standardpakets LinearAlgebra
julia> eigen(randn(5,5))          # Berechnung der Eigenwerte und -vektoren einer
                                  # (5x5)-Matrix von normalverteilten Zufallszahlen
julia> exit()                     # REPL beenden
```

Julia besitzt einen integrierten Paketmanager, der mit `using Pkg` eingebunden wird. Die wichtigsten Befehle sind `Pkg.add("paket")` und `Pkg.update()`. Für die im Folgenden beschriebene Nutzung von JuMP werden benötigt:

- Modellierungssprache und Solver: `JuMP.jl`, `Ipopt.jl`, `Clp.jl`
- Grafik: `Plots.jl`, `PyPlot.jl`

Weitere sinnvolle Pakete sind

- Solver: `Cbc.jl`, `OSQP.jl`, `AmplNLWiter.jl`
- LTI-Systeme: `ControlSystems.jl`
- Matlab-Dateiformat: `MAT.jl`

Die Installation dieser Zusatzpakete kann mit der Hilfsdatei `inst.jl` erfolgen

```
julia> include("inst.jl")
```

und zieht die automatische Installation weiterer Pakete nach sich.

Mögliche Probleme:

Keine bekannten Probleme mit Julia 1.1.0

Mögliche Probleme mit älteren Julia-Versionen:

In Julia 1.0.2 funktioniert die automatische Installation von `MAT.jl` u. U. nicht korrekt, und es muss ein aktualisiertes Paket installiert werden:

```
 julia> ] (v1.0) pkg> rm MAT (v1.0) pkg> add
https://github.com/halleysfifthinc/MAT.jl#v0.7-update (v1.0) pkg>
<backspace> julia> using MAT
```

in Julia 1.0.2 funktioniert die automatische Installation von `Cbc.jl` u. U. nicht korrekt. Im Verzeichnis `.julia\packages\WinRPM\Y9QdZ\cache\2` muss die Datei `mingw64-libstdc++6-8.2.0-2.5.noarch.cpio` von Hand aus dem entsprechenden `rpm`-Archiv extrahiert werden. Danach wird das Paket mit `using Pkg; Pkg.build("Cbc")` neu erstellt.

Unter Umständen kommt es nach Installation von Julia 0.6.2, `JuMP.jl` und `Ipopt.jl` beim ersten Aufruf von `Ipopt.jl` zu einem Fehler

```
ERROR: LoadError: ReadOnlyMemoryError()
```

In diesem Fall sind die Dateien **libgcc_s_seh-1.dll**, **libgfortran-4.dll**, **libquadmath-0.dll**, **libstdc++-6.dll** und **libwinpthread-1.dll** aus dem Verzeichnis
C:\Users\<user>\.julia\v0.6\WinRPM\deps\usr\x86_64-w64-mingw32\sys-root\mingw\bin
nach **%JULIA_HOME%** zu kopieren.

Unter Umständen tritt beim ersten Aufruf einer **plot()**-Anweisung aus der **PyPlot**-Bibliothek ein Laufzeitfehler der Microsoft Visual C++ Laufzeit-Bibliothek **Runtime Error! R6034** auf. In diesem Fall ist in der Initialisierungsdatei **C:\Users\<user>\.juliarc.jl** die Anweisung

```
ENV["MPLBACKEND"]="qt4agg"
```

zu ergänzen.

JuMP kann mittels **AmplNLWriter.jl** beliebige Solver mit AMPL-Schnittstelle nutzen. Eine im Funktionsumfang eingeschränkte AMPL-Version („Studentenlizenz“: maximal 300-500 Variable und 300 Beschränkungen) ist frei verfügbar und kann zusammen mit mehreren Solvern als Archivdatei http://ampl.com/demo/ampl.mswin64.zip heruntergeladen werden. **ampl.mswin64.zip** wird in ein beliebiges Verzeichnis (z. B. **C:\Program Files\ampl.mswin64**) entpackt. Eine gesonderte Installation ist nicht erforderlich.

Alternativen zur Standard-Installation

Einen einfachen Einstieg in die Arbeit mit Julia ohne Installation ermöglicht JuliaBox https://www.juliabox.com. Hier läuft Julia komplett im Browser. Mit dem Paketmanager (siehe unten) sind lediglich die entsprechenden Zusatzpakete zu installieren.

JuliaPRO https://juliacomputing.com/products/juliapro ist ein Julia-Komplettpaket mit Entwicklungsumgebung(en), Debugger, Profiler und diversen Paketen.

Die Nutzung von Jupyter-Notebooks https://jupyter.org ist eine weitere Alternative.

5 Beispiel: Problem von Rosenbrock

Das aus der Vorlesung [1] bekannte Problem von Rosenbrock, ein unbeschränktes nichtlineares Optimierungsproblem mit zwei Variablen

$$\min_{x_1,\,x_2} \left\{ Q(x_1,x_2) \;=\; 100 \cdot \left(x_2 - x_1^2\right)^2 + (x_1-1)^2 \right\} \tag{1}$$

soll mit JuMP gelöst werden.

`julia.exe` startet ein interaktives Kommandozeilenprogramm (REPL). Zunächst werden die zu nutzenden Zusatzpakete `JuMP` und `Ipopt` geladen. Anschließend wird ein Modell `m` initialisiert, und der Solver Ipopt wird ausgewählt. Die Problembeschreibung (hier die Deklaration der Optimierungsvariablen und die Definition der Zielfunktion) sowie die Daten (hier die Startwerte für die Optimierungsvariablen) werden hinzugefügt. Dann wird der Solver aufgerufen und (nach erfolgreichem Optimierungslauf) werden die Optimierungsergebnisse ausgegeben.

```
julia> using JuMP, Ipopt
julia> m = Model(with_optimizer(Ipopt.Optimizer))
A JuMP Model
Feasibility problem with:
Variables: 0
Model mode: AUTOMATIC
CachingOptimizer state: EMPTY_OPTIMIZER
Solver name: SolverName() attribute not implemented by the optimizer.
julia> @variable(m, x[1:2])
2-element Array{VariableRef,1}:
 x[1]
 x[2]
julia> set_start_value(x[1], -1.2)
-1.2
julia> set_start_value(x[2], -1.0)
-1.0
julia> @NLobjective(m, Min, 100*(x[2]-x[1]^2)^2+(x[1]-1)^2)
julia> print(m)
Min 100.0 * (x[2] - x[1] ^ 2.0) ^ 2.0 + (x[1] - 1.0) ^ 2.0
Subject to
julia> optimize!(m)
...
julia> termination_status(m)
LOCALLY_SOLVED::TerminationStatusCode = 4
julia> objective_value(m)
2.238160324149676e-17
julia> value.(x)
2-element Array{Float64,1}:
 0.9999999953964535
 0.9999999906838674
```

Es wird die bekannte Lösung $x_1 = x_2 = 1$ gefunden.

Dieser interaktive Modus ist für kleine Optimierungsaufgaben oder zur Fehlersuche geeignet. Bei größeren Optimierungsproblemen empfiehlt sich die Nutzung von Julia-Funktionen und/oder Speicherung in eigenständigen Dateien.

6 Beispiel: Transportproblem

Das Transportproblem aus [8] (siehe Vorlesung [1])

$$Q(\mathbf{x}) = \sum_{i=1}^{4} K_i x_i \longrightarrow \min_{x_i,\, i=1,\ldots,4}! \tag{2a}$$

$$0 \leq x_i \leq C_i, \quad i = 1, \ldots, 4 \tag{2b}$$

$$V_1 = x_1 + x_3 \tag{2c}$$

$$V_2 = x_2 - x_3 + x_4 \tag{2d}$$

ist eine lineare Optimierungsaufgabe.

Eine einfache Formulierung des Problems in JuMP zeigt das folgende Listing.

Listing 1: Transportproblem (einfache Formulierung).

```
using JuMP, Clp, LinearAlgebra
# Parameter
C = [30, 20, 30, 60]
K = [40, 20, 10, 10]
V = [50, 40]
# Modell
m = Model(with_optimizer(Clp.Optimizer))
# Optimierungsvariable
@variable(m, 0 <= x[i=1:length(C)] <= C[i])
# Beschraenkungen
@constraint(m, x[1]+x[3] == V[1])
@constraint(m, x[2]-x[3]+x[4] == V[2])
# Zielfunktion
@objective(m, Min, dot(x, K))
# Aufruf Solver
optimize!(m)
# Ergebnisausgabe
println("Solver Status: ", termination_status(m))
if termination_status(m) == JuMP.MathOptInterface.OPTIMAL
        println("Gesamtkosten: ", objective_value(m))
        println(value.(x))
end
```

In Zeile 1 werden die Zusatzpakete `JuMP.jl`, `Clp.jl` und `LinearAlgebra.jl` geladen. Die Parameter des Optimierungsproblems werden in den Zeilen 3-5 als Vektoren angegeben. In Zeile 7 wird das Modell `m` initialisiert und der Solver `Clp` ausgewählt. Die Optimierungsvariablen `x` werden Zeile 9 mit dem Makro `@variable()` als Vektor mit den Komponentenbeschränkungen (2b) definiert. In den Zeile 11 und 12 werden die Knotengleichungen (2c), (2d) werden mit dem Makro `@constraint()` als Gleichungsbeschränkungen (Operator `==`) definiert. In Zeile 14 wird die Zielfunktion (2a) mit dem Makro `@objective()` definiert, wobei das Symbol `Min` die Richtung der Optimierung angibt. Die Funktion `dot()` berechnet das Skalarprodukt.

Mit der Anweisung `optimize!()` in Zeile 16 wird der Solver gestartet und in den Zeilen 18 bis 22 wird das Optimierungsergebnis unter Nutzung der Funktionen `objective_value()` für den

Optimalwert der Zielfunktion und `value.()` für die Optimalwerte der Optimierungsvariablen ausgegeben.

Eine allgemeinere Formulierung unter Verwendung von Indexmengen ist in der Datei `Papageorgiou141_jump.jl`  gespeichert.

Listing 2: Transportproblem: Datei `Papageorgiou141_jump.mod`.

```
1  using JuMP, Clp
   # Parameter
3  Rohrleitungen = ["L1", "L2", "L3", "L4"]
   Verbraucher = ["V1", "V2"]
5  Kapazität = Dict(zip(Rohrleitungen, [30, 20, 30, 60]))
   Kosten = Dict(zip(Rohrleitungen, [40, 20, 10, 10]))
7  Verbrauch = Dict(zip(Verbraucher, [50, 40]))
   Zufluss = Dict("V1" => ("L1", "L3"), "V2" => ("L2", "L4"))
9  Abfluss = Dict("V1" => (), "V2" => ("L3",))
   # Modell
11 m = Model(with_optimizer(Clp.Optimizer, LogLevel=0))
   # Optimierungsvariable
13 @variable(m, 0 <= Durchfluss[r in Rohrleitungen] <= Kapazität[r])
   # Beschränkungen
15 @constraint(m, con[v in Verbraucher], sum(Durchfluss[r] for r in Zufluss[v]) -
    sum(Durchfluss[r] for r in Abfluss[v]) == Verbrauch[v])
   # Zielfunktion
17 @objective(m, Min, sum(Durchfluss[r]*Kosten[r] for r in Rohrleitungen))
   # Aufruf Solver
19 optimize!(m)
   # Ergebnisausgabe
21 println("Solver Status: ", termination_status(m))
   if termination_status(m) == JuMP.MathOptInterface.OPTIMAL
23         println("Gesamtkosten: ", objective_value(m))
           for i in Rohrleitungen
25                 println(i, " : ", value(Durchfluss[i]))
           end
27 end
```

In der Zeile 1 werden die Zusatzpakete `JuMP.jl` und `Clp.jl` geladen. In den Zeilen 3 und 4 werden die Indexmengen `Rohrleitungen` und `Verbraucher` als Vektoren von Zeichenketten definiert. Der maximale Durchfluss `Kapazitaet` und die Transportkosten `Kosten` werden in den Zeilen 5 und 6 als assoziative Datenfelder (Dictionary, `Dict`) über diesen Indexmengen angegeben. Schließlich werden in den Zeilen 8 und 9 die Flussrichtungen als assoziative Datenfelder von Tupeln der Rohrleitungen mit positiver bzw. negativer Flussrichtung in die jeweiligen Verbrauchsknoten angegeben.

In Zeile 11 wird das Optimierungsproblem `m` initialisiert und der Solver `Clp` ausgewählt. In Zeile 13 werden mit dem Makro `@variable()` die Optimierungsvariablen `Durchfluss` als Feld über der Indexmenge `Rohrleitungen` mit den Komponentenbeschränkungen (2b) definiert. Die Knotengleichungen (2c), (2d) werden mit dem Makro `@constraint()` als Gleichungsbeschränkungen (Operator `==`) definiert. Da mit der Anweisung in Zeile 15 mehrere Beschränkungen definiert werden sollen, ist die Indexmenge `Verbraucher` als 2. Argument anzugeben. Der Operator `sum()` erlaubt

die Summation über die zugehörigen Indexmengen (Tupel). In Zeile 17 wird schließlich die Zielfunktion (2a) mit dem Makro `@objective()` definiert, wobei das Symbol `Min` die Richtung der Optimierung angibt.

Der Aufruf des Solvers und die Ergebnisausgabe sind identisch zur oben beschriebenen einfachen Formulierung.

Mit dem Befehl `include("Papageorgiou141_jump.l")` wird die Datei in REPL geladen und die Ausführung gestartet:

```
julia> include("Papageorgiou141_jump.jl")
Solver Status: OPTIMAL
Gesamtkosten: 1900.0
L1 : 20.0
L2 : 10.0
L3 : 30.0
L4 : 60.0
```

Das Optimierungsproblem kann zur Kontrolle ausgegeben werden:

```
julia> print(m)
Min 40 Durchfluss[L1] + 20 Durchfluss[L2] + 10 Durchfluss[L3] + 10 Durchfluss[L4]
Subject to
 Durchfluss[L1] >= 0.0
 Durchfluss[L2] >= 0.0
 Durchfluss[L3] >= 0.0
 Durchfluss[L4] >= 0.0
 Durchfluss[L1] <= 30.0
 Durchfluss[L2] <= 20.0
 Durchfluss[L3] <= 30.0
 Durchfluss[L4] <= 60.0
 Durchfluss[L1] + Durchfluss[L3] == 50.0
 Durchfluss[L2] + Durchfluss[L4] - Durchfluss[L3] == 40.0
```

Alternativ kann man `julia.exe` beim Aufruf den Namen einer JULIA-Programmdatei übergeben, die dann im nicht-interaktiven Modus ausgeführt wird:

```
D:\Users\Arnold>julia Papageorgiou141_jump.jl
Solver Status: OPTIMAL
Gesamtkosten: 1900.0
L1 : 20.0
L2 : 10.0
L3 : 30.0
L4 : 60.0
```

7 Beispiel: Optimalsteuerung van der Pol-Oszillator

Das in diesem Abschnitt beschriebene Beispiel ist [7] entnommen. Betrachtet wird der gesteuerte VAN DER POL-Oszillator

$$\begin{aligned}
\dot{x}_1(t) &= x_2(t) && \text{(3a)}\\
\dot{x}_2(t) &= -x_1(t) + (1 - x_1^2(t)) \cdot x_2(t) + u(t) && \text{(3b)}\\
0 &\leq t \leq t_f\\
x_1(0) &= 1, \quad x_2(0) = 1 && \text{(3c)}\\
x_1^2(t_f) + x_2^2(t_f) &= r^2 && \text{(3d)}\\
-1 &\leq u(t) \leq 1, \quad 0 \leq t \leq t_f && \text{(3e)}
\end{aligned}$$

Der Parameter r ist zu $r = 0.2$ vorgegeben. Für das System sind vier Optimalsteuerungsaufgaben zu lösen.

Problem 1: Bei fester Endzeit $t_f = 4$ soll das Zielfunktional

$$J_1 = \frac{1}{2} \int_0^{t_f} x_1^2(t) + x_2^2(t) + u^2(t)\, dt \tag{4}$$

minimiert werden.

Problem 2: Bei freier Endzeit t_f soll das System durch Minimierung von

$$J_2 = t_f \tag{5}$$

zeitoptimal gesteuert werden.

Problem 3: Bei fester Endzeit $t_f = 4$ soll das Zielfunktional (4) minimiert werden. Zusätzlich ist die Zustandsbeschränkung

$$-0.4 \leq x_2(t) \tag{6}$$

zu berücksichtigen.

Problem 4: Bei fester Endzeit $t_f = 4$ und geänderten Anfangsbedingungen $x_1(0) = 0$, $x_2(0) = 1$ anstelle von (3c) ist das Zielfunktional

$$J_4 = \frac{1}{2} \int_0^{t_f} x_1^2(t) + x_2^2(t)\, dt \tag{7}$$

zu minimieren. Die Endbedingung (3d) entfällt.

Die Optimalsteuerungsaufgabe wird mittels direkter Kollokation (siehe Vorlesung [1]) unter Verwendung von JuMP gelöst.

Hierzu wird ein äquidistantes Zeitgitter

$$t^k = k \cdot \Delta t, \quad k = 0, \dots N, \quad \Delta t = \frac{t_f}{N} \tag{8}$$

eingeführt. $\mathbf{x}^k \approx \mathbf{x}(t^k)$, $k = 0, \dots, N$ bezeichnen die Näherungswerte für die Zustandsgrößen in den Gitterpunkten und $u^k \approx u(t^k + \frac{1}{2}\Delta t)$, $k = 0, \dots, N-1$ die Näherungswerte für die Eingangsgröße im Intervallmittelpunkt.

Eine direkter Kollokationsansatz mittels impliziter Mittelpunktregel liefert die folgende diskrete Approximation des Optimalsteuerungsproblems (3a)–(4):

$$x_1^{k+1} - x_1^k = \Delta t \cdot \frac{x_2^k + x_2^{k+1}}{2} \tag{9a}$$

$$x_2^{k+1} - x_2^k = \Delta t \cdot \left(-\frac{x_1^k + x_1^{k+1}}{2} + \left(1 - \left(\frac{x_1^k + x_1^{k+1}}{2}\right)^2\right) \cdot \frac{x_2^k + x_2^{k+1}}{2} + u^k \right) \tag{9b}$$

$$0 \le k \le N-1$$

$$x_1^0 = 1, \quad x_2^0 = 1 \tag{9c}$$

$$\left(x_1^N\right)^2 + \left(x_2^N\right)^2 = r^2 \tag{9d}$$

$$-1 \le u^k \le 1, \quad k = 0, \dots N-1 \tag{9e}$$

$$\tilde{J}_1 = \frac{\Delta t}{2} \sum_{k=0}^{N-1} \left(\left(\frac{x_1^k + x_1^{k+1}}{2}\right)^2 + \left(\frac{x_2^k + x_2^{k+1}}{2}\right)^2 + \left(u^k\right)^2 \right) \tag{9f}$$

Für die Probleme 2–4 ist die Aufgabe entsprechend zu modifizieren.

Im folgenden Listing ist das JuMP-Modell der diskreten Approximation (9a)–(9f) angegeben (Datei `vdp_jump.jl` ).

Listing 3: Optimalsteuerung: Problem 1 (Datei `vdp_jump.jl`).

```
# Zusatzpakete
using JuMP, Ipopt, Plots
pyplot(overwrite_figure=false)
# Modell und Solver
mod = Model(with_optimizer(Ipopt.Optimizer, max_iter=1000))
# benutzerdefinierte Funktion
fun(x1, x2) = -x1+(1-x1^2)*x2
JuMP.register(mod, :fun, 2, fun, autodiff=true)
# Modellparameter
@NLparameter(mod, x10 == 1.0)
x20 = 1.0
tff = 4.0
r = 0.2
umin = -1.0
umax = 1.0
@NLparameter(mod, p4 == 1.0)
# Diskretisierung
N = 99
# Optimierungsvariable
```

```
20 @variable(mod, umin <= u[0:N-1] <= umax)      # Steuergröße
   @variable(mod, x1[0:N])                       # Zustandsgröße 1
22 @variable(mod, x2[0:N])                       # Zustandsgröße 2
   @variable(mod, tff <= tf <= tff)              # Optimierungshorizont [0, tf]
24 # Randbedingungen
   @NLconstraint(mod, x1[0] == x10)              # Anfangsbedingung x1
26 @constraint(mod, x2[0] == x20)                # Anfangsbedingung x2
   @NLconstraint(mod, p4*(x1[N]^2+x2[N]^2) == p4*r^2)     # Endbedingung
28 # DGL: Kollokationsbedingungen implizite Mittelpunktregel
   @NLconstraint(mod, koll_x1[k=0:N-1], x1[k+1] - x1[k] == tf/N*(x2[k+1] + x2[k])
   /2.0)
30 @NLconstraint(mod, koll_x2[k=0:N-1], x2[k+1] - x2[k] == tf/N*(fun((x1[k+1]+x1[
   k])/2.0, (x2[k+1]+x2[k])/2.0)+u[k]))
   # Zielfunktional
32 @NLobjective(mod, Min, 0.5*tf/N*sum(((x1[k]+x1[k+1])/2.0)^2 + ((x2[k]+x2[k+1])
   /2.0)^2 + (u[k])^2 for k=0:N-1))
   # Aufruf Solver
34 optimize!(mod)
   # Ergebnis
36 println("Solver Status: ", termination_status(mod))
   if termination_status(mod) == JuMP.MathOptInterface.LOCALLY_SOLVED
38         println("Zielfunktional: ", objective_value(mod))
           ergebnis(1, tf, x1, x2, u, koll_x1, koll_x2)
40 end
```

In Zeile 2 werden die Zusatzpakete `JuMP.jl`, `Ipopt.jl` und `Plots.jl` geladen. In Zeile 3 wird das von `Plots` zu verwendende Grafikpaket und die Parameter angegeben.

Das Modell `mod` wird in Zeile 5 initialisiert. In Zeile 7 und 8 wird eine Funktion `fun()` definiert und für JuMP verfügbar gemacht.

Die Modellparameter werden in den Zeilen 10–16 festgelegt. Da der Anfangswert $x_1(0)$ für Problem 4 geändert werden soll, wird er als veränderlicher Modellparameter mit `@NLparameter()` definiert, ebenso wie der Parameter `p4`, siehe unten.

Die Anzahl der Diskretisierungsintervalle bzw. Gitterpunkte in Zeile 18 entspricht Gleichung (8).

Die Optimierungsvariablen sind die Werte der Steuer- und Zustandsgrößen in den Gitterpunkten bzw. Intervallen (Zeilen 20–22) sowie der Optimierungshorizont `tf` (Zeile 23). Da der Optimierungshorizont in Problem 1 mit $t_f = 4$ fest vorgegeben ist, wird die untere und obere Schranke für `tf` auf diesen vorgegebenen Wert gesetzt.

Die Anfangsbedingungen (9c) sowie die Endbedingung (9d) werden als Gleichungsbeschränkungen in den Zeilen 25–27 vorgegeben. Da `x10` ein nichtlinearer Modellparameter ist, ist die zugehörige Anfangsbedingung eine nichtlineare Beschränkung.

Die Kollokationsbedingungen für die Approximation der Zustandsdifferentialgleichungen mittels impliziter Mittelpunktregel (9a), (9b) finden sich in den Zeilen 29 und 30, dabei wird die oben definierte Funktion `fun()` verwendet.

Die diskretisierte Zielfunktion (9f) ist in Zeile 32 definiert. Es ist das Makro `@NLobjective()` zu verwenden, da gemäß (9f) die quadratische Zielfunktion mit $\frac{t_f}{N}$ multipliziert wird.

Der Aufruf des Solvers erfolgt in Zeile 34, anschließend werden die Ergebnisse ausgewertet und mit der Funktion `ergebnis()` grafisch dargestellt.

Nun wird das Modell entsprechend Problem 2, 3 und 4 modifiziert und die optimale Lösung wird jeweils erneut berechnet und ausgewertet.

Listing 4: Optimalsteuerung: Problem 2 (Datei `vdp_jump.jl`).

```
# Problem 2: freie Endzeit, zeitoptimal
set_lower_bound(tf, 0.0)
set_upper_bound(tf, Inf)
set_start_value(tf, tff)
@NLobjective(mod, Min, tf)
# Aufruf Solver
optimize!(mod)
```

Für Problem 2 wird die Endzeit t_f als Optimierungsvariable freigegeben, indem die Schranken (Komponentenbeschränkungen) für die Variable `tf` auf `0` bzw. `Inf` (entsprechend $+\infty$) gesetzt werden. Gemäß (5) ist nun die Endzeit `tf` zu minimieren.

Listing 5: Optimalsteuerung: Problem 3 (Datei `vdp_jump.jl`).

```
# Problem 3: feste Endzeit, Zustandsbeschränkung
set_lower_bound(tf, tff)
set_upper_bound(tf, tff)
@NLobjective(mod, Min, 0.5*tf/N*sum(((x1[k]+x1[k+1])/2.0)^2 + ((x2[k]+x2[k+1])
/2.0)^2 + (u[k])^2 for k=0:N-1))
# Zustandsbeschränkung
@NLparameter(mod, x2min == -0.4)
@NLconstraint(mod, c_x2min[k=0:N-1], (x2[k+1]+x2[k])/2.0 >= x2min)
# Aufruf Solver
optimize!(mod)
```

Für Problem 3 werden die Komponentenbeschränkungen für die nun wieder feste Endzeit $t_f = 4$ auf die ursprünglichen Werte zurückgesetzt. Ebenso wird wieder die Zielfunktion von Problem 1 verwendet. In Zeile 99 wird die untere Schranke (6) für die Zustandsgröße x_2 an den Kollokationspunkten (Intervallmittelpunkten) ausgewertet. Der Wert der unteren Schranke wird als Parameter `x2min` eingeführt.

Listing 6: Optimalsteuerung: Problem 4 (Datei `vdp_jump.jl`).

```
# Problem 4: feste Endzeit, freier Endzustand
set_value(x2min, -1000.0)
set_value(x10, 0.0)
set_value(p4, 0.0)
@NLobjective(mod, Min, 0.5*tf/N*sum(((x1[k]+x1[k+1])/2.0)^2 + ((x2[k]+x2[k+1])
/2.0)^2 for k=0:N-1) + 1e-6/(tf/N)*sum((u[k+1]-u[k])^2 for k=0:N-2))
# Aufruf Solver
optimize!(mod)
```

Für Problem 4 wird der Wert des Parameters `x2min` so gewählt, dass die Beschränkungen nicht mehr aktiv werden. Anschließend wird der Anfangswert `x10` modifiziert. Da für Problem 4 die

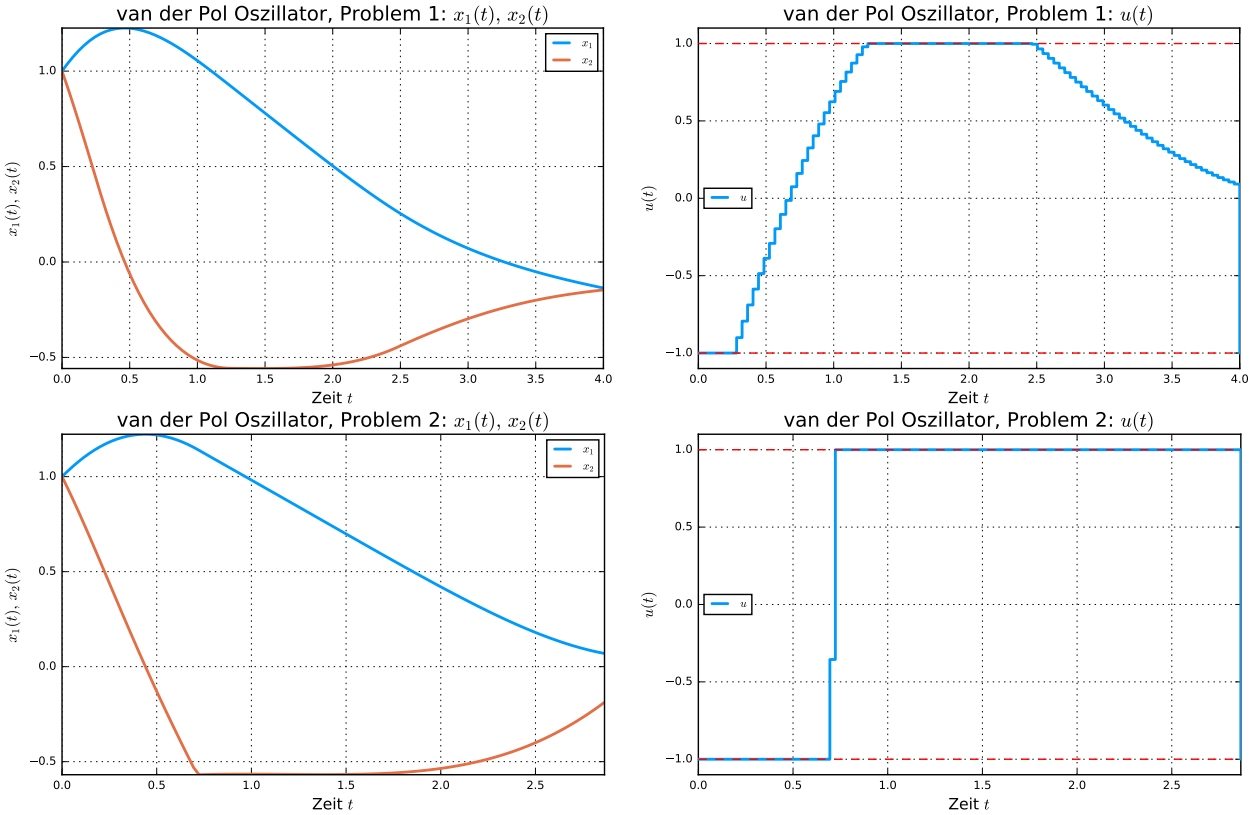


Abbildung 1: Gesteuerter VAN DER POL-OSZILLATOR: optimale Lösungen Problem 1 und 2.

Endzustandsbeschränkung (3d) entfallen soll, wird diese mit `p4=0` multipliziert. Die Zielfunktion ist nun (7) in zeitlich diskretisierter Form.

Der Regularisierungsterm

$$\frac{10^{-6}}{\Delta t} \sum_{k=0}^{N-2} \left(u^{k+1} - u^k\right)^2$$

reduziert numerisch bedingte Oszillationen der Steuergröße im singulären Lösungsabschnitt $t \in [\approx 2.45, 4]$.

Mit dem Befehl `include()` wird die Datei in REPL geladen und die Ausführung gestartet:

```
julia> include("vdp_jump.jl")
```

Die Abbildungen 1 und 2 zeigen die Lösungen der vier Probleme für $N = 99$ Diskretisierungsintervalle.

Literatur

[1] E. ARNOLD. *Numerische Methoden der Optimierung und Optimalen Steuerung*. Vorlesung Universität Stuttgart. 2019. URL: https://www.isys.uni-stuttgart.de/lehre/lehrveranstaltungen/nmopt (siehe S. 9, 11, 14).

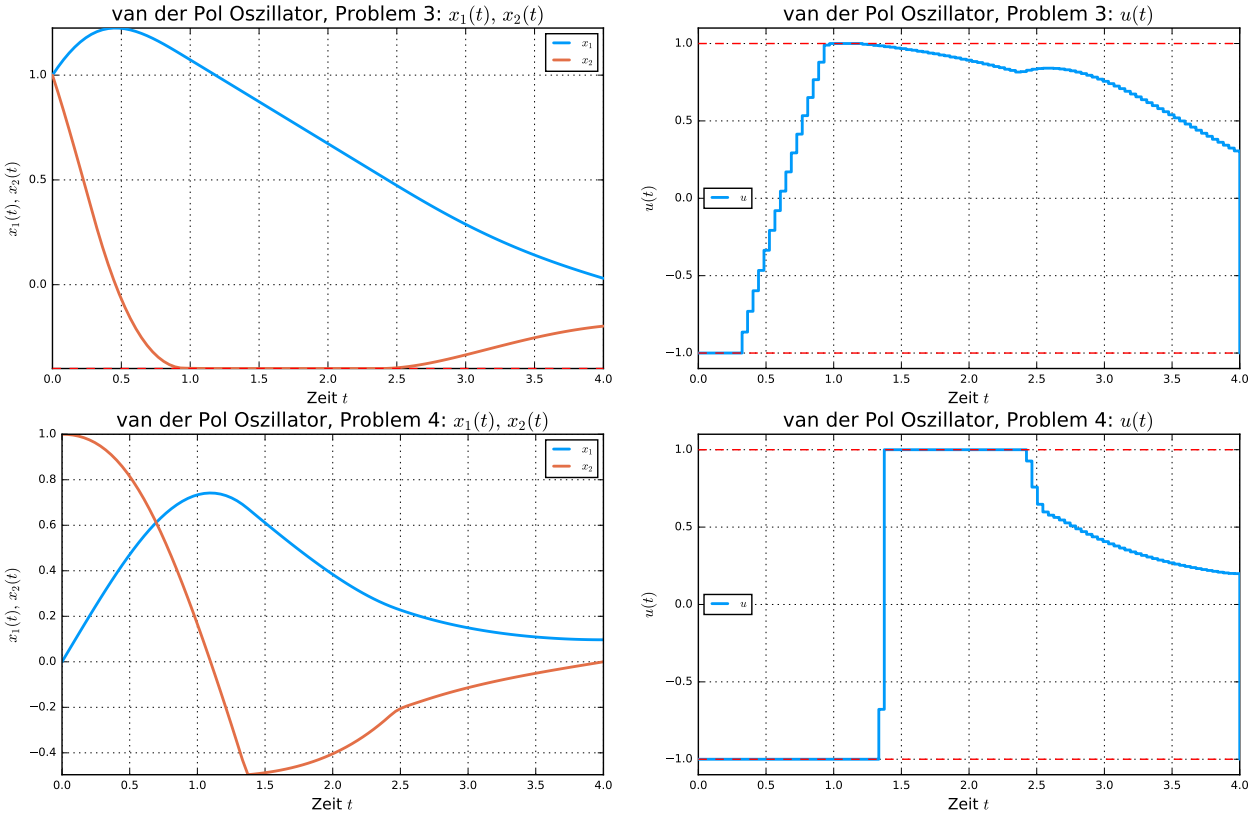


Abbildung 2: Gesteuerter VAN DER POL-OSZILLATOR: optimale Lösungen Problem 3 und 4.

[2] F. BORNEMANN. *Numerische lineare Algebra: Eine konzise Einführung mit MATLAB und Julia*. Wiesbaden: Springer, 2016. DOI: 10.1007/978-3-658-12884-5 (siehe S. 3).

[3] S. BOYD und L. VANDENBERGHE. *Introduction to Applied Linear Algebra - Vectors, Matrices, and Least Squares - Julia language companion*. 2018. URL: https://web.stanford.edu/~boyd/vmls/vmls-julia-companion.pdf (siehe S. 3).

[4] I. DUNNING, J. HUCHETTE und M. LUBIN. "JuMP: A modeling language for mathematical optimization". In: *arXiv:1508.01982 [math.OC]* (2015). URL: https://arxiv.org/abs/1508.01982 (siehe S. 3).

[5] C. KWON. *Julia programming for Operations Research*. Kwon, 2016. ISBN: 978-1533328793 (siehe S. 3).

[6] M. LUBIN und I. DUNNING. "Computing in Operations Research Using Julia". In: *INFORMS Journal on Computing* 27.2 (2015), S. 238–248. DOI: 10.1287/ijoc.2014.0623 (siehe S. 3).

[7] H. MAURER. *Tutorial on control and state constrained optimal control problems. Part I: Examples*. SADCO Summer School, Imperial College London. 2011. URL: https://hal.inria.fr/inria-00629518/PDF/Maurer_Part1.pdf (siehe S. 14).

[8] M. PAPAGEORGIOU, M. LEIBOLD und M. BUSS. *Optimierung: statische, dynamische, stochastische Verfahren für die Anwendung*. Berlin: Springer, 2012. DOI: 10.1007/978-3-540-34013-3 (siehe S. 11).

[9] M. SHERRINGTON. *Mastering Julia*. Packt Publishing, 2015 (siehe S. 3).

[10] A. Wächter und L. T. Biegler. “On the implementation of a primal-dual interior point filter line search algorithm for large-scale nonlinear programming”. In: *Mathematical Programming* 106.1 (2006), S. 25–57. DOI: 10.1007/s10107-004-0559-y. URL: http://cepac.cheme.cmu.edu/pasilectures/biegler/ipopt.pdf (siehe S. 6).