TransWORHP

Benutzerhandbuch / user guide

Steinbeis-Forschungszentrum für
Optimierung, Steuerung und Regelung

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Optimale Steuerung

Während bei Problemen der Nichtlinearen Optimierung ein Vektor von Variablen bestimmt wird, um eine Zielfunktion zu minimieren, sucht man bei Problemen der Optimalen Steuerung nach Vektoren von Funktionen (und Vektoren von Variablen), die ein Zielfunktional minimieren.

While a vector of variables has to be determined for nonlinear programming problems in order to minimize an objective function, for optimal control problems a vector of functions (and additional variables) has to minimize an objective functional.

1.1 Aufgabenstellung

Bevor das Optimalsteuerungsproblem formuliert wird, sollen zuerst einige Begriffe eingeführt werden.

Let us introduce some terms before formulating the optimal control problem.

Systemvariablen

Der Zustand eines Systems zu einem Zeitpunkt $t \in [t_0; t_f]$ lasse sich durch einen Vektor

$$x(t) = (x_1(t), \ldots, x_n(t))^T \in \mathbb{R}^n$$

ausdrücken, der *Zustandsvektor* genannt wird. Für $i = 1, \ldots, n$ heißen seine Einträge $x_i(t)$ die *Zustandsvariablen*.
Speziell für den Anfangszeitpunkt $t_0 \in \mathbb{R}$ nennt man $x(t_0)$ den *Anfangszustand*, und für den Endzeitpunkt $t_f \in \mathbb{R}$ entsprechend $x(t_f)$ den *Endzustand*. Ohne Einschränkung wird $t_0 = 0$ angenommen. Die Endzeit t_f kann fest vorgegeben oder frei wählbar sein.
Das Systemverhalten soll über einen *Steuervektor*

$$u(t) = (u_1(t), \ldots, u_m(t))^T \in \mathbb{R}^m$$

kontrolliert werden, der aus den *Steuervariablen* oder kurz *Steuerungen* $u_j(t)$, $j = 1, \ldots, m$ besteht.

Dabei sei $x \in C_p^1([0; t_f], \mathbb{R}^n)$ ein Vektor stückweise stetig differenzierbarer Funktionen und die Komponenten von $u \in C_p^0([0; t_f], \mathbb{R}^m)$ seien stückweise stetig.
Zusätzlich kann das Systemverhalten noch von *freien Parametern*

$$p = (p_1, \ldots, p_{\widehat{p}})^T \in \mathbb{R}^{\widehat{p}}$$

abhängen, die ebenfalls bestimmt werden müssen.

Variables of the System

The state of a system at time $t \in [t_0; t_f]$ may be expressed by a vector

$$x(t) = (x_1(t), \ldots, x_n(t))^T \in \mathbb{R}^n$$

which is called *state vector*. For $i = 1, \ldots, n$ its entries $x_i(t)$ are called *state variables*.

Especially for the initial time $t_0 \in \mathbb{R}$ we call $x(t_0)$ the *initial state*, and for the final time $t_f \in \mathbb{R}$ analogously $x(t_f)$ the *final state*. Without limitation we assume $t_0 = 0$. The final time t_f can be fixed or free.

The behavior of the system has to be controlled using a *control vector*

$$u(t) = (u_1(t), \ldots, u_m(t))^T \in \mathbb{R}^m,$$

consisting of *control variables* or simply *controls* $u_j(t)$, $j = 1, \ldots, m$.
Here, $x \in C_p^1([0; t_f], \mathbb{R}^n)$ is a vector of piecewise continuously differentiable functions, and the components of $u \in C_p^0([0; t_f], \mathbb{R}^m)$ are piecewise continuous.
Additionally the behaviour of the system can depend on *free parameters*

$$p = (p_1, \ldots, p_{\widehat{p}})^T \in \mathbb{R}^{\widehat{p}},$$

which have to be determined as well.

Systemdynamik

Die Änderungen, die der Systemzustand über die Zeit erfährt, lassen sich über ein System von Differentialgleichungen erster Ordnung

$$\dot{x}(t) = f(x(t), u(t), p, t), \quad t \in [0; t_f], \tag{1.1}$$

der *Dynamik des Systems*, ausdrücken. Wie gewohnt bezeichnet $\dot{x}(t)$ die Ableitung einer Variablen $x(t)$ nach der Zeit. Dabei sei die stetige Funktion $f : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^{\widehat{p}} \times [0; t_f] \to \mathbb{R}^n$ bezüglich x, u und p stetig differenzierbar.
Erfüllt ein Tupel $(x(t), u(t), p)$ die Systemdynamik (1.1), so heißt es *Lösung des Systems*.

Hängt die Funktion f nicht explizit von der Zeit ab, also $f = f(x(t), u(t), p)$, wie es bei mechanischen Problemen oft der Fall ist, so spricht von von einem *autonomen* System.

Dynamic of the System

The changes in time of the state of the system can be expressed by a system of differential equations of first order

the *dynamic of the system*. As usual $\dot{x}(t)$ denotes the derivative of a function $x(t)$ with respect to time. Here, the continuous function $f : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^{\widehat{p}} \times [0; t_f] \to \mathbb{R}^n$ is continuously differentiable with respect to x, u and p.
A tupel $(x(t), u(t), p)$ holding the dynamic of the system (1.1), is called *solution of the system*.

If the function f does not depend explicitly on time, i.e. $f = f(x(t), u(t), p)$, as it is common for mechanical problems, the system is called *autonomous*.

Randbedingungen

Der Anfangszustand $x(0)$ und der Endzustand $x(t_f)$ des Systems, oder nur einzelne Komponenten davon, können über *Randbedingungen* vorgeschrieben werden. Allgemein werden die Bedingungen über eine stetig differenzierbare Funktion $\omega : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^{\widehat{p}} \to \mathbb{R}^r$ gefordert ($r \leq 2n$):

$$\omega(x(0), x(t_f), p) = 0 \tag{1.2}$$

In der Praxis kennt man oft den vorgeschriebenen Anfangszustand $x_0 \in \mathbb{R}^n$ explizit. Hier genügt die Möglichkeit der Angabe von Randbedingungen in der Form

$$\begin{aligned} x(0) &= x_0, \\ \omega(x(t_f), p) &= 0, \end{aligned}$$

mit $\omega : \mathbb{R}^n \times \mathbb{R}^{\widehat{p}} \to \mathbb{R}^r, r \leq n$.

Boundary Conditions

The initial state $x(0)$ or the final state $x(t_f)$ of the system, or just individual components of these, can be limited with *boundary conditions*. Generally, these conditions are formulated using a continuously differentiable function $\omega : \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^{\widehat{p}} \to \mathbb{R}^r$ ($r \leq 2n$):

In practical problems the initial state $x_0 \in \mathbb{R}^n$ is often known explicitly. In these cases boundary conditions are formulated in the form

$$\begin{aligned} x(0) &= x_0, \\ \omega(x(t_f), p) &= 0, \end{aligned}$$

with $\omega : \mathbb{R}^n \times \mathbb{R}^{\widehat{p}} \to \mathbb{R}^r, r \leq n$.

Steuer- und Zustandsbeschränkungen

Control and State Constraints

Zusätzlich können für jeden Zeitpunkt t über Ungleichungsnebenbedingungen Beschränkungen an die Steuerungen $u(t)$ und die Zustände $x(t)$ vorgegeben werden. Über eine hinreichend oft stetig differenzier-

bare Funktion $g : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^{\widehat{p}} \to \mathbb{R}^l$ formuliert man *gemischte Beschränkungen* derart, dass

$$g(x(t), u(t), p) \leq 0, \quad t \in [0; t_f]. \tag{1.3}$$

Tritt die Steuerung $u(t)$ in einer Ungleichungsnebenbedingung nicht explizit auf,

$$g_i(x(t), p) \leq 0, \quad t \in [0; t_f],\ i \in \{1, \dots, l\},$$

mit $g_i : \mathbb{R}^n \to \mathbb{R}$, so spricht man von *Zustandsbeschränkungen*.
Hängt hingegen die Ungleichungsnebenbedingung nur von der Steuerung $u(t)$ ab, dann heißt

$$g_i(u(t)) \leq 0, \quad t \in [0; t_f],\ i \in \{1, \dots, l\},$$

mit $g_i : \mathbb{R}^m \to \mathbb{R}$ eine *Steuerbeschränkung*. Oft sind die Steuerungen $u_i(t)$ einfachen Box-Beschränkungen

$$u_{i,\min} \leq u_i(t) \leq u_{i,\max}, \quad t \in [0; t_f],\ i = 1, \dots, m,$$

unterworfen. Dann bezeichnet

$$U := [u_{1,\min}; u_{1,\max}] \times \cdots \times [u_{m,\min}; u_{m,\max}] \subset \mathbb{R}^m$$

den zulässigen *Steuerbereich*.

Die Zielfunktion

Die Kostenfunktion

$$I[x, u] = \phi(x(t_f), p) + \int_0^{t_f} f_0(x(t), u(t), p, t)dt \tag{1.4}$$

wird als *Zielfunktion* bezeichnet. Dabei seien $\phi : \mathbb{R}^n \times \mathbb{R}^{\widehat{p}} \to \mathbb{R}$ und $f_0 : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}^{\widehat{p}} \times [0; t_f] \to \mathbb{R}$ stetig differenzierbare Funktionen bezüglich aller Argumente.
Diese Darstellung der Zielfunktion wird als Bolza-Form bezeichnet. Natürlich können auch die Fälle $\phi \equiv 0$ (Lagrange-Form) oder $f_0 \equiv 0$ (Mayer-Form) auftreten. Insbesondere lässt sich die Zielfunktion (1.4) in diese Spezialfälle umtransformieren.

Speziell bei mechanischen Vorgängen treten in der Zielfunktion nur Funktionen $f_0 = f_0(x(t), u(t), p)$ auf, die nicht direkt von der Zeit abhängen.

Optimaler Steuerprozess

Der *optimale Steuerprozess* bezeichnet das Problem, Steuerungen $u(t)$ so zu bestimmen, dass die vorgegebene Zielfunktion (1.4) unter Einhaltung der Systemdynamik (1.1) und den *Zulässigkeitsbedingungen* (1.2) und (1.3) minimiert wird.

$$\begin{aligned} \min_{x,u,p,t_f} \quad & I[x, u] &&= \phi(x(t_f), p) + \int_0^{t_f} f_0(x(t), u(t), p, t)dt \\ \text{unter} \quad & \dot{x}(t) &&= f(x(t), u(t), p), \\ & \omega(x(0), x(t_f), p) &&= 0, \\ & g(x(t), u(t), p) &&\leq 0, \quad t \in [0; t_f]. \end{aligned} \tag{1.5}$$

Lösungen des Problems (1.5) lassen sich für feste Verfahrzeit t_f wie folgt charakterisieren:

Zulässige Lösung und optimale Lösung. Erfüllt eine Lösung $(x^\star, u^\star, p^\star)$ des Systems, bestehend aus einem Parametervektor $p^\star \in \mathbb{R}^{\widehat{p}}$, der Steuerung $u^\star : [0; t_f] \to \mathbb{R}^m$, und der daraus resultierenden Trajektorie $x^\star : [0; t_f] \to \mathbb{R}^n$ die für (1.5) genannten Zulässigkeitsbedingungen, so heißt die Lösung *zulässig*. Erfüllt die zulässige Lösung $(x^\star, u^\star, p^\star)$ außerdem

$$I[x^\star, u^\star] \leq I[x, u]$$

für alle zulässigen Lösungen (x, u, p), mit $p \in \mathbb{R}^{\widehat{p}}$, $u : [0; t_f] \to \mathbb{R}^m$ und $x : [0; t_f] \to \mathbb{R}^n$, dann heißt sie *optimale Lösung*.

In diesem Fall bezeichnet der Vektor $x^\star$ die *optimale Trajektorie* und $u^\star$ die *optimale Steuerung* des Problems (1.5).
Ist in der Aufgabenstellung die Endzeit t_f frei wählbar, muss zusätzlich noch die Endzeit optimal bestimmt werden. Durch Einführung eines weiteren freien Parameters p_0 kann jedoch das System auf ein System mit fester Endzeit transformiert werden.
Statt

$$\dot{x}(t) = f(x(t), u(t), p) \qquad \text{für } t \in [0, t_f]$$

verwende nun

$$\dot{x}(t) = f(x(t), u(t), p) \cdot p_0 \qquad \text{für } t \in [0, 1].$$

1.2 Berechnung optimaler Steuerungen durch direkte Verfahren

Bei *direkten Verfahren* wird durch eine Diskretisierung der Steuerungen und eventuell der Zustandsvariablen das unendlich dimensionale Optimalsteuerungsproblem (1.5) durch ein nichtlineares Optimierungsproblem mit endlich vielen Parametern approximiert.

1.2.1 Diskretisierung

Das Prinzip, mit dem sich das Integral der Zielfunktion und die Differentialgleichungen von Problem (1.5) numerisch auswerten lassen, besteht darin, an ausgewählten Stützstellen Näherungswerte für das Integral und die rechte Seite der Differentialgleichungen zu bestimmen.
Für die nicht notwendigerweise äquidistanten Stützstellen t_i gelte

$$0 = t_1 \leq t_2 \leq \cdots \leq t_l = t_f, \quad l \in \mathbb{N}.$$

Die Näherungen der Steuerung $u(t_i)$ und des Zustands $x(t_i)$ an den Gitterpunkten t_i seien mit u^i bzw. x^i bezeichnet:

$$u^i \approx u(t_i), \quad x^i \approx x(t_i)$$

Eine einfache Näherung für das Integral der Zielfunktion zwischen zwei Sützstellen t_i und t_{i+1} ist dann beispielsweise gegeben durch

$$\int_{t_i}^{t_{i+1}} f_0(x(t), u(t), p)dt \approx (t_{i+1} - t_i) f_0(x^i, u^i, p).$$

Allgemeiner erhält man daraus mit dem *Eulerschen Polygonzugverfahren*die Iterationsvorschrift für $i = 1, \ldots, l-1$:

$$\int_{t_1}^{t_{i+1}} f_0(x(t), u(t), p)dt \approx \int_{t_1}^{t_i} f_0(x(t), u(t), p)dt + (t_{i+1} - t_i) f_0(x^i, u^i, p)$$

Wählt man das Euler-Verfahren als *Integrationsverfahren* zur Lösung des Differentialgleichungssystems, besitzt die durch Diskretisierung approximierte Form von (1.5) die freien Parameter $(u^i)_{i=1,\dots,l}$ für die Steuerung und $(x^i)_{i=1,\dots,l}$ für den Zustand und lautet mit $h_i := t_{i+1} - t_i$, $i = 1, \dots, l-1$:

$$
\begin{array}{lrcl}
\min\limits_{x,u,p} & \phi(x^l) + \sum\limits_{i=1}^{l-1} h_i f_0(x^i, u^i, p) & & \\
\text{unter} & x^{i+1} & = & x^i + h_i f(x^i, u^i, p), \quad i = 1, \dots, l-1 \\
& \omega(x^1, x^l, p) & = & 0 \\
& g(x^i, u^i, p) & \leq & 0, \quad i = 1, \dots, l
\end{array}
\tag{1.6}
$$

Somit erhält man ein nichtlineares Optimierungsproblem

$$
\begin{array}{ll}
\min\limits_{z} & f(z) \\
\text{unter} & g_i(z) \leq 0 \quad i = 1, \dots, k
\end{array}
\tag{1.7}
$$

Dieses kann z.B. mit WORHP gelöst werden (TODO Referenz).

1.2.2 Integration der Systemdynamik

Das Euler-Verfahren lässt sich mit $f_i := f(x^i, u^i, p)$, $i = 1, \dots, l$ kompakter schreiben in der Form

$$0 = x^{i+1} - x^i - h_i f_i$$

Da mit WORHP ohnehin bereits Gleichungssysteme gelöst werden, können auch implizite Verfahren ohne Mehraufwand eingesetzt werden.

Das Trapezverfahren besitzt $\mathcal{O}(h^2)$:

$$0 = x^{i+1} - x^i - \frac{h_i}{2}(f_i + f_{i+1}) \tag{1.8}$$

Das Hermite-Simpson-Verfahren benötigt eine weitere Funktionsauswertung an der Zwischenstelle $t_{i+\frac{1}{2}} = \frac{1}{2}(t_i + t_{i+1})$ und besitzt $\mathcal{O}(h^4)$:

$$
\begin{array}{rcl}
0 & = & x^{i+\frac{1}{2}} - \frac{1}{2}(x^{i+1} + x^i) - \frac{h_i}{8}(f_i - f_{i+1}) \\
0 & = & x^{i+1} - x^i - \frac{h_i}{6}(f_{i+1} + 4 f_{i+\frac{1}{2}} + f_i)
\end{array}
\tag{1.9}
$$

Das Hermite-Simpson-Verfahren lässt sich durch dieses Butcher-Tableau darstellen:

0	0	0	0
1/2	5/24	1/3	−1/24
1	1/6	2/3	1/6
	1/6	2/3	1/6

Ausgeschrieben heißt das:

$$\begin{aligned} f_i &= f(x^i, t^i) \\ f_{i+\frac{1}{2}} &= f(\underbrace{x^i + \frac{5}{24}h_i f_i + \frac{1}{3}h_i f_{i+\frac{1}{2}} - \frac{1}{24}h_i f_{i+1}}_{x^{i+\frac{1}{2}}}, t^i + \frac{h_i}{2}) \\ f_{i+1} &= f(\underbrace{x^i + \frac{1}{6}h_i f_i + \frac{2}{3}h_i f_{i+\frac{1}{2}} + \frac{1}{6}h_i f_{i+1}}_{x^{i+1}}, t^i + h_i) \\ x^{i+1} &= x^i + \frac{h_i}{6}(f_i + 4f_{i+\frac{1}{2}} + f_{i+1}) \end{aligned}$$

Dabei ist die letzte Zeile die Simpson-Regel.
Aus erster Klammer folgt:

$$x^{i+\frac{1}{2}} = x^i + \frac{5}{24}h_i f_i + \frac{1}{3}h_i f_{i+\frac{1}{2}} - \frac{1}{24}h_i f_{i+1}$$

Aus zweiter Klammer folgt:

$$f_{i+\frac{1}{2}} = \frac{3}{2h_i}\left(x_{i+1} - x_i - \frac{h_i}{6}f_i - \frac{h_i}{6}f_{i+1}\right)$$

Einsetzen von Gleichung 2 in Gleichung 1 ergibt die Hermite-Gleichung:

$$x^{i+\frac{1}{2}} = \frac{1}{2}x^i + \frac{1}{2}x^{i+1} + \left(\frac{5}{24} - \frac{1}{12}\right)h_i f_i + \left(-\frac{1}{24} - \frac{1}{12}\right)h_i f_{i+1}$$

Struktur der Systemdynamik

Die gegebene Darstellung der Integrationsschemata liefert dünn-besetzte Strukturen in den Ableitungsmatrizen.

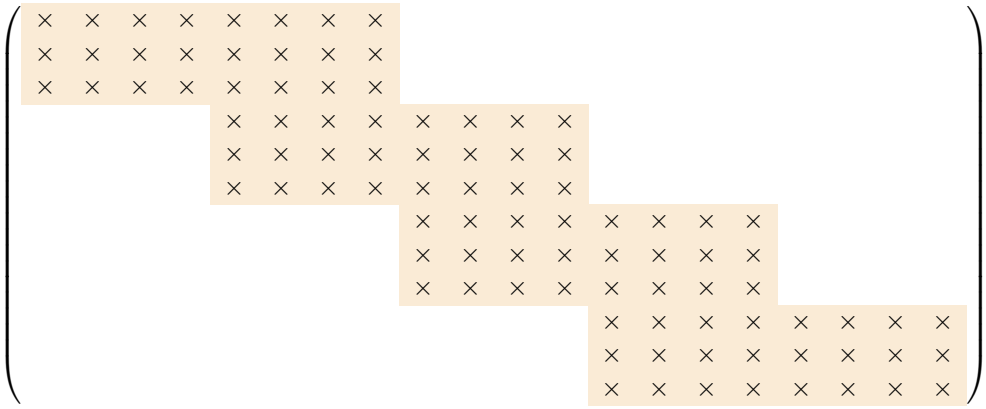
Integriere z.B. dieses System $l = 5$ diskreten Punkten:

$$\begin{aligned} \dot{x}_1 &= x_2 \\ \dot{x}_2 &= u \\ \dot{x}_3 &= u^2 \end{aligned}$$

Das (NLP) besitzt dann diesen Variablenvektor

$$x = \begin{pmatrix} x^1 \\ u^1 \\ x^2 \\ u^2 \\ \dots \\ x^l \\ u^l \end{pmatrix} \text{ mit } x^i = \begin{pmatrix} x_1^i \\ x_2^i \\ x_3^i \end{pmatrix}$$

Da in Gleichung (1.8) nur Variablen zum diskreten Punkt i und $i+1$ verbunden werden, hat die Jacobi-Matrix der Nebenbedingungen diese Gestalt:



Außerdem besitzt auch die Ableitung der rechten Seite des Differentialgleichungssystems eine interne Struktur:

$$\begin{pmatrix} 0 & \times & 0 & 0 \\ 0 & 0 & 0 & \times \\ 0 & 0 & 0 & \times \end{pmatrix}$$

Dass lässt sich auf die Jacobi-Matrix übertragen. Leitet man Gleichung (1.8) nach x^i bzw. x^{i+1} ab,

entstehen (negative und positive) Einheitsmatrizen, die sich mit der eben genannten Struktur überlagern:

$$\begin{pmatrix}
-1 & \times & & & 1 & \times & & & & & & & & & & & & & & \\
 & -1 & & \times & & 1 & & \times & & & & & & & & & & & & \\
 & & -1 & \times & & & 1 & \times & & & & & & & & & & & & \\
 & & & & -1 & \times & & & 1 & \times & & & & & & & & & & \\
 & & & & & -1 & & \times & & 1 & & \times & & & & & & & & \\
 & & & & & & -1 & \times & & & 1 & \times & & & & & & & & \\
 & & & & & & & & -1 & \times & & & 1 & \times & & & & & & \\
 & & & & & & & & & -1 & & \times & & 1 & & \times & & & & \\
 & & & & & & & & & & -1 & \times & & & 1 & \times & & & & \\
 & & & & & & & & & & & -1 & & \times & & 1 & & \times & & \\
 & & & & & & & & & & & & -1 & \times & & & 1 & \times & & \\
 & & & & & & & & & & & & & -1 & & \times & & 1 & & \times
\end{pmatrix}$$

Analog lässt sich auch die Dünn-Besetztheit des Hermite-Simpson-Verfahrens angeben.

1.2.3 Integration des Lagrange-Terms

Ein Optimalsteuerungsproblem mit Zielfunktional

$$I[x,u] = \phi(x(t_f),p) + \int_0^{t_f} f_0(x(t),u(t),p,t)\,dt$$

lässt sich stets in Mayer-Form umwandeln ($f_0 \equiv 0$).
Alternativ lässt es sich der Lagrange-Term auch diskretisiert berechnen. Verwende die Kurzschreibweise $f_0^i := f_0(x^i,u^i,p,t_i)$, $i = 1,\dots,l$.
Bei Euler-Verfahren:

$$I[x,u] = \phi(x^l,p) + \sum_{i=1}^{l-1} h_i f_0^i$$

Beim Trapezverfahren:

$$I[x,u] = \phi(x^l,p) + \sum_{i=1}^{l-1} \frac{h_i}{2}\left(f_0^i + f_0^{i+1}\right)$$

Beim Hermite-Simpson-Verfahren:

$$I[x,u] = \phi(x^l,p) + \sum_{i=1}^{l-1} \frac{h_i}{6}\left(f_0^i + 4f_0^{i+\frac{1}{2}} + f_0^{i+1}\right)$$

Auswertung an den Zwischen-Stützstellen des Hermite-Simpson-Verfahrens:

$$\int_{t_i}^{t_{i+\frac{1}{2}}} f_0(x(t),u(t),p,t)\,dt \approx \frac{h_i}{24}\left(5f_0^i + 8f_0^{i+\frac{1}{2}} - f_0^{i+1}\right)$$

Die Zahl der Optimierungsvariablen des (NLP) lässt sich reduzieren, wenn Mehrfach-Schießen eingeführt wird.

1.2.4 Approximation der Steuerungen und des Zustands

Durch die Transformation des Optimalsteuerproblems auf ein diskretes Problem ist eine Lösung lediglich an den Stützstellen verfügbar. Um Werte der Lösung zwischen den Stützstellen zu berechnen, interpoliert man die Lösung geeignet.
Es sei $D_i(t)$ eine Basis von kubischen Splines.
Approximiere ($n_1 = 2l$)

$$x(t) \approx \widetilde{x}(t) = \sum_{i=1}^{n_1} \gamma_i D_i(t)$$

Der Spline soll durch die Stützstellen laufen

$$\widetilde{x}(t_i) = x^i, i = 1, ..., l$$

und die rechte Seite des Differentialgleichungssystems erfüllen

$$\dot{\widetilde{x}}(t_i) = f_i, i = 1, ..., l$$

Ebenso für Steuerungen

$$u(t) \approx \widetilde{u}(t) = \sum_{i=1}^{n_2} \beta_i C_i(t)$$

Trapez-Verfahren: $C_i(t)$ sind lineare B-Splines mit $n_2 = l$

$$\widetilde{u}(t_i) = u^i, k = 1, ..., l$$

Hermite-Simpson: quadratische B-Spline: $n_2 = 2l - 1$

$$\widetilde{u}(t_i) = u^i, i = 1, ..., l$$

$$\widetilde{u}(t_{i+1/2}) = u^{i+1/2}, i = 1, ..., l - 1$$

1.2.5 Fehlerabschätzung

Aus der Bachelorarbeit von Matthias Rick, angepasst nach aktuellem Stand von TransWORHP

Vorteil: Direkte Vefahren brauchen keine notwendigen Bedingungen (Pontryagin).
Nachteil: Genauigkeit der Lösung kann nicht mit notwendigen Bedingungen überprüft werden.

Hier: Annahme, dass $\widetilde{u}(t)$ optimal ist. Nur für $h_k \to 0$ gehen die KKT-Bedingungen des NLP in die notwendigigen Bedinungen des OCP über. Schätze Fehler zwischen $\widetilde{x}(t)$ und $x(t)$ ab. Fehlerabschätzung für $u(t)$ als C^0-Spline, $x(t)$ als C^1-Spline aus Spline-Lösung $\widetilde{u}(t)$, $\widetilde{x}(t)$.
Im Folgenden bezeichne $x(t_k)$ die exakte Lösung, $\tilde{x}(t_k)$ eine durch Splines genäherte Lösung und x_k die diskrete Lösung an der Stelle t_k. Analoges gilt für die Steuerung u.
Werden Optimalsteuerungsprobleme mit TransWORHPgelöst, so erhält man diskrete Punkte als Lösung. Als *globalen Diskretisierungsfehler* an der Stelle t_k bezeichnet man die Abweichung dieser von der exakten Lösung

$$|x_k - x(t_k)|\,.$$

Der *lokale Diskretisierungsfehler* ist die Differenz der berechneten Lösung x_{k+1} und der Lösung der Differentialgleichung, die durch den vorherigen Punkt verläuft. Hierbei bezeichne t_{k+1} die Stelle $t_k + h_k$. Die Fehlerordnung ergibt sich als $\mathcal{O}(h^p)$, wobei p die Ordnung des Diskretisierungsverfahren bezeichne, für den globalen und $\mathcal{O}(h^{p+1})$ für den lokalen Fehler. Beispiele für die Ordnung der Diskretisierungsverfahren sind $p = 2$ für das Trapezverfahren und $p = 4$ für Hermite-Simpson. Der Diskretisierungsfehler ist nach [**?**] Betts allgemein von der Form

$$\epsilon_k \approx ||c_k h^{p+1}||,$$

wobei c_k geeignete Faktoren sind. Man erkennt, dass für $h \to 0$ der Fehler gegen Null strebt. Dies motivierte den Ansatz, eine sehr hohe Anzahl an Gitterpunkten zu wählen, um somit h klein werden zu lassen und einen kleinen Fehler zu bekommen. Unter Umständen kann sich der Exponent noch um den Wert r reduzieren, wenn Beschränkungen aktiv werden, sodass sich das Folgende ergibt

$$\epsilon_k \approx ||c_k h^{p-r+1}||. \tag{1.10}$$

Dieses Verhalten wird als *Ordnungsreduktion* bezeichnet. Da sich in TransWORHPdiese Darstellung nicht findet, wird im Folgenden zwar auf die Reduktion eingegangen, allerdings bei der praktischen Umsetzung nicht betrachtet.
Da die exakte Lösung nicht bekannt ist, benötigen wir eine Abschätzung für den Diskretisierungsfehler. Hierzu betrachten wir die exakte Lösung x an der Stelle t_{k+1}

$$\begin{aligned} x(t_{k+1}) = x(t_k + h_k) &= x(t_k) + \int_{t_k}^{t_k+h_k} \dot{x}(t)\,\mathrm{d}t \\ &= x(t_k) + \int_{t_k}^{t_k+h_k} f(x(t), u(t), t)\,\mathrm{d}t. \end{aligned}$$

Unter dem Integral stehen die unbekannten Lösungen für x und u, sodass wir diese durch Splines approximieren und uns eine neue Zustandsfunktion $\hat{x}(t_{k+1})$ definieren als

$$\hat{x}(t_{k+1}) := x(t_k) + \int_{t_k}^{t_k+h_k} f(\tilde{x}(t), \tilde{u}(t), t)\,\mathrm{d}t.$$

Zur besseren Lesbarkeit sei im Folgenden $\tilde{f}(t) := f(\tilde{x}(t), \tilde{u}(t), t)$.

$$\begin{aligned} \hat{x}(t_{k+1}) &= x(t_k) + \int_{t_k}^{t_k+h_k} \tilde{f}(t)\,\mathrm{d}t \\ &= x(t_k) + \int_{t_k}^{t_k+h_k} \dot{\tilde{x}}(t)\,\mathrm{d}t - \int_{t_k}^{t_k+h_k} \dot{\tilde{x}}(t)\,\mathrm{d}t + \int_{t_k}^{t_k+h_k} \tilde{f}(t)\,\mathrm{d}t \\ &= x(t_k) + \tilde{x}(t_{k+1}) - \tilde{x}(t_k) - \int_{t_k}^{t_k+h_k} \dot{\tilde{x}}(t) - \tilde{f}(t)\,\mathrm{d}t \end{aligned}$$

Wie man sich leicht überlegen kann, ist die Differenz $|x(t_k) - \tilde{x}(t_k)| = 0$, da dies eine Bedingung des Splines ist. Somit folgt

$$\tilde{x}(t_{k+1}) - \hat{x}(t_{k+1}) = \int_{t_k}^{t_k+h_k} \dot{\tilde{x}}(t) - \tilde{f}(t)\,\mathrm{d}t.$$

Nun lässt sich der Fehler konkret abschätzen. Dazu betrachten wir den Betrag jeder Komponente der Zustandsfunktion. So gilt für die i-te Komponente

$$\begin{aligned}|(\tilde{x}(t_{k+1}))_i - (\hat{x}(t_{k+1}))_i| &= \left| \int_{t_k}^{t_k+h_k} (\dot{\tilde{x}}(t))_i - (\tilde{f}(t))_i \, \mathrm{d}t \right| \\ &\leq \int_{t_k}^{t_k+h_k} \left| (\dot{\tilde{x}}(t))_i - (\tilde{f}(t))_i \right| \, \mathrm{d}t\end{aligned}$$

Somit können wir den *absoluten* lokalen Diskretisierungsfehler für die i-te Komponente im k-ten Intervall $\eta_{i,k}$ definieren als

$$\eta_{i,k} := \int_{t_k}^{t_{k+1}} |\varepsilon_i(t)| \, \mathrm{d}t \tag{1.11}$$
$$\text{wobei } \varepsilon(t) := \dot{\tilde{x}}(t) - f(\tilde{x}(t), \tilde{u}(t), t)$$

Weiterhin können wir den *relativen* lokalen Diskretisierungsfehler ϵ_k im k-ten Intervall definieren. Dieser ergibt sich als das gewichtete Maximum über alle Komponenten als

$$\epsilon_k := \max_i \frac{\eta_{i,k}}{(\omega_i + 1)} \tag{1.12}$$

mit den Gewichten

$$\omega_i := \max_k \left\{ |(\tilde{x}(t_k))_i| \, , \left| (\dot{\tilde{x}}(t_k))_i \right| \right\} .$$

In (1.12) sorgt die 1 dafür, dass nicht durch Null geteilt wird, falls $\omega_i = 0$.

Ordnungsreduktion

Um die Ordnungsreduktion zu verdeutlichen, betrachten wir ein Intervall, in welches $I \in \mathbb{N}$ neue Punkte eingefügt werden sollen. Der Fehler im alten Intervall sei mit θ und der Fehler im neuen mit η bezeichnet, welche sich nach (1.10) ergeben. Nach [**?**] Betts treffen wir die Annahme, dass die Konstanten c und r auf dem alten und dem neuen Intervall gleich sind. Damit ergeben sich

$$\theta = ch^{p-\hat{r}+1} \quad \text{und} \quad \eta = c \left(\frac{h}{1+I} \right)^{p-\hat{r}+1} .$$

Formt man beide Ausdrücke nach c um und setzt sie gleich, ergibt sich

$$\hat{r} = p + 1 - \frac{\log(\theta) - \log(\eta)}{\log(1+I)}.$$

Da die Ordnungsreduktion eine ganze Zahl sein soll, wird sie wie folgt abgeschätzt

$$r = \max\{0, \min\{\text{runden}(\hat{r}), p\}\}.$$

Damit gilt beispielsweise für die Reduktion beim Trapezverfahren, dass $r \in \{0, 1, 2\}$.

Diese Berechnung ist nicht in TransWORHPintegriert. Es ist allerdings möglich über das XML-File eine globale Reduktion der Ordnung anzugeben.

Neues Gitter

Das vorrangige Ziel ist es, dass der Fehler auf dem neuen Gitter kleiner ist als auf dem vorherigen Gitter. Damit man allerdings schon vorher abschätzen kann, wie sich der Fehler in einem Intervall durch das Hinzufügen einer gewissen Anzahl an Punkten ändert, wird eine Abschätzung hergeleitet.
Setzt man (1.10) und (1.12) gleich, so erhalten wir

$$\|c_k\| = \frac{1}{h^{p-r_k+1}} \max_i \frac{\eta_{i,k}}{(\omega_i+1)}$$

wobei r_k die Ordnungsreduktion im k-ten Intervall bezeichnet. Werden nun I_k Punkte dem Intervall zugefügt, so gilt für den Fehler der neuen Intervalle

$$\epsilon_k \approx \|c_k\| \left(\frac{h}{1+I_k}\right)^{p-r_k+1} = \max_i \frac{\eta_{i,k}}{(\omega_i+1)} \left(\frac{1}{1+I_k}\right)^{p-r_k+1}. \tag{1.13}$$

Dies ist eine Abschätzung, mit der man vorhersagen kann, wie sich der Fehler im neuen Intervall verringert, wenn I_k Punkte eingefügt werden. Wie bereits angemerkt, wird in TransWORHP keine Ordnungsreduktion betrachtet, sodass sich (1.13) vereinfacht zu

$$\epsilon_k \approx \|c_k\| \left(\frac{h}{1+I_k}\right)^{p+1} = \max_i \frac{\eta_{i,k}}{(\omega_i+1)} \left(\frac{1}{1+I_k}\right)^{p+1}.$$

Zuletzt können wir den durchschnittlichen Fehler $\overline{\epsilon}$ über alle Intervalle definieren. Dieser ist der Mittelwert aller relativen Fehler

$$\overline{\epsilon} := \frac{1}{N_t} \sum_{k=1}^{N_t} \epsilon_k.$$

Dieser Wert ist nützlich um zu bestimmen, wie gleichverteilt der Fehler über alle Intervalle ist.

1.2.6 Fehlerabschätzung 2

Eine weitere Möglichkeit den Diskretisierungsfehler abzuschätzen besteht darin das Problem auf zwei Arten zu diskretisieren. Hierzu müssen zwei verschiedene Verfahren gewählt werden, die sich in ihrer Ordnung unterscheiden. Es muss hierbei darauf geachtet werden, dass das Verfahren, für welches der Fehler bestimmt wird, eine nidrigere Ordnung hat als das andere Verfahren. Somit lässt sich beispielsweise das Euler-Verfahren mit dem Trapez-Verfahren abschätzen.
Die konkrete Fehlerabschätzung ist dann das Maximum der Differenz der einzelnen Komponenten der Dynamik.

$$\epsilon_k = \max_i \left| (x(t_k+h_k))_i^{[1]} - (x(t_k+h_k))_i^{[2]} \right|, \quad \text{im } k\text{-ten Intervall}$$

wobei $[\cdot]$ die verschiedenen Verfahren andeuten soll.
Auf diese Weise lässt sich der Fehler deutlich schneller ausrechnen. Allerdings funktioniert das Verfahren nur, wenn es ein Verfahren höherer Ordnung gibt. Deshalb lässt sich in TransWORHPder Fehler für das Hermite-Simpson-Verfahren nicht auf diese Art bestimmen.

1.2.7 Gitteranpassung

Der Algorithmus der Gitteranpassung gliedert sich in mehrere Schritte, die im Weiteren genau erklärt werden, wobei sich an [**?**] Betts orientiert wurde. Der erste Schritt sieht dabei wie folgt aus

i. Diskretisiere Problem mit äquidistantem Gitter

ii. Berechne Diskretisierungsfehler

iii. Füge neue Gitterpunkte ein

Diskretisierung

Zunächst muss das Problem diskretisiert werden, was bereits ausführlich beschrieben wurde. Als Startgitter wählt man eine relativ grobe Folge von Gitterpunkten, um den Rechenaufwand zu Beginn gering zu halten. Anschließend wird das Problem auf diesem Gitter mit TransWORHPgelöst.

Berechnung des Diskretisierungsfehlers

Sobald eine erste Lösung des Problems besteht, wird der Diskretisierungsfehler nach (1.12) berechnet. Dazu muss zunächst der Zustand durch einen kubischen und die Steuerung durch einen linearen Spline approximiert werden. Für die Steuerung ist der Funktionswert am linken und rechten Gitterpunkt bekannt, sodass ein linearer Spline erzeugt werden kann. Beim Zustand sind ebenfalls die Funktionswerte bekannt, allerdings wird für einen kubischen Spline die Steigung an den entsprechenden Stellen benötigt, welche wir aus der Dynamik des Systems am linken und rechten Punkt erhalten.
Anschließend wird (1.11) ausgewertet. Um einen möglichst genauen Wert für das Integral zu erhalten, verwenden wir das Romberg-Verfahren.

Erstellung des neuen Gitters

Als nächstes wird das geschickte Einfügen von neuen Gitterpunkten behandelt. Dies ist der entscheidende Schritt des gesamten Algorithmus und wird daher ausführlich beschrieben.
Man muss sich zunächst überlegen, wie viele Punkte maximal ins neue Gitter eingefügt werden sollen, denn es ist sicherlich nicht sinnvoll mit einem Gitter mit beispielsweise $N_t = 10$ Punkten zu starten und auf ein Gitter mit $N_t = 1000$ zu wechseln, denn dies würde die bestehenden Intervallen zu fein werden lassen. Deshalb ist es ein sinnvoller Ansatz, dass man das neue Gitter höchstens doppelt so fein werden lässt. Dies bedeutet, dass sich aus N_t Punkten höchstens $2 \cdot N_t - 1$ Punkte ergeben können. Das wäre genau dann der Fall, wenn jedes Intervall halbiert würde.
In der Praxis stellt man fest, dass es Intervalle geben wird, in denen der Fehler deutlich größer ist, sodass dort eine feinere Diskretisierung wünschenswert ist. Um zu verhindern, dass die maximale Anzahl an neuen Punkten in ein einziges Intervall eingefügt wird, führen wir eine Schranke M_1 ein, die den Höchstwert an neuen Punkten pro Intervall festlegt. Nach [**?**] Betts wählen wir $M_1 = 5$. Eine Folge dieser Beschränkung ist, dass das neue Gitter in einem Schritt nicht zwangsläufig maximal fein wird, d.h. $2 \cdot N_t - 1$ Punkte hat. Dies ist gewünscht, da bei zu kleinen Schrittweiten auch verstärkt numerische Probleme auftreten können. Es werden nun solange Punkte in das Intervall mit dem größten Diskretisierungsfehler eingefügt,

bis

$$\begin{aligned} &\text{mindestens } M' \text{ Punkte eingefügt wurden, wobei } M' \geq \min\{M_1, 0.1 \cdot N_t\} \textbf{ und} \\ &\quad \text{der maximaler Fehler ist kleiner als Schranke oder} \\ &\quad \text{der geschätzter Fehler ist kleiner als } 0.1 \cdot \text{Schranke oder} \\ &\quad N_t - 1 \text{ Punkte wurden eingefügt oder} \\ &\quad M_1 \text{ wurden in ein Intervall eingefügt} \end{aligned} \tag{1.14}$$

wobei man beachten muss, dass sich der Fehler in einem Intervall durch das Hinzufügen zusätzlicher Punkte entsprechend (1.13) verändert.
Sobald man weiß, wie viele Punkte in welches Intervall eingefügt werden sollen, stellt sich die Frage, an welcher Stelle die Punkte eingefügt werden. Wird beispielsweise festgestellt, dass in ein Intervall I_k neue Punkte eingefügt werden sollen, so könnte man die Punkte normal-, exponential- oder auch gleichverteilt einfügen. Wir entscheiden uns dazu, die neuen Gitterpunkte gleichverteilt, d.h. äquidistant einzufügen, da wir innerhalb des Intervalls kein Wissen über die Struktur einbringen können.

Erneute Optimierung

Nachdem das neue Gitter erstellt wurde, wird das Problem erneut mit TransWORHPgelöst. Um die Konvergenz zu beschleunigen, wird die bereits im vorherigen Schritt berechnete Lösung als Startschätzung genommen, sodass im besten Fall nur wenige Iterationen notwendig sind. Je nach Feinheit des Gitters stellt man fest, dass TransWORHPbereits nach einem Schritt eine optimale Lösung berechnet. Dabei ist zu beachten, dass an den neuen Stützstellen die bereits bestimmte Lösung durch TransWORHPinterpoliert wird. Anschließend wird erneut der Diskretisierungsfehler bestimmt und neue Punkte eingefügt. Dies wird solange wiederholt, bist der größte Fehler über alle Intervalle kleiner als eine vorgegebene Schranke ist.
Der beschriebe Vorgang ist als Ablaufdiagramm in Abbildung 1.1, welche sich im Anhang befindet, aufgezeigt. Im Weiteren wird der Algorithmus an Beispielen getestet.

Ablaufdiagramm

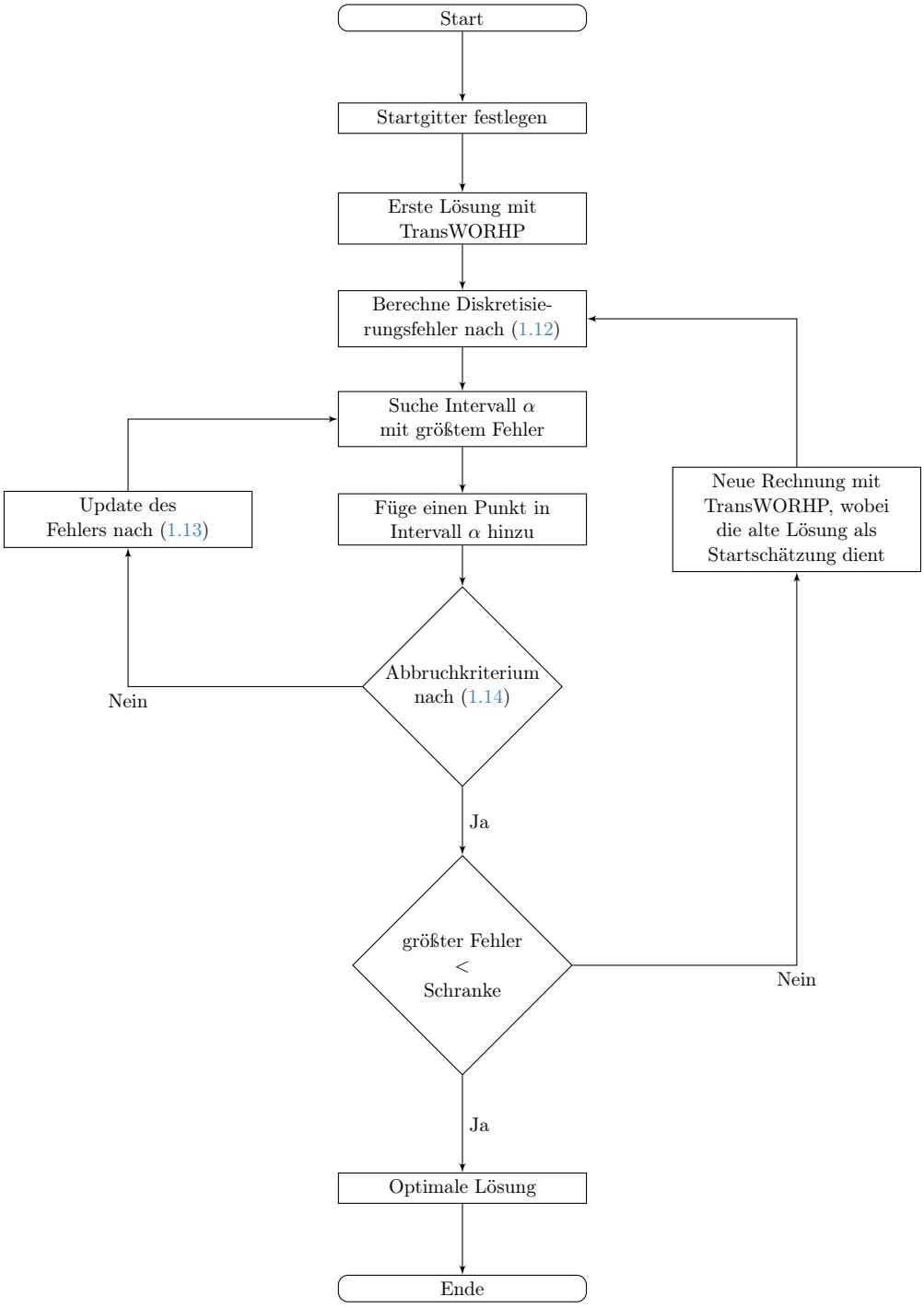


Abbildung 1.1: Ablaufdiagramm der Gitteranpassung

2 Starting with TransWORHP

Für Anwender wird TransWORHP als Bibliothek mit mehreren einfachen und einigen größeren Beispielen ausgeliefert. Um eigene Optimalsteuerprobleme zu implementieren muss von der Klasse `TransWORHP` abgeleitet werden.
Für Entwickler steht der gesamte Quellcode zur Verfügung.

For users TransWORHP is generally shipped as a library, with some trivial and advanced examples. You have to overload the base class `TransWorhp` in order to implement your own optimal control problems.
For developers there is also the full source code available.

2.1 TransWORHP for linux users

Laden Sie sich die aktuelle Version von TransWORHP von der Homepage herunter und entpacken Sie die tgz Dateien. Die Bibliotheken für WORHP und TransWORHP befinden sich im Order `lib` und die header-Dateien liegen im Ordner `include`.
Kompilieren Sie die Beispiel-Probleme und das Tutorial:

Download the current version from the TransWORHP web page and extract the tgz file. You find the libraries for WORHP and TransWORHP in the `lib` folder, and the relevant header files in the `include` folder.
Compile the examples from the folders `example` and `tutorial`:

```
make
```

Wenn keine Fehler auftraten, liegen die fertig kompilierten Dateien in den Ordnern `example_release` and `tutorial_release`.

If everything works fine, you will get the executables in the folders `example_release` and `tutorial_release`.

If the SDL2 library is missing, you will have to install the package libsdl2-dev or compile it on your own from https://www.libsdl.org/download-2.0.php.

Wenn Sie keine grafische Ausgabe benötigen, benutzen Sie die TransWORHP Version ohne Grafik. Kompilieren Sie:

If you don't care about the output, there is a smaller version of TransWORHP which has no graphical output. Just compile our standard example:

```
make make-tutorial-mini
```

Dieser Befehle erstellt die Datei `tutorial_release/spline4_mini`, welche nicht OpenGL oder SDL2 benötigt.

You'll get the file `tutorial_release/spline4_mini`, which does not depend on OpenGL or SDL2.

- tutorial: schrittweise Entwicklung des Spline-Problems
- example: größere Beispiele (z.B die Laufkatze), vgl. Kapitel 6

- tutorial: sources for stepwise example of a spline problem
- example: advanced examples (try the container crane = laufkatze), see chapter 6

2.2 TransWORHP for linux developers

TODO.

2.3 Running TransWORHP

Welchen Sie ins Verzeichnis `example_release` und tippen Sie

Change into `example_release` and try

```
laufkatze
```

Don't forget to copy your license file `worhp.lic` to this folder!

Das Programm kann über die XML-Dateien von WORHP und TransWORHP angepasst werden.
Jedes Programm versteht den Parameter
ohne grafische Ausgabe:

The program behavouir can be modified using `worhp.xml` and `transworhp.xml`
Each program understands these parameters:
no graphics:

```
laufkatze -p
```

Anzahl der diskreten Punkte verändern (hier 100 Gitterpunkte)

Change number of discrete points (here to 100)

```
laufkatze -n 100
```

2.3.1 Structure

Have a look at the DG-Window of any run, where you can see best, how constraints and variables are ordered!

- blue = any value
- grey = 0
- red = -1
- green = +1

Variables are ordered like:

- states of first discrete point,[1]
- controls of first discrete point,
- states of second discrete point,
- controls of second discrete point,
- ...
- free parameters

[1] if using hermite simpson you will have the additional point (states and controls) inbetween

Constraints are ordered like:

- ode constraints connecting first and second point [2]
- ode constraints connecting second and 3rd point
- ...
- non-trivial boundary values in given order (using the rand-function) rand is german for boundary :-(
- mixed state and control constraints for first point (using neben)
- mixed state and control constraints for 2nd point ...

If you move the mouse to the top of the window, a menu appears. Try selecting Data->NLP Constraints (and press RETURN to activate... it's a bug) and you will see all WORHP-variables and constraints with their boundaries on the console.

2.3.2 Classes

The class TWfolder contains all worhp-objects. In WorhpLoop() you will find the reverse communication, where objective and constraints and their derivatives are provided.
Optimal Control Problems are implemented as subclass of class TransWorhp. An instance is then added to the TWfolder. Note, that you can add several problems to the folder and generate huge problems. `example_release/laufkatze_phase` does this nicely!
In the reverse communication then, the objective is calculated as the sum of all problems in the folder.

2.3.3 XML

In `transworhp.xml` we also modify some worhp-parameters:

```
<WORHP param="worhp.xml">
    <USERDF>1</USERDF>
    <USERDG>1</USERDG>
    <USERHM>1</USERHM>
</WORHP>
```

Set everything to 0 to keep worhp.xml-settings.

[2] if using hermite simpson you will have hermite equation of 1. ode, simpson eq. of 1. ode, herm. eq. of 2. ode and so on.

3 Tutorial

Die Beispielprogramme zu diesem Tutorial finden sich im Ordner `tutorial`.

The example files for this tutorial can be found in the folder `tutorial`.

3.1 Beispielaufgabe

Als einfaches Optimalsteuerungsproblem soll das Spline-Problem gelöst werden. Die mathematische Aufgabenstellung lautet:

The spline problem serves as a simple example for an optimal control problem. Its mathematical formulation is:

Zielfunktional	$\min\limits_{u,x}$	$\int_0^1 u_0^2(t)\,dt$				objective functional
Differentialgleichungssystem	unter	$\dot{x}_0(t)$	$=$	$x_1(t)$	für $t \in [0,1]$	ode system
		$\dot{x}_1(t)$	$=$	$u_0(t)$	für $t \in [0,1]$	
Randbedingungen		$x_0(0)$	$=$	0		boundary conditions
		$x_1(0)$	$=$	1		
		$x_0(1)$	$=$	0		
		$x_1(1)$	$=$	1		
Beschränkungen		$u_0(t)$	$\in$	$[-6,6]$	für $t \in [0,1]$	constraints

Wir geben zunächst das Zielfunktional in Mayer-Form an, und transformieren das Problem daher um. Siehe ... für Angabe in Lagrange-Form.

For the Mayer formulation of the objective functional, we have to transform the problem. See ... for Lagrange formulation.

$$
\begin{aligned}
\min_{u,x} \quad & x_2(1) \\
\text{unter} \quad & \dot{x}_0(t) = x_1(t) \quad \text{für } t \in [0,1] \\
& \dot{x}_1(t) = u_0(t) \quad \text{für } t \in [0,1] \\
& \dot{x}_2(t) = u_0(t)^2 \quad \text{für } t \in [0,1] \\
& x_0(0) = 0 \\
& x_1(0) = 1 \\
& x_2(0) = 0 \\
& x_0(1) = 0 \\
& x_1(1) = 1 \\
& u_0(t) \in [-6,6] \quad \text{für } t \in [0,1]
\end{aligned}
$$

TransWORHP diskretisiert dieses Optimalsteuerproblem, und gibt das so erhaltene nichtlineare Optimierungsproblem an die Routine WORHP weiter.

TransWORHP diskretisiert dieses Optimalsteuerproblem, und gibt das so erhaltene nichtlineare Optimierungsproblem an die Routine WORHP weiter.

3.2 Erste Implementierung mit TransWORHP

Ein Optimalsteuerungsproblem wird mit TransWORHP als Ableitung der Klasse `TransWorhpProblem` implementiert. Die einzelnen Komponenten des Optimalsteuerungsproblems werden dabei als Methoden der abgeleiteten Klasse zur Verfügung gestellt.

3.1: Minimalbeispiel Optimalsteuerungsproblem: spline0.cpp

```
class Spline0 : public TransWorhpProblem {
public:

    Spline0(const TWdimension &TWdata) : TransWorhpProblem(TWdata) {}

    double obj() override {

        return x(n_dis-1,2);
    }

    void ode(double *dx, double t, const double *x, const double *u,
             const double *p) override {

        dx[0] = x[1];
        dx[1] = u[0];
        dx[2] = u[0]*u[0];
    }

    void u_boundary(double *u_low, double *u_upp) override {

        u_low[0] = -6;
        u_upp[0] = +6;
    }

    void x_boundary(double *x_low, double *x_upp) override {

        x_low[2] = 0;
    }

    void var_boundary(double *x_low, double *x_upp) override {

        x_low[x_index(0,0)] = x_upp[x_index(0,0)] = 0;
        x_low[x_index(0,1)] = x_upp[x_index(0,1)] = 1;
        x_low[x_index(0,2)] = x_upp[x_index(0,2)] = 0;

        x_low[x_index(n_dis-1,0)] = x_upp[x_index(n_dis-1,0)] = 0;
        x_low[x_index(n_dis-1,1)] = x_upp[x_index(n_dis-1,1)] = 1;
    }

};
```

3.2.1 `TWdimension`

Über das Struct `TWdimension` lassen sich alle benötigten Problemgrößen setzen. Werden einzelne Einträge nicht gesetzt, so werden diese mit sinnvollen Standardwerten belegt.

`ID`	Text zur kurzen Beschreibung des Problems
`n_dis`	Anzahl der diskreten Punkte n
`n_ode`	Anzahl der Zustandsgrößen x (d.h. Anzahl der Gleichungen im Differentialgleichungssystem)
`n_ctrl`	Anzahl der Steuergrößen u
`n_param`	Anzahl der freien Parameter (z.B. Prozesszeit)
`n_rand`	Anzahl der Randbedingungen
`n_neben`	Anzahl der (nichttrivialen) Steuer- und Zustandsbeschränkungen
`n_integral`	Anzahl an Integraltermen
`n_zen`	d

Für die Mehrziel-Methode lassen sich weitere Parameter setzen.

`multinode`	Vektor mit Indezies der Mehrzielknoten
`BOXneben`	Vektor mit Indezies der beschränkten Zustände

Eine äquidistante Verteilung der Mehrzielknoten lässt sich weiterhin mit

```
void setMultinodes(int multi);
```

erzielen.

3.2.2 Konstruktor

Im Konstruktor von `TransWorhp` wird die Anzahl der Diskretisierungspunkte festgelegt, sowie die Problemgrößen des Optimalsteuerungsproblems.
Die Syntax lautet:

```
TransWorhpProblem(const TWdimension &TWdata);
```

`TWdata`	Struct mit den Problemgrößen

3.2.3 Zielfunktional

In dieser zu überladenden Methode wird das Zielfunktional berechnet. Hierfür stehen die Steuerungen und Zustände zu allen diskreten Zeitpunkten $\{t_0, \ldots, t_{n-1}\}$ zur Verfügung.

```
double obj();
```

Der Zugriff auf die Zustände erfolgt über:

```
double x(int dis, int i);
```

`dis`	Index des diskreten Punkts
`i`	Koordinate des Zustandsvektors

Analog lässt sich auf die Steuerungen zugreifen:

```
double u(int dis, int i);
```

`dis`	Index des diskreten Punkts
`i`	Koordinate des Steuervektors

Auf freie Parametern, lässt sich zugreifen mit

```
double p(int i);
```

`i`	Koordinate des Parametervektors

3.2.4 Differentialgleichungssystem

In dieser zu überladenden Methode wird rechte Seite des Differentialgleichungssystems berechnet. Hierfür stehen Zustand und Steuerung zu einem festen Zeitpunkt zur Verfügung.

```
void ode(double *dx, double t, const double *x, const double *u, const double *p);
```

`dx`	Rückgabewert: Berechnung der rechten Seite
`t`	Zeitpunkt
`x`	Zustandsvektor
`u`	Steuervektor
`p`	Parametervektor

3.2.5 Box-Beschränkungen

In diesen zu überladenden Methoden werden Beschränkungen für Zustände, Steuerungen oder Parameter angegeben:

```
void x_boundary(double *x_low, double *x_upp);
void u_boundary(double *u_low, double *u_upp);
void p_boundary(double *p_low, double *p_upp);
```

`x_low`	Untergrenzen des Zustandsvektors
`x_upp`	Obergrenzen des Zustandsvektors
`u_low`	Untergrenzen des Steuervektors
`u_upp`	Obergrenzen des Steuervektors
`p_low`	Untergrenzen des Parametervektors
`p_upp`	Obergrenzen des Parametervektors

In diesem Beispiel wurde zusätzlich $x_2(t) \geq 0$ vorgeben, um die Konvergenz zu beschleunigen.

3.2.6 Anfangs- und Endbedingungen

In dieser zu überladenden Methode können Anfangs- und Endbedingungen für Zustände angegeben werden. Ebenso können auch Box-Beschränkungen, die vom diskreten Punkt abhängen, formuliert werden.

```
void var_boundary(double *x_low, double *x_upp);
```

x_low	Untergrenzen des Optimierungsvektors
x_upp	Obergrenzen des Optimierungsvektors

Die anzugebenden Grenzen beziehen sich auf den gesamten Optimierungsvektor, der an WORHP übergeben wird. Um auf die einzelnen Einträge zuzugreifen, sollten Index-Funktionen verwendet werden. Der Index der Zustandsgrößen bestimmt sich mit:

```
int x_index(int dis, int i);
```

dis	Index des diskreten Punkts
i	Koordinate des Zustandsvektors

Analog lässt sich der Index der Steuerungen bestimmen:

```
int u_index(int dis, int i);
```

dis	Index des diskreten Punkts
i	Koordinate des Steuervektors

Auf freie Parameter lässt sich mit diesem Index zugreifen:

```
double p_index(int param);
```

param	Koordinate des Parametervektors

3.2.7 Hauptprogramm

3.2: Minimalbeispiel Aufruf: spline0.cpp

```
1  int main(int argv, char* argc[]) {
2
3      TWparameter twparameter("transworhp.xml");
4      TWfolder folder(&twparameter, 0);
5
6      TWdimension TWdim;
7      TWdim.ID = "Spline0";
8      TWdim.n_ode = 3;
9      TWdim.n_ctrl = 1;
10     TWdim.n_dis = 11;
11
12     Spline0 ph(TWdim);
13     ph.setSolver(&twparameter);
14     folder.Add(&ph);
15
16     folder.Init();
17
18     folder.Loop();
19
20     return 0;
21 }
```

Viele Einstellungen von TransWORHP und WORHP werden über XML-Dateien angegeben (vgl. 8.1). Im Hauptprogramm wird zunächst die Datei `transworhp.xml` geöffnet und die XML-Datei geparst.
Das von uns implementierte Optimalsteuerungsproblem besteht aus drei Zuständen und einer Steuerung wird mit einer Anzahl an diskreten Punkten instanziert und mit `Add()` dem `TWfolder` hinzugefügt. Zuvor wird mittels `setSolver()` dem Problem ein Löser zugeteilt. Die Anzahl der diskreten Punkte wird dabei aus den Parametern ausgelsen (`twparameter.NDIS`). Mit der Funktion `Init()` wird der Arbeitsspeicher für WORHP angelegt und mit TransWORHP verbunden. Anschließen wird mit `Loop()` die Standard-Optimierung gestartet.
Da keine feste Anzahl von diskreten Punkten vorgegeben wird, lässt sich die Diskretisierung über den Konsolenaufruf einstellen:
Mit `spline0 -n 31` werden z.B. 31 diskrete Punkte verwendet.

3.3 Grafische Ausgabe der Optimierung

TransWORHP besitzt ein Interface für grafische Ausgabe, um den Optimierungsfortschritt oder Systemmatrizen bequem beobachten zu können. Das Ausgabefenster wird Viewer genannt.

3.3: Grafische Ausgabe: spline1.cpp

```
1  int main(int argv, char* argc[]) {
2
3      TWparameter twparameter("transworhp.xml");
4      twparameter.Arguments(argv, argc);
5      TWfolder folder(&twparameter, 0);
6
7      TWdimension TWdim;
8      TWdim.ID = "Spline1";
9      TWdim.n_ode = 3;
10     TWdim.n_ctrl = 1;
11     TWdim.n_dis = twparameter.NDIS;
12
13     Spline1 ph(TWdim);
14     ph.setSolver(&twparameter);
15     folder.Add(&ph);
16
17     Viewer *viewer = nullptr;
18     if (twparameter.PLOT) viewer = new Viewer(&twparameter);
19
20     folder.Init();
21     folder.Init(viewer);
22     folder.Loop();
23
24     delete viewer;
25
26     return 0;
27 }
```

Zunächst muss ein `*viewer` erstellt werden. Nach der Initialisierung von TransWORHP mit `Init()` muss `Init(viewer)` aufgerufen werden, damit das Anzeigefenster mit TransWORHP verbunden wird. Die Einstellungen des Viewers werden ebenfalls über die XML-Datei angepasst.
Wenn das Programm ausgeführt wird, erscheint neben der Konsolenausgabe ein Fenster mit den Zustands- und Steuerfunktionen, vgl. Abb. 3.1.
Zusätzlich werden die für die Optimierung mit WORHP benötigte Jacobi-Matrix `DG` und Hesse-Matrix `HM` angezeigt (sparsity). Die Bedeutung der verwendeten Farben zeigt Tab. 3.1.

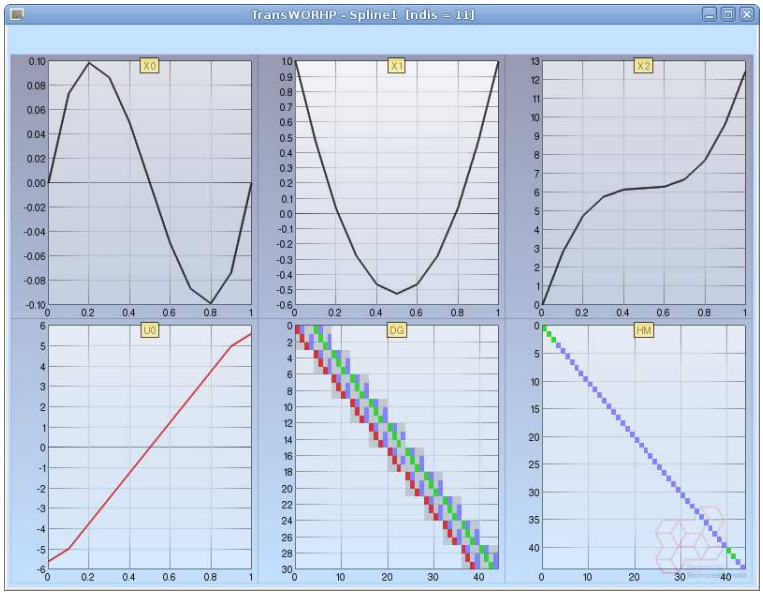


Abbildung 3.1: Grafische Ausgabe

Nicht eingefärbte Einträge werden in der Matrix nicht angelegt, und sparen so Speicherplatz und Rechenzeit.

Die Blöcke der Jacobi-Matrix entstehen dadurch, dass Zustände zu diskreten Zeitpunkten nur von benachbarten Zuständen abhängen.

Abspeichern der Abbildungen als eps mit F1.

3.4 Ableitungsstrukturen in TransWORHP

3.4.1 Angabe von Ableitungs-Strukturen

Die Rechenzeit kann verringert werden, wenn die Struktur der Jacobi-Matrix besser bekannt ist. In diesem Beispiel lässt sich die Struktur der Ableitung der rechten Seite leicht angeben.

Die rechte Seite lautete:

blau	zufälliger Wert
rot	-1
rosa	$-\frac{1}{2}$
grün	1
grau	Null

Tabelle 3.1: Bedeutung der Farben der angezeigten Matrizen

$$\begin{aligned} dx_0 &= x_1 \\ dx_1 &= u_0 \\ dx_2 &= u_0^2 \end{aligned}$$

Damit hängt dx_0 nur von x_1 ab, dx_1 und dx_2 nur von u_0.
Diese Struktur wird durch überladen dieser Methode angegeben:

```
bool ode_structure(DiffStructure &s);
```

In `DiffStructure` wird angegeben, welche Gleichung von welchen Zuständen, Steuerungen und Parametern abhängt. Dazu dient der Klammer-Operator

```
double& DiffStructure::operator()(int eq, int var);
```

`eq`	Index der zu definierenden Gleichung
`var`	Index der Optimierungsgröße. Dieser wird mit `x_indexode`, `u_indexode` oder `p_indexode` bestimmt.

```
int p_indexode(int param);
```

`param`	Koordinate des Parametervektors

Der Rückgabewert von `ode_structure` bestimmt, ob die Struktur auch verwendet wird.

3.4: Ableitungsstrukturen der rechten Seite: spline2.cpp

```
1   bool ode_structure(DiffStructure &s) override {
2
3       s(0, x_indexode(1)); // dx[0] / dx[1]
4       s(1, u_indexode(0)); // dx[1] / du[0]
5       s(2, u_indexode(0)); // dx[2] / du[0]
6       return true;
7   }
```

Die entstehende Jacobi-Matrix ist in Abb. 3.2 dargestellt.
Analog wird in diesem Beispiel auch die Struktur der Ableitung der Zielfunktion angegeben. Hier ist diese Methode zu überladen:

```
bool obj_structure(DiffStructure &s);
```

Da die Zielfunktion nur aus einer Gleichung besteht, ist hier der Parameter `eq` auf 0 zu setzen.

3.5: Ableitungsstrukturen der Zielfunktion: spline2.cpp

```
1   bool obj_structure(DiffStructure &s) override {
2
3       s(0, x_index(n_dis-1,2));
4       return true;
5   }
```

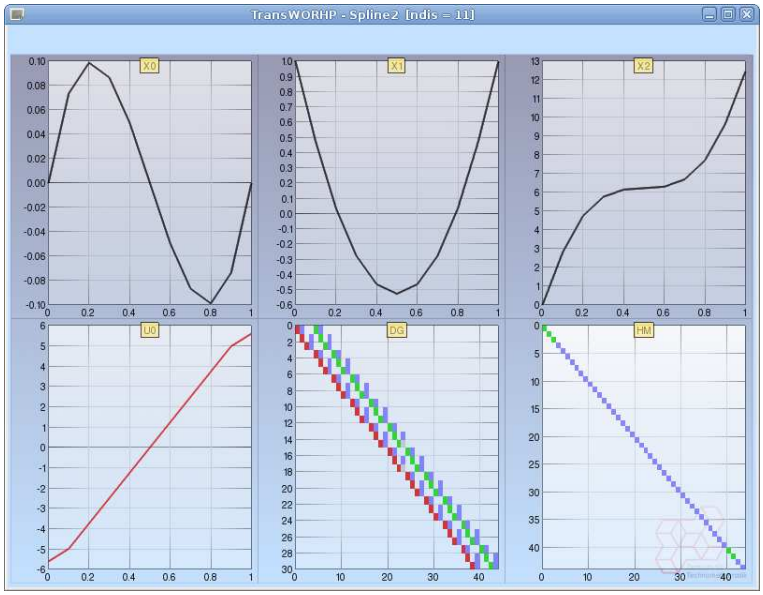


Abbildung 3.2: Jacobi-Matrix mit optimaler Ausnutzung der Sparsity

3.4.2 Angabe der ersten Ableitungen

Um numerische Ableitungen mit finiten Differenzen zu umgehen – diese werden bei sehr vielen diskreten Punkten ungenauer als der Diskretisierungsabstand – bietet sich die analytische Angabe von Ableitungen an.
Die rechte Seite lautete:

$$\begin{aligned} dx_0 &= x_1 \\ dx_1 &= u_0 \\ dx_2 &= u_0^2 \end{aligned}$$

Die nicht verschwindenden analytischen Ableitungen sind

$$\begin{aligned} \frac{\partial dx_0}{\partial x_1} &= 1 \\ \frac{\partial dx_1}{\partial u_0} &= 1 \\ \frac{\partial dx_2}{\partial u_0} &= 2u_0 \end{aligned}$$

Um diese in TransWORHP anzugeben, wird wieder eine Methode überladen:

```
bool ode_diff(DiffStructure &s, double t, const double *x, const double *u, const double *p);
```

`t`, `x`, `u` und `p` entspricht der Bedeutung aus der Funktion `ode`.
Der Klammer-Operator aus Abschnitt 3.4.1 gibt eine Referenz auf einen Speicherplatz zurück, der mit den berechneten Ableitungen beschrieben wird.

3.6: Analytische Ableitungen der rechten Seite: spline3.cpp

```
1     bool ode_diff(DiffStructure &s, double t, const double *x,
2                   const double *u, const double *p) override {
```

```
3
4           s(0, x_indexode(1))= 1;
5           s(1, u_indexode(0))= 1;
6           s(2, u_indexode(0))= 2*u[0];
7           return true;
8       }
```

Analog für die Zielfunktion:

```
bool obj_diff(DiffStructure &s);
```

3.7: Analytische Ableitungen der Zielfunktion: spline3.cpp

```
1       bool obj_diff(DiffStructure &s) override {
2
3           s(0, x_index(n_dis-1,2)) = 1;
4           return true;
5       }
```

3.5 Allgemeine Optimalsteuerprobleme mit TransWORHP

3.5.1 Angabe von Startschätzungen

Bei komplexeren Optimierungsproblemen ist die Angabe von sinnvollen Startschätzungen wichtig, um Konvergenz zu sichern oder die Konvergenzgeschwindigkeit zu verbessern.
Hierfür werden wieder Methoden überladen:

```
void x_init(double *x, int i, int dis)
void u_init(double *u, int i, int dis)
void p_init(double *p)
```

`x_init` und `p_init` werden für jeden diskreten Punkt aufgerufen. Der aktuelle Index `i` sowie die Gesamtzahl der Punkte `dis` stehen bereit.
Anmerkung: Je nach gewählter Diskretisierungsmethode (Trapez, Hermite-Simpson) kann `dis` variieren.

3.8: Startschätzung der Steuerung: spline4.cpp

```
1       void u_init(double *u, int i, int dis) override {
2
3           u[0] = -6 + (12.*i)/dis;
4       }
```

3.5.2 Randwerte

Einfache Anfangs- und Endwerte, bzw. beliebige Werte dazwischen können mit `var_boundary` angegeben werden.
Um komplexere Randbedingungen für einzelne Zeitpunkte zu formulieren, können Gleichungen der Form

$$r(x, u) = 0$$

hinzugefügt werden. Die Anzahl der Gleichungen ist im Konstruktor von TransWORHP anzugeben.
Die Randbedingungen können mit `x()`, `u()`, `p()` wie in Abschnitt 3.2.3 formuliert werden.

```
void rand(double *r);
```

Analog lassen sich auch wieder die Ableitung der Randwerte sowie deren Struktur angeben:

```
bool rand_structure(DiffStructure &s);
```

```
bool rand_diff(DiffStructure &s)
```

Es können Ober- und Untergrenzen angegeben werden für Bedingungen der Form:

$$r_{low} \leq r(x,u) \leq r_{upp}$$

```
void rand_boundary(double *r_low, double *r_upp);
```

Auf diese Weise lassen sich auch (weniger effizient) die gegebenen Anfangs- und Endbedingungen festlegen:

3.9: Randwerte: spline5.cpp

```
1   void rand(double *r) override {
2
3       r[0] = x(0,0);
4       r[1] = x(0,1) - 1;
5       r[2] = x(0,2);
6
7       r[3] = x(n_dis-1,0);
8       r[4] = x(n_dis-1,1) - 1;
9
10      r[5] = x__(3,0)-x__(5,0);
11      r[6] = x__(4,0)-x__(8,0);
12  }
13
14  bool rand_structure(DiffStructure &s) override {
15
16      s(0,x_index(0,0));
17      s(1,x_index(0,1));
18      s(2,x_index(0,2));
19      s(3,x_index(n_dis-1,0));
20      s(4,x_index(n_dis-1,1));
21
22      s(5,x_index__(3,0));
23      s(5,x_index__(5,0));
24
25      s(6,x_index__(4,0));
26      s(6,x_index__(8,0));
27
28      return true;
29  }
30
31  bool rand_diff(DiffStructure &s) override {
32
33      s(0,x_index(0,0)) = 1;
34      s(1,x_index(0,1)) = 1;
35      s(2,x_index(0,2)) = 1;
36      s(3,x_index(n_dis-1,0)) = 1;
37      s(4,x_index(n_dis-1,1)) = 1;
38
39      s(5,x_index__(3,0)) = 1;
40      s(5,x_index__(5,0)) = -1;
41
```

```
42          s(6,x_index__(4,0)) = 1;
43          s(6,x_index__(8,0)) = -1;;
44
45          return true;
46      }
```

3.5.3 Nebenbedingungen

Einfache Steuer- und Zustandsbeschränkungen (Box-Beschränkungen) können mit `x_boundary`, `u_boundary` und `p_boundary` formuliert werden. Diese Funktionen sollte man auch benutzen, da sie effektiver sind.

Die Anzahl der Nebenbedingungen (`n_neben`) muss im Konstruktor von TransWORHP angegeben werden. Für komplexere Beschränkungen wird diese Methode überladen:

```
void neben(double *c, double t, const double *x, const double *u, const double *p)
```

`t`, `x`, `u` und `p` entsprechen dem Aufruf von `ode()` in Abschnitt 3.2.3.
Es müssen Ober- und Untergrenzen angegeben werden:

```
void neben_boundary(double *c_low, double *c_upp);
```

Die Struktur und die Werte der Ableitungen der Nebenbedingungs-Gleichungen kann zusätzlich angegeben werden:

```
bool neben_structure(DiffStructure &s);
bool neben_diff(DiffStructure &s, double t, const double *x, const double *u, const double
*p);
```

Im Beispiel wird zusätzlich

$$-0.4 \leq x_0 + x_1 \leq 1$$

gefordert. Diese Gleichung hängt von x_0 und x_1 ab, was in der Struktur vorgegeben wird.

3.10: Nebenbedingungen: spline5.cpp

```
1       void neben(double *c, double t, const double *x,
2                  const double *u, const double *p) override {
3
4           c[0] = x[0] + x[1];
5       }
6
7           void neben_boundary(double *c_low, double *c_upp) override {
8
9           c_low[0] = -.4;
10          c_upp[0] = 1;
11      }
12
13      bool neben_structure(DiffStructure &s) override {
14
15          s(0,x_index(0,0));
16              s(0,x_index(0,1));
17              return true;
18          }
19
20      bool neben_diff(DiffStructure &s, double t, const double *x,
21                      const double *u, const double *p) override {
```

```
22
23          s(0,x_index(0,0)) = 1;
24          s(0,x_index(0,1)) = 1;
25          return true;
26      }
```

3.6 Nächste Schritte

Optionale Erweiterungen zur Lösung von Optimalsteuerungsproblemen werden hier vorgestellt. Grundlage ist jeweils das Spline-Problem in der Version spline4.cpp.

3.6.1 Startschätzung durch Integration

Sind die ungefähre Struktur der Steuerung und die Anfangswerte der Zustände bekannt, kann dieses als Startschätzung angegeben und damit die Zustände für alle Zeitpunkte hochintegriert werden. Dadurch entsteht eine Startschätzung für die Zustände. Hierzu wird die Funktion

```
int Integrate(int btableau);
```

genutzt, welcher durch den Löser (`solver`) bereitgestellt wird.

3.11: Startschätzung durch Integration: spline_ruku.cpp

```
1       // Startschaetzung aus Anfangszustand und Steuerung integrieren
2       // Anfangszustand festlegen
3       ph.solver->X[ph.x_index(0,0)]  = 0;
4       ph.solver->X[ph.x_index(0,1)]  = 1;
5       ph.solver->X[ph.x_index(0,2)]  = 0;
6
7       // Steuerung festlegen
8       for (int i = 0; i < twparameter.NDIS; i++) {
9           ph.solver->X[ph.u_index(i,0)] = -6 + 12 *  i/(twparameter.NDIS-1.);
10      }
11
12      // System integrieren
13      int steps = ph.solver->Integrate(twparameter.butchertableau);
14      std::cout << "Integrationsschritte:␣" << steps << std::endl;
```

3.6.2 Ergebnisse zwischenspeichern

Genauere Lösungen lassen sich durch eine höhere Anzahl an Diskretisierungspunkten erreichen. Da eine Lösung auf einem sehr feinen Gitter viel Rechenzeit in Anspruch nimmt, ist eine gute Startschätzung nötig. Diese lässt sich beispielsweise durch eine Lösung auf einem gröberen Gitter erzeugen. Zum Speichern einer Lösung wird die Funktion `ToMatlab()` benutzt. Anschließen kann vor der erneuten Optimierung (vor `Loop()`) die zuvor erstellte Lösung mit `FromMATLAB()` eingelesen werden. Beide Methoden werden vom `solver` bereitgestellt. Hierbei werden Zwischenpunkte interpoliert.

```
void ToMATLAB(const std::string& filename);
void FromMATLAB(const std::string& filename);
```

3.12: schreiben und laden einer Startschätzung: spline_load.cpp

```
    TWdimension TWdim;
    TWdim.ID = "Spline / Laden, Speichern";
    TWdim.n_ode = 3;
    TWdim.n_ctrl = 1;
    TWdim.n_dis = twparameter.NDIS;

    Viewer *viewer = nullptr;
    if (twparameter.PLOT) viewer = new Viewer(&twparameter);

    // Startschaetzung in Datei erstellen
    {
        SplineLoad ph(TWdim);
        TWfolder folder(&twparameter, 0);
        ph.setSolver(&twparameter);
        folder.Add(&ph);
        folder.Init();
        folder.Init(viewer);

        folder.Loop(1);
        ph.solver->ToMATLAB("spline11.m");
    }

    if (viewer) viewer->closeAll();

    // Startschaetzung aus Datei lesen (und fuer mehr Diskretisierungspunkte interpolieren)
    {
        //Anzahl diskreter Punkte erhoehen
        TWdim.n_dis = 1001;

        SplineLoad ph(TWdim);
        TWfolder folder(&twparameter, 0);
        ph.setSolver(&twparameter);
        folder.Add(&ph);
        folder.Init();
        folder.Init(viewer);

        ph.solver->FromMATLAB("spline11.m");
        folder.Loop(1);
    }
```

3.6.3 Automatische Differentiation

Experimentell. Bereitstellen der Funktionen obj und ode für MagicDoubles.
Beispiel:
TODO spline_ad.cpp
Details später in Abschnitt 4.4.

3.6.4 Lagrange-Term im Zielfunktional

Experimentell. Angabe von Integrandenfunktionen (mit Ableitungen) und Gewichtung, die zum Mayer-Zielfunktional obj() hinzuaddiert wird.
Vgl. 1.2.3
TODO spline_int.cpp

3.6.5 Explizite Integrationsverfahren

TODO spline_expl.cpp

3.6.6 Sensitivitätsanalyse mit WORHP Zen

TODO spline_zen.cpp

3.6.7 Adaptive Gitteranpassung

Für die adaptive Gitteranpassung muss lediglich `Loop()` durch `meshRef()` ersetzt werden. Einstellungen können in der XML-Datei vorgenommen werden (vgl. 8.1.9).

```
int meshRef();
```

Es ist darauf zu achte, dass während die Gitteranpassung mit grafischer Ausgabe (also mit aktiviertem `viewer`) läuft, das Fenster nicht via Klick auf Schließen geschlossen wird. Dies könnte Speicherfehler hervorbringen! Nach jedem Anpassungsschritt schließt und öffnet sich das Fenster selbstständig.

3.13: adaptive Gitteranpassung: rakete_gitter.cpp

```
1  int main(int argv, char* argc[]) {
2
3      TWparameter twparameter("transworhp.xml");
4      twparameter.Arguments(argv, argc);
5
6      Viewer *viewer = nullptr;
7      if (twparameter.PLOT) viewer = new Viewer(&twparameter);
8
9      TWdimension TWdim;
10     TWdim.ID = "Raketenwagen mesh-refinement";
11     TWdim.n_dis = twparameter.NDIS;
12     TWdim.n_ode = 2;
13     TWdim.n_ctrl = 1;
14     TWdim.n_param = 1;
15
16     RaketenPhase ph(TWdim);
17     ph.setSolver(&twparameter);
18
19     TWfolder folder(&twparameter, 0);
20     folder.Add(&ph);
21     folder.Init();
22     folder.Init(viewer);
23
24     folder.meshRef();
25
26     delete viewer;
27
28     return 0;
29 }
```

3.6.8 Mehrere Phasen

Es ist möglich mehrere Optimalsteuerprobleme (Phasen) in einem TWfolder zu vereinen. Dies ist exemplarisch in spline_phase.cpp dargestellt. Hier wurde das Splineproblem in zwei Phasen aufgeteilt. So wurden

in der ersten Phase die Anfangswerte angegeben, aber die Endwerte frei gelassen. In der zweiten Phasen entsprechend andersherum. Dies lässt sich mit Fallunterscheidungen in `var_boundary()` realisieren.

3.14: mehrere Phasen: spline_phase.cpp

```
1   void var_boundary(double *x_low, double *x_upp) override {
2
3       x_low[x_index(0,2)] = x_upp[x_index(0,2)] = 0;
4
5       if (mode == 0) {
6           x_low[x_index(0,0)] = x_upp[x_index(0,0)] = 0;
7           x_low[x_index(0,1)] = x_upp[x_index(0,1)] = 1;
8       } else if (mode == 1) {
9           x_low[x_index(n_dis-1,0)] = x_upp[x_index(n_dis-1,0)] = 0;
10          x_low[x_index(n_dis-1,1)] = x_upp[x_index(n_dis-1,1)] = 1;
11      }
12  }
```

Damit Stetigkeit zwischen beiden Phasen herrscht, müssen weitere Nebenbedingungen angegeben werden. Hierzu muss zunächst von `TWfolder` geerbt und die Anzahl der Nebenbedingungen angegeben werden. Hierzu stehen folgende Methoden zur Verfügung:

```
void g_boundary(double *x_low, double *x_upp);
void con(double *C);
bool con_structure(DiffStructure &s);
bool con_diff(DiffStructure &s, int colindex);
```

Diese Methoden verhalten sich analog zu den bereits beschriebene (vgl. z.B. `neben()`). Allerdings ist es (bis jetzt) nur möglich lineare Nebenbedingungen anzugeben. Zugriff auf die Phasen besteht über den Vektor `phases`:

```
std::vector<TransWorhp*> phases;
```

Zu beachten ist, dass beim Zugriff auf Zustände oder Steuerungen aus einzelnen Phasen ein Offset (`Delta1`) addiert werden muss. Weiterhin muss beachtet werden, dass der Zugriff direkt auf dem WORHP Optimierungsvektor erfolgt.

3.15: mehrere Phasen: spline_phase.cpp

```
1   void con(double *C) override {
2
3       // zunaechst nur lineare Nebenbedingungen erlaubt!
4       double *X = worhp_o.X;
5       int index1 = phases[0]->x_index(phases[0]->n_dis-1,0) + phases[0]->solver->Delta1;
6       int index2 = phases[1]->x_index(0,0)                  + phases[1]->solver->Delta1;
7
8       C[0] = X[index1] - X[index2];
9       C[1] = X[index1+1] - X[index2+1];
10  }
```

Internes zusammensetzen mehrere Phasen

Werden mehrere Phasen verbunden, so erstellt TransWORHP intern ein einziges (großes) Problem daraus. Hierzu werden die einzelnen Phasen nacheinander in den Optimierungsvektor von WORHP geschrieben. Dies ist in Abbildung 3.3 anhand der Struktur von `DF` bzw. `DG` gut zu erkennen.

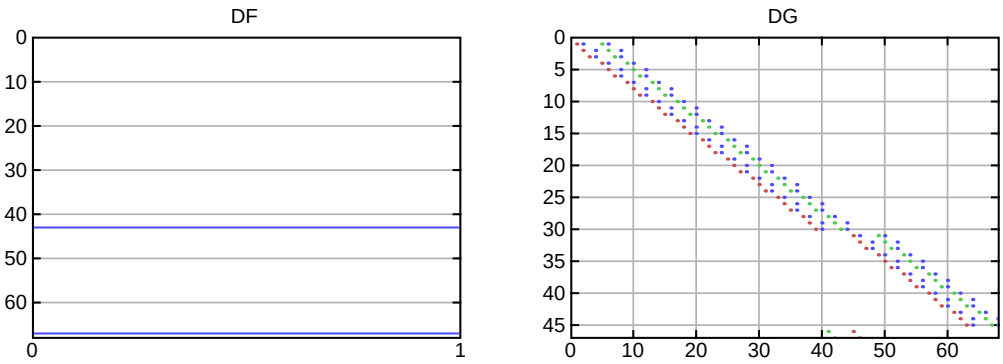


Abbildung 3.3: Spline-Problem mit zwei Phasen

4 Funktionsübersicht

In diesem Kapitel werden die wichtigsten Funktionen erklärt.

4.1 Die Klasse `TransWorhp`

Von dieser Klasse werden alle Probleme abgeleitet.

4.1.1 Systemmethoden

Konstruktor

```
TransWorhp(const std::string &s, int dis, int ode, int ctrl, int param, int rand, int neben);
```

Destruktor

```
~TransWorhp();
```

WORHP-Arbeitsspeicher anlegen und mit Instanz verknüpfen:

```
int Init(XMLNode *xml, OptVar &o, Workspace &w, Params &p, Control &c, Viewer *viewer=0);
```

XML-Dateien lesen:

```
static XMLNode *ReadParams(const std::string& filename);
```

Optimierungsschleife:

```
int Loop();
```

4.1.2 Zugriffsmethoden

Die Zustände und Steuerungen werden in den Optimierungsvektor von WORHP geschrieben. Damit komfortabel auf einzelne von ihnen zugegriffen werden kann, stehen verschiedene Funktionen bereit. Steuerung und der Zustand lassen sich zu jedem diskreten Zeitpunkt `dis` mit folgenden Methoden auslesen. Da freie Parameter für alle Zeitpunkte gleich sind, gibt es nur einen Wert.

```
double x(int dis, int ode) const;
double u(int dis, int ctrl) const;
double p(int param) const;
```

Wird das Optimalsteuerungsproblem mit Verfahren höherer Ordnung (z.B Hermite-Simpson) gelöst, werden Zwischenpunkte eingefügt. An diesen Punkten lässt sich ebenfalls der Zustand und die Steuerung auslesen. Die Methoden `x__()` und `u__()` laufen über alle Punkte, wohingegen die Methoden `x()` und `u()` nur an den 'Haupt'punkten auswerten und alle Zwischenpunkte überspringen.

```
double x__(int dis, int ode) const;
double u__(int dis, int ctrl) const;
```

Indexbestimmung

```
int x_index(int dis, int ode);
int u_index(int dis, int ctrl);
int p_index(int param);
int p_indexode(int param);
```

4.1.3 Implementierung des OCP

Überladbare Methoden

Zielfunktion

```
double obj();
bool obj_structure(DiffStructure &s);
bool obj_diff(DiffStructure &s);
```

ODE-System

```
void ode(double *dx, double t, const double *x, const double *u, const double *p);
bool ode_structure(DiffStructure &s);
bool ode_diff(DiffStructure &s, double t, const double *x, const double *u, const double *p);
bool ode_diff_p(DiffStructure &s, double t, const double *x, const double *u, const double
*p, int index);
```

(letztere Funktion bestimmt ausschließlich die Ableitung nach Parametern!)
Box-Beschränkungen

```
void x_boundary(double *x_low, double *x_upp);
void u_boundary(double *u_low, double *u_upp);
void p_boundary(double *p_low, double *p_upp);
void var_boundary(double *x_low, double *x_upp);
```

Rand-Bedingungen

```
void rand(double *r);
void rand_boundary(double *r_low, double *r_upp);
bool rand_structure(DiffStructure &s);
bool rand_diff(DiffStructure &s);
```

Nebenbedingungen

```
void neben(double *c, double t, const double *x, const double *u, const double *p);
void neben_boundary(double *c_low, double *c_upp);
bool neben_structure(DiffStructure &s);
bool neben_diff(DiffStructure &s, double t, const double *x, const double *u, const double
*p);
bool neben_diff_p(DiffStructure &s, double t, const double *x, const double *u, const double
*p, int index);
```

Startschätzung

```
void init();
void p_init(double *p);
void x_init(double *x, int i, int dis);
void u_init(double *u, int i, int dis);
```

4.1.4 I/O-Methoden

Überladbare Methoden

Aufruf in jedem Iterationschritt für Konsolen- oder Datei-Ausgabe:

```
int step();
```

Wird 0 zurückgegeben, wird die Optimierung unterbrochen.
Aufruf am Ende der Optimierung für Konsolen- oder Datei-Ausgabe:

```
void terminate();
```

Grafisches Interface

Weitere Plots erzeugen und Plots beschriften:

```
void OpenWindows(Viewer *gr);
std::string GetXTitle(int d);
std::string GetUTitle(int d);
```

4.1.5 Weiterführende Methoden

Zeitdiskretisierung anpassen, z.B. im Konstruktor:

```
void TimeAxis(double exponent);
```

oder manuell.
Zustand oder Steuerung an beliebigem Zeitpunkt auslesen:

```
void GetState(double *x, double t);
void GetControl(double *u, double t);
```

4.2 Die Struktur `DiffStructure`

Einträge in den Ableitungsmatrizen können angelegt und gesetzt werden durch den Klammeroperator:

```
double& operator()(int i, int j);
```

4.2.1 Zugriff auf DiffStructure

Mit Hilfsfunktionen wird der richtige Index zugeordnet.
lokale Strukturen (ode, integral, neben): Zugriff mit x_indexode
globale Strukturen (rand, obj): Zugriff mit x_index
TODO: Bild

4.3 Die Klasse `Viewer`

In `OpenWindows()` können neue Plots hinzugefügt werden:
Struktur der Ableitungs-Matrix plotten

```
void Matrix(const std::string &s, WorhpMatrix *m);
```

Daten plotten, die in einer Funktion `func` bereitgestellt werden:

```
void Data(const std::string &s, Funktionenzeiger2 func, int lo, int hi, int index);
```

Dabei ist `func` von diesem Typ:

```
typedef double (*Funktionenzeiger2) (int &len, int &ndgl, int &nsteuer, double *t,double *x,
double *u, int &i, int &index);
```

`index` kann zur internen Unterscheidung verwendet werden.
Phasen-Diagramm plotten (also Daten `func(..., d1)` gegen `func(..., d2)`).

```
void PhasePlot(const std::string &s, Funktionenzeiger2 func, int d1, int d2);
```

OpenGL-Plot hinzufügen.

```
void ThreeD(const std::string &s, XMLNode *xml, plot3d f);
```

Dabei ist `f` von diesem Typ:

```
typedef void (*plot3d) (glObject *obj, double *x, double t);
```

Animation von Objekten
Modell mit Cinema4D erstellen und mit riptide exportieren.

Vergleichswerte hinzufügen:

```
void AddCompareCurve(int &cmpstep, double* cmp, double* cmptime, int n);
```

Alle Plots schließen oder neu anordnen:

```
void CloseAll(); void TilePlots();
```

4.4 Die Klasse `MagicDouble`

Automatische Berechnung der 1. und 2. Ableitung.
TODO

5 Grafisches Interface

Erklärung der Menu-Punkte, Maus- und Tastatursteuerung, etc.

5.1 Überprüfen der analytischen Ableitungen

Angabe von analytische Ableitungen bevorzugt, da stabiler.
Wie lassen die sich auf Korrektheit überprüfen?
Man setze in `transworhp.xml`

```
<USERDF>-1</USERDF>
<USERDG>-1</USERDG>
<USERHM>-1</USERHM>
```

um die Ableitungsberechnung von WORHP zu verwenden.
Dann lässt man das Programm normal laufen, und wählt am Ende der Optimierung im erscheinenden Menu am oberen Fensterrand Debug->Check DF bzw. Debug->Check DG (oder drückt einfach F9 bzw. F10).
Dann erscheint eine lange Ausgabe bestehend aus Matrix-Index, numerischem Ableitungswert und dem vorgegebenen Ableitungswert.
Wo sich die Werte stark unterscheiden, erscheint ein Pfeil, und der Text ist rot.

6 Beispiele

Im Ordner `example` finden sich umfangreichere Beispiele:

6.1 Chemischer Reaktor

Bei diesem Modell handelt es sich um eine zweistufige Reaktionsgleichung zur Umwandlung einer chemischen Substanz. Der Temperatur im Reaktor kann durch die Steuerung beeinflusst werden, um die Umwandlung zu begünstigen. Ziel ist es, die Konzentration der entstehenden Substanz x_2 im Endpunkt zu maximieren.

$$
\begin{array}{lrcl}
\min\limits_{x,u} & -x_2(1) & & \\
\text{unter} & \dot{x}_1(t) & = & -u(t) \cdot x_1(t) + u(t)^2 \cdot x_2(t) \\
& \dot{x}_2(t) & = & u(t) \cdot x_1(t) - 3 \cdot u(t)^2 \cdot x_2(t) \\
& x_1(0) & = & 1 \\
& x_2(0) & = & 0 \\
& u(t) & \in & [0;1], \quad t \in [0;1]
\end{array}
$$

Quelle: 8.16 bzw. Skript?

Anzahl Punkte	Zielfunktionswert	Rechenzeit
21	-0.28810463835	

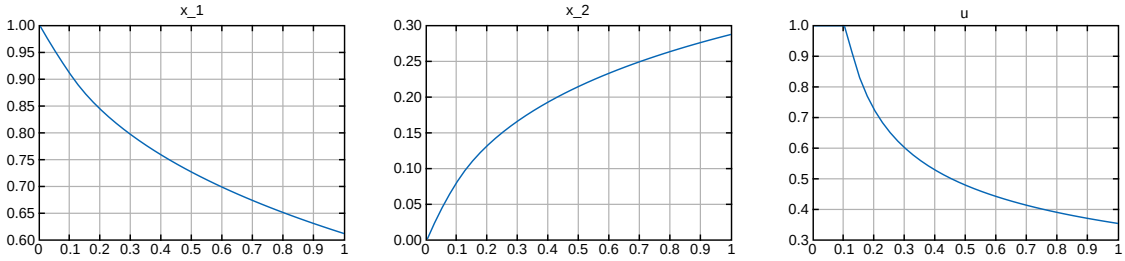


Abbildung 6.1: Chemischer Reaktor

6.2 Spline

Interpolation durch kubischen Spline als Optimierungsaufgabe.

$$
\begin{array}{lllllll}
\min\limits_{x,u} & \frac{1}{2}x_3(1) & & & & & \\
\text{unter} & \dot{x}_1(t) & = & x_2(t) & & & \\
& \dot{x}_2(t) & = & u(t) & & & \\
& \dot{x}_3(t) & = & u^2(t) & & & \\
& x_1(0) & = & 0 & x_1(1) & = & 0 \\
& x_2(0) & = & 1 & x_2(1) & = & -1 \\
& x_3(0) & = & 0 & & & \\
& x_1(t) & \leq & \alpha, \quad t \in [0;1] & & &
\end{array}
$$

$\alpha > 1/4$: unbeschränkt
$1/6 < \alpha < 1/4$: Berührpunkt
$1/6 > \alpha$: Randstück

Diskretisierung	21 Punkte
Zielfunktionswert $\alpha = 0$	1.9999986332
Zielfunktionswert $\alpha = \frac{1}{6}$	2.6666658193

Quelle: 10.14 bzw. Skript?

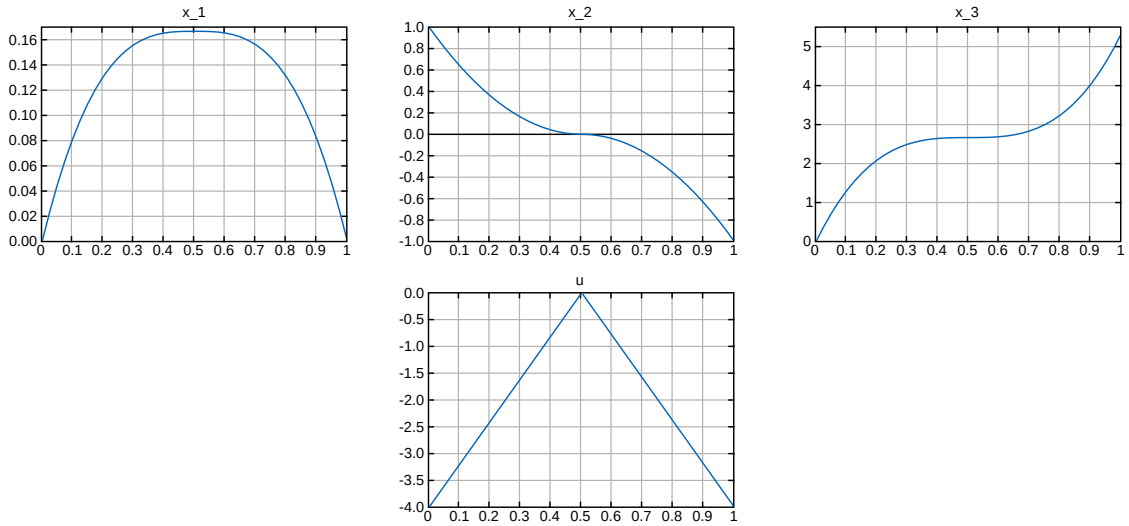


Abbildung 6.2: Lösung Spline-Problem mit $\alpha = \frac{1}{6}$

Das Programm zeigt, wie sich Parameter über die Kommandozeile variieren lassen. Hierzu wird eine map der Kommandozeilenparameter ausgelesen.
Die Beschränkung α lässt sich über –c steuern.

```
spline -n 21 --c 0.166666
```

6.3 Ungedämpfter harmonischer Oszillator

Beispiel mit freier Endzeit. Die freie Endzeit wird in p_0 abgelegt. Das DGl-System muss mit p_0 multipliziert werden.

$$
\begin{array}{lrcll}
\min\limits_{x,u,t_f} & t_f & & & \\
\text{unter} & \dot{x}_1(t) & = & x_2(t) & \\
 & \dot{x}_2(t) & = & -x_1(t) + u(t) & \\
 & x(0) & = & \begin{pmatrix} 3 \\ 1 \end{pmatrix} \qquad x(t_f) = \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \\
 & u(t) & \in & [-1;1], \qquad t \in [0;t_f] &
\end{array}
$$

Diskretisierung	51 Punkte
Zielfunktionswert	5.2160656622

Quelle: 5.19

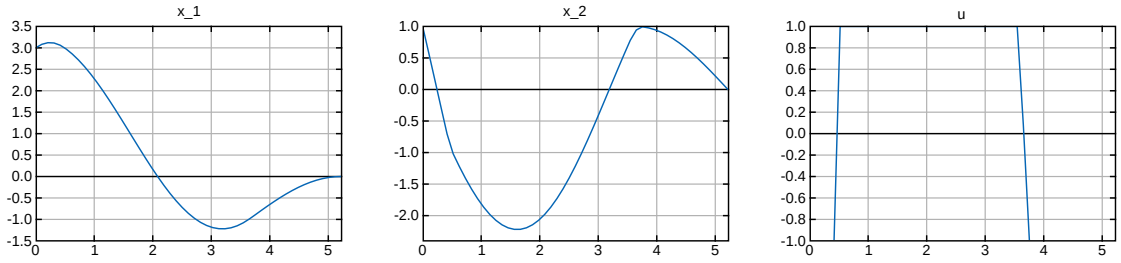


Abbildung 6.3: Lösung Oszillator-Problem

`freetime = 1;` stellt freie Endzeit auch im Viewer dar.

6.4 Raketenwagen

Gasgeben und Bremsen.

$$
\begin{array}{lrcll}
\min\limits_{x,u,t_f} & t_f & & & \\
\text{unter} & \dot{x}_1(t) & = & x_2(t) & \\
 & \dot{x}_2(t) & = & u(t) & \\
 & x(0) & = & \begin{pmatrix} 4 \\ -1 \end{pmatrix} \qquad x(t_f) = \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \\
 & u(t) & \in & [-1;1], \qquad t \in [0;t_f] &
\end{array}
$$

Quelle: ?

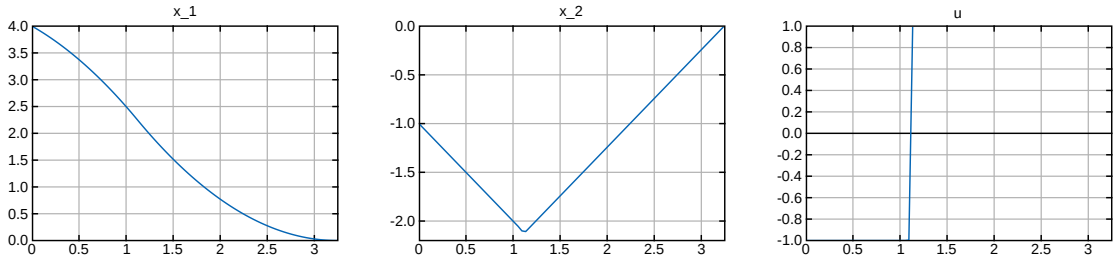


Abbildung 6.4: Lösung Raketenwagen-Problem

6.5 Erzentlader

$$
\begin{array}{lll}
\min\limits_{x,u,t_f} & t_f & \\
\text{unter} & \dot{x}_1(t) = x_2(t) & \\
& \dot{x}_2(t) = u(t) & \\
& \dot{x}_3(t) = x_4(t) & \\
& \dot{x}_4(t) = -x_3(t) + u(t) & \\
& x(0) = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \qquad x(t_f) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} & \\
& u(t) \in [-1;1], & t \in [0;t_f] \\
& x_2(t) \leq 0.2, & t \in [0;t_f]
\end{array}
$$

Diskretisierung	101 Punkte
Zielfunktionswert	5.84412332892

Quelle: 10.11

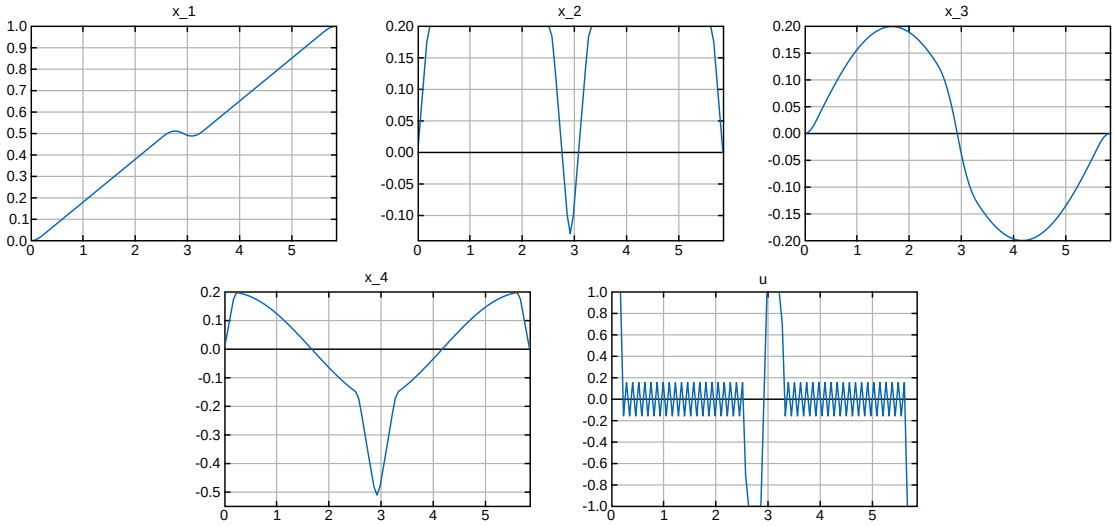


Abbildung 6.5: Lösung Erzentlader

6.6 Knickstab

$$
\begin{aligned}
\min_{x,u} \quad & \int_0^1 \frac{1}{2} u(t)^2 + \alpha \cdot \cos\theta(t)\, dt \\
\text{unter} \quad & \dot{x}(t) = \sin\theta(t) \\
& \dot{\theta}(t) = u(t) \\
& x(0) = 0 \qquad\qquad x(1) = 0 \\
& x(t) \in [-0.05; 0.05], \quad t \in [0;1]
\end{aligned}
$$

Alternative Formulierungen als `knickstab.cpp` und `knickstab_int.cpp` mit Lagrange-Term

TolOpti wird intern höher gesetzt.

Diskretisierung $\alpha = 9.90027$	60 Punkte
Zielfunktionswert	9.9002014316
Diskretisierung $\alpha = 27.2627$	51 Punkte
Zielfunktionswert	27.151069208
Diskretisierung $\alpha = 150$	180 Punkte
Zielfunktionswert	148.51217659

Quelle: ?

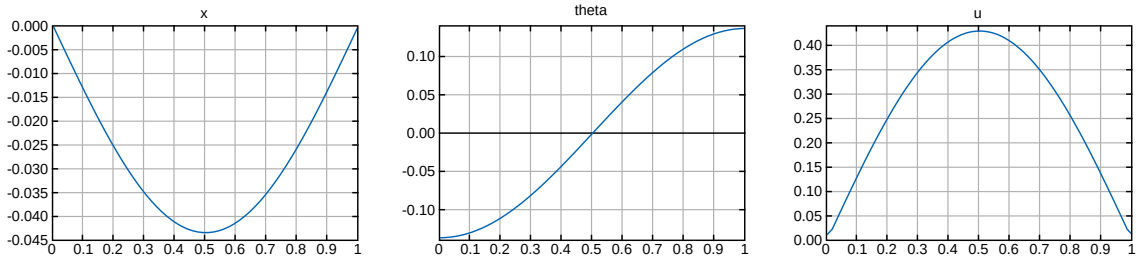


Abbildung 6.6: Lösung Knickstab für $\alpha = 9.90027$

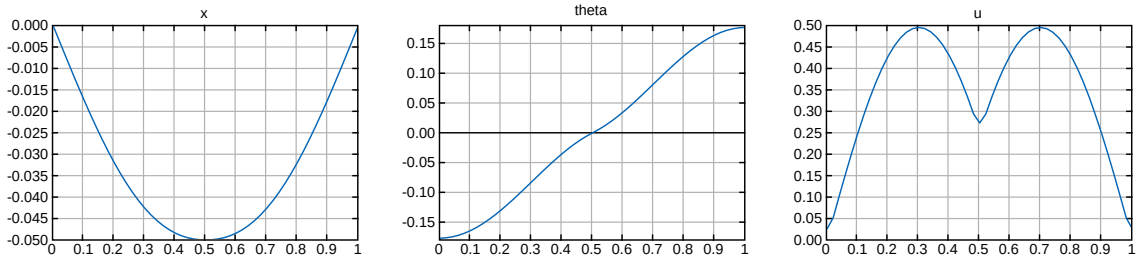


Abbildung 6.7: Lösung Knickstab für $\alpha = 27.2627$

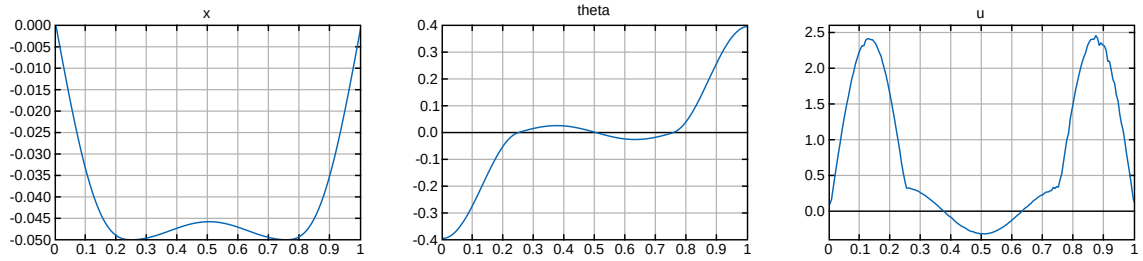


Abbildung 6.8: Lösung Knickstab für $\alpha = 150$

6.7 Reentry-Problem

$$\min \int_0^{t_f} 10v^3\sqrt{p_0 e^{-\beta R x i}}$$

$$\begin{aligned}
\text{unter} \quad \dot{v} &= -\tfrac{Spv^2}{2m}C_W(u) - \tfrac{g\sin\gamma}{(1+\xi)^2} \\
\dot{\gamma} &= -\tfrac{Spv}{2m}C_A(u) + \tfrac{v\cos\gamma}{R(1+\xi)} - \tfrac{g\sin\gamma}{v\cdot(1+\xi)^2} \\
\dot{\xi}(t) &= \tfrac{v\sin\gamma}{R} \\
\text{mit} \quad p &= p_0 e^{-\beta R\xi} \\
C_W(u) &= 1.174 - 0.9\cos u \\
C_A(u) &= 0.6\sin u \\
v(0) &= 0.36 \qquad\qquad v(t_f) = 0.27 \\
\gamma(0) &= -8.1\cdot\tfrac{\pi}{180} \qquad \gamma(t_f) = 0 \\
\xi(0) &= \tfrac{4}{R} \qquad\qquad \xi(t_f) = \tfrac{2.5}{R}
\end{aligned}$$

$$R = 209, \beta = 4.26, p_0 = 2.704\cdot 10^{-3}, g = 3.2172\cdot 10^{-4}, \frac{S}{m} = 53200$$

Diskretisierung	71 Punkte
Zielfunktionswert	0.027911689692
Final Time	225.427

Quelle: ?

Funktion terminate zur Ausgabe der Verfahrzeit nach Optimierung

Funktion step zur Manipulation der Lösung (hier abschneiden auf $[-\pi, \pi]$)

alternative formulierung mit sin/cos und Nebenbedingung

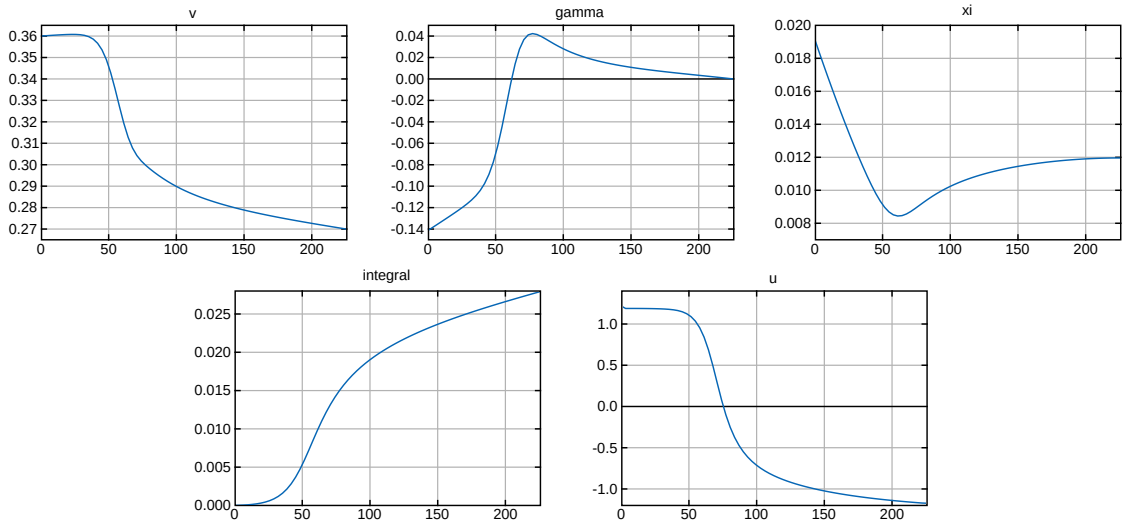


Abbildung 6.9: Lösung Reentry-Problem

6.8 Laufkatze

Kransystem

- Standardproblem
- mit Automatischer Differentiation
- in Lagrange-Formulierung

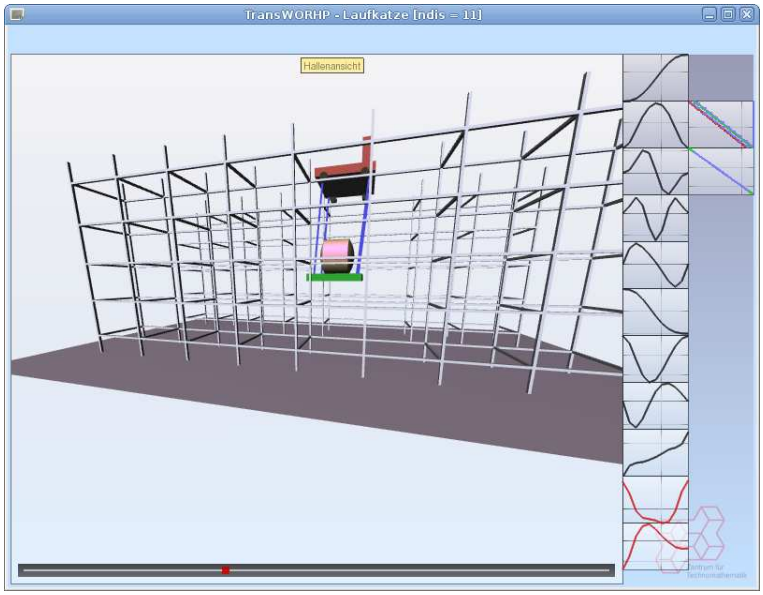


Abbildung 6.10: 3d-Plot der Laufkatze

$$
\begin{aligned}
\min_{x,u,t_f} \quad & x_8(t_f) + 0.1 \cdot t_f \\
\text{unter} \quad & \dot{x}_0 = x_1 \cdot t_f \\
& \dot{x}_1 = x_4 \cdot t_f \\
& \dot{x}_2 = x_3 \cdot t_f \\
& \dot{x}_3 = \left(x_4 - (9.81 - x_7) \cdot \frac{x_2}{x_5} \right) \cdot t_f \\
& \dot{x}_4 = u_0 \cdot t_f \\
& \dot{x}_5 = x_6 \cdot t_f \\
& \dot{x}_6 = x_7 \cdot t_f \\
& \dot{x}_7 = u_1 \cdot t_f \\
& \dot{x}_8 = \left(u_0^2 + u_1^2 \right) \cdot t_f \\
& x(0) = (0,0,0,0,0,5,0,0,0)^\top \\
& x(t_f) = (8,0,0,0,0,4,0,0,\text{frei})^\top \\
& -1 \le u_0(t) \le 1, -1 \le u_1(t) \le 1 \\
& 0 \le x_0(t) \le 100, -3 \le x_1(t) \le 3 \\
& -2 \le x_2(t) \le 2, -10 \le x_3(t) \le 10 \\
& -4 \le x_4(t) \le 4, 0.5 \le x_5(t) \le 15 \\
& -3 \le x_6(t) \le 3, -10 \le x_7(t) \le 10 \\
& 0 \le x_8(t), \text{ für alle } t \in [0, t_f]
\end{aligned}
$$

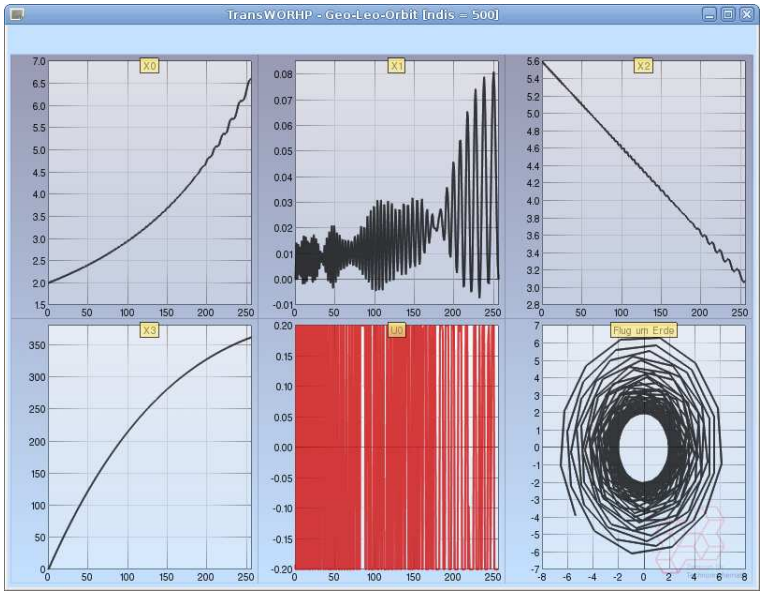


Abbildung 6.11: Phasenplot beim Geo-Leo-Transfer

6.9 Beschränkter Spline

wie im Tutorial, mit einer zusätzlichen Beschränkung

6.10 Brachistochrone

Kurve der schnellsten Fallzeit

6.11 Roboter

Industrie-Roboter mit 3 Freiheitsgraden

6.12 Geo-Leo-Transfer

Raumfähre fliegt um die Erde.

7 MATLAB-Interface

TODO.
Auf numerische Genauigkeit achten! Sogar backslash-Operator zum Lösen von Gleichungssystemen ist nicht immer genau genug.

8 Konfiguration von TransWORHP

8.1 XML-Datei `transworhp.xml`

Einige Einstellungen von TransWORHP lassen sich über eine XML-Datei konfigurieren, vgl. Abschnitt 3.2.7.
Es kann `id` oder `text-id` angegeben werden.

8.1.1 <WORHP>

Das Tag `<WORHP>` beschreibt, wie WORHP aufgerufen wird. Mit dem Attribut `param` wird die WORHP-Parameterdatei angegeben, z.B. `worhp.xml`.
Mit den Unter-Tags `USERDF`, `USERDF` und `USERHM` lassen sich Setzungen aus der WORHP-Parameterdatei überschreiben:

- +1 Ableitungen werden von TransWORHP berechnet. Entweder numerisch oder analytisch, falls vorhanden.
- -1 Ableitungen werden von WORHP numerisch bestimmt.
- 0 Setzung aus Paramterdatei übernehmen.

Steht USERHM auf -1, so wird z.B. die Hesse-Matrix von WORHP berechnet, je nach dortiger Setzung mit finiten Differenzen oder BFGS-Methoden.
Steht USERHM auf 1, berechnet TransWORHP die Hesse-Matrix. Möglicherweise längere Rechenzeit pro Iteration, aber insgesamt weniger Iterationen.
Beispiel:

```
<WORHP param="worhp.xml">
    <USERDF>1</USERDF>
    <USERDG>1</USERDG>
    <USERHM>-1</USERHM>
</WORHP>
```

8.1.2 <DISCRETIZATION>

Zur Auswahl stehen die Optionen aus Tab. 8.1

id	text-id	Beschreibung
1	`Trapez`	Trapezverfahren (1.8)
2	`HermiteSimpson`	Hermite-Simpson-Verfahren (1.9)

Tabelle 8.1: Optionen für <DISCRETIZATION>

Die erstgenannte Option wird verwendet.
Beispiel:

```
<DISCRETIZATION>Trapez</DISCRETIZATION>
```

8.1.3 <FINITEDIFF>

Parameter zur Berechnung der finiten Differenzen in TRANSWORHP, wenn keine expliziten analytischen Ableitungen angegeben werden.
Beispiel:

```
<FINITEDIFF>1e-6</FINITEDIFF>
```

8.1.4 <BUTCHER>

Eine Vorabintegration kann mit expliziten Integrationsverfahren (mit Schrittweitensteuerung) durchgeführt werden. Dazu müssen die Anfangswerte, die freien Parameter und die Steuerungen sinnvoll belegt werden. Siehe `spline_ruku.cpp` für ein Beispiel.
Zur Auswahl stehen die Optionen aus Tab. 8.2.

id	text-id	Beschreibung
0	HeunEuler	$s = 2$, $p = 2$, $q = 1$
1	BogackiShampine	$s = 4$, $p = 3$, $q = 2$
2	Uebung	$s = 4$, $p = 2$, $q = 3$
3	Fehlberg	$s = 6$, $p = 4$, $q = 5$
4	CashKarp	$s = 6$, $p = 5$, $q = 4$
5	DormandPrince	$s = 7$, $p = 4$, $q = 5$

Tabelle 8.2: Optionen für <BUTCHER>

Beispiel:

```
<BUTCHER>1</BUTCHER>
```

8.1.5 <HESSIAN>

Die Struktur <STRUCTURE> und Berechnungsweise <VALUES> der Hessematrix kann unabhängig voneinander festgelegt werden.
Zur Auswahl der Struktur stehen die Optionen 0-3 aus Tab. 8.3.
Um die gegebene Struktur zu erhalten, sollte in der WORHP-Parameterdatei

- entweder BFGS ausgeschaltet sein,

  ```
  <BOOL name="FidifHM">True</BOOL>
  ```

- oder strukturerhaltendes BFGS eingeschaltet sein.

  ```
  <BOOL name="FidifHM">True</BOOL>
  <INT name="BFGSmethod">100</INT>
  ```

Zur Auswahl der Werte stehen die Optionen 0-2 aus Tab. 8.4.
Beispiel:

id	text-id	Beschreibung
0	`Diagonal`	Nur Diagonalmatrix
1	`Full`	Volle untere Dreiecksmatrix
2	`Odeblocks2`	Für Optimalsteuerungsproblem geeignete Sparsity mit Verbindung benachbarter Blöcken
3	`Odeblocks`	Für Optimalsteuerungsproblem geeignete Sparsity
4		Verwendung der analytischen Ableitung bei AD
5		Verwendung der analytischen Ableitung bei AD (schneller)

Tabelle 8.3: Optionen für <STRUCTURE>

id	text-id	Beschreibung
0	`DiffDiff`	Berechnung der 2. Ableitung von $F + \mu \cdot G$
1	`DiffDG`	Berechnung der 1. Ableitung von DF + mu*DG
2		schnellere Berechnung der 1. Ableitung von DF + mu*DG
3		Berechnung der analytischen Ableitung von F + mu*G
4		Berechnung der analytischen Ableitung von F + mu*G, 2. Version

Tabelle 8.4: Optionen für <VALUES>

```
<HESSIAN>
    <STRUCTURE>3</STRUCTURE>
    <VALUES>2</VALUES>
</HESSIAN>
```

Bei Verwendung der analytischen Ableitungen wird lässt sich die Berechnung der Hessematrix durch <MAGIC_HESSIAN> parametrisieren.
Beispiel:

```
<MAGIC_HESSIAN>
    <STRUCTURE>5</STRUCTURE>
    <VALUES>4</VALUES>
</MAGIC_HESSIAN>
```

8.1.6 <PLOT>

Die Struktur von DF, DG und HM kann beobachtet werden.
Beispiel:

```
<PLOT>
    <!--SPARSITY>DF</SPARSITY-->
    <SPARSITY>DG</SPARSITY>
    <SPARSITY>HM</SPARSITY>
</PLOT>
```

8.1.7 <WINDOW>

Diese Einstellungen betreffen die grafische Darstellung.
Beispiel:

```
<WINDOW width="800" height="600">
    <OPENGL multisamplebuffers="1" multisamplesamples="4" />
    <LOGO>ZeTeM</LOGO>
    <BACKGROUND color_top="153,153,179,255" color_bottom="196,225,255,255" />
    <MENU color_back="153,153,179,255" color_fps="153,153,179,255" />
</WINDOW>
```

width und height setzt die Fensterbreite.
In OPENGL können multisamplebuffers aktiviert werden, um Anti-Aliasing bei der 3d-Darstellung zu erhalten. multisamplesamples legt die Qualität fest (erlaubte Werte abhängig von der Grafikkarte)
LOGO kann ZeTeM, UniBremen oder Steinbeis sein.
Die Farben für BACKGROUND und MENU sind im Format RGBA anzugeben.

8.1.8 <TERMINAL>

Dieses Tag betrifft Einstellungen der Ausgabekonsole unter Windows.
Wenn dieses Tag vorhanden ist, wird die Schriftart und Puffergröße des Konsolenfensters angepasst.
Beispiel:

```
<TERMINAL width="132" height="2000" >
    <FONT width="6" height="8" />
</TERMINAL>
```

8.1.9 <MESHREF>

Hiermit lassen sich Änderungen an der adaptiven Gitteranpassung vornehmen. Es stehen drei Modi bereit: neue Punkte in Intervall einfügen, in welchem der Fehler am größten ist (modus="0"), gezielt Auf-/Absprungpunkte der Steuerung finden (modus="1") oder beides (modus="2"). Weitere Einstellungen aus Tabelle 8.5 stehen zu Verfügung.
Beispiel:

```
<MESHREF modus="0" >
    <FEHLER_MOD>0</FEHLER_MOD>
    <TOL>1e-10</TOL>
    <maxIter>100</maxIter>
    <!-- Betts Parameter - Anfang -->
    <M1>5</M1>
    <R>0</R>
    <KAPPA>0.1</KAPPA>
```

Allgemein	
FEHLER_MOD	Art der Fehlerberechnung: 0:Betts, 1:über höhere Diskretisierung
TOL	Fehlerschranke
maxIter	maximale Anzahl an Iterationsschritten
(VERLAUF	Zwischenschritte anzeigen ein (1) oder aus (0)) **zur Zeit nicht verfügbar**

Betts-Parameter	
M1	maximale Anzahl neuer Punkte pro Intervall. Standard: 5
R	globale Ordnungsreduktion (falls nicht die gewünschte Genauigkeit erreicht wird)
KAPPA	Faktor nach Betts für Fehlerreduktion. Standard: 0.1
M	maximale Anzahl neuer Punkte über alle Intervalle; wenn leer: Standard

Plots (0=aus, 1=an)	
SCHRITTWEITE	Anzeige für Schrittweite pro Intervall
FEHLER	Anzeige für Fehler pro Intervall
GITTERPUNKTE	Position der Gitterpunkte
LAMBDA	Plot der Lambda-Multiplikatoren
MU	Plot der Adjungierten

Tabelle 8.5: Optionen für <MESHREF>

```
    <M></M>
    <!-- Betts Parameter - Ende -->
    <VERLAUF>1</VERLAUF>
    <PLOTS>
        <SCHRITTWEITE>0</SCHRITTWEITE>
        <FEHLER>1</FEHLER>
        <GITTERPUNKTE>1</GITTERPUNKTE>
        <LAMBDA>1</LAMBDA>
        <MU>1</MU>
    </PLOTS>
</MESHREF>
```

9 Weiteres

9.1 Verwenden der XML-Bibliothek

TODO.

9.2 Verwenden von Shared Memory

TODO Beschreibung der Funktionen

9.3 Optimale Steuerung

9.3.1 Parametrische Sensitivitätsanalyse

Stehen für eine zu optimierende Größe einer praxisbezogenen Anwendung ausreichend Informationen zur Verfügung, lässt sich daraus ein mathematisches Optimierungsproblem modellieren und im günstigsten Fall lösen.
Oft wird man sich jedoch mit der alleinigen Angabe der optimalen Lösung nicht zufrieden geben, sondern wünscht oder benötigt Informationen darüber, wie die berechnete Lösung auf kleine Störungen im System reagiert.

Mit einem Störparameter $p \in \mathbb{R}^{n_p}$ sollen Störungen, denen das System möglicherweise unterworfen ist, im *gestörten Problem der nichtlinearen Optimierung* berücksichtigt werden.

$$\begin{array}{lll} \min\limits_{x} & f(x,p) & \\ \text{unter} & g_i(x,p) = 0, & i \in \{1,\dots,m_e\}, \\ & g_i(x,p) \leq 0, & i \in \{m_e+1,\dots,m\}. \end{array} \tag{9.1}$$

Dabei hängen die Funktionen $f : \mathbb{R}^n \times \mathbb{R}^{n_p} \to \mathbb{R}$ und $g : \mathbb{R}^n \times \mathbb{R}^{n_p} \to \mathbb{R}^m$ nun zusätzlich von diesem Störparameter ab. Wegen letzterem ist auch die zulässige Menge $X(p)$ abhängig vom Parameter p.
Für einen Referenzwert $p = p_0 \in \mathbb{R}^{n_p}$ wird (9.1) als das *ungestörte* oder *nominelle Problem* bezeichnet.

Das Ziel einer *parametrischen Sens.A.* ist die Untersuchung der Auswirkungen, die kleine Abweichungen vom Referenzwert auf die berechnete optimale Lösung des nominellen Problems haben. Dazu werden die totalen Ableitungen der Systemgleichungen nach dem Störparameter benötigt.

Ließe sich die Lösung des gestörten Problems (9.1) explizit in Abhängigkeit vom Störparameter p in der Form $x(p) \in \mathbb{R}^n$ angeben, hätte man eine umfassende Auskunft über den Einfluss des Störparameters.
Für kleine Störungen lässt sich das lokale Verhalten der optimalen Lösung $x^\star$ des ungestörten Problems unter wesentlich geringerem Rechenaufwand durch Bestimmen der Sensitivitätsableitungen $\frac{dx}{dp}(p_0)$ und $\frac{d\lambda}{dp}(p_0)$ untersuchen, die nach Fiacco [**?**] durch Differentiation der KKT-Bedingungen und nach dem Störparameter gewonnen werden.
Es seien mit $m_0 \leq m$ die aktiven Indizes der Referenzlösung $x^\star$ ohne Einschränkung so sortiert, dass

$$\mathcal{I}(x^\star) = \{i \in \{m_e+1,\dots,m\} \mid g_i(x^\star,p_0) = 0\} = \{m_e+1,\dots,m_0\}.$$

Wurde ein Lösungskandidat $x^\star$ des Problems (9.1) gefunden, können inaktive Ungleichungsnebenbedingungen in (9.1) gestrichen, und aktive Ungleichungsnebenbedingungen durch Gleichungsnebenbedingungen ersetzt werden, ohne dass die Lösung $x^\star$ davon beeinflusst wird.
Mit der Funktion $\tilde{g} : \mathbb{R}^n \times \mathbb{R}^{n_p} \to \mathbb{R}^{m_0}$ mit $\tilde{g}_i(x,p) = g_i(x,p),\ i = 1,\dots,m_0$ lässt sich das so entstehende nichtlineare Optimierungsproblem formulieren als

$$\begin{array}{lll} \min & f(x,p) & \\ \text{unter} & \tilde{g}(x,p) = 0. & \end{array} \tag{9.2}$$

In der entsprechenden Lagrange-Funktion

$$L(x,\tilde{\lambda},p) = f(x,p) + \tilde{\lambda}(p)^T \tilde{g}(x,p),$$

werden nur die zu den Gleichungsbeschränkungen eindeutig bestimmten Lagrange-Multiplikatoren $\tilde{\lambda} : \mathbb{R}^{n_p} \to \mathbb{R}^{m_0}$ berücksichtigt.
Analog zum KKT-System (4.13) erhält man

$$\begin{pmatrix} \nabla_x^2 L(x^\star,\tilde{\lambda}^\star,p_0) & \nabla_x \tilde{g}(x^\star,p_0)^T \\ \nabla_x \tilde{g}(x^\star,p_0) & 0 \end{pmatrix} \begin{pmatrix} \frac{dx}{dp}(p_0) \\ \frac{d\lambda}{dp}(p_0) \end{pmatrix} + \begin{pmatrix} \nabla_{xp} L(x^\star,\tilde{\lambda}^\star,p_0) \\ \nabla_p \tilde{g}(x^\star,p_0) \end{pmatrix} = 0, \tag{9.3}$$

woraus sich aufgrund der Invertierbarkeit der Jacobi-Matrix mit den Abkürzungen $L := L(x^\star, \tilde{\lambda}^\star, p_0)$ und $\tilde{g} := \tilde{g}(x^\star, p_0)$ explizite Formeln zur Bestimmung der Sensitivitätsableitungen ablesen lassen:

Folgerung 9.1 Unter Voraussetzungen an Differenzierbarkeit und Regularität gilt für die Funktionen x und $\tilde{\lambda}$

$$\begin{pmatrix} \frac{dx}{dp}(p_0) \\ \frac{d\tilde{\lambda}}{dp}(p_0) \end{pmatrix} = - \begin{pmatrix} \nabla_x^2 L & \nabla_x \tilde{g}^T \\ \nabla_x \tilde{g} & 0 \end{pmatrix}^{-1} \begin{pmatrix} \nabla_{xp} L \\ \nabla_p \tilde{g} \end{pmatrix}.$$

Damit lassen sich die Sensitivitätsableitungen der Optimierungsvariablen $\frac{dx}{dp}(p_0)$ und der Lagrange-Multiplikatoren $\frac{d\tilde{\lambda}}{dp}(p_0)$ direkt aus der Nominallösung ermitteln.

Approximation gestörter Lösungen

Mit der optimalen Lösung $x^\star$ und $\tilde{\lambda}^\star$ und den nach Folgerung 9.1 berechneten Sensitivitätsableitungen lassen sich Schätzwerte für die optimale Lösung des gestörten Problems angeben.
Wird der Parameter p im gestörten Problem (9.1) nur in einer kleinen Umgebung um p_0 variiert, verändert sich die Menge der aktiven Indizes nicht, und (9.2) bleibt eine kompaktere Darstellung des Problems, bei der die nicht aktiven Ungleichungsbedingungen eliminiert werden. Die Taylor-Entwicklungen erster Ordnung liefern dann Schätzungen der Optimierungsvariablen und der Lagrange-Multiplikatoren:

$$\begin{aligned} x(p) &\approx x^\star + \tfrac{dx}{dp}(p_0) \cdot (p - p_0) \\ \tilde{\lambda}(p) &\approx \tilde{\lambda}^\star + \tfrac{d\tilde{\lambda}}{dx}(p_0) \cdot (p - p_0) \end{aligned}$$

Die Gleichungsbeschränkungen von (9.2) werden durch die Approximation von $x(p)$ im Allgemeinen mit kleinen Fehlern $\varepsilon \in \mathbb{R}^{m_0}$ verletzt:

$$g\left(x^\star + \frac{dx}{dp}(p_0) \cdot (p - p_0), p\right) = \varepsilon \neq 0$$

Wird dieser Fehler ebenfalls als Störparameter aufgefasst, lässt sich die Verletzung der Beschränkungen nach Büskens [**?**] schrittweise reduzieren. Dazu formuliert man ein unter den Parametern $(p, \varepsilon) \in \mathbb{R}^{n_p \times m_0}$ gestörtes Problem

$$\begin{aligned} \min \quad & f(x, p) \\ \text{unter} \quad & g(x, p) - \varepsilon \quad = \quad 0, \end{aligned}$$

dessen Lösung $x(p, \varepsilon)$ für $\varepsilon = \varepsilon_0 = 0$ der Lösung von (9.2) entspricht.
Für Störungen $(p, \varepsilon) \neq (p_0, 0)$ lassen sich mit den Sensitivitätsableitungen $\frac{dx}{d\varepsilon}(\varepsilon)$ iterativ Lösungen $x(p, \tilde{\varepsilon})$ berechnen, die gegen einen Fixpunkt $x(p, 0)$ konvergieren.

Übertragen auf Optimalsteuerungsprobleme

Die diskretisierten Steuer- und Zustandsvariablen entsprechen genau den Optimierungsvaraibalen des Nichtlinearen Problems. Damit stehen für die diskretisierten Variablen die Sensitivitätsableitungen zur Verfügung.

Beispiel: Notlandung

Mit einem Trägersystem wird ein Raumgleiter auf circa 30 km Höhe gebracht. Nach dem Abkoppeln des Gleiters wird sein Antrieb aktiviert, um eigenständig einen Orbit zu erreichen. Versagt der Antrieb, muss der Raumgleiter kontrolliert wieder zur Erde zurückgebracht werden.
In diesem Optimierungsproblem muss die Startgeschwindigkeit des Raumgleiters nach einer missglückten Abkopplung so gut ausgenutzt werden, dass er möglichst weit fliegt. (Modell nach Mayrhofer/Sachs)

Steuerungen

Luftwiderstand	C_L
Rollwinkel	μ

Zustände

Geschwindigkeit	v
Inklination	γ
Azimut	χ
Höhe	h
Breitengrad	$\dot{\Lambda}$
Längengrad	$\dot{\Theta}$

Optimalsteuerungsproblem:

$$
\begin{array}{lrcl}
\min\limits_{C_L,\mu,t_f} & -c_1(\Lambda(t_f)-\Lambda(0))^2 - c_2(\Theta(t_f)-\Theta(0))^2 & & \\
\text{unter} & \dot{v} &=& -D(v,h;C_L)\frac{1}{m} - g(h)\sin\gamma + \omega^2\cos\Lambda(\sin\gamma\cos\Lambda - \cos\gamma\sin\chi\sin\Lambda)R(h) \\
& \dot{\gamma} &=& L(v,h;C_L)\frac{\cos\mu}{mv} - \left(\frac{g(h)}{v} - \frac{v}{R(h)}\right)\cos\gamma \\
& & & +2\omega\cos\chi\cos\Lambda + \omega^2\cos\Lambda(\sin\gamma\sin\chi\sin\Lambda + \cos\gamma\cos\Lambda)\frac{R(h)}{v} \\
& \dot{\chi} &=& L(v,h;C_L)\frac{\sin\mu}{mv\cos\gamma} - \cos\gamma\cos\chi\tan\Lambda\frac{v}{R(h)} \\
& & & +2\omega(\sin\chi\cos\Lambda\tan\gamma - \sin\Lambda) - \omega^2\cos\Lambda\sin\Lambda\cos\chi\frac{R(h)}{v\cos\gamma} \\
& \dot{h} &=& v\sin\gamma \\
& \dot{\Lambda} &=& \cos\gamma\sin\chi\frac{v}{R(h)} \\
& \dot{\Theta} &=& \cos\gamma\cos\chi\frac{v}{R(h)\cos\Lambda} \\
& 0 &=& \frac{v\cos\gamma}{R(h)}(\sin\chi + \frac{\cos\chi}{\cos\Lambda}) \\
& q_{\max} &\geq& cv^3\sqrt{\varrho(h)}
\end{array}
$$

Anwendung der Sensitivitätsanalyse zur Echtzeitsteuerung:

Prädiktor

$$
p = \begin{pmatrix} \text{Starthöhe } h(0) \\ \text{Luftdichte } \rho(h) \end{pmatrix}
$$

$$
\frac{dC_L(t)}{dp}
$$

$$
\frac{d\mu(t)}{dp}
$$

Korrektor

$$
\epsilon = (500 - h(t_f))
$$

$$
\frac{dC_L(t)}{d\epsilon}
$$

$$
\frac{d\mu(t)}{d\epsilon}
$$

Abbildung 9.1: Sensitivitätsableitungen und nachkorrigierte Bahn für das Notlandemanöver

Beispiel: Spline mit Störparameter p

$$\begin{array}{lll} \min\limits_{u,x} & \int\limits_0^1 u_1^2(t)\,dt & \\ \text{unter} & \dot{x}_1(t) = x_2(t) + p_3 & \text{für } t \in [0,1] \\ & \dot{x}_2(t) = u_1(t) & \text{für } t \in [0,1] \\ & x_1(0) = p_1 & \\ & x_2(0) = p_2 & \\ & x_1(1) = 0 & \\ & x_2(1) = 1 & \end{array}$$

Abbildung 9.2: Zur Interpretation der Sensitivitätsableitungen für Optimalsteuerungsprobleme

9.3.2 Regelung

Die Trajektorie wurde erzeugt. Dank der Sensitivitätsanalyse kann man auch auf Störungen reagieren.

Fehler im Modell oder äußere Störungen lassen sich auch durch eine Regelung kompensieren:

$$u_f(t) = u_{\text{ref}}(t) - K(x(t) - x_{\text{ref}}(t)), \quad t \in [t_0, t_f]$$

Die Matrix K berechnet sich dabei aus der Lösung der algebraischen Riccati-Gleichung

$$A^T S + SA - PBR^{-1}B^T S + Q = 0$$

als

$$K = R^{-1}B^T S$$

mit den Gewichtungsmatrizen Q und R für Zustand und Steuerung, und den Matrizen des linearisierten Systems A und B:

$$A|_t = \frac{\partial}{\partial x} f(x_{\text{ref}}(t), u_{\text{ref}}(t))$$

$$B|_t = \frac{\partial}{\partial u} f(x_{\text{ref}}(t), u_{\text{ref}}(t))$$

Mit einer geeigneten Wahl der Gewichtungsmatrizen, lassen sich auf diese Weise große Störungen reduzieren, während der zusätzliche Energieverbrauch klein bleibt.

Abbildung 9.3: Mit Hilfe der Regelung lassen sich Störungen in der Anfangsposition ausgleichen.

9.3.3 Schwierigkeiten

Hier soll nur eine Problematik dargestellt werden.

Diese Optimalsteuerungsproblem mit freier Endzeit führt zu einer unstetigen Steuerung, einer sogenannten bang-bang-Steuerung.

$$
\begin{array}{lllll}
\min\limits_{u,x} & t_f & & & \\
\text{unter} & \dot{x}_1(t) & = & x_2(t) & \text{für } t \in [0,1] \\
 & \dot{x}_2(t) & = & u_1(t) & \text{für } t \in [0,1] \\
 & x_1(0) & = & 4 & \\
 & x_2(0) & = & -1 & \\
 & x_1(1) & = & 0 & \\
 & x_2(1) & = & 0 & \\
 & -1 \leq u \leq 1 & & &
\end{array}
$$

Die Steuerung ist zuerst -1, dann 1, vgl. Abb. 9.4.

Abbildung 9.4: Typische bang-bang-Steuerung

Für eine genauere Lösung müssen weitere Zwischenpunkte in der Nähe des Schaltpunktes eingeführt werden.
Das ganze Problem ist aber trivial lösbar, wenn das Optimierungsproblem nur den Umschalt-Zeitpunkt und die freie Endzeit optimieren soll.

9.3.4 TransWORHP

Das Transkriptionsverfahren TransWORHP kann Optimalsteuerungsprobleme zufriedenstellend lösen. Grafische Ausgabe für Iterationsverlauf hilft, die Lösung einzuschätzen.

Welche weiteren Anforderungen gibt es an ein Transkriptionsverfahren?

Automatische Skalierung. Für jede Steuerung und jeden Zustand kann ein typischer Bereich angegeben werden, zu dem skaliert wird. Gleichzeitig müssen auch die Gleichungsnebenbedingungen aus dem Trapez- bzw. Hermite-Simpson-Verfahren skaliert werden, ebenso wie alle weiteren Neben- und Randbedingungen.

DAE Systeme. Bisher werden Nebenbedingungen dieser Form berücksichtigt:

$$\dot{x}(t) = f(x(t), u(t), t)$$

Zusätzlich lassen sich noch algebraische Gleichungen einhalten:

$$0 = a(x(t), u(t), t)$$

Mehrphasen-Systeme. Stufen von Raketen, Längere Missionen in Erdnähe/Mondnähe unterschiedliche DGls, erfordern phasenweise Formulierung und Verbinden der Phasen mit Zusatzbedingungen

Zusätzlich auch komplexe Fragestellungen lösbar: Bilevel-Probleme: Wie muss man fliegen, um stets ein sicheres Alternativ-Manöver (z.B. Brems-Manöver) zu garantieren.

Vektorwertige Zielfunktion. Oft unterschiedliche Gütekriterien in Konkurrenz: Zeit, Energie, Verschleiß. Bestimmung der Gewichtungsfaktoren nicht intuitiv. Alle Gewichtungen ausprobieren, um eine "Pareto-Front" zu erzeugen, anhand derer die Auswahl leichter fällt.

Wie lässt sich diese Pareto-Front leicht finden? Nachbar-Lösung als Startschätzung, Sensitivitätsableitung bzgl. Gewichtungen.

10 Lizenzen

10.1 Third-party Lizenzen

10.1.1 Lapack

- Copyright (c) 1992-2013 The University of Tennessee and The University of Tennessee Research Foundation. All rights reserved.
- Copyright (c) 2000-2013 The University of California Berkeley. All rights reserved.
- Copyright (c) 2006-2013 The University of Colorado Denver. All rights reserved.

COPYRIGHT
Additional copyrights may follow
HEADER
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer listed in this license in the documentation and/or other materials provided with the distribution.
- Neither the name of the copyright holders nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

The copyright holders provide no reassurances that the source code provided does not infringe any patent, copyright, or any other intellectual property rights of third parties. The copyright holders disclaim any liability to any recipient for claims brought against recipient by any third party for infringement of that parties intellectual property rights.
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS ÄS ISÄND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.