Institute of
Applied Mechanics
Institut für Baumechanik

Graz University of Technology

Bachelorarbeit im Studiengang Bauingenieurwissenschaften
am Institut für Baumechanik

Technische Universität Graz

Modellierung und nachhaltige Optimierung von Seilkräften am Beispiel einer Straßenlaterne

Paula Gaugl

Graz, 14. November 2023

Betreuer: Ass.Prof.Dipl.-Ing.Dr.techn.Bsc
Michael Helmut Gfrerer

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Abstract

The aim of this scientific study is to investigate the forces in ropes within a structural system, focusing on a streetlamp suspended by two cables attached to upright supports. The research objectives encompass the calculation of rope forces, both when dealing with a rigid and when dealing with an elastic rope. Additionally, the study wants to establish a cost estimation model for the streetlamp, considering factors such as rope surface area, tension and sag. The overarching goal of this investigation is the optimization of costs (minimization).

The methodology employed in this work involves the formulation of a mathematical model and comprehensive calculations with several plots for a better visualization, aided by the web-based calculation platform Jupyter Notebook and the programming language Python. This enables the user to interact with the model directly.

The results of the study indicate that when dealing with rigid ropes, the solutions are relatively straightforward. However, even minor changes in the case of elastic ropes significantly complicate the calculations, leading to a substantial increase in computational complexity. The cost optimization shows that only if the weight if the lamp is high enough, there are significant changes in the costs.

Zusammenfassung

Die vorliegende wissenschaftliche Arbeit widmet sich der Berechnung und Analyse von Seilkräften in einem strukturellen System. Konkret geht es um eine Straßenlaterne, die symmetrisch von zwei Seilen gehalten wird, welche an Stützen befestigt sind. Ziel dieser Forschung ist die Berechnung der auf die Seile wirkenden Kräfte, zum einen wenn es sich um ein starres, zum anderen, wenn es ich um ein dehnbares Seil handelt. Darüber hinaus befasst sich die Arbeit mit der Entwicklung eines Kostenmodells für die Straßenlaterne, das die Faktoren Seilfläche, Seilkraft und Seildurchhang berücksichtigt. Das übergeordnete Ziel dieser Untersuchung besteht in der Optimierung der Kosten, in Form einer Minimierung.

Die angewandte Methodik dieser Arbeit besteht in der Formulierung eines mathematischen Modells sowie umfassenden Berechnungen und Visualisierungen der Ergebnisse auf der webbasierten, Online-Rechenplattform Jupyter Notebook mithilfe der Programmiersprache Python. Dadurch wird ein interaktiver Umgang mit dem Modell möglich.

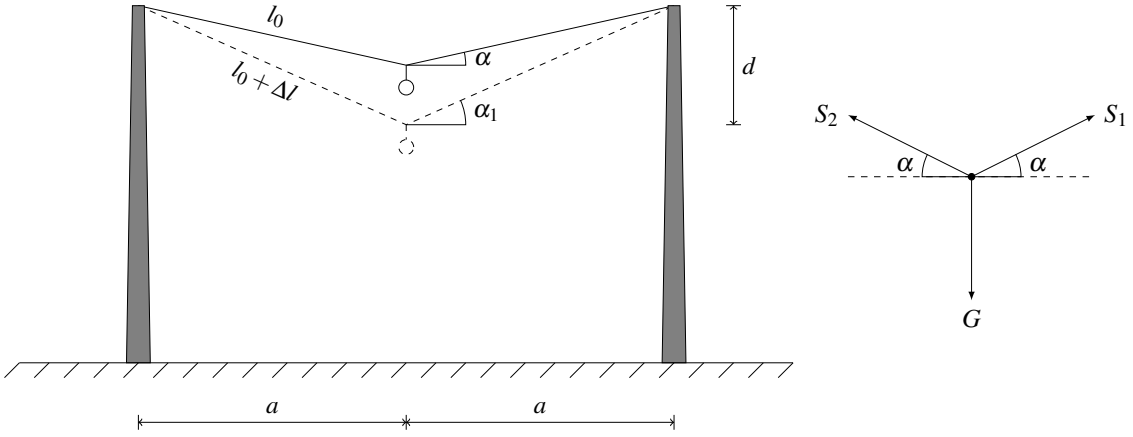
Die Ergebnisse der Untersuchung zeigen, dass die Berechnung der Seilkräfte bei starren Seilen ein vergleichsweise einfaches Modell darstellt, während bei dehnbaren Seilen selbst geringfügige Änderungen eine erhebliche Komplexität in den Berechnungen hervorrufen, was massive Auswirkungen auf den erforderlichen Rechenaufwand hat. Aus den Ergebnissen der Kostenoptimierung ist zu erkennen, dass das Kosteneinsparungspotential erst bei großem Eigengewicht des Beleuchtungskörpers ausgeschöpft wird.

INHALTSVERZEICHNIS

1. EINLEITUNG

Modellbildung und -beschreibung

Das System besteht aus zwei Stehern gleicher Höhe, zwei Seilen und einem Beleuchtungskörper, der symmetrisch aufgehängt ist. Es wird angenommen, dass die Steher durch ein dreiwertiges Auflager starr mit dem Untergrund verbunden sind. Die Lampe wird idealisiert als Massepunkt dargestellt. Das Seil wird zuerst als starr, in weiterer Folge als dehnbar angenommen.
Der horizontale Abstand des Mittelpunktes der Lampe zur Symmetrieachse der Steher wird als a, die Ursprungslänge der Seile als l_0, der Winkel als α und der Durchhang als d bezeichnet. Ist das Seil dehnbar ergibt sich die Seillänge zu $l_0 + \Delta l$ und der sich daraus ergebende Winkel zu α_1. Die Variablen S_1 und S_2 beschreiben die Seilkraft (siehe Abbildung 1.1a).
Während für die Berechnung der Seilkraft mit einem starren Seil ein Knotenfreischnitt um den Massepunkt des Beleuchtungskörpers (vgl. Abbildung 1.1b) und zwei Kräftegleichgewichte (horizontal und vertikal) ausreichen, wird beim dehnbaren Seil durch die zusätzliche Unbekannte Δl eine neue Gleichung benötigt. Dazu wird das Hookesche Gesetz herangezogen, das einen linearen Zusammenhang zwischen Spannung (σ) und Dehnung (Verzerrung) (ϵ) mithilfe des E-Modul beschreibt. Die Spannung wird dabei als Kraft/Fläche, die Dehnung als $\frac{\Delta l}{l_0}$, das heißt als relative Längenänderung des Seils im Vergleich zur Ursprungslänge, dargestellt werden. Der Winkel α_1 wird durch einen trigonometrischen Zusammenhang näher bestimmt. Dadurch kann die Seilkraft des dehnbaren Seiles, wenn auch nur in impliziter Form, ausgedrückt werden.



(a) Schematische Darstellung (b) Freischnitt

Abbildung 1.1: Modell Straßenaterne

Jupyter Notebook

Jupyter Notebook ist eine kostenlose webbasierte interaktive Rechenplattform. Das Besondere daran ist, dass Code und Output direkt untereinander einsehbar sind. Der Output ist interaktiv und leicht zu teilen. Der Code kann damit einfach online geöffnet und angepasst werden. Die Änderungen sind dabei unmittelbar einsehbar. Durch sogenannte Markdown-Zellen, die nur Text enthalten, ist es möglich den Code ausführlich und anschaulich zu beschreiben und dadurch die Nachvollziehbarkeit zu erhöhen.
Als Programmiersprache wird Python verwendet.

2. LÖSUNGSMETHODEN

Nachfolgend sind die beiden Jupyter Notebooks eingefügt. Im ersten werden die Gleichungen zur Berechnung der Seilkräfte des starren und des dehnbaren Seils aufgestellt, gelöst und die Ergebnisse visualisiert. Das zweite beinhaltet die Kostenrechnung und -minimierung.
Die Informationen über das Freischneiden der Lampe und das Aufstellen der Kräftebilanzen können [2], [3] und [8] entnommen werden. Für das Differenzieren und Anwenden von diversen mathematischen Aussagen wie dem Satz der impliziten Funktionen oder dem Satz von Schwarz wurden die Quellen [1] und [6] herangezogen. Die Anwendungsmöglichkeiten und der Jupyter Notebook Bibliotheken NumPy, Matplotlib, Sympy und SciPy sind in [4], [5], [7] und [9] beschrieben.

1 Berechnung der Seilkräfte - Vergleich starres mit dehnbarem Seil

1.1 starres Seil:

Kräftegleichgewicht in horizontaler Richtung:
$\Sigma F_h = 0 : S_1 = S_2$
Kräftegleichgewicht in vertikaler Richtung:
$\Sigma F_v = 0 : G - 2S_1 \sin(\alpha) = 0$
daraus folgt:
$S_1 = \frac{G}{2\sin(\alpha)}$ mit $\alpha = \arccos(\frac{a}{l_0})$

1.2 dehnbares Seil:

Hookesches Gesetz: $\sigma = E\epsilon = E\frac{\Delta l}{l_0}$ mit $\sigma = \frac{F}{A}$
daraus folgt: $\Delta l = \frac{S_1 l_0}{AE}$
Trigonometrie: $\alpha_1 = arccos(\frac{a}{l_0+\Delta l})$
umgeformt und Δl eingesetzt ergibt sich für S_1 eine implizite Gleichung: $S_1 = \frac{G}{2\sin\left(\arccos(\frac{a}{l_0(1+\frac{S_1}{AE})})\right)}$

mit $sin(arccos(x)) = \sqrt{(1-x^2)}$ erhält man: $S_1 = \frac{G}{2\sqrt{1-x^2}}$ mit $x = \frac{a}{l_0(1+\frac{S_1}{AE})}$

[2]:
```
import sympy as sp
import numpy as np
import scipy as sc
from scipy import optimize
import matplotlib.pyplot as plt
```

[3]:
```
G = sp.Symbol('G') #Eigengewicht Lampe
S_1 = sp.Symbol('S_1') #Seilkraft
a = sp.Symbol('a') #horizontaler Abstand Lampe Steher
l_0 = sp. Symbol('l_0') #ursprüngliche Seillänge
A = sp.Symbol('A') #Querschnittsfläche Seil
E = sp.Symbol('E') #E-Modul Seil
x = sp.Symbol('x') #Variable
implizite_funktion=sp.lambdify([x,G,a,A,E,l_0],x-G/(2*(1-(a/(l_0*(1+x/A/
 ↪E)))**2)**(1/2)))
explizite_funktion_starr=sp.lambdify([x,G,a,l_0],x-G/(2*(1-(a/(l_0))**2)**(1/2)))
```

[4]:
```
G = 10*9.81 #N
a = 4 #m
l_0 = 4.001 #m
E = 200000 * 10**6 #N/m^2
A = 0.002**2*np.pi/4 #m^2
```

Die Schwierigkeit beim Ermitteln der Seilkraft des dehnbaren Seils besteht darin, dass S_1 nicht explizit ermittelt werden kann (d.h. die Gleichung kann nicht nach S_1 umgeformt werden) und sich die Funktion damit nur in impliziter Form darstellen lässt.
Um diese implizite Gleichung zu lösen, müssen die Nullstellen der Funktionen gefunden werden.

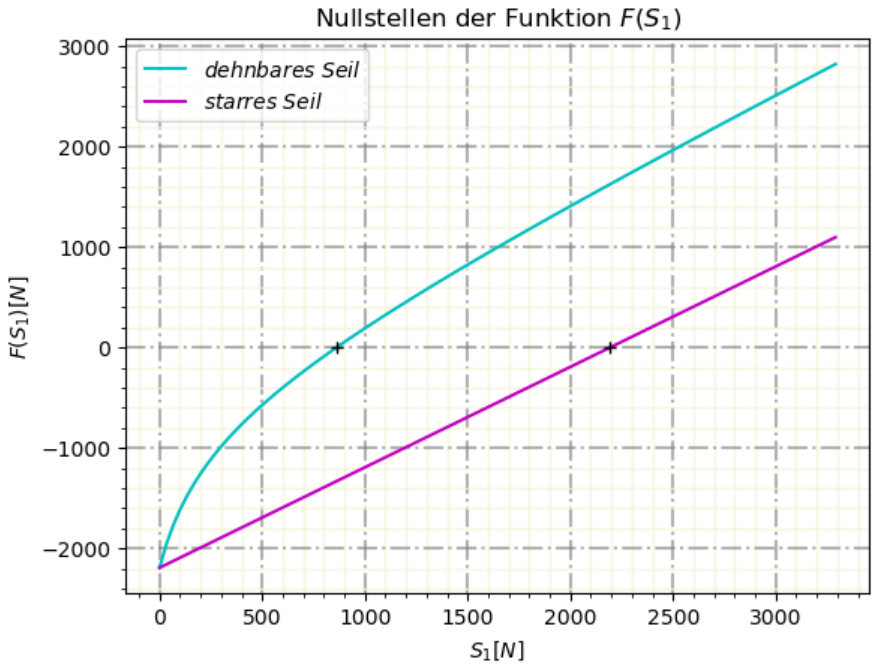
Dehnbares Seil: $F(S_1) = S_1 - \frac{G}{2\sqrt{1-x^2}} = 0$ mit $x = \frac{a}{l_0\left(1+\frac{S_1}{AE}\right)}$

Um das dehnbare mit dem starren Seil vergleichen zu können, werden auch von der Funktion des starren Seils die Nullstellen ermittelt, wobei hier eine explizite Darstellung der Seilkraft möglich ist.

Starres Seil: $F(S_1) = S_1 - \frac{G}{2\sqrt{1-\left(\frac{a}{l_0}\right)^2}} = 0$

Als Eingabewerte für die weiteren Berechnungen werden hier beispielshaft die Materialkennwerte eines Stahlseils eingesetzt. Der Steherabstand beträgt 8m, die Masse des Beleuchtungskörpers 10kg und der Seildurchmesser 2mm.

[8]:
```
sol = optimize.root(implizite_funktion, [0], args=(G,a,A,E,l_0), jac=False)

sol2 = optimize.root(explizite_funktion_starr, [0], args=(G,a,l_0), jac=False)

x1 = np.linspace(0*sol2.x,1.5*sol2.x, 100)
y1 = x1-G/(2*(1-(a/(l_0*(1+x1/A/E)))**2)**(1/2))
y2 = x1-G/(2*(1-(a/(l_0))**2)**(1/2))


plt.figure()

plt.plot(x1, y1, 'c')
plt.plot(x1, y2, 'm')
plt.plot(sol.x, 0, 'k+')
plt.plot(sol2.x, 0, 'k+')

plt.grid(which='major', color='gray', alpha=0.6, linestyle='dashdot', lw=1.5)
plt.minorticks_on()
plt.grid(which='minor', color='beige', alpha=0.8, ls='-', lw=1)
plt.title('Nullstellen der Funktion $F(S_1)$')
plt.xlabel('$S_1 [N]$')
plt.ylabel('$F(S_1) [N]$')
plt.legend([r'$dehnbares$ $Seil$', r'$starres$ $Seil$'], loc='best')

plt.show()
```



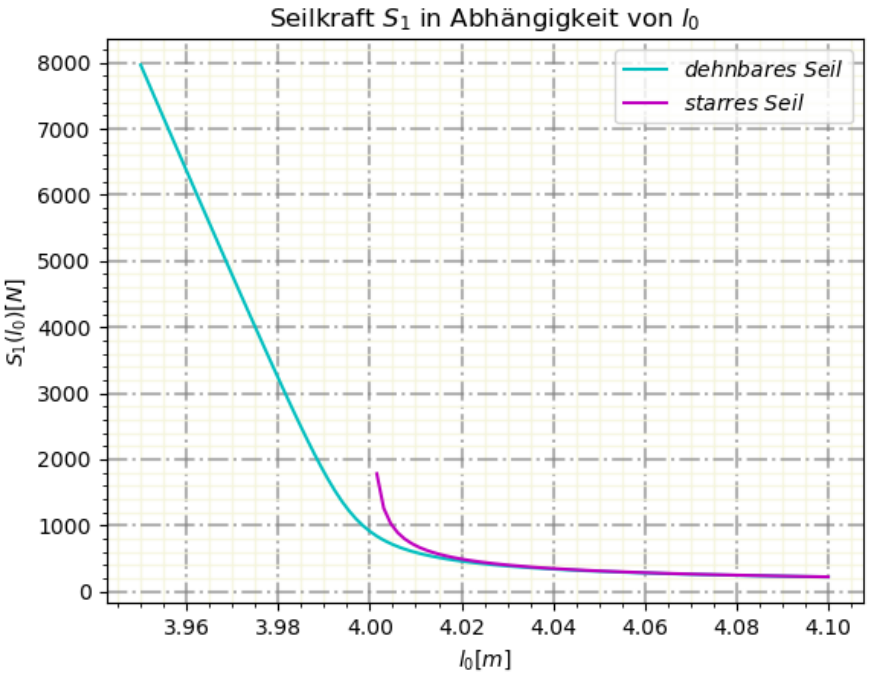
1.3 Seilkraft S_1 in Abhängigkeit der Seillänge l_0

Mithilfe einer for-Schleife wird die Seillänge l_0 variabel gehalten und erlaubt so eine Darstellung der Seilkraft S_1 in Abhängigkeit von l_0.

Aufgrund der Tatsache, dass die Länge des starren Seils nicht kleiner als der horizontale Abstand zwischen der Lampe und der Stütze a werden kann, muss die Schleife des starren Seils bei $l_0 = a$ abgebrochen werden, um einen negativen Radikand (Ausdruck unter der Wurzel) zu vermeiden.
Im Gegensatz dazu kann das dehnbare Seil über seine ursprüngliche Länge hinaus gedehnt/gestreckt d.h. vorgespannt werden. Der Wert, dem sich die Funktion unendlich annähert (die Asymptote), ist 0.

Aus der Abbildung ist ersichtlich, dass die Seilkraft sowohl für das starre, als auch für das dehnbare Seil mit Abnahme der Seillänge zunimmt - das heißt: Je kleiner der Winkel zwischen dem Seil und der Horizontalen bzw. je größer das Verhältnis $\frac{a}{l_0}$ ist, desto größer ist die Seilkraft.
Geht l_0 gegen 0, dann steigt der Wert der Seilkraft des dehnbaren Seils rechnerisch ins Unendliche. Weiters ist die Vorspannung des dehnbaren Seils gut erkennbar. Durch das Einbauen eines zu kurzen Seils, wird schon vor dem Einbau des Seils Spannung bzw. Kraft in das System eingebracht.

[6]:
```
i = sp.Symbol('i') #i entspricht l_0
sol3=[]
sol4 = []
ll = np.linspace(3.95,4.1,100) #Bereich, über den iteriert wird
ll4=[]

for i in ll:
    sol3.append(optimize.root(implizite_funktion, [max(E*A*(a/i-1)*1.001, 0.
 ↪001)],args=(G,a,A,E,i), jac=False).x)

    if i<=a:
        continue
    sol4.append(G/(2*(1-(a/(i*(1)))**2)**(1/2)))
    ll4.append(i)

plt.plot(ll, sol3, 'c-')
plt.plot(ll4, sol4, 'm-')

plt.grid(which='major', color='gray', alpha=0.6, linestyle='dashdot', lw=1.5)
plt.minorticks_on()
plt.grid(which='minor', color='beige', alpha=0.8, ls='-', lw=1)
plt.title('Seilkraft $S_1$ in Abhängigkeit von $l_0$')
plt.xlabel('$l_0 [m]$')
plt.ylabel('$S_1(l_0) [N]$')
plt.legend([r'$dehnbares$ $Seil$', r'$starres$ $Seil$'], loc='best')

plt.show()
```



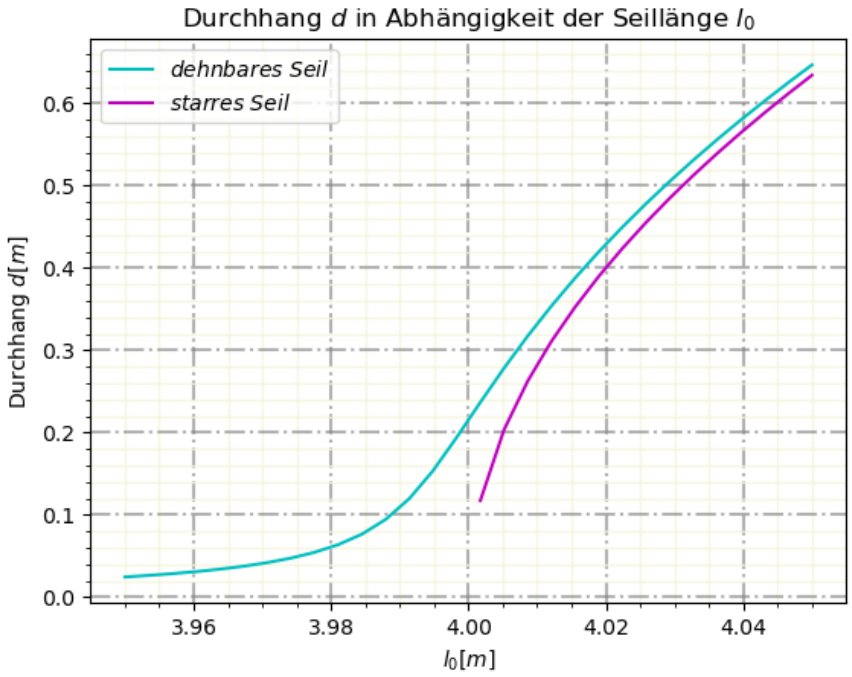
1.4 Ermitteln des Durchhangs d in Abhängigkeit der Seillänge l_0

Mithilfe des Satz des Pythagoras kann der Durchhang d des Seils berechnet werden. Beim dehnbaren Seil verlängert sich die ursprüngliche Seillänge l_0 um Δl.

Grundsätzlich gilt: Je größer l_0 ist, desto größer ist auch der Durchhang d. Während das starre Seil bei $l_0 = a$ den minimalen Durchhang erreicht, kann l_0 beim dehnbaren Seil auch kleinere Werte als a annehmen. Der minimale Durchhang wird dann bei einem unendlich kleinen l_0 erreicht.

[7]:
```
j = sp.Symbol('j') #j entspricht l_0
d_dehnbar = []
d_starr = []
ll5 = np.linspace(3.95,4.05,30)
ll6 = []

for j in ll5:
    sol5 = optimize.root(implizite_funktion, [max(E*A*(a/j-1)*1.001, 0.
↪001)],args=(G,a,A,E,j), jac=False)
    l = (1+sol5.x/A/E) * j #delta l_0
```

```
    d_dehnbar.append((l**2-a**2)**(1/2)) #d_dehnbarr...Durchhang dehnbares Seil
    
    if j<=a:
        continue
    d_starr.append((j**2-a**2)**(1/2)) #d_starr...Durchhang starres Seil
    ll6.append(j)

plt.plot(ll5, d_dehnbar, 'c-')
plt.plot(ll6, d_starr, 'm-')


plt.grid(which='major', color='gray', alpha=0.6, linestyle='dashdot', lw=1.5)
plt.minorticks_on()
plt.grid(which='minor', color='beige', alpha=0.8, ls='-', lw=1)
plt.title('Durchhang $d$ in Abhängigkeit der Seillänge $l_0$')
plt.xlabel('$l_0 [m]$')
plt.ylabel('Durchhang $d [m]$')
plt.legend([r'$dehnbares$ $Seil$', r'$starres$ $Seil$'], loc='best')

plt.show()
```



2 Kostenfunktion des dehnbaren Seils

[1]:
```
import sympy as sp
import numpy as np
from scipy import optimize
import matplotlib.pyplot as plt
```

Für die Berechnung werden verschiedene erweiternde Module eingebunden, die auf der Programmiersprache Python basieren.
Das Modul "SymPy" wird verwendet um symbolische (algebraische) Berechnungen durchzuführen, "NumPy" steht für Numeric Python und ermöglicht effizientes (numerisches) Rechnen mit mathematischen Funktionalitäten und großen Matrizen, "SciPy" stellt fundamentale Algorithmen für wissenschaftliches Arbeiten zur Verfügung und "Matplotlib" ist eine Bibliothek zur Visualisierung von Daten.

2.1 Ermitteln der Ableitungen

Alle nachfolgenden Berechnungen basieren auf einer Funktion, die sich aus dem Seilvolumen (Querschnitt und Ursprungsänge des Seils), der Seilkraft und dem Durchhang (Berechnung siehe zuvor) zusammensetzt. Konstanten ermöglichen eine unterschiedliche Gewichtung der einzelnen Komponenten. In weiterer Folge wird diese Funktion Kostenfunktion genannt und lässt folgendermaßen darstellen: $K = c_1 \cdot A \cdot l_0 + c_2 \cdot S_1 + c_3 \cdot d$.
Es ist anzumerken, dass das Ziel dieser Berechnungen nicht ist, die "realen" Kosten einer Straßenbeleuchtungskonstruktion abzubilden. Es geht vielmehr um die mathematische Vorgehensweise und die Anschaulichkeit der Ergebnisse.
Die Zustandsgleichung ist die homogene Form der Gleichung zur Berechnung der Seilkraft.

[2]:
```
G = sp.Symbol('G') #Eigengewicht Lampe
a = sp.Symbol('a') #horizontaler Abstand Lampe Steher
l_0 = sp.Symbol('l_0') #ursprüngliche Seillänge
S = sp.Function('S')(l_0) #Seilkraft abhängig von l_0 (Funktion)
S_1 = sp.Symbol('S_1') #Seilkraft (symbolisch)
A = sp.Symbol('A') #Querschnittsfläche Seil
E = sp.Symbol('E') #E-Modul Seil
c1 = sp.Symbol('c1') #Konstante
c2 = sp.Symbol('c2') #Konstante
c3 = sp.Symbol('c3') #Konstante

Kostenfunktion = c1*A*l_0+c2*S_1+c3*(((1+S_1/A/E) * l_0)**2-a**2)**(1/2)
kosten_funktion=sp.lambdify([c1, c2, c3, a, A, E, l_0, S_1], Kostenfunktion)
 ↪#lambdify: SymPy -> NumPy d.h. symbolisch -> numerisch
Zustandsgleichung = S_1-G/(2*(1-(a/(l_0*(1+S_1/A/E)))**2)**(1/2))
implizite_funktion=sp.lambdify([S_1,G,a,A,E,l_0],Zustandsgleichung)
```

Für die Kostenoptimierung ist es notwendig die Nullstellen der Ableitung der Kostenfunktion zu suchen. Ziel ist es, eine Seillänge l_0 zu finden, für die die Kosten minimal werden. Da die Kostenfunktion jedoch von der Seilkraft abhängt, die ihrerseits ebenfalls von der Seillänge abhängig ist, wird zuerst von dieser gesondert die Ableitung berechnet. Die Größen a, E, A werden als konstant angenommen.

Zustandsgleichung: $f(l_0, S_1(l_0)) = S_1 - \frac{G}{2\sqrt{1-x^2}} = 0$ mit $x = \frac{a}{l_0\left(1+\frac{S_1}{AE}\right)}$

Ableiten und Nullsetzen der Zustandsgleichung: $\frac{df}{dl_0} = \frac{\partial f}{\partial l_0} + \frac{\partial f}{\partial S_1}\frac{dS_1}{dl_0} = 0$ (Kettenregel)

Umgeformt ergibt dies die erste Ableitung der Seilkraft: $\frac{dS_1}{dl_0} = -\left(\frac{\partial f}{\partial S_1}\right)^{-1}\frac{\partial f}{\partial l_0}$ (Satz der impliziten Funktionen)

[3]:
```
f = Zustandsgleichung.subs(S_1,S)
dS_l0 = sp.solve(sp.diff(f,l_0),sp.diff(S,l_0))[0].simplify() #erste Ableitung
↪der Zustandsgleichung nach l_0
dS_l0
```

[3]:
$$-\frac{A^2E^2Ga^2\left(AE+S(l_0)\right)}{l_0\left(A^2E^2Ga^2+2.0l_0^2\left(-\frac{A^2E^2a^2-l_0^2(AE+S(l_0))^2}{l_0^2(AE+S(l_0))^2}\right)^{1.5}(AE+S(l_0))^3\right)}$$

Anstatt *solve* könnte hier auch eine einfache Division verwendet werden. Ist S_1 jedoch kein Skalar sondern ein Vektor (mehrere Seilkräfte in einem System), ergibt die Ableitung eine Matrix. Um eine solche lösen zu können, wird der Befehl *solve* benötigt.

Durch das Substituieren von S_1 (Variable) durch S (Funktion, abhängig von l_0) wird die symbolische Rechenfähigkeit von SymPy ausgenutzt. Dadurch wird in weiterer Folge z.B. beim Ableiten die Kettenregel automatisch angewandt.

Die zweite Ableitung der Zustandsgleichung f berechnet sich händisch wie folgt:

$$\begin{aligned}\frac{d}{dl_0}\left(\frac{df}{dl_0}\right) &= \frac{\partial^2 f}{\partial^2 l_0} + \frac{\partial f}{\partial S_1\partial l_0}\frac{dS_1}{dl_0} + \frac{\partial f}{\partial l_0\partial S_1}\frac{dS_1}{dl_0} + \frac{\partial f}{\partial^2 S_1}\left(\frac{dS_1}{dl_0}\right)^2 + \frac{\partial f}{\partial S_1}\frac{d^2S_1}{d^2l_0}\\ &= \frac{\partial^2 f}{\partial^2 l_0} + 2\frac{\partial f}{\partial S_1\partial l_0}\frac{dS_1}{dl_0} + \frac{\partial^2 f}{\partial^2 S_1}\left(\frac{dS_1}{dl_0}\right)^2 + \frac{\partial f}{\partial S_1}\frac{d^2S}{d^2l_0}\\ &= 0\end{aligned}$$

Für die Vereinfachung von $\frac{\partial f}{\partial S_1\partial l_0}\frac{dS_1}{dl_0}$ und $\frac{\partial f}{\partial l_0\partial S_1}\frac{dS_1}{dl_0}$ wird der Satz von Schwarz angewandt. Er besagt, dass die Reihenfolge, in der die partiellen Differentiationen nach den einzelnen Variablen durchgeführt werden, nicht entscheidend für das Ergebnis ist und somit vertauscht werden können. Somit kann man diese zu $2\frac{\partial f}{\partial S_1\partial l_0}\frac{dS_1}{dl_0}$ zusammenfassen.

Die zweite Ableitung der Seilkraft ergibt sich damit zu:

$$\frac{d^2S_1}{d^2l_0} = -\left(\frac{\partial f}{\partial S_1}\right)^{-1}\left[\frac{\partial^2 f}{\partial^2 l_0} + 2\frac{\partial f}{\partial S_1\partial l_0}\frac{dS_1}{dl_0} + \frac{\partial^2 f}{\partial^2 S_1}\left(\frac{dS_1}{dl_0}\right)^2\right]$$

[4]:
```
ddS_l0 = sp.solve(sp.diff(f,l_0,l_0),sp.diff(S,l_0,l_0))[0].subs(sp.
↪diff(S,l_0),dS_l0) #zweite Ableitung der Zustandsgleichung nach l_0
#ddS_l0.subs(S,S_1) (Aufgrund der Länge des Ausdrucks nicht dargestellt)
```

Nun wird die Kostenfunktion nach l_0 abgeleitet, wobei die zuvor berechnete Ableitung der Seilkraft darin eingesetzt wird. Durch nochmaliges Differenzieren und Einsetzten der ersten und zweiten Ableitung der Seilkraft ergibt sich dann die zweite Ableitung der Kostenfunktion.

[5]:
```
dK_l0=sp.diff(Kostenfunktion.subs(S_1,S),l_0).subs(sp.diff(S,l_0),dS_l0).
↪subs(S,S_1) #erste Ableitung der Kostenfunktion nach l_0
```

[6]:
```
ddK_l0=sp.diff(Kostenfunktion.subs(S_1,S),l_0,l_0).subs(sp.
↪diff(S,l_0,l_0),ddS_l0).subs(sp.diff(S,l_0),dS_l0).subs(S,S_1)
#zweite Ableitung der Kostenfunktion nach l_0
```

Im nächsten Schritt werden zwei Funktionen definiert, die die erste und zweite Ableitung der Kostenfunktion darstellen. Durch den Befehl *lambdify* werden die symbolischen Variablen aus SymPy in die numerische Bibliothek von NumPy verschoben. Es sind dadurch numerische Berechnungen möglich.

[7]:
```
fun=sp.lambdify([A, E, G, S_1, a, c1, c2, c3, l_0],dK_l0) #erste Ableitung der␣
↪Kostenfunktion
fun_second_derivative_K=sp.lambdify([A, E, G, S_1, a, c1, c2, c3, l_0],ddK_l0)␣
↪#zweite Ableitung der Kostenfunktion
```

2.2 Kostenverteilung

Nun sollen Plots erzeugt werden, die sowohl die Teil- als auch die Gesamtkosten in Abhängigkeit von der Seillänge l_0 darstellen. Es werden zwei Abbildungen (Plot A und Plot B) geplottet, deren Eingangsdaten sich nur in der Gewichtskraft des Beleuchtungskörpers unterscheiden. Den Konstanten c_1, c_2, c_3 sind beliebige Werte zugewiesen.
Die "implizite_Funktion" wurde bereits zu Beginn definiert und entspricht der Zustandsgleichung für die Berechnung der Seilkraft.

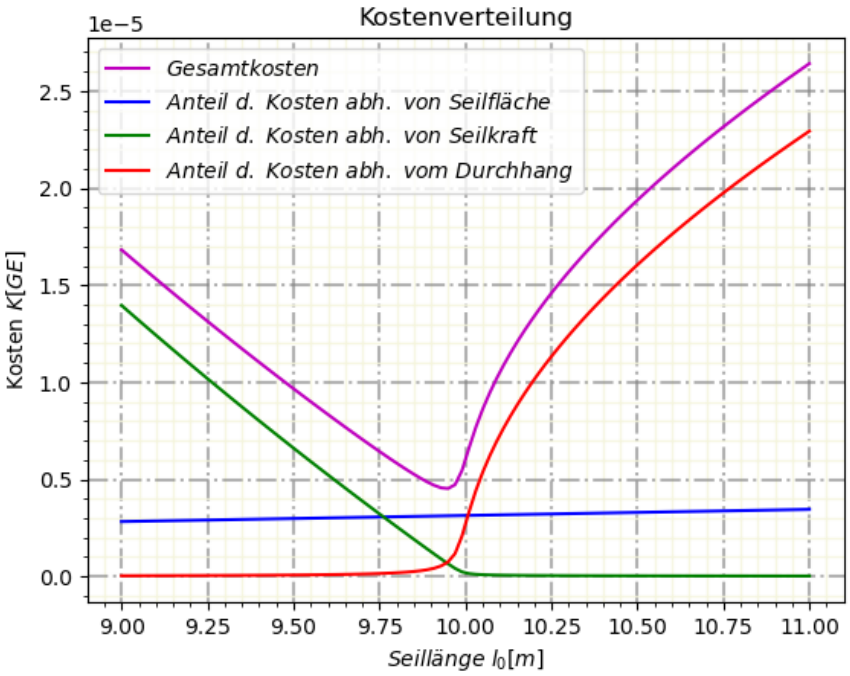
[8]:
```
def plotKosten():
    intervall=np.linspace(a*0.9,a*1.1,100)
    ges=[]
    k1=[]
    k2=[]
    k3=[]
    for j in intervall: #j entspricht l_0
        sol1 = optimize.root(implizite_funktion, [max(E*A*(a/j-1)*1.01, 0.
↪001)],args=(G,a,A,E,j), jac=False,tol=10**-9,method='lm')
        sol1=sol1.x #Seilkraft
        ges.append(kosten_funktion(c1, c2, c3, a, A, E, j, sol1)) #Gesamtkosten
        k1.append(c1*A*j) #Anteil der Kosten abhängig von der Seilfläche
        k2.append(c2*sol1) #Anteil der Kosten abhängig von der Seilkraft
        d = (((1+sol1/A/E) * j)**2-a**2)**(1/2) #Durchhang
```

```
        k3.append(c3*(((1+sol1/A/E) * j)**2-a**2)**(1/2)) #Anteil der Kosten␣
↪abhängig vom Durchhang

    plt.plot(intervall, ges, 'm-')
    plt.plot(intervall,k1, 'b-')
    plt.plot(intervall,k2, 'g-')
    plt.plot(intervall,k3, 'r-')

    plt.grid(which='major', color='gray', alpha=0.6, linestyle='dashdot', lw=1.5)
    plt.minorticks_on()
    plt.grid(which='minor', color='beige', alpha=0.8, ls='-', lw=1)
    plt.title('Kostenverteilung')
    plt.xlabel('$Seillänge$ $l_0 [m]$')
    plt.ylabel('Kosten $K [GE]$')
    plt.legend([r'$Gesamtkosten$', r'$Anteil$ $d.$ $Kosten$ $abh.$ $von$␣
↪$Seilfläche$', r'$Anteil$ $d.$ $Kosten$ $abh.$ $von$ $Seilkraft$', r'$Anteil$␣
↪$d.$ $Kosten$ $abh.$ $vom$ $Durchhang$'], loc='best');
```

2.2.1 Plot A:

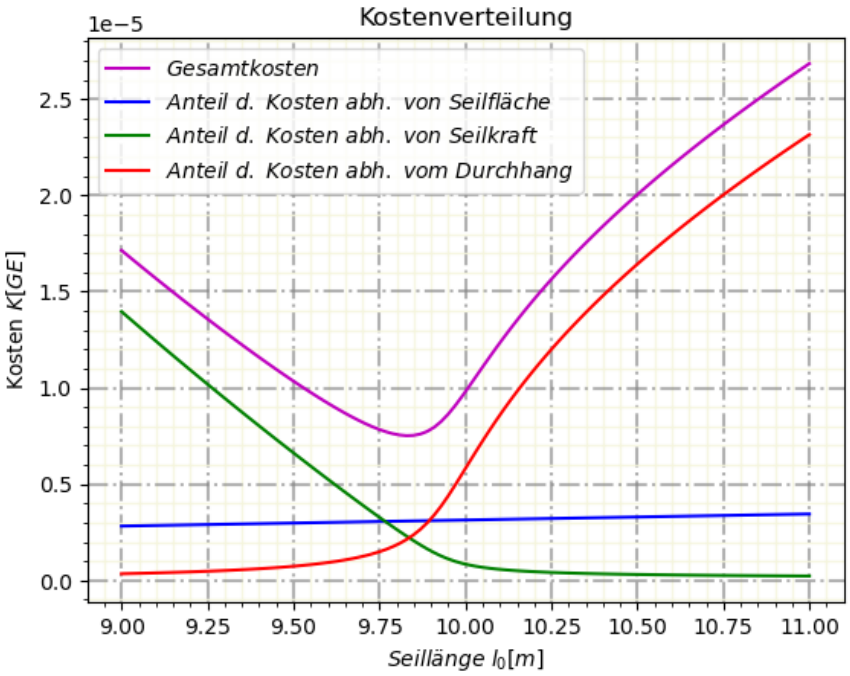
[9]:
```
#Plot A
G = 10*9.81 # N
a = 10 # m
l_0 = 4.1 # m
E = 200000 * 10**6 # N/m^2
A = 0.002**2*np.pi/4 # m^2
c1 = 0.1 #Konstante für die Seilfläche
c2 = 0.0000000002 #Konstante für die Seilkraft
c3 = 0.000005 #Konstante für den Durchhang

plotKosten()
```

```
<lambdifygenerated-2>:2: RuntimeWarning: invalid value encountered in power
  return -1/2*G*(-a**2/(l_0**2*(1 + S_1/(A*E))**2) + 1)**(-0.5) + S_1
```



2.2.2 Plot B:

[10]:
```
#Plot B: die Gewichtskraft der Lampe ist im Vergleich zu Plot A um den Faktor 10
 ↪größer
G = 100*9.81 # N
a = 10 # m
l_0 = 4.1 # m
E = 200000 * 10**6 # N/m^2
A = 0.002**2*np.pi/4 # m^2
c1=0.1
c2=0.0000000002
c3=0.000005

plotKosten()
```

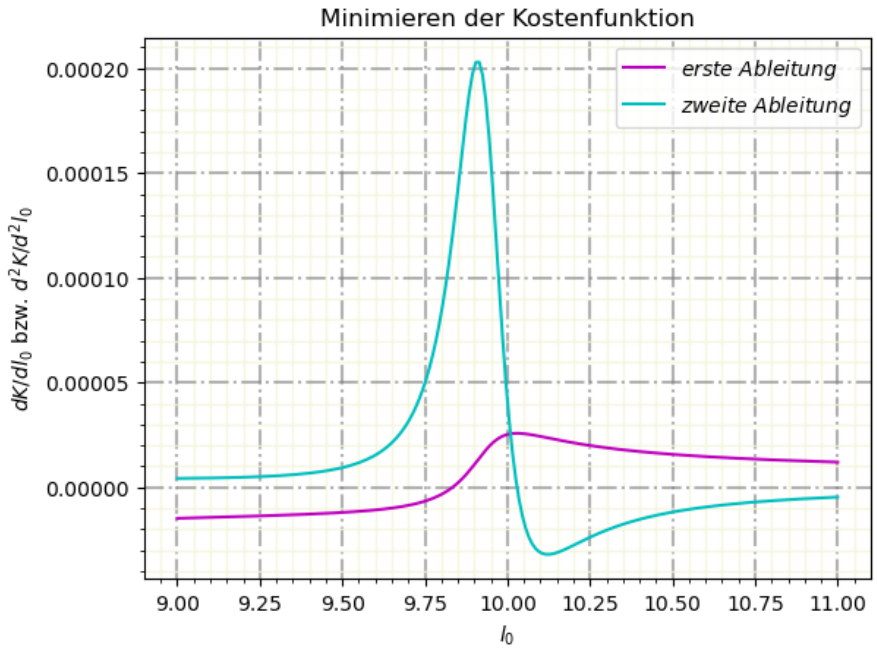


Den geplotteten Graphen A und B kann entnommen werden, dass das Minimum der Gesamtkosten etwas unter dem horizontalen Abstand von Lampe zu Steher etwa bei 9.8 liegt ($a = 10$). Das heißt, um die Kosten zu minimieren ist es sinnvoll ein etwas kürzeres Seil einzubauen und dieses vorzuspannen. Je kürzer die Ursprungslänge des verwendeten Seils ist, desto dominanter ist jedoch der Kostenanteil aus der Seilkraft und desto ausschlaggebender somit die Seilgüte. Wird ein längeres Seil eingebaut, geht der Kostenanteil aus der Seilkraft gegen 0, da das Seil keine Vorspannkräfte aufnehmen muss. Dafür werden dann hier die Kosten abhängig vom Durchhang sehr relevant. Bei $l_0 > a$ entstehen durch die größere Seillänge höhere Kosten. Der Anteil resultierend aus der Seilfäche hat einen relativ konstanten Einfluss auf die Gesamtkosten. Die Form der Gesamtkostenfunktion wird also maßgeblich von den Kostenanteilen aus Seilkraft und Durchhang bestimmt.

Werden Plot A und Plot B miteinander verglichen, fällt auf, dass sich der Tiefpunkt der Gesamtkosten von Plot B weiter rechts befindet. Das bedeutet, je schwerer die Lampe ist, desto mehr sollte das Seil vorgespannt werden.
Die Größe der jeweiligen Teilkosteneinflüsse kann durch die Kontanten c_1, c_2, c_3 geregelt werden.

2.3 Minimieren der Kostenfunktion

[11]:
```
intervall=np.linspace(a*0.9,a*1.1,200)
first_deriv = []
second_deriv = []

for j in intervall: #j entspricht l_0
    sol1 = optimize.root(implizite_funktion, [max(E*A*(a/j-1)*1.01, 0.
 ↪001)],args=(G,a,A,E,j), jac=False, tol=10**-9,method='lm').x #Seilkraft
    first_deriv.append(fun(A,E,G,sol1,a,c1,c2,c3,j))
    second_deriv.append(fun_second_derivative_K(A,E,G,sol1,a,c1,c2,c3,j))

plt.plot(intervall, first_deriv, 'm-')
plt.plot(intervall, second_deriv, 'c-')


plt.grid(which='major', color='gray', alpha=0.6, linestyle='dashdot', lw=1.5)
plt.minorticks_on()
plt.grid(which='minor', color='beige', alpha=0.8, ls='-', lw=1)
plt.title('Minimieren der Kostenfunktion')
plt.xlabel('$l_0$')
plt.ylabel('$dK/dl_0$ bzw. $d^2K/d^2l_0$')
plt.legend([r'$erste$ $Ableitung$', r'$zweite$ $Ableitung$'], loc='best');
```



Die Abbildung “Minimieren der Kostenfunktion” zeigt, dass die Nullstelle der ersten Ableitung wie zuvor beschrieben etwas keiner als der horizontale Abstand zwischen Lampe und Steher ist. Die Kosten sind hier minimal. Die zweite Ableitung nimmt an dieser Stelle einen Extremwert an (Maximum).

Nachfolgend wird mit verschiedenen Methoden die kosteneffizienteste Seillänge l_0 rechnerisch ermittelt. Dazu werden drei verschiedene Varianten verwendet und miteinander verglichen. Die dazu verwendeten Funktionen stammen aus dem SciPy Modul *optimize*. Während *optimize.root* die Nullstellen einer Funktion ermittelt, wird *optimize.minimize* verwendet, um das Minimum einer Funktion zu finden. Mithilfe von *optimize.root* kann so von der Ableitung der Kostenfunktion (k_diff) die Nullstelle und damit die kostenffizienteste Seillänge gefunden werden. Diese kann ebenfalls durch das Ermitteln des Minimum der Kostenfunktion (k_fun) unter Zuhilfenahme von *optimize.minimize* gefunden werden. Dazu sind sieben Iterationsschritte (nit) und 26 Auswertungen der Zielfunktion (=Kostenfunktion) (nfev) notwendig, um eine Genauigkeit (tol) von 10^{-11} zu erreichen. Wird jedoch der Funktion die Ableitung in Form der Jacobi-Matrix (jac=k_diff) bereits als Argument übergeben, wird das Ergebnis mit der erforderlichen Toleranz bereits nach 13 Auswertungen der Zielfunktion erreicht. Alle drei Varianten führen so mit unterschiedlichen Herangehensweisen und Rechenaufwand zum gefragten Ergebnis.

[12]:
```
def k_fun(j,c1, c2, c3, a, A, E,G): #Kostenfunktion
    sol1 = optimize.root(implizite_funktion, [max(E*A*(a/j-1)*1.01, 0.
 ↪001)],args=(G,a,A,E,j), jac=False).x
    return kosten_funktion(c1, c2, c3, a, A, E, j, sol1)
def k_diff(j,c1, c2, c3, a, A, E,G): #erste Ableitung der Kostenfunktion
    sol1 = optimize.root(implizite_funktion, [max(E*A*(a/j-1)*1.001, 0.
 ↪001)],args=(G,a,A,E,j), jac=False).x
    return fun(A,E,G,sol1,a,c1,c2,c3,j)[0]
def k_diff2(j,c1, c2, c3, a, A, E,G): #zweite Ableitung der Kostenfunktion
    sol1 = optimize.root(implizite_funktion, [max(E*A*(a/j-1)*1.001, 0.
 ↪001)],args=(G,a,A,E,j), jac=False).x
    return fun_second_derivative_K(A,E,G,sol1,a,c1,c2,c3,j)[0]
def callbackF(Xi):
    print(Xi)

print('root-Funktion:')
print(optimize.root(k_diff, a, args=(c1, c2, c3, a, A, E,G)).x)
print('minimize-Funktion:')
print(optimize.minimize(k_fun, a, args=(c1, c2, c3, a, A,␣
 ↪E,G),tol=1e-11,callback=callbackF))
print('minimize-Funktion mit Ableitungsinformation:')
print(optimize.minimize(k_fun, a, args=(c1, c2, c3, a, A,␣
 ↪E,G),tol=1e-11,jac=k_diff,callback=callbackF))
```

```
root-Funktion:
[9.83360523]
minimize-Funktion:
[9.96553716]
[9.89106896]
[9.84041987]
[9.83523095]
[9.83366255]
[9.83360556]
[9.83360521]
      fun: 7.5213433558742635e-06
 hess_inv: array([[8305.62316898]])
      jac: array([1.70530257e-12])
  message: 'Optimization terminated successfully.'
     nfev: 26
      nit: 7
     njev: 13
   status: 0
  success: True
        x: array([9.83360521])
minimize-Funktion mit Ableitungsinformation:
[9.96553715]
[9.8910689]
[9.84041984]
[9.83523096]
[9.83366243]
[9.83360572]
[9.83360523]
      fun: 7.521343355874389e-06
 hess_inv: array([[8324.56852086]])
      jac: array([1.77707665e-14])
  message: 'Optimization terminated successfully.'
     nfev: 13
      nit: 7
     njev: 13
   status: 0
  success: True
        x: array([9.83360523])
```

Eine weitere Methode die Kostenfunktion zu minimieren ist das Anwenden der Newton-Methode. Diese ist ein iteratives Verfahren zur Näherungslösung von Gleichungen, um die Nullstellen einer Funktion zu finden. Ausgehend von einem beliebigen Funktionswert wird die Ableitung der Funktion an dieser Stelle als Tangente dargestellt. Die Nullstelle der Ableitung wird dann als Näherung für die Nullstelle der Funktion angesehen und dient als Ausgangspunkt für die nächste Interation.

Die Iterationsvorschrift für die Newton-Methode lautet: $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$

dabei ist:

x_n... aktueller Schätzwert

x_{n+1}... nächster Schätzwert

$f(x_n)$... Funktionswert an der Stelle x_n

$f'(x_n)$... Ableitung der Funktion an der Stelle x_n

Voraussetzung für die Anwendung der Newton-Methode ist, dass die Ableitung der Funktion bekannt ist und nicht null sein darf (sonst Division durch 0). Weiters ist die Konvergenz der Methode stark vom gewählten Startwert und der Form der Funktion abhängig.

In dem Fall der Minimierung der Kostenfunktion ist die Nullstelle der Ableitung der Kostenfunktion gesucht. Deshalb wird diese als Basisfunktion $f(x)$ verwendet. Anstelle des Funktionswertes $f(x_n)$ ist deshalb die erste Ableitung und anstelle von $f'(x_n)$ die zweite Ableitung in die Iterationsvorschrift einzusetzen.

Zur Konvergenz dieser Methode ist zu sagen, dass die Wahl des Startwertes sehr entscheidend ist. Dieser muss sehr nahe an der tatsächlichen Nullstelle der Funktion liegen (z.B. Startwert 9.75 bei tats. Lsg. 9.83). Der Grund dafür liegt in der natürlichen Form der Funktion (siehe Abb. "Minimieren der Kostenfunktion"). Da diese im Allgemeinen relativ flach ist, ist auch die Steigung der Tangente sehr gering. Das führt dazu, dass die Stelle, an der die Tangente die Abszisse schneidet, weit vom dezeitigen x_n entfernt ist. Der neue Schätzwert x_{n+1} entfernt sich dadurch immer weiter von der eigentlich gesuchten Nullstelle. Das Verfahren ist damit nicht robust.

[13]:
```
def k_diff(j): #j entspricht l_0, Ableitung der Kostenfunktion
    sol1 = optimize.root(implizite_funktion, [max(E*A*(a/j-1)*1.001, 0.
 ↪001)],args=(G,a,A,E,j), jac=False).x
    return fun(A,E,G,sol1,a,c1,c2,c3,j)[0]
def k_diff2(j): #zweite Ableitung der Kostenfunktion
    sol1 = optimize.root(implizite_funktion, [max(E*A*(a/j-1)*1.001, 0.
 ↪001)],args=(G,a,A,E,j), jac=False).x
    return fun_second_derivative_K(A,E,G,sol1,a,c1,c2,c3,j)[0]

def newtonMethod(x0,iterationNumber, f,df):
    x=x0

    for i in range(iterationNumber):
        print(f'Iteration {i}: l_0 = {x}')
        print(f'Ableitung der Kostenfunktion: {f(x)}')
        print(f'zweite Ableitung der Kostenfunktion: {df(x)}')
        x=x-f(x)/df(x) #Iterationsvorschrift
```

```
    residual=np.abs(f(x))
    return x, residual

solution, residual = newtonMethod(9.75,6,k_diff,k_diff2)
print('============================================================')
print(f'Ergebnis:')
print(f'l_0 = {solution}')
print(f'K(l_0) = {residual}')
```

```
Iteration 0: l_0 = 9.75
Ableitung der Kostenfunktion: -6.62816201931516e-06
zweite Ableitung der Kostenfunktion: 4.963740497371312e-05
Iteration 1: l_0 = 9.883531598253883
Ableitung der Kostenfunktion: 7.689033170046583e-06
zweite Ableitung der Kostenfunktion: 0.00018711352351807582
Iteration 2: l_0 = 9.842438720382319
Ableitung der Kostenfunktion: 1.111434205693368e-06
zweite Ableitung der Kostenfunktion: 0.00013166686904064265
Iteration 3: l_0 = 9.833997460908378
Ableitung der Kostenfunktion: 4.72010347471308e-08
zweite Ableitung der Kostenfunktion: 0.00012058853072961475
Iteration 4: l_0 = 9.833606038652368
Ableitung der Kostenfunktion: 9.74724077120522e-11
zweite Ableitung der Kostenfunktion: 0.00012009073913968958
Iteration 5: l_0 = 9.833605226996045
Ableitung der Kostenfunktion: 4.1835465484658246e-16
zweite Ableitung der Kostenfunktion: 0.00012008970847937495
============================================================
Ergebnis:
l_0 = 9.833605226992562
K(l_0) = 1.700842158086635e-18
```

3. FAZIT

Bei der einfachsten Modellierung (starres Seil) kann die Lösung relativ einfach ermittelt werden. Mithilfe eines Freischnittes, Kräftebilanzen und einem geometrischen Zusammenhang kann die gesuchte Seilkraft ermittelt werden.
Wird die Komplexität der Modellierung erhöht, indem eine Dehnung des Seils zugelassen wird, erhöht sich der Rechenaufwand drastisch, da die Seilkraft nun nur noch in impliziter Form ausgedrückt werden kann. Infolgedessen ergeben sich Abhängigkeiten zwischen den Variablen, wodurch beim Differenzieren unter anderem die Kettenregel angewandt werden muss. Dieser erhöhte Rechenaufwand ist für diese Problemstellung jedoch nicht unbedingt sichtbar, da zur Berechnung viele Standardfunktionen von Bibliotheken von Jupyter Notebook verwendet werden können. Dass diese ausreichend genaue Ergebnisse liefern, liegt daran, dass das Modell nur eine Variable, d.h. eine Seilkraft beinhaltet.
Weiters ist anzumerken, dass das Programm sehr feinfühlig auf Änderungen der Eingabewerte reagiert. Sind beispielsweise die Startwerte bei der numerischen Seilkraftberechnung nicht bedacht gewählt, stößt Jupyter Notebook schnell an die Laufzeitgrenzen.
Zu den Ergebnissen der Kostenoptimierung muss gesagt werden, dass die Relevanz dieser für die Realität eher gering ist, da eine signifikante Kostenreduktion erst bei sehr großem Eigengewicht der Lampe auftritt. Allerdings könnte dieses Modell eine Annäherung für komplexere Problemstellungen, Kräfte in Tragseilen betreffend, wie Hänge- oder Schrägseilbrücken sein. Dort wäre es sowohl notwendig mehrere verschiedene Seilkräfte vektoriell als auch das Eigengewicht des Seils (Kettenlinie) zu berücksichtigen. Ist diese Annäherung tatsächlich ausreichend gut, bringt diese Modellierung wirtschaftliches und nachhaltiges Einsparungs- und Optimierungspotential (Materialgüte, Seillänge) mit sich.

LITERATURVERZEICHNIS

[1] Herbert Amann and Joachim Escher. Analysis 2. 2. Auflage, 2008.

[2] Dietmar Gross, Werner Hauger, Jörg Schröder, and Wolfgang A. Wall. Technische Mechanik 1. 14. Auflage, 2019.

[3] Dietmar Gross, Werner Hauger, Jörg Schröder, and Wolfgang A. Wall. Technische Mechanik 2. 14. Auflage, 2021.

[4] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[5] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[6] Klaus Jänich. Mathematik 2. 2. Auflage:81–99, 2002.

[7] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017.

[8] Martin Schanz. Baumechanik 1 + 2. Statik und Festigkeitslehre. 2020.

[9] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.