Effiziente Netzwerke

Klaus Rheinberger, FH Vorarlberg

15. Februar 2024

Inhaltsverzeichnis

1. Vorwort

1.1. Leitfaden

Die Lehrveranstaltung ist in viert Teile aufgeteilt:

- Im Teil Programmieren wiederholen wir die Grundlagen der Programmiersprache Python und lernen die algebraische Modellierungssprache Pyomo kennen.
- Im Teil Theorie werden die Konzepte und mathematischen Beschreibungen von Netzwerken, Flexibilitäten und Zielgrößen dargestellt und daraus Optimierungsmodelle erstellt.
- Im Teil Beispiele werden die ersten beiden Teile anhand von Beispielen erarbeitet und angewandt.
- Im Teil Aufgaben werden den Studierenden Aufgaben gestellt, die sie selbstständig lösen und präsentieren sollen.

1.2. Literatur

1.2.1. Empfohlene Literatur

- Linge, Svein; Langtangen, Hans Petter (2019): Programming for Computations - Python: A Gentle Introduction to Numerical Simulations with Python 3.6. 2nd edition, Springer. [LL19]
- Hands-On Optimization with Python insbesondere Pyomo style guide
- Gurobi modeling-examples
- Hillier, Frederick; Lieberman, Gerald (2020): Introduction to Operations Research. 11th edition. New York, NY: McGraw-Hill Education Ltd. [HL20]
- Jensen, Paul A. (2008): Operations Research Models and Methods. John Wiley & Sons. [JB08]
- Williams, H. Paul (2013): Model Building in Mathematical Programming. Auflage, Wiley. [Wil13]
- Sioshansi, Ramteen; Conejo, Antonio J. (2017): Optimization in Engineering: Models and Algorithms. Springer. [SC17]
- Schellong, Wolfgang (2016): Analyse und Optimierung von Energieverbundsystemen. Springer. [Sch16]
- Hamacher, Horst W. (2006): Lineare Optimierung und Netzwerkoptimierung: Zweisprachige Ausgabe Deutsch Englisch. 2. Auflage: Vieweg+Teubner. [Ham06]
- Sierksma, Gerard; Zwols, Yori (2015): Linear and Integer Optimization: Theory and Practice. 3rd edition. CRC Press. [SZ15]
- Sierksma, Gerard; Ghosh, Diptesh (2012): Networks in Action: Text and Computer Exercises in Network Optimization. Springer. [SG12]
- Bertsekas, Dimitri P. (1998): Network Optimization: Continuous And Discrete Models. Athena Scientific. [Ber98]
- Nickel, Stefan u.a. (2022): Operations Research. 3. Aufl. Berlin, Heidelberg: Springer. [Nic+22]
- Papageorgiou, Markos; Leibold, Marion; Buss, Martin (2015): Optimierung: Statische, dynamische, stochastische Verfahren für die Anwendung. 4. Auflage, Springer. [PLB15]
- Suhl, Leena; Mellouli, Taïeb (2013): Optimierungssysteme: Modelle, Verfahren, Software, Anwendungen. 3. Aufl. Gabler Verlag. [SM13]

1.2.2. Weiterführende Literatur

- Boyd, Stephen; Vandenberghe, Lieven (2018): Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares. Cambridge University Press. [BV18]
- Easley, David; Kleinberg, Jon (2010): Networks, Crowds, and Markets: Reasoning about a Highly Connected World. Cambridge University Press. [EK10]

- Knieps, Gunter (2007): Netzökonomie. Gabler. [Kni07]
- Turau, Volker; Weyer, Christoph (2015): Algorithmische Graphentheorie. 4th edition, De Gruyter. [TW15]
- Bazaraa, Mokhtar S.; Jarvis, John J.; Sherali, Hanif D. (2010): Linear Programming and Network Flows. Wiley. [BJS10]
- Barbato, Antimo; Capone, Antonio (2014): „Optimization Models and Methods for Demand-Side Management of Residential Users: A Survey.“ In: Energies, 7 (2014), 9, S. 5787–5824. Online im Internet: DOI: 10.3390/en7095787. [BC14]

Teil I.

Programmieren

2. Python

2.1. Allgemeines

Python ist eine freie, offene, plattformunabhängige, üblicherweise interpretierte, höhere Programmiersprache. Python Code ist gut lesbar und einfach zu lernen. Es gibt sehr viele Pakete und Einsatzgebiete für Python, siehe PyPI - the Python Package Index. Für den Bereich des wissenschaftlichen Rechnens sind insbesondere die SciPy-Pakete nützlich. Python erfreut sich einer großen und wachsenden Beliebtheit und besitzt daher eine umfangreiche und breitgefächert Community. So ziemlich jedes Problem mit Lösung/en findet man z. B. unter stackoverflow. Der Name bezieht sich übrigens auf die englische Komikergruppe Monty Python.

Installation: Python und Python Pakete können auf sehr unterschiedliche Art und Weise installiert werden. Wir verwenden die von vielen empfohlene Variante der Python Distribution Anaconda:

- Um die umfangreiche Anaconda Distribution zu installieren, folgen Sie der Dokumentation entsprechend Ihrem Betriebssystem.
- Sparsamer ist die minimale Version Miniconda.

Zu den **Alternativen** im Bereich wissenschaftliches Rechnen zählen

- Julia: frei, offen und plattformunabhängig
- R: frei, offen und plattformunabhängig
- Matlab: kommerziell
- Mathematica: kommerziell

Es gibt, grob gesagt, zwei Typen von **Entwicklungsumgebungen**:

- Ein Text-Editor kombiniert mit der Kommandozeile (englisch: command prompt, console, terminal). Entwicklungsumgebungen wie Spyder, Visual Studio Code oder PyCharm kombinieren beides und mehr in einer Applikation.
- Wir verwenden die web-basierte, interaktive, literate programming Entwicklungsumgebung JupyterLab, die neben Python noch viele andere Programmiersprachen mittels Jupyter-Notebooks unterstützt. Im Wikipedia-Eintrag zum Projekt Jupyter steht (Stand 2020-06-18) unter anderem: "Das Jupyter Notebook hat sich als Benutzeroberfläche für Cloud Computing verbreitet. Große Cloud-Anbieter haben angepasste Tools für Cloud-Anwender entwickelt. Beispiele dafür sind Amazon SageMaker, Googles Colaboratory und Microsofts Azure Notebook."

2.2. JupyterLab

Starten Sie JupyterLab via dem Anaconda Navigator oder einfacher über eine Kommandozeile mit dem Befehl `jupyter lab`. Das meiste der browserbasierten Oberfläche ist selbsterklärend. Wenn Sie ein (neues) **Jupyter Notebook** (Dateiendung .ipynb) starten, wird ein **Kernel** gestartet. Der Kernel führt Ihre Python Befehle aus und beinhaltet alle verwendeten Objekte (Variablen, Funktionen, Pakete etc.).

Achtung: Verwenden Sie generell für Ordner- und und Dateinamen keine Umlaute, keine Sonderzeichen und keine Leerzeichen!

Ein Jupyter-Notebook besteht aus einer Liste von **Zellen** (engl. cells). Zellen können im `Command Mode` oder im `Edit Mode` bearbeitet werden. Die wichtigsten zwei Zelltypen sind `Code` und `Markdown`.

Code Zellen:

- Wir schreiben unseren Python Code in die Code-Zellen eines Jupyter-Notebooks. Zum **Ausführen des Codes einer Code-Zelle** drücken Sie `Strg+Return` oder `Shift+Return`. Bei der ersten Variante bleibt der Fokus auf der Code-Zelle, bei der zweiten springt man in die nächste Zelle.
- Der Rückgabewert des letzten Befehls einer Code Input-Zelle wird in einer folgenden Code Output-Zelle ausgegeben. Sie können die Ausgabe mit einem Strichpunkt am Ende des letzten Befehls unterdrücken.
- Kommentare in Code-Zellen beginnen mit dem Rautezeichen `#`.
- Funktionen haben runde Klammern. Hilfe z. B. zur Funktion `print` erhalten Sie durch `help(print)` oder `print?` oder SHIFT-TAB drücken, wenn der Cursor nach der ersten runden Klammer von `print()` steht.
- Eine Liste der von Ihnen definierten Variablen erhalten Sie mit dem magic command `%whos`. Löschen einzelner Variablen, hier z. B. der Variable `x`, erfolgt mit dem Befehl `del(x)`. Löschen aller selber definierten Variablen erfolgt mit dem Befehl `%reset -s`.
- *Tipp:* Verwenden Sie die Tabulator-Vervollständigung beim Coden!
- *Tipp:* Mit der Tastenkombination `Strg+I` öffnet sich der sehr hilfreiche **Contextual Help** Tab.
- *Tipp:* Öffnen Sie eine Console für Ihr Notebook, um außerhalb des Notebooks interaktiv Befehle im Namespace des Notebook-Kernels auszuführen.

Markdown Zellen: Markdown ist eine vereinfachte Auszeichnungssprache, die bereits in der Ausgangsform ohne weitere Konvertierung leicht lesbar ist. Sie können in Markdown sehr leicht folgenden strukturierten Text erstellen:

- Überschriften: Rautesymbol(e) vor der Überschrift
- Listen: mit Minus-, Plus- oder Sternzeichen
- Links: mit Syntax `[Name](URL)`
- Bilder: mit Syntax `![Name](Pfad-zu-Bilddatei)`
- Mathematische Formeln wie z. B. $K = \frac{mv^2}{2}$ via dem LaTeX-Code `$K = \frac{mv^2}{2}$`
- *Tipp:* Cheatsheet

Keyboard Shortcuts

Die Keyboard Shortcuts (Tastenkürzel) sind in den Menüs neben den Auswahlen angegeben. Hier eine persönliche Auswahl:

Shortcut	Effekt
Ctrl+s	save notebook
Ctrl+Shift+q	close and shutdown notebook
Ctrl+f	Find
Enter	enter edit mode
Escape	enter command mode
Ctrl+Enter	run cell
Shift+Enter	run cell and got to cell below
Alt+Enter	run cell and insert a new cell below
Ctrl+b	toggle left sidebar
Shift+m	merge selected cells
in edit mode: Ctrl+Shift+-	split cell
in command mode: m	set cell type to markdown
in command mode: y	set cell type to code
in command mode: d d	delete cell
in command mode: z	undo deleting of cell
in command mode: a	insert cell above
in command mode: b	insert cell below
in command mode: x/c/v	cut/copy/paste cells
in command mode: UP/DOWN arrow	selected previous/next cell
in command mode: i+i	interrupt kernel
in command mode: 0+0	restart kernel
in command mode: Shift+l	toggle all line numbers

Bevor Sie **JupyterLab beenden**, vergessen Sie nicht, Ihre Jupyter-Notebooks zu speichern, zu schließen sowie die zugehörigen Kernel herunterzufahren. Zum Beenden von JupyterLab wählen Sie im Menü `File/Shut Down`.

Exportieren: Sie könne Jupyter-Notebooks über den Menüeintrag `File/Export Notebook As` oder über Systembefehle in andere Dateiformate exportieren. Systembefehle können Sie in einem System-Terminal, im `Anaconda Command Prompt` ausführen, oder in einer Codezelle, wenn Sie ein Rufezeichen vor den Systembefehl setzen. Hier eine Auswahl:

- HTML: Systembefehl `jupyter nbconvert Mein_Notebook.ipynb`
- PDF: Systembefehl `jupyter nbconvert --to pdf Mein_Notebook.ipynb`
- LaTeX: Systembefehl `jupyter nbconvert --to latex Mein_Notebook.ipynb`
- Python-Script: Systembefehl `jupyter nbconvert --to script Mein_Notebook.ipynb`

Unter nbconvert finden Sie die Syntax für noch weitere Outputformate und Infos zu evtl. Zusatzsoftware, die für die Konvertierung notwendig ist.

Hinweis: Um Ihren Code bei Bedarf außerhalb der JupyterLab-Umgebung auszuführen, können Sie

- Ihr Jupyter-Notebook `mynotebook.ipynb` in ein Python Script mit Endung `.py` exportieren und anschließend von einem System-Terminal aus mit `python mynotebook.py` starten oder
- in einem System-Terminal `jupyter nbconvert --to notebook --execute mynotebook.ipynb` verwenden.

Navigationsbefehle im Dateisystem: Für die Arbeit in einem System-Terminal, dem `Anaconda Command Prompt` oder in einer Codezelle sind oft folgende Navigationsbefehle nützlich:

- `pwd`: print working/current directory
- `ls` oder `dir`: list files in current directory
- `cd DIR`: change into absolute or relative directory DIR
- `cd ..`: change to parent directory

Extensions: Folgende der vielen JupyterLab Extensions könnten Sie interessieren:

- Variable Inspector
- Interactive Widgets

Tipps: Schauen Sie mal in die Gallery of interesting Jupyter Notebooks und in das Jupyter-Tutorial rein.

Einführung in Python: Sollten Sie noch keine Erfahrung mit Python haben, dann können Sie zum Beispiel das folgende Tutorial als Einstieg verwenden.

2.3. Tutorial

Die folgende Einführung lehnt sich stark an die Kapitel 1 bis 4 des empfehlenswerten Buchs **"Programming for Computations - Python. A Gentle Introduction to Numerical Simulations with Python 3.6."** von Svein Linge und Hans Petter Langtangen, 2. Auflage, 2020, an. Hier der Link zum Verlag.

Weitere nützliche Quellen:

- die Lehrveranstaltung Programmiertechniken
- Official Python 3 documentation
- Python Tutorial bei www.w3schools.com

2.3.1. Erste Schritte

Hier ein erstes **Beispiel:** Der Code unten berechnet die Höhe $y(t) = v_0 t - \frac{1}{2} g t^2$ eines Balls zum Zeitpunkt $t \geq 0$, der zum Zeitpunkt $t = 0$ aus der Höhe $y = 0$ mit der Geschwindigkeit v_0 vertikal in die Höhe geworfen wird.

Beachten Sie:

- **Code-Kommentare** beginnen mit dem Rautezeichen `#`.
- **Funktionen** haben immer runde Klammern.
- Das Potenzieren von Zahlen wird mit `**` implementiert.

```
# program for computing the height of a ball in vertical motion

v0 = 5     # initial velocity in m/s
g  = 9.81  # acceleration of gravity in m/s^2
t  = 0.6   # time in s

y = v0*t - 0.5*g*t**2  # vertical position in m

print(f"At time t = {t} s the ball is at height y(t) = {y} m.") # formated printing
```

```
At time t = 0.6 s the ball is at height y(t) = 1.2342 m.
```

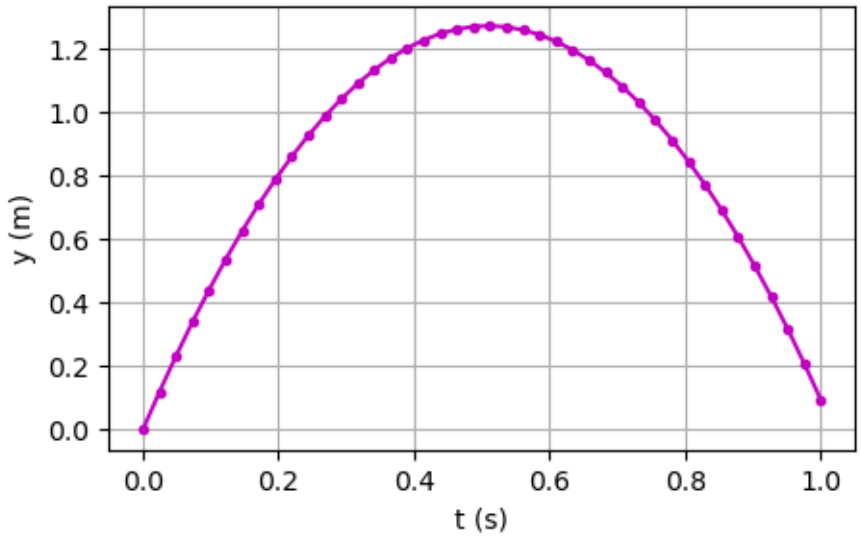
Um Funktionen außerhalb der Python Standard Library, in der sich z. B. die Funktion `print` befindet, zu verwenden, importieren wir die gewünschten Python Pakete in den Namespace unseres Jupyter-Notebooks. Die Pakete NumPy und Matplotlib verwenden wir unter anderen in der Lehrveranstaltung.

```
# imports into the namespace:
import numpy as np
import matplotlib.pyplot as plt
```

Die Funktionen und Module des Pakets `numpy` können nach dem obigen Import mit `np.*` angesprochen werden. Analoges gilt für die Funktionen von `matplotlib.pyplot`.

Mit NumPy können wir die Berechnung der vertikalen Position des Balls vektorisieren, und mit Matplotlib können wir diese grafisch darstellen.

```
t = np.linspace(0, 1, 42)  # creates an array of 42 numbers from 0 to 1

y = v0*t - 0.5*g*t**2

plt.figure(figsize=(5, 3)) # sets the size of the figure
plt.plot(t, y, '.-m')      # plots all y coordinates vs. all t coordinates
plt.xlabel("t (s)")        # places the text t (s) on x-axis
plt.ylabel("y (m)")        # places the text y (m) on y-axis
plt.grid(True)             # adds a grid to the figure
```



Jetzt aber der Reihe nach!

2.3.2. Import von Paketen

Anstatt das gesamte Paket `some_package` mit `import some_package as some_prefix` mit einem selbstgewählten Präfix `some_prefix` zu importieren, können auch einzelne Funktionen importiert werden.

```
from numpy import linspace, pi

# after the import linspace and pi are part of the namespace, e. g.:
linspace(0, 5, 11)
```

```
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ])
```

```
from numpy import cos as cosinus

cosinus(pi)
```

```
-1.0
```

Für den Import von Paketen müssen nicht unbedingt Präfixe wie z. B. `np` vergeben werden.

```
import numpy

numpy.linspace(0, 5, 11)
```

```
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ])
```

Achtung: Es ist **nicht empfohlen**, alle Funktionen und Module eines Pakets mit einem Import vom Typ `from math import *` in den Namespace zu importieren! Warum?

2.3.3. Datentypen - eine Auswahl

Von den **integrierten Datentypen** verwenden wir die unten angeführten. Dabei besprechen wir deren Funktionalitäten nicht vollständig!

String: immutable

```
x = "Hello World"  # or 'Hello World'
print(type(x))
print("Applied " + "Mathematics") # adding is concatenating
```

```
<class 'str'>
Applied Mathematics
```

Integer: immutable

```
x = 42
print(type(x))
```

```
<class 'int'>
```

Float: immutable

```
# float is short for floating point number

x = 42.123456789
print(type(x))
```

```
<class 'float'>
```

```
print(type(12/3)) # result is of type float
```

```
<class 'float'>
```

```
print(4/4*3)
print(4/(4*3))  # do not forget the brackets if you wanted this.
```

```
3.0
0.3333333333333333
```

```
# computations:

print(4.5 + 1.5)  # addition
print(4.5 - 5)    # subtraction
print(2*7.1)      # multiplication
print(2**5)       # exponentiation
print(7.5/3)      # division
print(7.5//3)     # floor division, integer division
print(7.5%3)      # remainder, modulo
```

```
6.0
-0.5
14.2
32
2.5
2.0
1.5
```

List: mutable

```
# A list is a collection which is ordered and changeable (= mutable).

x = ['Eric', 42, 0.3]
print(type(x))
print(len(x))  # length, i. e. number of items, of the list
```

```
<class 'list'>
3
```

```
# accessing list items:
print(x[0])     # Python starts counting with 0!
print(x[1:3])   # item with index 1 included, item with index 3 excluded!
print(x[:2])    # items up to index 2 excluded
print(x[1:])    # items starting from index 1 included
print(x[-1])    # last item
print(x[-2])    # second last item
print(x[-2:])   # from the end up to second last item included
print(x[-2:-1]) #  should be clear now
```

```
Eric
[42, 0.3]
['Eric', 42]
[42, 0.3]
0.3
42
[42, 0.3]
[42]
```

```
# With the dot notation methods of the object can be accessed.
x.append(137)
x
```

```
['Eric', 42, 0.3, 137]
```

```
# lists can also be added and multiplied:
x = ['Eric', 42, 0.3]
print(x + ['John', 137])
print(2*x)
```

```
['Eric', 42, 0.3, 'John', 137]
['Eric', 42, 0.3, 'Eric', 42, 0.3]
```

Tuple: immutable

```
# A tuple is a collection which is ordered and unchangeable (= immutable).

x = ('Eric', 42, 1/3)
print(type(x))
print(x[1])
# But x has for example no append method.
# you can make it a list with
list(x)
```

```
<class 'tuple'>
42

['Eric', 42, 0.3333333333333333]
```

Dictionary: mutable

A dictionary allows you to map arbitrary key values to pieces of data. Any immutable Python object can be used as a key: an integer, a floating-point number, a string, or even a tuple.

```
x = {"name" : "Eric",
     "age"  : 42}
print(type(x))
```

```
<class 'dict'>
```

```
# accessing dictionary items:
x['name']
```

```
'Eric'
```

Boolean: immutable

```
# boolean:

print(type(True))
print(type(False))
print(4 == 12/3)
print(4 >= 3)
print(4  < 3)
print(4 != 3)
print(42 in ['Eric', 42, 1/3])
```

```
<class 'bool'>
<class 'bool'>
True
True
False
True
True
```

Achtung! Zu den unveränderlichen (immutable) Objekten zählen integers, floats, strings und andere, während Numpy Arrays (siehe unten) und Listen Beispiele für veränderliche (mutable) Objekte sind. Das hat folgende wichtige Auswirkungen:

```
x = 1
y = x
y = 4
print(x)
```

```
1
```

```
# however:
x = [1,2,3]
y = x   # Python creates a reference to the mutable object x
y[0] = 4
print(x)
```

```
[4, 2, 3]
```

```
# workaround with copying values:
x = [1,2,3]
y = x.copy()   # Python creates a new object y
y[0] = 4
print(x)
```

```
[1, 2, 3]
```

NumPy Arrays, short arrays: mutable

- 1-dimensionale Arrays für Vektorrechnung
- 2-dimensionale Arrays für Matrizenrechnung

```
x = np.array([1, 2.1, -5])
print(x)
print(type(x))
print(len(x))     # length, i. e. number of items
print(np.ndim(x)) # number of dimensions
```

```
[ 1.   2.1 -5. ]
<class 'numpy.ndarray'>
3
1
```

```
# accssing list items works the same as with lists:
print(x[0])     # Python starts counting with 0!
print(x[1:3])   # item with index 1 included, item with index 3 excluded!
print(x[:2])    # items up to index 2 excluded
print(x[1:])    # items starting from index 1 included
print(x[-1])    # last item
print(x[-2])    # second last item
print(x[-2:])   # from the end up to second last item included
print(x[-2:-1]) # should be clear now
```

```
1.0
[ 2.1 -5. ]
[1.  2.1]
[ 2.1 -5. ]
-5.0
```

```
2.1
[ 2.1 -5. ]
[2.1]
```

```
# With the dot notation many methods be accessed.
# Use the TAB-key after the dot to see them! Here' an example:
x.mean()
```

```
-0.6333333333333333
```

```
# Arrays can also be added and multiplied,
# but now in the sense of vector algebra!
```

```
y = np.array([3, -1.9, 42])

x + y  # vector addition: elementwise!
```

```
array([ 4. ,  0.2, 37. ])
```

```
print(x)
print(x + 100)   # adding a scalar to all elements
```

```
[ 1.   2.1 -5. ]
[101.  102.1  95. ]
```

```
3*x   # multiplying a scalar to all elements
```

```
array([  3. ,   6.3, -15. ])
```

Das innere Produkt zweier Vektoren wird im Englischen oft “dot product” genannt.

```
np.dot(x, y)
```

```
-210.99
```

```
# alternatively with the @ operator:
x@y
```

```
-210.99
```

```
np.cross(x, y) # cross product: only für vectors with 3 elements
```

```
array([ 78.7, -57. ,  -8.2])
```

```
x*x   # caution: elementwise!
```

```
array([ 1.  ,  4.41, 25.  ])
```

```
x/x   # elementwise!
```

```
array([1., 1., 1.])
```

```
x**3  # elementwise!
```

```
array([   1.   ,    9.261, -125.   ])
```

```
# constructors:

x = np.arange(start = 0, stop = 5, step=2)
print(x)

x = np.linspace(start = 0, stop = 5, num=11)
print(x)

x = np.zeros(4)
print(x)

x = np.ones(7)
print(x)
```

```
[0 2 4]
[0.  0.5 1.  1.5 2.  2.5 3.  3.5 4.  4.5 5. ]
[0. 0. 0. 0.]
[1. 1. 1. 1. 1. 1. 1.]
```

```
# The thing about copying ... :

x = np.array([1,2,3])
y = x
y[0] = 4
print(x)

x = np.array([1,2,3])
y = x.copy()
y[0] = 4
print(x)
```

```
[4 2 3]
[1 2 3]
```

```
# 2-dim arrays, aka matrices

M = np.array([[1,2,3],
              [4,5,6]])

print(M)
print(np.ndim(M)) # number of dimensions, number of square brackets
print(M.shape)    # number of rows and number of columns
```

```
[[1 2 3]
 [4 5 6]]
2
(2, 3)
```

```
# accessing items and slices:
print(M[1,2])
print(M[1,:])
print(M[:,0])
print(M[:,[0]])
```

```
6
[4 5 6]
[1 4]
[[1]
 [4]]
```

Overview of mutable and immutable data types:

- mutable data types: list, dictionary, array, ...
- immutable data types: int, float, bool, string, tuple, ...

2.3.4. Formatiertes Drucken

Wir verwenden **"f-strings"** für das formatierte Drucken. Zwei weitere Methoden werden z. B. im Buch von Linge und Langtangen beschrieben. f-strings beginnen mit einem `f` vor den Anführungszeichen, die den string kennzeichnen. Variablenwerte werden geschwungenen Klammern eingebunden. Zeichenketten (=strings), die über mehrere Zeilen laufen, können mit dreifachen Anführungszeichen geschrieben werden. `\n` bewirkt eine neue Zeile im Ausdruck. Mehr zu f-strings finden Sie z. B. im Python f-string tutorial.

```
my_name  = "Klaus"
my_age   = 47
my_float = 123.123456789

print(f"Hallo! My name is {my_name}.")
print(f"Two of my objects:\n  {my_name = }\n  {my_age = }.") # self-documenting expression
print(f"I am {my_age} years old. These are about {my_age*365} days.")
print(f"""My number in different formats:
  {my_float:.3f} or
  {my_float:10.3f} or
  {my_float:.3e} or
  {my_float:10.3e}""")
```

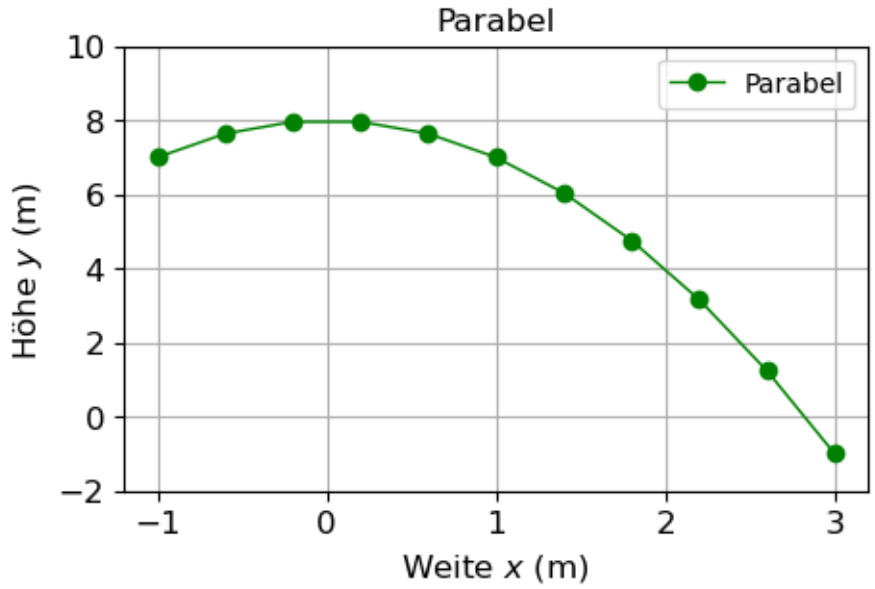
```
Hallo! My name is Klaus.
Two of my objects:
  my_name = 'Klaus'
  my_age = 47.
I am 47 years old. These are about 17155 days.
My number in different formats:
  123.123 or
     123.123 or
  1.231e+02 or
   1.231e+02
```

2.3.5. Grafiken

Wir werden nur wenige Beispiele in diesem Abschnitt aufzeigen. Die große Vielfalt an grafischen Darstellungsmöglichkeiten mit dem Python-Paket Matplotlib finden Sie z. B. unter Matplotlib Gallery. Es gibt neben Matplotlib noch viele andere Grafik-Pakete. Interaktive Grafiken können Sie z. B. mit ipywidgets erstellen.

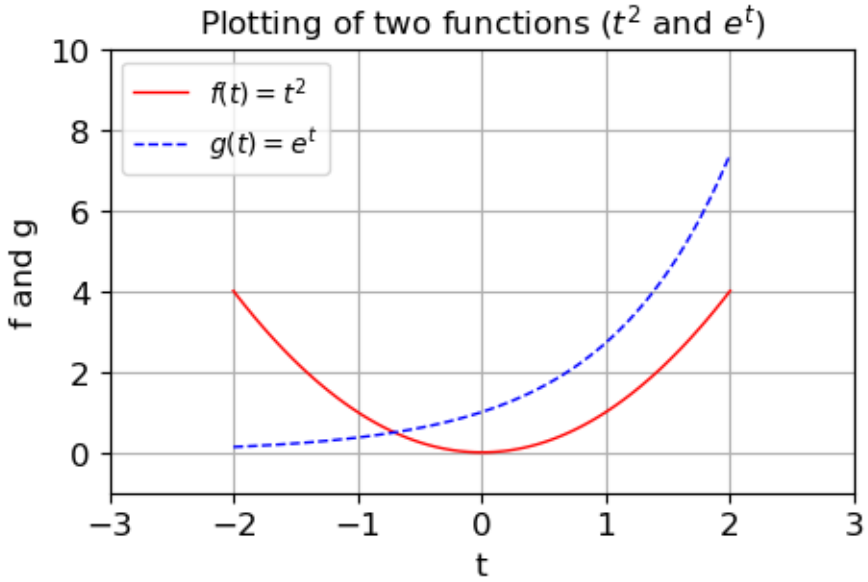
```
# set default values for all plotting:
plt.rcParams['axes.titlesize']  = 12
plt.rcParams['axes.labelsize']  = 12
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
plt.rcParams['legend.fontsize'] = 10
plt.rcParams['lines.linewidth'] = 1
```

```
x = np.linspace(-1, 3, num = 11)        # 11 points between -1 and 3
y = -x**2 + 8

plt.figure(figsize=(5, 3))              # sets the size of the figure
plt.plot(x, y, 'o-g', label='Parabel')  # plots all y coordinates vs. all x coordinates
plt.xlabel('Weite $x$ (m)')             # sets the x-label
plt.ylabel('Höhe $y$ (m)')              # sets the y-label
plt.ylim(-2, 10)                        # sets the limits of the y-axis
plt.title('Parabel')                    # title of the figure
plt.legend(numpoints=1, loc='best')     # legend uses the label set in plt.plot
plt.grid(True)                          # adds a grid to the figure

plt.savefig('abbildungen/Parabel.pdf')  # saves the figure as pdf
```




```
t = np.linspace(-2, 2, 100)
f_values = t**2
g_values = np.exp(t)

plt.figure(figsize=(5, 3))
plt.plot(t, f_values, 'r',
         t, g_values, 'b--')
plt.xlabel('t')
plt.ylabel('f and g')
plt.legend(['$f(t) = t^2$',
            '$g(t) = e^t$'])
plt.title('Plotting of two functions ($t^2$ and $e^t$)')
```

```
plt.grid(True)
plt.axis([-3, 3, -1, 10]);  # sets the limits of the x- and y-axis
```



2.3.6. Kontrollstrukturen

for-Schleife: hat die Struktur

```
for loop_variable in iterable_object:
    <code line 1>
    <code line 2>
    etc.
# first code line after the loop
```

```
# example 1: iteration over a list
for item in ["1", 2, np.pi]:
    print(item)
```

```
1
2
3.141592653589793
```

```
# example 2: iteration over an array

v = np.linspace(0, 3, 7)
for x in v:
    print(x, end=", ")
```

```
0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0,
```

```
# example 3: using enumerate to get the indices too

my_list= ['Sokrates', 'Platon', 'Aristoteles']
for index, value in enumerate(my_list):
    print(f"{index + 1}. {value}")
```

```
1. Sokrates
2. Platon
3. Aristoteles
```

```
# list comprehension:

my_list_1 = [-x for x in np.arange(-3, 4)]
print(my_list_1)

my_list_2 = [-x for x in np.arange(-3, 4) if x < 0]
print(my_list_2)

my_list_3 = [-x if x < 0 else x for x in np.arange(-3, 4)]
print(my_list_3)
```

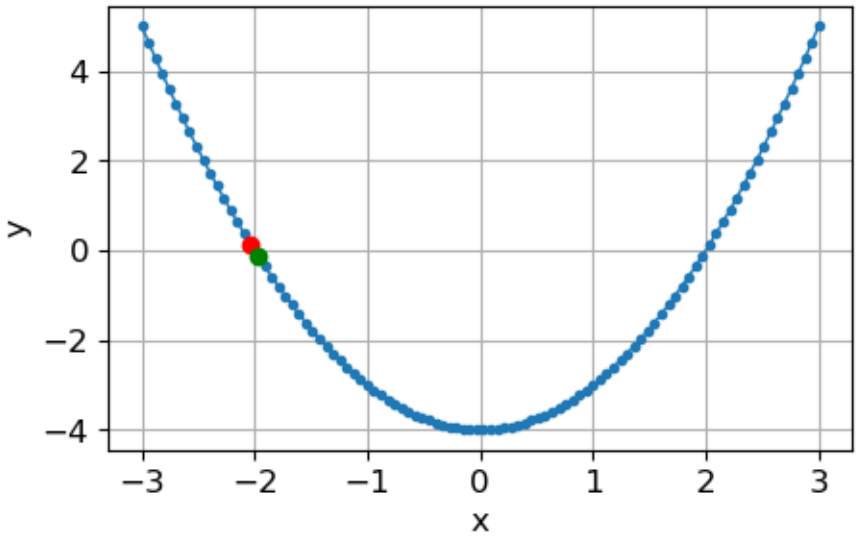
```
[3, 2, 1, 0, -1, -2, -3]
[3, 2, 1]
[3, 2, 1, 0, 1, 2, 3]
```

while-Schleife: hat die Struktur

```
while some_condition:
    <code line 1>
    <code line 2>
    etc.
# first code line after the loop
```

```
# find the first data points nearest to zero:
x = np.linspace(-3, 3, num=100)
y = x**2 - 4

plt.figure(figsize=(5, 3))
plt.plot(x, y, '.-')
plt.xlabel('x')
plt.ylabel('y')

ind = 0  # staring index, proceed to increasing values
sign_change = False
while not sign_change:
    old_sign = np.sign(y[ind])
    ind += 1
    new_sign = np.sign(y[ind])
    if new_sign != old_sign:
        sign_change = True

print(f"y[{ind -1}] = {y[ind - 1]}")
print(f"y[{ind   }] = {y[ind    ]}")
plt.plot(x[ind - 1], y[ind - 1], 'or')
plt.plot(x[ind    ], y[ind    ], 'og')
plt.grid(True)
```

```
y[16] = 0.12213039485766775
y[17] = -0.12029384756657491
```



if-elif-else-Abfragen: hat die Struktur

```
if condition_1:
    <code line 1>
    <code line 2>
    ...
elif condition_2: # optinonal
    <code line 1>
    <code line 2>
    ...
elif condition_3: # optinonal
    <code line 1>
    <code line 2>
    ...
else:             # optinonal
    <code line 1>
    <code line 2>
    ...
# First line after if-elif-else construction
```

```
T = 5 # select a water temperature in degree centigrade

if T <= 20:
    print("Do not swim. Too cold!")
elif T <= 36:
    print("Great, jump in!")
else:
    print("Do not swim. Too hot!")
```

```
Do not swim. Too cold!
```

Tipps: - Mit der `break`-Anweisung gelangen Sie direkt zur ersten Codezeile nach einer Schleife . - Mit der `continue`-Anweisung fahren Sie direkt mit der nächsten Iteration fort.

2.3.7. Funktionen

Eine Funktion hat die Struktur

```
def function_name(pos1, pos2, ..., kw1=default_1, kw2=default_2, ...):   # function header
    """This is a docstring"""         # function body
    <code line>                       # function body
    <code line>                       # function body
    ...                               # function body
    return result_1, result_2, ...    # last line in function body
# First line after function definition
```

Die `return` Zeile, die die Rückgabewerte definiert und die Argumente `pos1, ..., kw1 = ...` sind jeweils optional. Eine Funktion kann daher auch so aussehen:

```
def my_function():
    """function without arguments
    and without return values"""
    print("Ahoi!")
```

```
my_function()
```

```
Ahoi!
```

Der **Doc-String** ist der Hilfetext zur Funktion und kann aufgrund der dreifachen doppelten Anführungszeichen über mehrere Zeilen laufen.

```
help(my_function)
```

```
Help on function my_function in module __main__:

my_function()
    function without arguments
    and without return values
```

Argumente:

- Die Argumente `pos1, pos2, ...` sind sogenannte *positional parameters*.
- Die Argumente `kw1, kw2, ...` sind sogenannte *keyword parameters*.

Positional parameters müssen vor keyword parameters angeführt werden. Die Werte der Parameter, die beim Aufruf der Funktion übergeben werden heißen positional bzw. keword arguments. Die Werte `default_1` und `default_2` sind Default-Werte, die verwendet werden, wenn keine entsprechenden keyword arguments beim Aufruf der Funktion angegeben werden.

Hier ein Beispiel:

```
def my_greet(first_name, second_name, msg="Good morning!"):
    print(f"Hello {first_name} {second_name}! {msg}")
```

```
my_greet('Albert', 'Einstein') #
my_greet('Albert', 'Einstein', msg='Good night!')
my_greet('Albert', 'Einstein', 'Good night!')
my_greet('Good night!', 'Albert', 'Einstein')  # gives no error, but is not what you intended!
my_greet(first_name="Albert", second_name='Einstein', msg="How do you do?")
my_greet("Albert", second_name='Einstein', msg="How do you do?")
```

```
my_greet(msg = "How do you do?", first_name="Albert", second_name='Einstein')
my_greet(second_name='Einstein', first_name="Albert", msg = "How do you do?")
# my_greet(msg="How do you do?", "Albert", "Einstein") # gives "SyntaxError: positional argument follows keyw
```

```
Hello Albert Einstein! Good morning!
Hello Albert Einstein! Good night!
Hello Albert Einstein! Good night!
Hello Good night! Albert! Einstein
Hello Albert Einstein! How do you do?
Hello Albert Einstein! How do you do?
Hello Albert Einstein! How do you do?
Hello Albert Einstein! How do you do?
```

Erkenntnisse:

- Keyword arguments können auch ohne keyword angegeben werden.
- Es ist möglich, die Namen von positional parameters als keywords zu verwenden.
- Die Reihenfolge von keyword arguments kann geändert werden.
- Eine Funktion kann mit positional oder mit keyword arguments oder aus einer Mischung aufgerufen werden. Bei der Mischvariante müssen die positional arguments jedoch zuerst angeführt werden müssen.
- Solange in einem Funktionsaufruf für alle Argumente keywords verwendet werden, kann eine beliebige Reihenfolge der Argumente verwendet werden.

Geltungsbereich von Variablen (engl. Scope of Variables):

```
if "y" in locals():
    del(y)
```

```
k =   2  # global variable defined in the main program
d = 100  # global variable defined in the main program

def my_line_value(x):
    # The global variables k and d are known from "outside".
    k = 10  # k is now also the name of a local variable.
    # The assignment k = 10 changes only the local variable inside the function.
    # The "name-brother" global variable outside is not affected!
    print(f"inside values: k = {k}, d = {d}")
    y = k*x + d
    print(f"inside computation {y} = {k}*{x} + {d}")
    return y

x = 3

print(f"outside values before function call: k = {k}, d = {d}")
print(f"return value of input {x}: {my_line_value(x)}.")
print(f"outside values after function call: k = {k}, d = {d}")
# print(y) # gives "NameError: name 'y' is not defined"
```

```
outside values before function call: k = 2, d = 100
inside values: k = 10, d = 100
inside computation 130 = 10*3 + 100
return value of input 3: 130.
outside values after function call: k = 2, d = 100
```

Achtung: Veränderliche äußere Objekte werden in Funktionen geändert!

```
x = [1, 2]  # mutable object
print(f"outside value: {x = }")

def change(y):
    y[0] = 4
    print(f"inside value:  {y = }")

change(x)
print(f"outside value: {x = }")
```

```
outside value: x = [1, 2]
inside value:  y = [4, 2]
outside value: x = [4, 2]
```

```
# Auch ohne Übergabe:

x = [1, 2]  # mutable object
print(f"outside value: {x = }")

def change():
    x[0] = 4
    print(f"inside value:  {x = }")

change()
print(f"outside value: {x = }")
```

```
outside value: x = [1, 2]
inside value:  x = [4, 2]
outside value: x = [4, 2]
```

Lambda Funktionen: sind eine Möglichkeit, auf kompakte Arte "Einzeiler"-Funktionen zu definieren.

```
g = lambda x: x**2
# is equivalent to:
# def g(x):
#     return x**2

g(3)
```

```
9
```

Allgemeine Form einer Lambda Funktion:

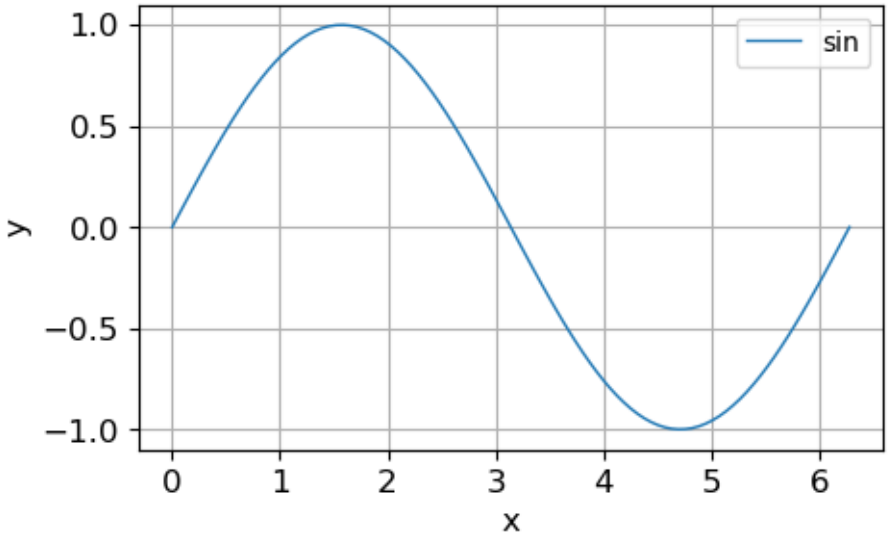
```
function_name = lambda arg1, arg2, ... : <some_expression>
```

Funktionen als Argumente von Funktionen:

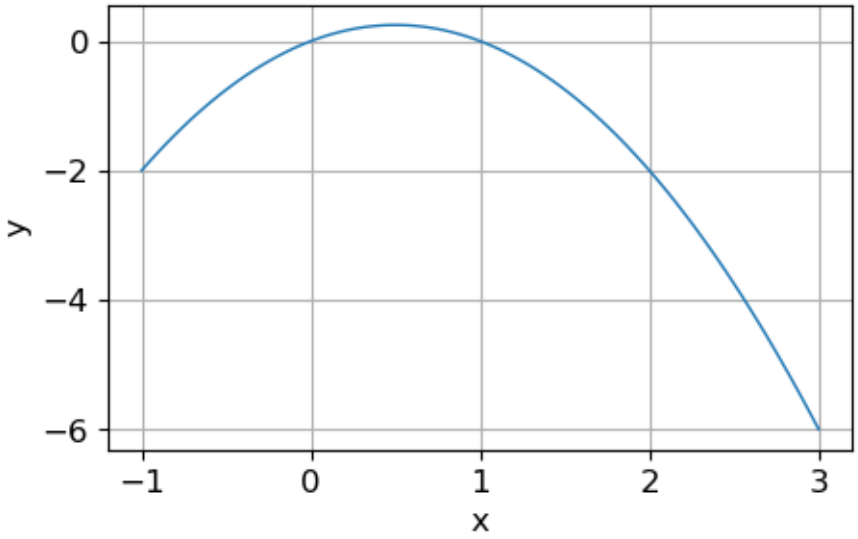
```
def my_plot(fct, start=-1, stop=3, legend=True):
    x = np.linspace(start, stop, num=100)
    y = fct(x)
    plt.figure(figsize=(5, 3))
    plt.plot(x, y, label=fct.__name__)
    plt.xlabel('x')
    plt.ylabel('y')
```

```
    if legend:
        plt.legend()
    plt.grid(True)
```

```
my_plot(np.sin, 0, 2*np.pi)
```




```
my_plot(lambda x: x - x**2, legend=False)
```



2.3.8. Daten IO

Es gibt sehr viele Arten Daten und Dateitypen. Wir beschränken uns hier auf das Laden und Speichern von einfachen CSV-Dateien. Als Beispiel nehmen wir die global temperature anomalies with respect to the 20th century average, genauer die dort bereitgestellte csv-Datei data.csv. Anstatt die Daten Zeile für Zeile

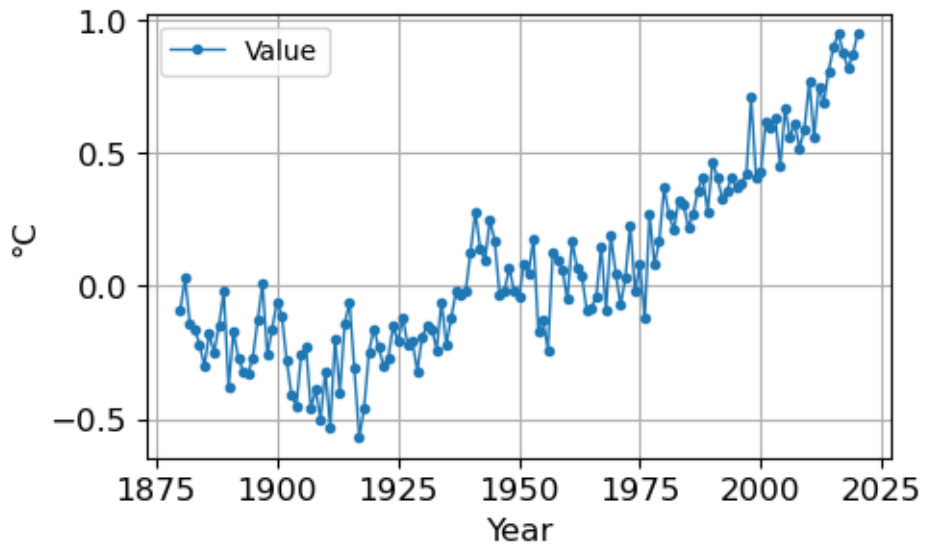
einzulesen und zu *parsen*, verwenden wir das sehr mächtige und allen Data Scientists empfehlenswerte Paket **pandas**.

```
import pandas as pd
```

```
filepath = "daten/data.csv"
df = pd.read_csv(filepath, skiprows=4, index_col=0) # read file into a pandas DataFrame
df
```

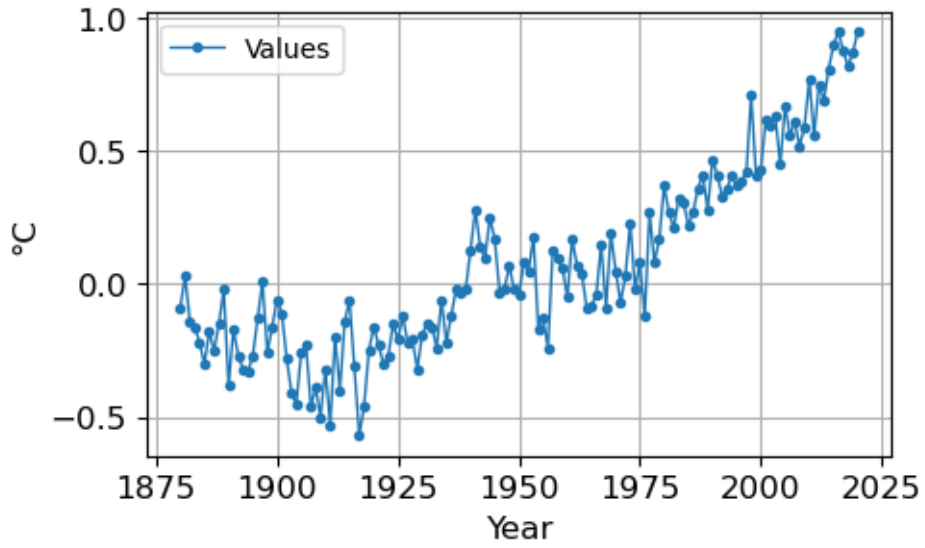
Year	Value
1880	-0.09
1881	0.03
1882	-0.14
1883	-0.16
1884	-0.22
...	...
2016	0.95
2017	0.88
2018	0.82
2019	0.87
2020	0.95

```
# We can directly plot the DataFrame ...

df.plot(figsize=(5, 3), grid=True, marker='.')
plt.ylabel('°C');
```




```
# ... or we convert the data to data types we already know ...

years = df.index.values      # to numpy array
values = df['Value'].values  # to numpy array
```

```
# ... and plot these:

plt.figure(figsize=(5, 3))
plt.plot(years, values, '.-', label='Values')
plt.xlabel('Year')
plt.ylabel('°C')
plt.legend()
plt.grid(True)
```




```
# exporting data to files:

# from DataFrame to Excel:
my_filepath = "daten/data_exported_with_pandas.xlsx"
df.to_excel(my_filepath)

# from numpy array to csv:
my_filepath = "daten/data_exported_with_numpy.csv"
np.savetxt(my_filepath, values, delimiter=',')  # using numpy
```

3. Pyomo

3.1. Überblick

Wir verwenden in der Lehrveranstaltung das Python-Paket Pyomo. Pyomo ist ein Open-Source-Projekt und frei verfügbar. Pyomo ist eine algebraische Modellierungssprache zur benutzerfreundlichen Formulierung von mathematischen Optimierungsproblemen. Zum Lösen eines mit Pyomo formulierten Optimierungsproblems wird ein Solver benötigt. Pyomo unterstützt eine Vielzahl von kommerziellen und Open-Source-Solvern.

Links:

- Pyomo Dokumentation zu Pyomo.
- Hands-On Optimization with Python insbesodere Pyomo style guide.

3.2. Alternativen

In diesem Abschnitt geben wir einen Überblick über andere High-Level Software zur Formulierung von Optimierungsproblemen, weil:

- Die Links führen Sie zu vielen, oft sehr guten Tutorials inkl. Code-Beispielen.
- In Realprojekten ist es wichtig, das passende Software-Setting unter Berücksichtigung folgender Punkte auszuwählen: kommerziell versus open source, Schnittstellen zu Programmiersprachen, Benutzerfreundlichkeit, Performance etc.. Dafür sollte man seinen Überblick über die LP-Software aktuell halten.

Liste von bekannten Algebraischen Modellierungssprachen:

- AMPL: you can find examples e. g. in the AMPL book
- GAMS: for some simple examples see e. g.the GAMS Quick Start Tutorial
- AIMMS: in AIMMS_modeling.pdf you find many examples and tricks
- Mosel: for examples see e. g. the FICO Xpress Optimization Examples Repository
- GNU MathProg where you can find a short example, see also GLPK and this online solver. The GNU MathProg language is a subset of the AMPL language.
- OPL: for examples see e. g. the OPL Language User's Manual
- MPL: you can find examples e. g. here

Für Python gibt es unter anderem folgende Modellierungs- und Entwicklungsumgebungen für (MI)LPs:

- SciPy LP and MILP with `linprog` and `milp`: open source solver and interface, modelling in matrix/vector form
- CVXOPT: open source interface to own solver and other solvers, can also solve quadratic programs, user's guide, see also cvxpy
- gurobipy: interface to commercial Gurobi solver
- Cplex: commercial solver and interface, academic license available
- lp_solve: open source solver and interface
- GLPK interfaces: open source solver and interfaces
- Pyomo: open source interface to different solvers
- PuLP: open source interface to own solver and other solvers documentation, github
- linopy: open source interface to different solvers
- Python-MIP: open source interface to different solvers
- PySCIPOpt: interface to the SCIP Optimization Suite, scipbook
- Optlang: open source interface to different solvers, github
- PYMPROG: open source interface to different solvers

3.3. Solver

Liste der bekanntesten Solver:

- CBC: open source solver
- HiGHs: open source solver
- GLPK: open source solver
- SCIP Optimization Suite: non-commercial solver for mixed integer programming (MIP) and mixed integer nonlinear programming (MINLP)
- lp_solve: open source solver
- Gurobi: commercial solver, academic license available
- Cplex: commercial solver and interface, academic license available
- Xpress: commercial solver
- Ipopt: open source software package fo large-scale nonlinear optimization, documentation

3.4. Installation

TIPP: Wir empfehlen die Installation von Pyomo und den Solvern in einer **conda environment**! Eine Anleitung dafür finden Sie hier in der Lehrveranstaltung Programmiertechniken.

3.4.1. Pyomo

- using conda: `conda install -c conda-forge pyomo`
- using pip: `pip install pyomo`

3.4.2. Solver

3.4.2.1. Windows

- GLPK using conda: `conda install -c conda-forge glpk`
- CBC: download the cbc zip-file for Windows from AMPL. Extract the zip-file and copy `cbc.exe` to the folder of your Python code. Use `solver = pyo.SolverFactory('cbc.exe')`.
- HiGHS: `conda install -c conda-forge highs`, and `pip install highspy`. Copy the file `highs.exe` from `C:\Users\USERNAME\...\pkgs\highs-XY\Library\bin` to the folder of your Python code.

3.4.2.2. macOS and Linux

- GLPK: install GLPK with your package manager, or `conda install -c conda-forge glpk`
- CBC: install CBC with your package manager, or `conda install -c conda-forge coincbc`.
- HiGHS: install HiGHS with your package manager, or `conda install -c conda-forge highs` and `pip install highspy`.

3.4.2.3. Alle Plattformen

- Gurobi: first install the gurobi solver using your `students.fhv.at` email account, cf. academic license. For the Python interface you have the following options:
  - using conda: `conda config --add channels https://conda.anaconda.org/gurobi`, `conda install gurobi`
  - using pip: `pip install gurobipy`

3.5. Tutorial

3.5.1. Beispiel: Butter und Eiscreme

Wir werden nun das folgende kleine LP mit Pyomo modellieren und mit verschiedenen Solvern lösen.

Fragestellung: Ein Bauer hat 3 Kühe, die in Summe 22 Gallonen (1 Gallone $\simeq$ 3.785 Liter) Milch pro Woche geben. Aus der Milch kann er Eiscreme und Butter machen. Er braucht 2 Gallonen Milch für 1 kg Butter und 3 Gallonen Milch für 1 Gallone Eiscreme. Es gibt keine Lagerrestriktionen für Butter. Er kann maximal 6 Gallonen Eiscreme lagern. Er hat 6 Arbeitsstunden pro Woche für die Herstellung zur Verfügung. Für 4 Gallonen Eiscreme benötigt er 1 Stunde, für 1 kg Butter benötigt er ebenfalls 1 Stunde. Die gesamte Produktion kann er zu folgenden Preisen verkaufen (vollständiger Absatz): 5 USD pro Gallone Eiscreme, 4 USD pro kg Butter. Seine Kosten belaufen sich auf 1.5 USD pro Gallone Eiscreme und 1 USD pro kg Butter. Wie viele Gallonen Eiscreme und wie viele kg Butter soll er herstellen, sodass er seinen Profit maximiert?

Modellierung:

Entscheidungsvariablen:

- x: produzierte Gallonen Eiscreme
- y: produzierte kg Butter

Zielfunktion: Maximiere den Profit $5x + 4y - (1.5x + y) = 3.5x + 3y$.

Nebenbedingungen:

- Lagerung von Eiscreme: $x \leq 6$
- Arbeitszeit: $\frac{1}{4}x + y \leq 6$
- verfügbare Milch: $3x + 2y \leq 22$
- Positivität der Variablen: $x \geq 0, y \geq 0$

Das gesamte lineare Programm (LP):

$$\begin{aligned} \max.\ & 3.5x + 3y \\ \text{s. t. } & x \leq 6 \\ & \frac{1}{4}x + y \leq 6 \\ & 3x + 2y \leq 22 \\ & x \geq 0 \\ & y \geq 0 \end{aligned}$$

Quellen:

- Problemstellung: Ferris, Mangasarian, Wright: Linear Programming with MATLAB. SIAM (Society for Industrial and Applied Mathematics), 2008
- Pyomo: A basic Pyomo model.

```
# The module pyomo.environ provides the components
# most commonly used for building Pyomo models:
import pyomo.environ as pyo
```

```
# create a model with an optional problem title:
model = pyo.ConcreteModel("butter_and_ice_cream")
```

```
# Display the model. Currently the model is empty:
model.display()
```

```
Model butter_and_ice_cream

  Variables:
    None

  Objectives:
    None

  Constraints:
    None
```

Entscheidungsvariablen:

- Standardmäßig ist die Domäne die Menge aller reellen Zahlen. Andere häufig verwendete Domänen sind `pyo.NonNegativeReals`, `pyo.NonNegativeIntegers` und `pyo.Binary`, siehe hier.
- `bounds` ist ein optionales Schlüsselwort zur Angabe eines Tupels, das Werte für die untere und obere Schranke enthält.

```
model.x = pyo.Var(bounds=(0, 6))
model.y = pyo.Var(domain=pyo.NonNegativeReals)

# display updated model:
model.display()
```

```
Model butter_and_ice_cream

  Variables:
    x : Size=1, Index=None
        Key  : Lower : Value : Upper : Fixed : Stale : Domain
        None :     0 :  None :     6 : False :  True :  Reals
    y : Size=1, Index=None
        Key  : Lower : Value : Upper : Fixed : Stale : Domain
        None :     0 :  None :  None : False :  True : NonNegativeReals

  Objectives:
    None

  Constraints:
    None
```

Ausdrücke (engl. expressions): Pyomo-Ausdrücke sind mathematische Formeln, die Entscheidungsvariablen enthalten. Pyomo-Ausdrücke werden verwendet, um die Zielfunktion und Nebenbedingungen zu definieren.

```
# create some expressions:
model.revenue = 5*model.x + 4*model.y
model.cost = 1.5*model.x + 1*model.y

# expressions can be printed:
print(model.revenue)
print(model.cost)
```

```
5*x + 4*y
1.5*x + y
```

Zielfunktion:

```
# set the objective function with another expression:
model.profit = pyo.Objective(expr = model.revenue - model.cost, sense = pyo.maximize)

# alternatively:
# model.profit = pyo.Objective(expr = 3.5*model.x + 3*model.y, sense = pyo.maximize)
```

```
# pretty print the model:
model.pprint()
```

```
2 Var Declarations
    x : Size=1, Index=None
        Key  : Lower : Value : Upper : Fixed : Stale : Domain
        None :     0 :  None :     6 : False :  True :  Reals
    y : Size=1, Index=None
        Key  : Lower : Value : Upper : Fixed : Stale : Domain
        None :     0 :  None :  None : False :  True : NonNegativeReals

1 Objective Declarations
    profit : Size=1, Index=None, Active=True
        Key  : Active : Sense    : Expression
        None :   True : maximize : 5*x + 4*y - (1.5*x + y)

3 Declarations: x y profit
```

Nebenbedingungen: Eine Nebenbedingung besteht aus zwei Ausdrücken, die durch eine der logischen Beziehungen Gleichheit (==), Kleiner-als (<=) oder Größer-als (>=) getrennt sind.

```
model.time = pyo.Constraint(expr = 0.25*model.x + model.y <= 6)
model.milk = pyo.Constraint(expr = 3*model.x + 2*model.y <= 22)

model.pprint()
```

```
2 Var Declarations
    x : Size=1, Index=None
        Key  : Lower : Value : Upper : Fixed : Stale : Domain
        None :     0 :  None :     6 : False :  True :  Reals
    y : Size=1, Index=None
        Key  : Lower : Value : Upper : Fixed : Stale : Domain
        None :     0 :  None :  None : False :  True : NonNegativeReals

1 Objective Declarations
    profit : Size=1, Index=None, Active=True
        Key  : Active : Sense    : Expression
        None :   True : maximize : 5*x + 4*y - (1.5*x + y)

2 Constraint Declarations
    milk : Size=1, Index=None, Active=True
        Key  : Lower : Body      : Upper : Active
        None :  -Inf : 3*x + 2*y :  22.0 :   True
    time : Size=1, Index=None, Active=True
        Key  : Lower : Body       : Upper : Active
        None :  -Inf : 0.25*x + y :   6.0 :   True

5 Declarations: x y profit time milk
```

Lösen des Optimierungsproblems: Ein Solver-Objekt wird mit SolverFactory erstellt und dann auf das Modell angewendet. Das optionale Schlüsselwort `tee=True` bewirkt, dass der Solver seine Ausgabe ausdruckt.

```
solver = pyo.SolverFactory('cbc')
# solver = pyo.SolverFactory('glpk')
# solver = pyo.SolverFactory('appsi_highs')
# solver = pyo.SolverFactory('gurobi')

assert solver.available(), f"Solver {solver} is not available."
# If the solver is not available, i.e., solver.available() returns False,
# then the assert statement will raise an AssertionError with the message
# f"Solver {solver} is not available."
```

```
# solve the model with the chosen solver:
results = solver.solve(model, tee=True)
```

```
Welcome to the CBC MILP Solver
Version: 2.10.11
Build Date: Oct 26 2023

command line - /bin/cbc -printingOptions all -import /tmp/tmp4mgaf1j0.pyomo.lp -stat=1 -solve -solu /tmp/tmp4
Option for printingOptions changed from normal to all
 CoinLpIO::readLp(): Maximization problem reformulated as minimization
Coin0009I Switching back to maximization to get correct duals etc
Presolve 2 (0) rows, 2 (0) columns and 4 (0) elements
Statistics for presolved model


Problem has 2 rows, 2 columns (2 with objective) and 4 elements
Column breakdown:
1 of type 0.0->inf, 1 of type 0.0->up, 0 of type lo->inf,
0 of type lo->up, 0 of type free, 0 of type fixed,
0 of type -inf->0.0, 0 of type -inf->up, 0 of type 0.0->1.0
Row breakdown:
0 of type E 0.0, 0 of type E 1.0, 0 of type E -1.0,
0 of type E other, 0 of type G 0.0, 0 of type G 1.0,
0 of type G other, 0 of type L 0.0, 0 of type L 1.0,
2 of type L other, 0 of type Range 0.0->1.0, 0 of type Range other,
0 of type Free
Presolve 2 (0) rows, 2 (0) columns and 4 (0) elements
0  Obj 0 Dual inf 6.4999998 (2)
0  Obj 0 Dual inf 6.4999998 (2)
3  Obj 29
Optimal - objective value 29
Optimal objective 29 - 3 iterations time 0.002
Total time (CPU seconds):       0.00   (Wallclock seconds):       0.00
```

Zusammenfassung:

```
import pyomo.environ as pyo

model = pyo.ConcreteModel("butter_and_ice_cream")

model.x = pyo.Var(bounds=(0, 6))
model.y = pyo.Var(domain=pyo.NonNegativeReals)
```

```
model.profit = pyo.Objective(expr = 3.5*model.x + 3*model.y, sense = pyo.maximize)

model.time = pyo.Constraint(expr = 0.25*model.x + model.y <= 6)
model.milk = pyo.Constraint(expr = 3*model.x + 2*model.y <= 22)

solver = pyo.SolverFactory('cbc')
# solver = pyo.SolverFactory('glpk')
# solver = pyo.SolverFactory('appsi_highs')
# solver = pyo.SolverFactory('gurobi')

results = solver.solve(model, tee=True)
```

```
Welcome to the CBC MILP Solver
Version: 2.10.11
Build Date: Oct 26 2023

command line - /bin/cbc -printingOptions all -import /tmp/tmpwogw20x9.pyomo.lp -stat=1 -solve -solu /tmp/tmpw
Option for printingOptions changed from normal to all
 CoinLpIO::readLp(): Maximization problem reformulated as minimization
Coin0009I Switching back to maximization to get correct duals etc
Presolve 2 (0) rows, 2 (0) columns and 4 (0) elements
Statistics for presolved model


Problem has 2 rows, 2 columns (2 with objective) and 4 elements
Column breakdown:
1 of type 0.0->inf, 1 of type 0.0->up, 0 of type lo->inf,
0 of type lo->up, 0 of type free, 0 of type fixed,
0 of type -inf->0.0, 0 of type -inf->up, 0 of type 0.0->1.0
Row breakdown:
0 of type E 0.0, 0 of type E 1.0, 0 of type E -1.0,
0 of type E other, 0 of type G 0.0, 0 of type G 1.0,
0 of type G other, 0 of type L 0.0, 0 of type L 1.0,
2 of type L other, 0 of type Range 0.0->1.0, 0 of type Range other,
0 of type Free
Presolve 2 (0) rows, 2 (0) columns and 4 (0) elements
0  Obj 0 Dual inf 6.4999998 (2)
0  Obj 0 Dual inf 6.4999998 (2)
3  Obj 29
Optimal - objective value 29
Optimal objective 29 - 3 iterations time 0.002
Total time (CPU seconds):       0.00   (Wallclock seconds):       0.00
```

Analyse der Lösung:

```
# pretty print the whole model:
model.pprint()
```

```
2 Var Declarations
    x : Size=1, Index=None
        Key  : Lower : Value : Upper : Fixed : Stale : Domain
        None :     0 :   4.0 :     6 : False : False :  Reals
    y : Size=1, Index=None
        Key  : Lower : Value : Upper : Fixed : Stale : Domain
        None :     0 :   5.0 :  None : False : False : NonNegativeReals
```

```
1 Objective Declarations
    profit : Size=1, Index=None, Active=True
        Key  : Active : Sense    : Expression
        None :   True : maximize : 3.5*x + 3*y

2 Constraint Declarations
    milk : Size=1, Index=None, Active=True
        Key  : Lower : Body      : Upper : Active
        None :  -Inf : 3*x + 2*y :  22.0 :   True
    time : Size=1, Index=None, Active=True
        Key  : Lower : Body       : Upper : Active
        None :  -Inf : 0.25*x + y :   6.0 :   True

5 Declarations: x y profit time milk
```

```
# display a component of the model:
model.profit.pprint()
```

```
profit : Size=1, Index=None, Active=True
    Key  : Active : Sense    : Expression
    None :   True : maximize : 3.5*x + 3*y
```

```
# display the solution values of model components:
print(f"Profit = {pyo.value(model.profit): 5.2f} USD")
print(f"     x = {pyo.value(model.x): 5.2f} units")
print(f"  Time = {pyo.value(model.time): 5.2f} hours")
```

```
Profit =  29.00 USD
     x =  4.00 units
  Time =  6.00 hours
```

```
# Alternative display of variable values:
# After a solution has been computed, a function with the same name as decision variable
# is created that will report the solution value.

print(f"optimal x = {model.x()} gallons of ice cream")
print(f"optimal y = {model.y()} kg of butter")
```

```
optimal x = 4.0 gallons of ice cream
optimal y = 5.0 kg of butter
```

3.5.2. Beispiel: Biomasse(heiz)kraftwerk

Sie wollen über die nächsten 24 Stunden $t = 0, \dots, 23$ Ihr Biomasse(heiz)kraftwerk mit maximalem Gesamterlös betreiben. Für jede Stunde können Sie die erzeugte Energiemenge x_t (MWh) zwischen 0 und 100 MWh wählen. Die Verkaufspreise p_t (€/MWh) für Ihre erzeugten Energiemengen x_t sind für alle 24 Stunden bekannt. Sie können in den nächsten 24 Stunden in Summe maximal 1000 MWh erzeugen. Von einer auf die nächste Stunde kann sich die erzeugte Energiemenge um maximal 20 MWh ändern. Bestimmen Sie einen optimalen Betriebsplan x_t für die nächsten 24 Stunden $t = 0, \dots, 23$.

Modellierung als LP:

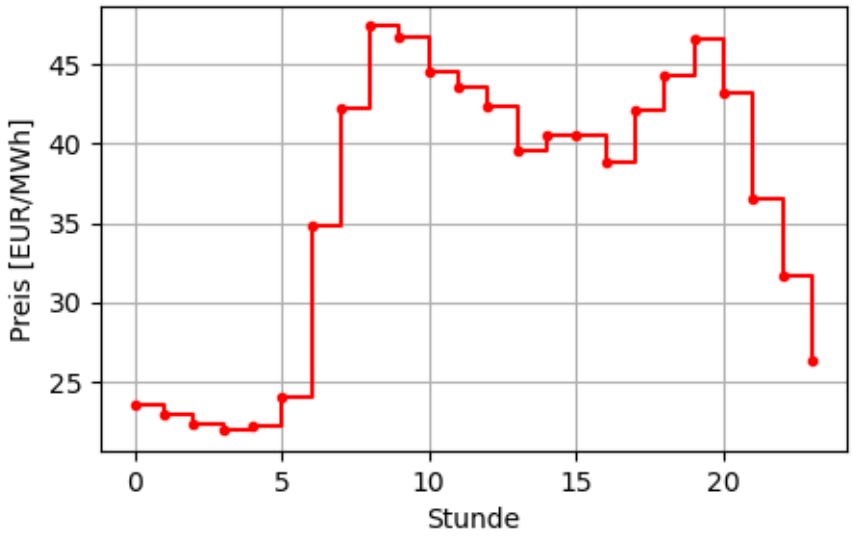
- Entscheidungsvariablen: Energiemengen x_t (MWh) für jede Stunde $t = 0, \dots, 23$ mit Schranken $0 \leq x_t \leq 100$ MWh

- Zielfunktion: Maximiere den Gesamterlös $\sum_{t=0}^{23} p_t x_t$
- Nebenbedingungen:
  - $\sum_{t=0}^{23} x_t \leq 10000$ MWh
  - $-20 \leq x_t - x_{t-1} \leq 20$ MWh für alle $t = 1, \ldots, 23$

Implementierung mit Pyomo:

```
import pyomo.environ as pyo
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
```

```
times = np.arange(0, 24, step=1)
prices = np.array([23.46, 22.91, 22.35, 21.91, 22.13, 23.96,
                   34.83, 42.24, 47.45, 46.80, 44.58, 43.56,
                   42.33, 39.61, 40.49, 40.49, 38.83, 42.07,
                   44.27, 46.68, 43.27, 36.50, 31.71, 26.29])

plt.figure(figsize=(5, 3))
plt.step(times, prices, where='post', marker='.', color='red')
plt.xlabel('Stunde')
plt.ylabel('Preis [EUR/MWh]')
plt.grid(True)
```



Wir lösen das LP in Pyomo mit einer Indexmenge zum Indizieren der Entscheidungsvariablen und Nebenbedingungen, vgl. Pyomo: Sets:

```
model = pyo.ConcreteModel()

# Add the set of times to the model. This set will be used to index the variables:
model.T = pyo.Set(initialize=times)

# Add the variables to the model: energy (MWh) x_t produced in each hour
# include bounds on the variables: 0 <= x_t <= 100
model.x = pyo.Var(model.T, bounds=(0, 100))
```

```
# Add the objective function to the model including the sense of the optimization:
model.revenue = pyo.Objective(expr = pyo.quicksum(model.x[t]*prices[t] for t in model.T),
                              sense = pyo.maximize)

# Add the total energy constraint to the model:
model.resource = pyo.Constraint(expr = pyo.quicksum(model.x[t] for t in model.T) <= 1000)

# Add the constraints for ramping up: The @ decorates
# the ramp_up function as constraints for all times of the set model.T.
@model.Constraint(model.T)
def ramp_up(model, t):
    if t >= 1:
        # return the constraint expression:
        return model.x[t] - model.x[t - 1] <= 20
    else:
        # skip this index:
        return pyo.Constraint.Skip

# add the constraints for ramping down:
@model.Constraint(model.T)
def ramp_down(model, t):
    if t >= 1:
        # return the constraint expression:
        return -20 <= model.x[t] - model.x[t - 1]
    else:
        # skip this index:
        return pyo.Constraint.Skip

# model.pprint()
```

If you want to know what a decorator really does read for example this primer on decorators.

```
# Let's solve the model:

solver = pyo.SolverFactory('cbc')
# solver = pyo.SolverFactory('glpk')
# solver = pyo.SolverFactory('appsi_highs')
# solver = pyo.SolverFactory('gurobi')

results = solver.solve(model, tee=False)
print(results.solver.status)
```

```
ok
```

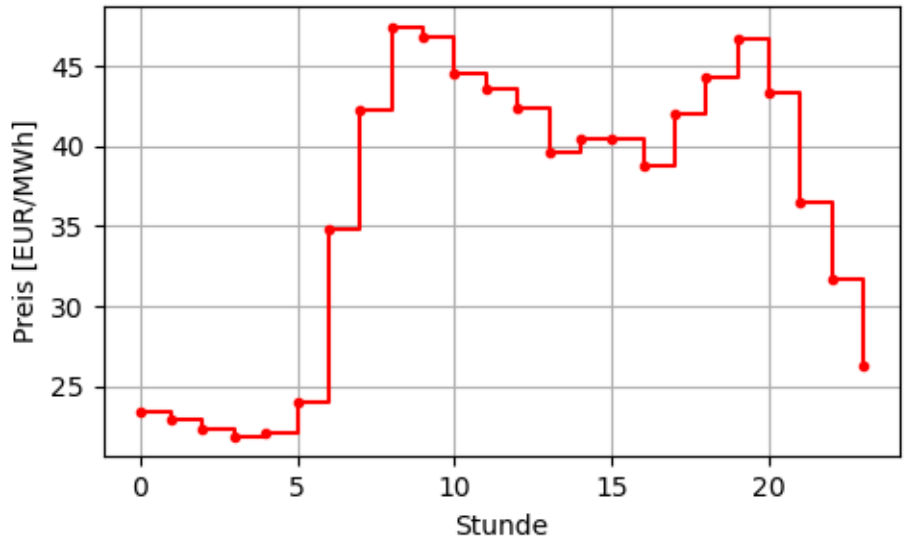
```
# Print the optimal revenue:
print(f"optimal revenue = {pyo.value(model.revenue):.2f} EUR")
```

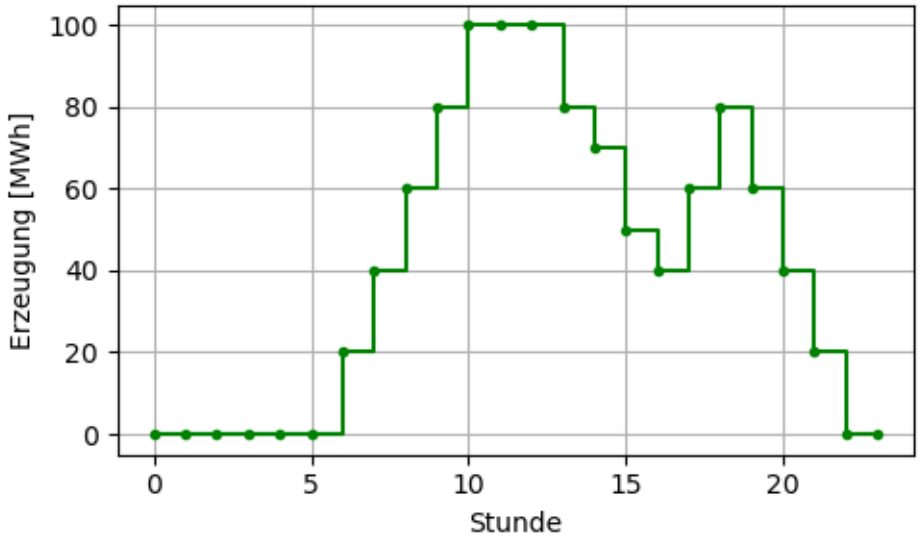
```
optimal revenue = 42932.40 EUR
```

```
# optimal production schedule x_t as a dictionary with the times as keys:
sol_dict = model.x.extract_values()
sol_dict
```

```
{0: 0.0,
 1: 0.0,
 2: 0.0,
 3: 0.0,
 4: 0.0,
 5: 0.0,
 6: 20.0,
 7: 40.0,
 8: 60.0,
 9: 80.0,
 10: 100.0,
 11: 100.0,
 12: 100.0,
 13: 80.0,
 14: 70.0,
 15: 50.0,
 16: 40.0,
 17: 60.0,
 18: 80.0,
 19: 60.0,
 20: 40.0,
 21: 20.0,
 22: 0.0,
 23: 0.0}
```

```
# plot of the optimal production schedule:
plt.figure(figsize=(5, 6))
plt.subplot(2, 1, 1)
plt.step(times, prices, where='post', marker='.', color='red')
plt.xlabel('Stunde')
plt.ylabel('Preis [EUR/MWh]')
plt.grid(True)

plt.subplot(2, 1, 2)
plt.step(times, [sol_dict[t] for t in times], where='post', marker='.', color='green')
plt.xlabel('Stunde')
plt.ylabel('Erzeugung [MWh]')
plt.grid(True)

plt.tight_layout()
```

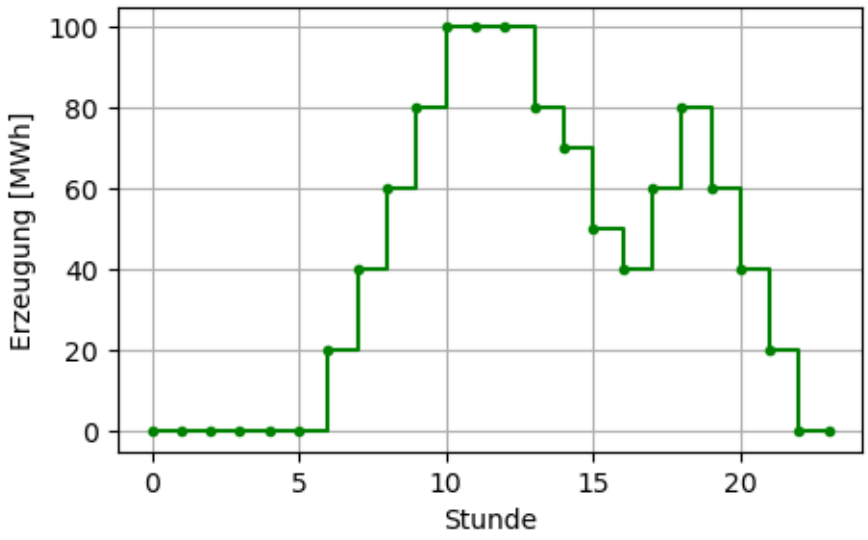





```
# Alternative plot of the optimal production schedule
# as a pandas dataframe with the times as index:
df = pd.DataFrame.from_dict(sol_dict, orient='index', columns=['x'])
display(df)

df.plot(drawstyle="steps-post", figsize=(5, 3), legend=False, color='green',
        xlabel='Stunde', ylabel='Erzeugung [MWh]', marker='.', grid=True);
```

	x
0	0.0
1	0.0
2	0.0
3	0.0
4	0.0
5	0.0

	x
6	20.0
7	40.0
8	60.0
9	80.0
10	100.0
11	100.0
12	100.0
13	80.0
14	70.0
15	50.0
16	40.0
17	60.0
18	80.0
19	60.0
20	40.0
21	20.0
22	0.0
23	0.0



3.5.3. Hinweise

- Pyomo und die Pyomo Dokumentation sind sehr umfangreich. Wir verwenden davon nur einen Teil, z. B. nur die sogenannten *konkrete Modelle* und keine *abstrakten Modelle*.
- Unter Pyomo kann man den Größen Einheiten geben und so die Ausdrücke auf Konsistenz bzgl. Einheiten prüfen lassen, siehe Units Handling in Pyomo.

3.6. Übung

Lösen Sie das folgende lineare Programm (LP) mit Pyomo, d. h.:

1. Bestimmen Sie die Entscheidungsvariablen inklusive Einheiten, Datentyp und Schranken.
2. Formulieren Sie die Zielfunktion und die Nebenbedingungen.

3. Modellieren Sie das LP mit Pyomo.
4. Lösen Sie das LP mit verschiedenen Solvern, und geben Sie die Lösung formatiert aus.
5. Interpretieren Sie die Lösung! Ist sie plausibel? Ändern Sie z. B. die Daten, um Ihre Lösung an einfachen Spezialfällen zu überprüfen.

Gasoline-Mixture Problem: A gasoline refiner needs to produce a cost-minimizing blend of ethanol and traditional gasoline. The blend needs to have at least 65 % burning efficiency and a pollution level no greater than 85 %. The burning efficiency, pollution level, and per-ton cost of ethanol and traditional gasoline are given in the following table:

Product	Efficiency [%]	Pollution [%]	Cost [$/ton]
Gasoline	70	90	200
Ethanol	60	80	220

Quelle: Sioshansi, Ramteen; Conejo, Antonio J. (2017): Optimization in Engineering: Models and Algorithms. 2017, Springer. Seite 24 f.

Teil II.

Theorie

4. Netzwerke

4.1. Graphentheorie

Ein **Graph** ist ein Modell einer realen Gegebenheit, das deren relevante Objekte (=Knoten) und die relevanten Verbindungen (=Kanten) zwischen den Objekten abbildet. Es gibt sehr viele Situationen, die mittels Graphen modelliert werden. Hier eine kleine Auswahl:

- wirtschaftliche oder soziale Beziehungen
- Kommunikationsnetze wie das Internet
- Verkehrsnetze: Straßennetze, U-Bahnnetz, Flugverbindungen
- Stromverteilungsnetze
- Nahwärmenetze
- Molekülstrukturen
- Stammbäume
- Ordnerstrukturen

Ein **ungerichteter Graph** besteht einfach aus einer Menge von **Knoten** (Ecken, Punkt, englisch vertices, nodes, points) und einer Menge von **Kanten** (Bögen, engl. edges, links, lines). Eine Kante wird durch ein ungeordnetes Paar von Knoten, den Endknoten der Kante, angegeben und als **Linie** gezeichnet.

In Abbildung Abbildung 4.1 ist ein ungerichteter Graph mit 5 Knoten und 6 Kanten dargestellt.

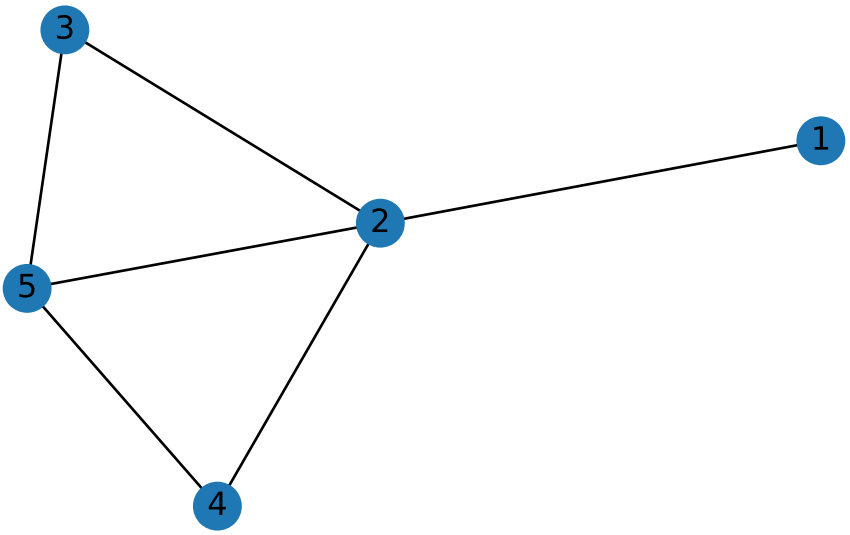


Abbildung 4.1.: Ungerichteter Graph mit 5 Knoten und 6 Kanten

Ein **gerichteter Graph** (englisch directed graph) besteht aus einer Menge von Knoten und einer Menge von **gerichteten Kanten**, die durch geordnete Paare von Knoten, den Anfangsknoten und den Endknoten, bestimmt sind. Die gerichteten Kanten werden statt durch Linien durch **Pfeile** gekennzeichnet, wobei der Pfeil von ihrem Anfangs- zu ihrem Endknoten zeigt.

In Abbildung Abbildung 4.2 ist ein gerichteter Graph mit 4 Knoten und 5 gerichteten Kanten dargestellt.

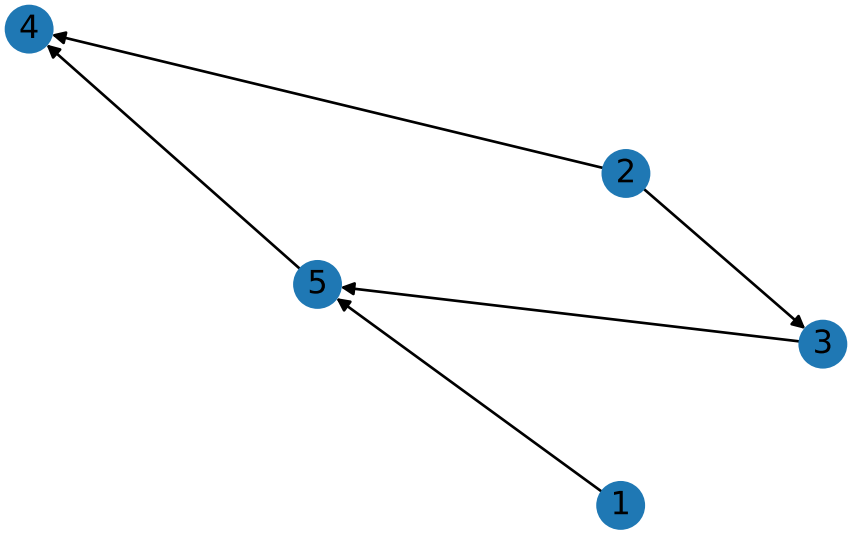


Abbildung 4.2.: Gerichteter Graph mit 4 Knoten und 5 Kanten

4.2. Netzwerke

Ein (Fluss- oder Transport-) **Netzwerk** (engl. network) ist ein zusammenhängender, **gerichteter Graph**, bei dem jede Kante einen **Fluss** aufnehmen kann und jede Kante eine **Kapazität** (allgemeiner: untere und obere Flussschranken) für den Fluss hat. Die Menge des Flusses auf einer Kante kann die Kapazität der Kante nicht überschreiten. Ein Fluss muss die Einschränkung erfüllen, dass die **Menge des Flusses in einen Knoten gleich der Menge des Flusses aus ihm heraus** ist. Ein **externer Fluss** in oder aus einem Knoten ist erlaubt und wird neben dem Knoten oder als Fluss auf einer zusätzlichen Kante des Knotens angegeben.

Ein **Fluss-Netzwerk** (engl. flow network) ist ein Netzwerk, dessen Kanten zusätzlich **Kosten pro Mengeneinheit** des Flusses zugeordnet sind. Typischerweise will man einen Fluss durch die Kanten bestimmen, der den Einschränkungen des Netzwerks genügt und dessen Gesamtkosten minimal sind.

Vorzeichenkonventionen:

- Ein positiver Fluss fließt in Pfeilrichtung, ein negativer Fluss fließt gegen die Pfeilrichtung.
- Ein positiver externer Fluss fließt dabei in den Knoten, ein negativer externer Fluss bedeutet, dass der zugehörige positive Fluss aus dem Knoten herausfließt.

Default-Werte: Falls einem Knoten oder einer Kante keine Parameterwerte zugeschrieben werden, gelten Default-Werte.

Abbildung Abbildung 4.3 zeigt ein Fluss-Netzwerk. Die Kanten sind mit ihren Kapazitäten und Kosten beschriftet. Die Knoten sind mit ihren externen Flüssen beschriftet.

Anwendungen: Ein Netzwerk kann verwendet werden, um den Verkehr in einem Computer- oder Straßennetzwerk, Flüssigkeiten in Rohren, Energien in Leitungen, Ströme in einem elektrischen Stromkreis oder Ähnliches zu modellieren, solange “etwas” durch ein Netzwerk von Knoten “fließt”.

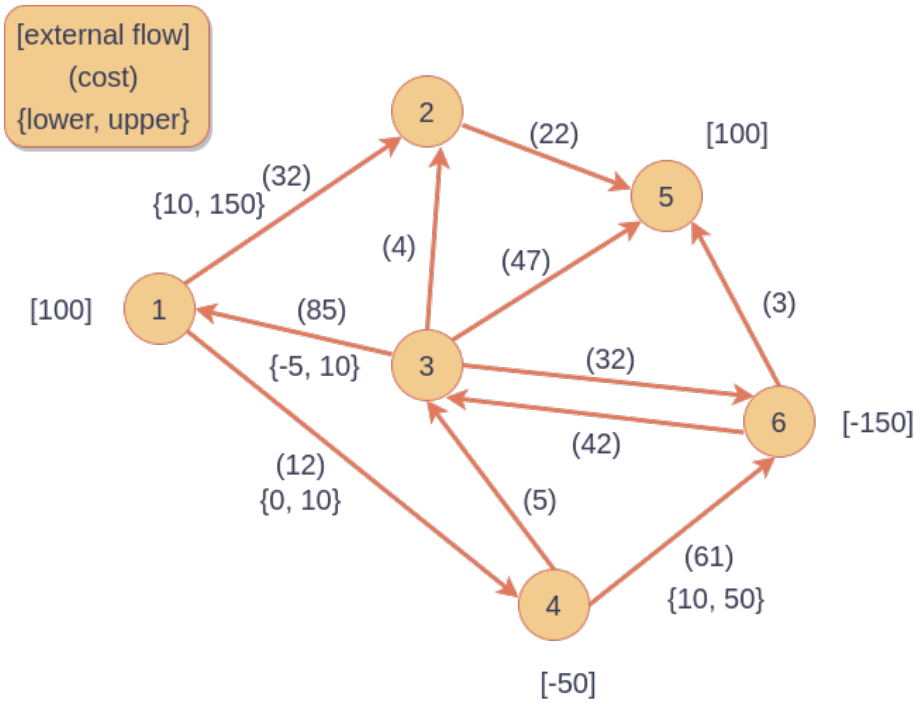


Abbildung 4.3.: Beispiel eines Fluss-Netzwerks

4.3. Energienetzwerke

In Energienetzwerken ist der Fluss gleich der (elektrischen oder thermischen) **Leistung**, die z. B. in kW angegeben wird. Die Kapazität ist die maximale Leistung, die über die Kante fließen kann. Die externen Flüsse sind die Leistungen, die in den Knoten eingespeist oder entnommen werden. Die oben erwähnte **Knotenregel**, dass die Menge des Flusses in einen Knoten gleich der Menge des Flusses aus ihm heraus ist, entspricht der **Energieerhaltung**.

Der Leistung (Last, Einspeisung, Erzeugung etc.) p auf einer Kante ist in vielen Anwendungen zeitabhängig, also eine Funktion $p(t)$ mit kontinuierlicher Zeit t. Die Knotenregel gilt dann zu jedem Zeitpunkt t. Ein zeitkontinuierlicher (=analoger) Leistungsverlauf $p(t)$ wird für datentechnische Anwendungen typischerweise in stückweise konstante Werte p_i **gesampelt**, siehe Beispiel: (Re-)Sampling.

5. Dynamische Probleme

5.1. Zeitdiskretisierung

Wir betrachten Energienetzwerke, deren Flüsse (=Leistungen auf den Kanten) sich im Zeitverlauf ändern können. Dabei verwenden wir gesampelte Leistungen und Energien, d. h. wir betrachten diskrete Zeitperioden und Zeitpunkte. Die Leistungen (typischerweise in kW) sind über eine Periode konstant, und die Energien (typischerweise in kWh) werden zu den Zeitpunkten, die die Perioden begrenzen, betrachtet. Für die Implementierung unter Python bietet sich folgende Konvention der Indexierung der Zeitpunkte und -perioden an:

- Samplingintervall Δt, typischerweise in h
- n Zeitperioden (=Intervalle)
- Gesamtdauer $T = \Delta t \cdot n$, typischerweise in h
- Zeitpunkte $t_i = \Delta t \cdot i$ für Indizes $i = 0, 1, 2, \dots, n$
- Energien zu den Zeitpunkten: E_i für Indizes $i = 0, 1, 2, \dots, n$, typischerweise in kWh
- Zeitperioden $[t_j, t_{j+1})$ für Indizes $j = 0, 1, 2, \dots, n-1$
- Leistungen während der Zeitperioden: p_j für Indizes $j = 0, 1, 2, \dots, n-1$, typischerweise in kW

Siehe Abbildung Abbildung 5.1.

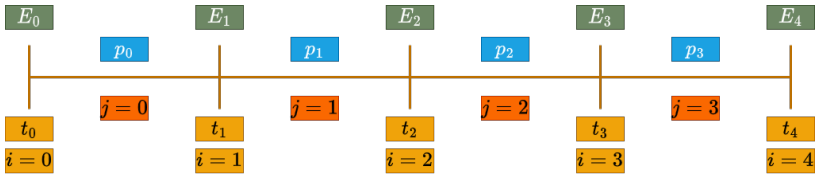


Abbildung 5.1.: Indexierung von Zeitpunkten, -perioden, Energien und Leistungen

5.2. Flexibilitäten

5.2.1. Ladeflexibilität

5.2.1.1. Verlustfrei

Lade den Energieinhalt E_i von E_{start} auf E_{end} in n Zeitperioden:

- Die Energien sind begrenzt durch $0 \leq E_i \leq E_{\max} \quad \forall i = 0, 1, 2, \dots, n.$
- Die Ladeleistungen p_j sind begrenzt durch $0 \leq p_j \leq p_{\max} \quad \forall j = 0, 1, 2, \dots, n-1.$
- Update des Energieinhalts: $E_{i+1} = E_i + p_i \Delta t \quad \forall i = 0, 1, 2, \dots, n-1.$
- Anfangs- und Endenergie: $E_0 = E_{\text{start}}$ und $E_n = E_{\text{end}}$.

5.2.1.2. Ladeverluste

Der Update des Energieinhalts ändert sich zu

$$E_{i+1} = E_i + \eta p_i \Delta t \quad \forall i = 0, 1, 2, \dots, n-1,$$

wobei $0 < \eta < 1$ der Ladewirkungsgrad ist.

5.2.1.3. Minimale Ladeleistung

Eine Größe p, wie z. B. die Ladeleistung eines E-Mobils während eines bestimmten Zeitintervalls, kann der Einschränkung unterworfen sein, dass sie entweder Null oder im Intervall $[l, u]$ liegt. Um diese Nebenbedingung zu implementieren, führen wir die binäre Variable $\lambda \in \{0, 1\}$ ein und formulieren die Nebenbedingung als

$$l\lambda \leq p \leq u\lambda.$$

Die binäre Variable λ ist 1, wenn das E-Mobil geladen wird und 0, wenn nicht.

5.2.2. Batterie

5.2.2.1. Verlustfreie Batterie

Entscheidungsvariablen:

- E_i … Energieinhalt der Batterie zum i-ten Zeitpunkt, $i = 0, 1, 2, \dots, n$
- p_j … Ladeleistung der Batterie während der j-ten Zeitperiode, $j = 0, 1, 2, \dots, n-1$

Nebenbedingungen:

- Die Energien sind begrenzt durch $0 \leq E_i \leq E_{\max}$ $\forall i = 0, 1, 2, \dots, n$.
- Die Ladeleistungen p_j sind begrenzt durch $p_{\min} \leq p_j \leq p_{\max}$ $\forall j = 0, 1, 2, \dots, n-1$, wobei $p_{\min} < 0$.
- Update des Energieinhalts: $E_{i+1} = E_i + p_i \Delta t$ $\forall i = 0, 1, 2, \dots, n-1$.
- Anfangs- und Endenergie: $E_0 = E_{\text{start}}$ und $E_n = E_{\text{end}}$.

5.2.2.2. Lade- und Entladeverluste

In Abbildung Abbildung 5.2 sind die Lade- und Entladeverluste einer Batterie berücksichtigt.

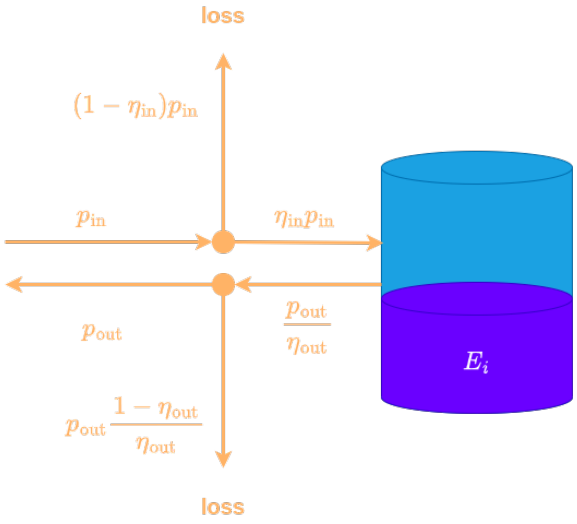


Abbildung 5.2.: Batterie mit Lade- und Entladeverluste

Entscheidungsvariablen:

- $p_{\text{in},j}$: Ladeleistung in kW während der Zeitperioden j, $0 \leq p_{\text{in},j} \leq p_{\text{in,max}}$
- $p_{\text{out},j}$: Entladeleistung in kW während der Zeitperioden j, $0 \leq p_{\text{out},j} \leq p_{\text{out,max}}$
- E_i: Energieinhalt der Batterie in kWh zu den Zeitpunkten i

- $b_{\text{in},j}$: binäre Variable, die angibt, ob die Batterie in der Zeitperiode j geladen wird ($b_{\text{in},j} = 1$) oder nicht ($b_{\text{in},j} = 0$)
- $b_{\text{out},j}$: binäre Variable, die angibt, ob die Batterie in der Zeitperiode j entladen wird ($b_{\text{in},j} = 1$) oder nicht ($b_{\text{in},j} = 0$)

Nebenbedingungen:

- Anfangs- und Endenergie: $E_0 = E_{\text{start}}$ und $E_n = E_{\text{end}}$.
- Die Energien sind begrenzt durch $0 \leq E_i \leq E_{\max}$ $\forall i = 0, 1, 2, \dots, n$.
- zeitliche Änderung der Energie der Batterie:

$$E_{i+1} = E_i + (\eta_{\text{in}} p_{\text{in},i} - \frac{p_{\text{out},i}}{\eta_{\text{out}}})\Delta t \quad \forall i = 0, 1, 2, \dots, n-1$$

- Kein gleichzeitiges Laden und Entladen der Batterie:
  - $p_{\text{in},j} \leq p_{\max} b_{\text{in},j}$ für alle $j = 0, 1, 2, \dots, n-1$
  - $p_{\text{out},j} \leq p_{\max} b_{\text{out},j}$ für alle $j = 0, 1, 2, \dots, n-1$
  - $b_{\text{in},j} + b_{\text{out},j} \leq 1$ für alle $j = 0, 1, 2, \dots, n-1$

5.2.3. Verschiebbare Lasten

Eine verschiebbare Last habe den Lastgang l_k in kW über m Perioden mit Indizes $k = 0, 1, 2, \dots, m-1$. Wir betrachten n Zeitperioden mit Indizes $j = 0, 1, 2, \dots, n-1$ und $n > m$. Der Startzeitpunkt i der verschiebbaren Last mit $0 \leq i \leq n-m$ sei frei wählbar. Um den resultierenden flexiblen Lastgang p_j über alle $j = 0, 1, \dots, n-1$ zu beschreiben, führen wir die binären Variablen s_i für $i = 0, 1, \dots, n-m$ ein, die angeben, dass die Last zum Zeitpunkt i gestartet wird, falls $s_i = 1$. Mit den folgenden Nebenbedingungen lässt sich dann der Lastgang p_j beschreiben:

- $\sum_{i=0}^{n-m} s_i = 1$: Die Last wird genau einmal gestartet.
- $p_j = \sum_{k=0}^{m-1} l_k s_{j-k}$ für alle $j = 0, 1, \dots, n-1$, falls $j - k \leq n - m$: Falls die Last zum Zeitpunkt $j - k$ gestartet wurde, dann hat sie in der Zeitperiode j die Leistungsaufnahme l_k.

5.2.4. Thermische Speicher

Wir betrachten einen elektrisch betriebenen Warmwasserboiler mit einem Volumen von 150 Litern. Das Wasser hat daher eine Masse m von ca. 150 kg. Die spezifische Wärmekapazität von Wasser beträgt ca. $c_p = 4.2$ kJ/(kg K). Die Wärmekapazität des Wassers beträgt daher $c = mc_p = 630$ kJ/K, was 0.175 kWh/K entspricht. Die Energiebilanz des Wassers lautet

$$c\dot{T}(t) = P_{\text{el}}(t) - P_{\text{loss}}(t) - P_{\text{dem}}(t).$$

Dabei ist $T(t)$ die Temperatur des Wassers in °C, $P_{\text{el}}(t)$ die elektrische Leistung in kW, die beim Aufheizen in das Wasser eingebracht wird, $P_{\text{loss}}(t)$ die Leistung in kW, die das Wasser an die Umgebung verliert, und $P_{\text{dem}}(t)$ die Leistung in kW, die durch Austausch von warmem mit kaltem Wasser an die Verbraucher abgegeben wird. Die Verluste an die Umgebung sind wegen Netwons Abkühlgesetz gegeben durch

$$P_{\text{loss}}(t) = k(T(t) - T_{\text{env}}),$$

wobei $k = 1.2$ W/K die Wärmeübertragungseigenschaften (Konvektion und Konduktion) beschreibt, und T_{env} die Umgebungstemperatur in °C angibt. Einsetzen von $P_{\text{loss}}(t)$ liefert die lineare Differentialgleichung erster Ordnung

$$c\dot{T}(t) + k(T(t) - T_{\text{env}}) = P_{\text{el}}(t) - P_{\text{dem}}(t).$$

Wir definieren $E(t) = cT(t)$ als Energieinhalt des Wassers in kWh. Dann lautet die Differentialgleichung für $E(t)$

$$\dot{E}(t) + \frac{k}{c}(E(t) - cT_{\text{env}}) = P_{\text{el}}(t) - P_{\text{dem}}(t).$$

Wenn die Umgebungstemperatur, die elektrische Leistung und der Verbrauch während einer Dauer Δt konstant sind, dann ist

$$E(t+\Delta t) = E(t)e^{-\frac{k}{c}\Delta t} + \frac{c}{k}(1-e^{-\frac{k}{c}\Delta t})(P_{\text{el}} - P_{\text{dem}} + kT_{\text{env}}),$$

siehe z. B. Ingenieurmathematik: Lineare DGL 1. Ordnung.

Plausibilitätschecks:

- Für $\Delta t = 0$ h ist $E(t+\Delta t) = E(t)$.
- Für kleine Δt ist $E(t+\Delta t) \approx E(t)(1-\frac{k}{c}\Delta t) + \Delta t(P_{\text{el}} - P_{\text{dem}} + kT_{\text{env}})$, da dann $e^{-\frac{k}{c}\Delta t} \simeq 1 - \frac{k}{c}\Delta t$.
- Für große Δt und $P_{\text{el}} = P_{\text{dem}} = 0$ ist $E(t+\Delta t) \approx cT_{\text{env}}$, da dann $e^{-\frac{k}{c}\Delta t} \simeq 0$. Das heißt, ohne Aufheizen und Verbrauch ist für große Δt die Temperatur des Wassers gleich der Umgebungstemperatur.

Für die oben angegebenen Werte von c, k sowie eine Umgebungstemperatur $T_{\text{env}} = 15$ °C und eine Dauer $\Delta t = 0.25$ h ist

$$E(t+\Delta t) = 0.9983E(t) + 0.2498(P_{\text{el}} - P_{\text{dem}} + 0.0012T_{\text{env}}).$$

In Abbildung Abbildung 5.3 ist für dieses Setting und $P_{\text{el}} = P_{\text{dem}} = 0$ die Temperatur des Wassers über eine Woche dargestellt.

```
import numpy as np
import matplotlib.pyplot as plt

dt = 0.25   # h
c = 0.175   # kWh/K
k = 0.0012  # kW/K
T_env = 15  # °C
T = 24*7    # h
times = np.arange(0, T + dt, dt) # h
E = np.zeros_like(times)
E[0] = c*70.0  # 70 °C
for i in range(0, len(times)-1):
    E[i+1] = E[i]*np.exp(-k/c*dt) + c/k*(1 - np.exp(-k/c*dt))*k*T_env

plt.figure(figsize=(5, 3))
plt.plot(times/24, E/c)
plt.xlabel("time (d)")
plt.ylabel("temperature (°C)")
plt.grid()
```

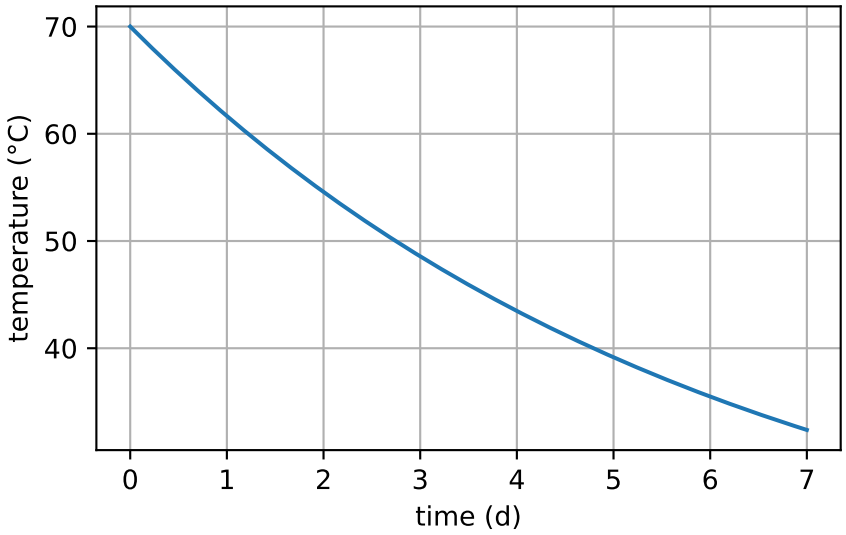


Abbildung 5.3.: Temperaturverlauf eines thermischen Speichers

Literaturhinweis: Peter Kepplingers Doktorarbeit

5.3. Zielfunktionen

5.3.1. Lineare Energiekosten

Gegeben sind Energiepreise c_j, die während der Zeitperioden mit Indizes $j = 0, 1, 2, \ldots, n-1$ für die Leistungen p_j gelten - unabhängig vom Vorzeichen der Leistungen. Zielfunktion:

$$\min \sum_{j=0}^{n-1} c_j p_j \Delta t.$$

5.3.2. Bezugskosten und Einspeisevergütung

Wir betrachten der Einfachheit halber nur eine Zeitperiode: Wenn sich die Bezugspreise c_{Bezug} in EUR/kWh und die Einspeisevergütung $c_{\text{Einsp.}}$ in EUR/kWh unterscheiden, dann ist die Kostenfunktion nicht-linear. Falls, wie üblich, $c_{\text{Bezug}} \geq c_{\text{einsp.}}$, dann ist die Kostenfunktion konvex und kann als Maximum zweier linearer Funktionen geschrieben werden:

$$c(E) = \max\{c_{\text{Bezug}} E, c_{\text{Einsp.}} E\}.$$

Dabei ist E die bezogene ($E > 0$) bzw. eingespeiste ($E < 0$) Energie in kWh, siehe Abbildung Abbildung 5.4.

Die **Minimierung** einer solchen Zielfunktion lässt sich in einem LP mit einer zusätzlichen Variablen K umsetzen:

$$\begin{aligned} &\min K \\ &\text{s. t. } c_{\text{Bezug}} E \leq K \\ &\qquad c_{\text{Einsp.}} E \leq K \end{aligned}$$

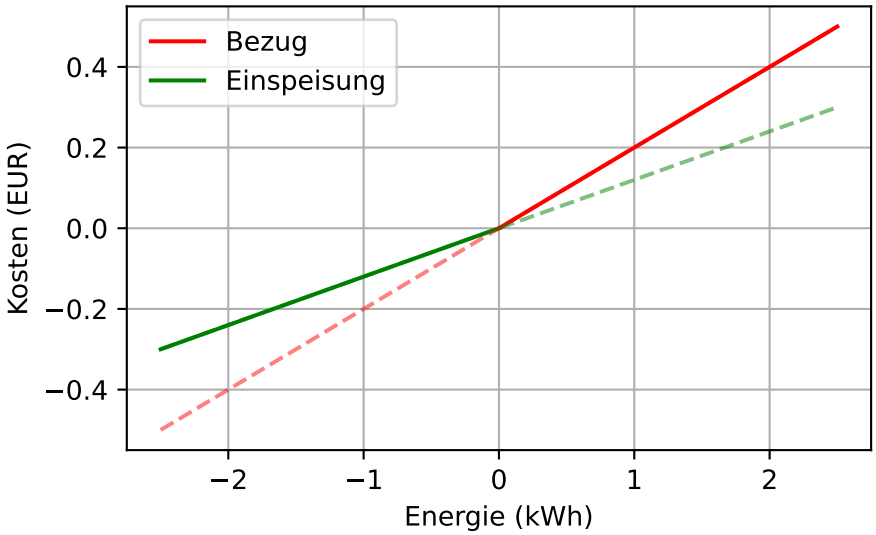


Abbildung 5.4.: konvexe, stückweise lineare Kostenfunktion

5.3.3. Eigenverbrauchsanteil und Autarkiegrad

5.3.3.1. Fixe Produktion und fixer Verbrauch

Wir betrachten den Fall eines fixen (bekannten, unflexiblen) Produktionsverlaufs in kW (z. B. einer PV-Anlage) und eines fixen (bekannten, unflexiblen) Verbrauchsverlaufs in kW (z. B. eines Haushalts) über eine fixierte Anzahl an Zeitperioden. Der Produktionsüberschuss werde ins Netz eingespeist und bei Bedarf aus dem Netz bezogen. Siehe Abbildung Abbildung 5.5.

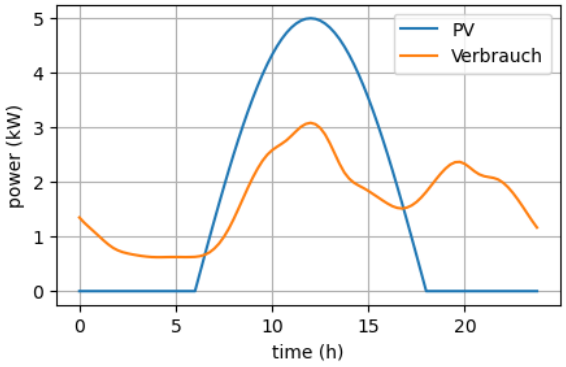


Abbildung 5.5.: Produktionsverlauf und Verbrauchsverlauf

Eigenverbrauch: Die eigenverbrauchte Leistung ist das Minimum aus der Produktionsleistung und der Verbrauchsleistung. Der Eigenverbrauch ist die Summe der eigenverbrauchten Leistungen mal dem Zeitintervall über alle Zeitperioden. Der Eigenverbrauch kann auch berechnet werden als produzierte Energie minus ins Netz eingespeiste Energie.

Eigenverbrauchsanteil: Der Eigenverbrauchsanteil (auch Eigenverbrauchsquote genannte) ist definiert als der Eigenverbrauch dividiert durch die produzierte Energie. Der Eingenverbrauchsanteil gibt an, wie viel der selbst produzierten Energie man selbst verbraucht hat.

Autarkiegrad: Der Autarkiegrad ist definiert als der Eigenverbrauch dividiert durch die verbrauchte Energie. Der Autarkiegrad gibt an, wie viel der gesamt verbrauchten Energie von der eigenen PV-Anlage gedeckt wurde.

Da die Größen produzierte Energie und verbrauchte Energie fixiert sind, ist die Maximierung des Eigenverbrauchsanteils und die Maximierung des Autarkiegrads äquivalent

- zur Maximierung des Eigenverbrauchs und
- zur Minimierung der ins Netz eingespeisten Energie.

5.3.3.2. Fixe Produktion und variabler Verbrauch

Durch den variablen Verbrauch ist auch die Einspeisung variabel. Für diesen Fall definieren wir den Eigenverbrauch als produzierte Energie minus ins Netz eingespeiste Energie.

Die Maximierung des Eingenverbrauchs entspricht daher der Minimierung der ins Netz eingespeisten Energie. Wenn die ins Netz eingespeiste Energie als negativer Anteil der Leistung g_j in kW definiert ist, d. h. als $\max(-g_j, 0)$, dann lautet die Minimierung der ins Netz eingespeisten Energie:

$$\min \sum_{j=0}^{n-1} \max(-g_j, 0)\Delta t.$$

Diese Zielfunktion ist aufgrund des Maximumoperators nicht-linear. Sie lässt sich jedoch in einem LP mit einer zusätzlichen Variablen f_j (in kW, für *feed in*) umsetzen:

$$\begin{aligned} \min & \sum_{j=0}^{n-1} f_j \Delta t \\ \text{s. t. } & -g_j \leq f_j \quad \forall j = 0, 1, 2, \dots, n-1 \\ & 0 \leq f_j \quad \forall j = 0, 1, 2, \dots, n-1 \end{aligned}$$

5.3.4. Spitzenlast

Die Spitzenlast ist die maximale Leistung $\max_{j \in J} p_j$, die während der Zeitperioden J auftritt. Aufgrund des Maximumoperators ist die Zielfunktion nicht-linear von den p_j abhängig. Die Minimierung der Spitzenlast kann allerdings durch die Minimierung einer zusätzlichen Variablen $m > 0$ in Kombination mit den folgenden zusätzlichen Nebenbedingung linear modelliert werden:

$$\begin{aligned} \min & \; m \\ \text{s. t. } & p_j \leq m \quad \forall j \in J. \\ & \dots \end{aligned}$$

Wie erwähnen hier die dazu verwandte Variante der Minimierung der l_∞ (Chebyshev) Norm: Die l_∞ (Chebyshev) Norm eines Vektors $y \in \mathbb{R}^n$ ist definiert als das Maximum der Absolutbeträge seiner Komponenten:

$$||y||_\infty = \max_{i=1,\dots,n} |y_i|$$

Die Minimierung der Zielfunktion $||y||_\infty$ lässt sich in einem LP mit einer zusätzlichen Variablen t und $2n$ zusätzlichen Nebenbedingungen umsetzen:

$$\begin{aligned} \text{min. } & t \\ \text{s. t. } & -t \leq y_i \leq t, i = 1, \dots, n \end{aligned}$$

5.3.5. Variation

Die Variation eines Leistungsverlaufs ist gegeben durch die absoluten Änderungen der Leistungen zwischen den Zeitperioden: $|p_j - p_{j-1}|$ für alle $j = 1, 2, \dots, n-1$.

Die *Minimierung der Variationssumme* $\min \sum_{j=1}^{n-1} |p_j - p_{j-1}|$ lässt sich als LP mit $n-1$ zusätzlichen Variablen $v_j \geq 0$ und $2n-2$ zusätzlichen Nebenbedingungen umsetzen:

$$\begin{aligned} &\text{min.} \sum_{j=1}^{n-1} v_j \\ &\text{s. t. } -v_j \leq p_j - p_{j-1} \leq v_j, j = 1, \dots, n-1 \end{aligned}$$

Die *Minimierung der Variationsspitze* $\min \max_{j=1,\dots,n-1} |p_j - p_{j-1}|$ lässt sich als LP mit einer zusätzlichen Variablen $m \geq 0$ und $2n-2$ zusätzlichen Nebenbedingungen umsetzen:

$$\begin{aligned} &\text{min. } m \\ &\text{s. t. } -m \leq p_j - p_{j-1} \leq m, j = 1, \dots, n-1 \end{aligned}$$

5.3.6. Tracking

Beim Tracking wird ein vorgegebener Leistungsverlauf q_j von einem flexiblen Leistungsverlauf p_j in kW möglichst gut nachgefahren. Der Unterschied zwischen den Leistungen p_j und q_j kann auch unterschiedliche Weisen quantifiziert werden.

5.3.6.1. Summe der absoluten Abweichungen

Die Summe der absoluten Abweichungen von p_j und q_j ist gegeben durch

$$\sum_{j=0}^{n-1} |p_j - q_j|$$

Die Minimierung dieser Zielfunktion lässt sich in einem LP mit n zusätzlichen Variablen a_j und $2n$ zusätzlichen Nebenbedingungen umsetzen:

$$\begin{aligned} &\text{min} \sum_{j=0}^{n-1} a_j \\ &\text{s. t. } -a_j \leq p_j - q_j \leq a_j, j = 0, 1, \dots, n-1 \end{aligned}$$

5.3.6.2. Maximale absolute Abweichung

Die maximale absolute Abweichung von p_j und q_j ist gegeben durch

$$\max_{j=0,1,\dots,n-1} |p_j - q_j|$$

Die Minimierung dieser Zielfunktion lässt sich in einem LP mit einer zusätzlichen Variablen m und $2n$ zusätzlichen Nebenbedingungen umsetzen:

$$\begin{aligned} &\text{min } m \\ &\text{s. t. } -m \leq p_j - q_j \leq m, j = 0, 1, \dots, n-1 \end{aligned}$$

5.3.6.3. Summe der quadratischen Abweichungen

Die Summe der quadratischen Abweichungen von p_j und q_j ist gegeben durch

$$\sum_{j=0}^{n-1}(p_j - q_j)^2$$

Die Minimierung dieser Zielfunktion kann nicht als LP formuliert werden. Diese Problemstellung wird als Regression oder Quadratic Programming bezeichnet, für die es eigene Lösungsmethoden gibt.

5.4. Literaturhinweise

- AIMMS Optimization Modeling: Part II "General Optimization Modeling Tricks"
- Sioshansi, Ramteen; Conejo, Antonio J. (2017): Optimization in Engineering: Models and Algorithms. Springer. Kapitel 3 "Mixed-Integer Linear Optimization" insbesondere Abschnitt 3.3 "Linearizing Nonlinearities Using Binary Variables".
- Williams, H. Paul (2013): Model Building in Mathematical Programming. 5. Auflage, Wiley. Kapitel 8 "Integer programming", Abschnitt 8.2 "The applicability of integer programming"
- Vandenberghe, Lieven: lecture notes on Linear Programming insbesondere "Lecture 2 Piecewise-linear optimization"

6. Stationäre Probleme

Unter stationären Problemen verstehen wir Optimierungsprobleme, die sich auf einen bestimmten Zeitpunkt beziehen oder keine zeitliche Entwicklung der Entscheidungsvariablen beinhalten und eine "Dauerlösung" suchen.

6.1. Transportproblem

6.1.1. Problemstellung

Bei einem Transportproblem (Englisch: transportation problem) geht es darum, eine Ware von gewissen Lieferzentren, auch Quellen genannt, an gewisse Empfangszentren, auch Ziele genannt, so zu verteilen, dass die Gesamtverteilungskosten minimiert werden. Es muss dabei nicht unbedingt um Waren, Liefer- und Empfangszentren gehen. Wichtig ist die Struktur von Quellen und Zielen.

Gegeben sind dabei im Detail:

- die Quellen (Lieferzentren, sources) und ihre Produktionsmengen
- die Ziele (Empfangszentren, targets) und ihre Bedarfsmengen
- die Transportkosten pro Mengeneinheit der Ware von jeder Quelle zu jedem Ziel

Gesucht sind die Warenmengen, die von jeder Quelle zu jedem Ziel transportiert werden sollen, sodass die Gesamtkosten minimal sind.

Das Transportproblem kann als ungerichteter Graph dargestellt werden, in dem die Quellen und Ziele als Knoten modelliert sind und jede Quelle mit jedem Ziel durch eine Kante verbunden ist. Die Kanten haben dabei die Transportkosten pro Mengeneinheit als Gewicht, siehe das Beispiel: Transportproblem.

6.1.2. LP-Formulierung

- Entscheidungsvariablen: Anzahlen x_{ij} der zu verteilenden Warenmengen von Quelle i an Ziel j, $i = 1, 2, \dots, m$, $j = 1, 2, \dots, n$.
- Zielfunktion: $\sum_{i=1}^{m} \sum_{j=1}^{n} c_{ij} x_{ij}$ mit Transportkosten c_{ij} pro Wareneinheit von jeder Quelle i zu jedem Ziel j
- Nebenbedingungen:
  - $\sum_{j=1}^{n} x_{ij} = s_i$ für alle $i = 1, 2, \dots, m$, mit s_i der Produktionsmenge der Quelle i
  - $\sum_{i=1}^{m} x_{ij} = d_j$ für alle $j = 1, 2, \dots, n$, mit d_j der Bedarfsmenge des Ziels j
  - $x_{ij} \geq 0$ für alle i und j

Dabei wird implizit angenommen, dass die **Transportkosten linear in den Warenmengen** x_{ij} sind.

Bedingung für die Existenz zulässiger Punkte: Ein Transportproblem hat genau dann zulässige Punkte, wenn gilt, dass die Summe der Produktionsmengen gleich der Summe der Bedarfsmengen ist: $\sum_{i=1}^{m} s_i = \sum_{j=1}^{n} d_j$.

Ganzzahlige Lösungen: Wenn alle s_i und d_j Werte ganzzahlig sind, dann sind die Ecken des Nebenbedingungspolyeders auch ganzzahlig. Unter diesen Umständen ist der vom Simplex-Algorithmus berechnete optimale Punkt daher ganzzahlig, und es müssen keine zusätzlichen Ganzzahligkeitsnebenbedingungen verwendet werden.

Algorithmik: Transportprobleme sind eine besondere Art von LPs und können daher mit dem Simplex-Algorithmus gelöst werden. Die besondere Struktur von Transportprobleme kann jedoch ausgenutzt werden, um den Simplex-Algorithmus effizienter zu machen. Siehe z. B. Hillier, Lieberman: Introduction to Operations Research. 10th edition, 2015. p. 333 ff. für den die "transportation simplex method" und Wikipedia: Netzwerk-Simplexmethode.

6.1.3. Modellierungstricks

Dummy-Ziel: Wenn die Summe der Produktionsmengen größer ist als die Summe der Bedarfsmengen, kann man ein Dummy-Ziel einführen, das den restlichen Bedarf mit Null Transportkosten übernimmt, siehe Abbildungen Abbildung 6.1 und Abbildung 6.2.

		DESTINATIONS			SUPPLY	
		1	2	3		
SOURCE	A	2	3	1	20	
	B	5	4	8	15	
	C	5	6	8	30	
DEMAND		20	15	25	60	65

Abbildung 6.1.: Dummy Ziel: vorher

		DESTINATIONS				SUPPLY	
		1	2	3	DUMMY DESTINATION		
SOURCE	A	2	3	1	0	20	
	B	5	4	8	0	15	
	C	5	6	8	0	30	
DEMAND		20	15	25	65-60= 5	65	65

Abbildung 6.2.: Dummy Ziel: nachher

Quelle: Link

Dummy-Quelle: Wenn die Summe der Bedarfsmengen größer ist als die Summe der Produktionsmengen, kann man eine Dummy-Quelle einführen, die die restliche Produktion mit Null Transportkosten übernimmt, siehe Abbildungen Abbildung 6.3 und Abbildung 6.4.

		DESTINATIONS			SUPPLY
		1	2	3	
SOURCE	A	2	3	1	20
	B	5	4	8	15
	C	5	6	8	30
DEMAND		20	30	25	75 / 65

Abbildung 6.3.: Dummy Quelle: vorher

		DESTINATIONS			SUPPLY
		1	2	3	
SOURCE	A	2	3	1	20
	B	5	4	8	15
	C	5	6	8	30
	DUMMY SOURCE	0	0	0	75-65= 10
DEMAND		20	30	25	75 / 75

Abbildung 6.4.: Dummy Quelle: nachher

Quelle: Link

6.2. Zuordnungsproblem

6.2.1. Problemstellung

- Es gibt n Tätigkeiten (Aufträge, englisch: tasks, jobs) und n Arbeiter (Auftragsempfänger, englisch: assignees).
- Jedem Arbeiter wird genau eine Tätigkeit zugeordnet.
- Wenn dem Arbeiter i die Tätigkeit j zugeordnet wird, entstehen Kosten c_{ij}.

- Ziel ist es, eine gesamtkostenoptimale Zuordnung zu bestimmen.

Es muss dabei nicht unbedingt um Tätigkeiten und Arbeiter gehen. Wichtig ist die Struktur von Quellen und Zielen. Das Zuordnungsproblem ist ein Spezialfall des Transportproblems und kann ebenso als ungerichteter Graph dargestellt werden.

6.2.2. LP-Formulierung

- Binäre Entscheidungsvariablen: x_{ij} für $i, j = 1, 2, \dots, n$ mit $x_{ij} = 1$, wenn dem Arbeiter i die Tätigkeit j zugeordnet wird, andernfalls 0.
- Zielfunktion: $\sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}$
- Nebenbedingungen
  - $\sum_{j=1}^{n} x_{ij} = 1$ für alle $i = 1, 2, \dots, n$
  - $\sum_{i=1}^{n} x_{ij} = 1$ für alle $j = 1, 2, \dots, n$

Relaxierung: Die Bedingung, dass die x_{ij} binär sein müssen, kann relaxiert, d. h. fallen gelassen, werden. Zuordnungsprobleme können nämlich als Spezialfälle von Transportproblemen mit $s_i = 1$ und $d_j = 1$ interpretiert werden. Alle, und daher auch die optimalen, Ecken sind daher automatisch ganzzahlig. Daher genügt es $x_{ij} \geq 0$ für alle i und j für kontinuierlichen Variablen x_{ij} zu fordern.

Algorithmik: Zuordnungsprobleme können mit der Ungarischen Methode sehr effizient gelöst werden. Siehe dazu z. B. Hillier, Lieberman: Introduction to Operations Research. 10th edition, 2015. p. 356 ff.

6.2.3. Modellierungstricks

Neben Dummy-Größen und der Big-M-Methode sind folgende Modellierungstricks interessant:

- Bestimmte Arbeiter können mehr als einer Aufgabe zugeordnet werden: Jeder dieser Arbeiter wird in separate (aber identische) neue Arbeiter aufgespaltet, wobei jedem neuen Arbeiter genau eine Aufgabe zugewiesen wird.
- Eine Aufgabe kann von mehreren Arbeitern ausgeführt werden: Diese Aufgabe wird in separate (aber identische) neue Aufgaben aufgespaltet, wobei jede neue Aufgabe von genau einem Arbeiter ausgeführt wird.

6.3. Umladeproblem

6.3.1. Problemstellung

Manchmal schließt der Transport einer Ware die Möglichkeit des Umschlags ein, wobei die Ware über zwischengeschaltete Umschlagspunkte transportiert wird. Dabei können Quellen oder Ziele auch Umschlagspunkte sein. Diese Erweiterung des Transportproblems um die Routing-Entscheidungen wird als Umladeproblem (engl. transshipment problem) bezeichnet. Es ist ein Spezialfall des Minimum-Cost-Flow-Problems, das wiederum ein Spezialfall eines Network-Flow-Problems ist. Umladeprobleme können aber auch in Transportprobleme umformuliert werden.

6.3.2. LP-Formulierung

Wir verwenden die Formulierung als Network Flow Problem: Die Netzwerk-Darstellung des Problems besteht dabei aus einem **gerichteten Graphen** mit

- **Knoten (nodes):** alle Quellen (sources), alle Umschlagspunkte (junctions), alle Ziele (targets)
- **Kanten (edges):** alle erlaubten gerichteten Verbindungen zwischen den Knoten.

Voraussetzung für die Lösbarkeit des Problems ist, dass die Summe der Produktionsmengen aller Quellen gleich der Summe der Bedarfsmengen aller Ziele ist.

- **Entscheidungsvariablen:** Warenmenge $x_e \geq 0$ auf jeder Kante e
- **Zielfunktion:** Summe der Kosten $\sum_{e \in \text{edges}} c_e x_e$
- **Nebenbedingungen:** Flussbilanz für jeden Knoten
  - $\sum_{\substack{\text{out-edges} \\ o \text{ of } s}} x_o - \sum_{\substack{\text{in-edges} \\ i \text{ of } s}} x_i = \text{production}(s)$ for all sources s
  - $\sum_{\substack{\text{out-edges} \\ o \text{ of } j}} x_o = \sum_{\substack{\text{in-edges} \\ i \text{ of } j}} x_i$ for all junctions j
  - $\sum_{\substack{\text{in-edges} \\ i \text{ of } t}} x_i - \sum_{\substack{\text{out-edges} \\ o \text{ of } t}} x_o = \text{demand}(t)$ for all targets t

6.4. Problemklassen

Unter dem Begriff **Network Flow Problem** werden verschiedene Problemklassen zusammengefasst, siehe z. B. Wikipedia: network flow problem. Manche Autoren verwenden den Begriff Network Flow Problem äquivalent mit **Minimum-Cost Flow Problem**. Jedes Network Flow Problem kann als LP modelliert werden. In der Graphik Abbildung 6.5 sind die Problemklassen und ihre Unterklassenstruktur dargestellt.

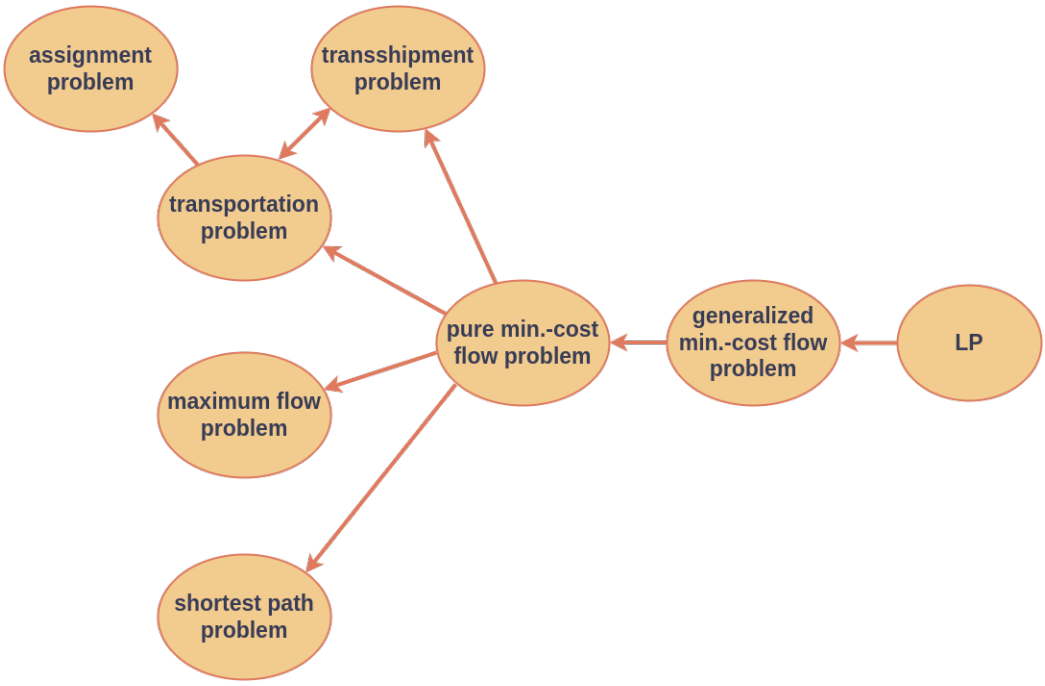


Abbildung 6.5.: Problemklassen

Umformulierungen: Die Unterklassen können als Minimum-Cost Flow Probleme formuliert werden, was den Begriff Unterklasse rechtfertigt. Siehe die Literaturhinweise.

6.5. Minimum-Cost Flow Problem

Wir betrachten ein Netzwerk mit Knotenmenge N (set of **n**odes) und Kantenmenge E (set of **e**dges). Das **Minimum-Cost Flow Problem** (kostenminimaler Fluss) lautet: "Bestimme einen Fluss durch die Kanten, der den Einschränkungen genügt und dessen Gesamtkosten minimal sind." Das zugehörige LP besteht aus:

- **Entscheidungsvariablen:** Fluss x_e durch Kante $e \in E$
- **Zielfunktion:** Minimiere die Summe der Kosten $\sum_{e \in E} c_e x_e$ mit c_e den Kosten pro Wareneinheit entlang der Kante e
- **Nebenbedingungen:**
  - **Flussbilanz für jeden Knoten:** Summe der Flüsse aus einem Knoten n in andere Knoten weniger Summe der Flüsse von anderen Knoten in den Knoten n ist gleich externer Fluss b_n des Knoten, d. h. $\sum_{\substack{\text{out-edges} \\ o \text{ of } n}} x_o - \sum_{\substack{\text{in-edges} \\ i \text{ of } n}} x_i = b_n$ für alle Knoten $n \in N$.
  - **Kapazitätsbeschränkungen für jede Kante:** Jeder Fluss kann nur Werte zwischen seiner unteren und oberen Schranke annehmen, d. h. $l_e \leq x_e \leq u_e$ für alle Kanten $e \in E$.

Existenz zulässiger Punkte: Eine notwendige Bedingung dafür, dass es einen zulässigen Punkt gibt, ist, dass $\sum_{n \in N} b_n = 0$, d. h. der gesamte externe Fluss in das Netzwerk ist gleich dem gesamten externen Fluss aus dem Netzwerk.

Ganzzahlige Lösungen: Wenn alle b_n, l_n und u_n ganzzahlig sind, dann ist der vom Simplex-Algorithmus gefundene optimale Punkt auch ganzzahlig, und es müssen keine zusätzlichen Ganzzahligkeitsnebenbedingungen verwendet werden.

Algorithmik: Für Minimum-Cost Flow Probleme gibt es den angepassten und dadurch effizienteren (als der Standard-Simplex-Algorithmus) LP-Algorithmus Network Simplex Algorithm.

6.6. Maximaler Fluss

6.6.1. Problemstellung

Ein Maximum Flow Problem kann allgemein wie folgt beschrieben werden:

1. Der gesamte Fluss durch ein gerichtetes und zusammenhängendes Netzwerk hat seinen Ursprung an einem Knoten, der **Quelle** (engl. source) genannt wird, und endet an einem anderen Knoten, der **Senke** (engl. sink, target) genannt wird.
2. Alle übrigen Knoten sind Umladeknoten.
3. Der Fluss durch die **gerichteten Kanten** ist nur in die angegebenen Richtungen erlaubt, wobei die Flussmenge einer Kante durch die **Kapazität** dieser Kante beschränkt ist. An der Quelle zeigen alle Kanten vom Knoten weg. An der Senke zeigen alle Kanten in den Knoten.
4. Das Ziel ist es, die **Gesamtmenge des Flusses** von der Quelle zur Senke zu **maximieren**. Diese Größe wird auf eine von zwei äquivalenten Arten gemessen, nämlich entweder die Menge, die die Quelle verlässt oder die Menge, die in die Senke eintritt.

In der Abbildung Abbildung 6.6 ist der Knoten s die Quelle und der Knoten t die Senke.

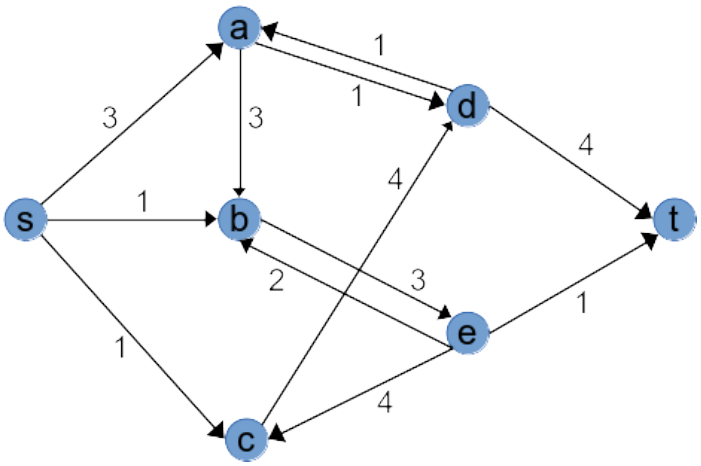


Abbildung 6.6.: Beispiel eines Maximum Flow Problems

6.6.2. LP-Formulierung

- **Entscheidungsvariablen:** Fluss x_e durch Kante $e \in E$
- **Zielfunktion:** Maximiere
  - die Summe der Flüsse $\sum_{\substack{\text{out-edges} \\ o \text{ of } s}} x_o$ aus der Quelle s oder
  - die Summer der Flüsse $\sum_{\substack{\text{in-edges} \\ i \text{ of } t}} x_i$ in die Senke t
- **Nebenbedingungen:**
  - Flussbilanz für jeden Knoten n außer Quelle und Senke: $\sum_{\substack{\text{out-edges} \\ o \text{ of } n}} x_o = \sum_{\substack{\text{in-edges} \\ i \text{ of } n}} x_i$
  - Kapazitätsbeschränkungen für jede Kante: $0 \leq x_e \leq u_e$

6.7. Kürzester Weg

6.7.1. Problemstellung

Ein Graph heißt **bewertet**, wenn jeder Kante ein Gewicht zugeordnet ist. In bewerteten Graphen ist die **Länge eines Weges** die Summe der Gewichte aller Kanten des Weges. Wir nehmen an, dass alle Kanten-Bewertungen positiv sind, sodass Wege von Knoten zu Knoten immer länger werden.

6.7.2. Dijkstra-Algorithmus

Kürzeste Wege in solchen bewerteten, ungerichteten Graphen können mit dem Algorithmus von Dijkstra gefunden werden, den dieser 1959 formuliert hat. Im Buch Hartmann: “Mathematik für Informatiker: Ein praxisbezogenes Lehrbuch”, Springer, 2019 ist der Algorithmus auf den Seiten 290 ff. ausführlicher beschrieben und begründet. Der Dijkstra-Algorithmus ist kein LP kann auch für gerichtete Graphen formuliert werden, siehe z. B. Wikipedia: Dijkstra's_algorithm oder GeeksforGeeks: Shortest path in a directed graph by Dijkstra's algorithm.

6.7.3. LP-Formulierung

Formulierung als Minimum-Cost Flow Problem: Wir betrachten das Shortest Path Problem in einem gerichteten Graphen, das alle kürzesten Wege von einem bestimmten Konten S zu allen anderen Knoten sucht. Ungerichtete Graphen können als gerichtete interpretiert werden, indem jede ungerichtete Kante durch zwei entgegengesetzte gerichtete Kanten ersetzt wird. Jede dieser gerichteten Kanten erhält dabei das Gewicht der ungerichteten Kante.

- Die Gewichte der Kanten werden als Kosten pro Flussmengeneinheit betrachtet.
- Dem Knoten S wird der externe Fluss $n-1$ zugewiesen mit n der Anzahl Knoten im Graphen.
- Jedem anderen Knoten wird der externe Fluss -1 zugewiesen.
- Die Kapazität jeder Kante wird unbeschränkt oder mindestens auf $n-1$ gesetzt.

Die optimalen Flüsse des damit definierten Minimum-Cost Flow Problems definieren die kürzesten Wege von S zu allen anderen Knoten. Die Summe der Gewichte entlang dieser Wege sind die zugehörigen kürzesten Distanzen.

6.8. Literaturhinweise

- Jensen, Bard: Operations Research Models and Methods. John Wiley & Sons, 2008. Seite 167
- Hillier, Lieberman: Introduction to Operations Research. 10th edition, 2015. ab Seite 397
- Sierksma, Zwols: Linear and Integer Optimization: Theory and Practice. 3rd edition, 2015. Seite 351

Teil III.

Beispiele

7. (Re-)Sampling

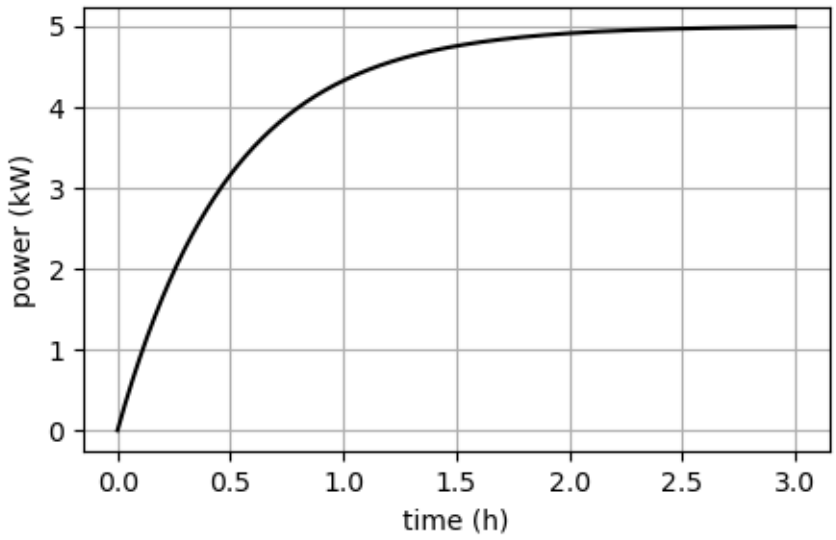
7.1. Kontinuierlicher Lastgang

Wir betrachten den Verlauf einer elektrischen Last, d. h., den Verlauf der Leistungsaufnahme $p(t)$ in kW eines elektrischen Verbrauchers über die Zeit t in h:

$$p(t) = 5(1 - e^{-2t})$$

Als erstes plotten wir diesen Lastgang:

```
import numpy as np
import matplotlib.pyplot as plt

t = np.linspace(0, 3, num=1000)   # (almost continuous) time in h
p = 5*(1 - np.exp(-2*t))          # power in kW

plt.figure(figsize=(5, 3))
plt.plot(t, p, 'k-')
plt.xlabel('time (h)')
plt.ylabel('power (kW)')
plt.grid(True)
```



7.2. Sampling mit Energieerhaltung

Gemessene Lastgänge werden z. B. alle $\Delta t = 15$ Minuten mit einem zugehörigen Wert aufgezeichnet. Dabei werden die kontinuierlichen Werte derart in konstante Werte für jedes Zeitintervall umgewandelt, dass die

in den Intervallen enthaltene Energie gleich bleibt. Wir bestimmen daher für unseren Beispiellastgang die mittlere Leistung $\bar{p}$ zwischen zwei Zeitpunkten t_1 und t_2, d. h. $\Delta t = t_2 - t_1$:

$$\begin{aligned}\bar{p} &= \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} p(t)\,\mathrm{d}t \\ &= \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} 5(1 - e^{-2t})\,\mathrm{d}t \\ &= \frac{5}{t_2 - t_1} \left[t_2 - t_1 + \frac{1}{2}(e^{-2t_2} - e^{-2t_1}) \right]\end{aligned}$$

Das Integral $E = \int_{t_1}^{t_2} p(t)\,\mathrm{d}t$ ist gleich der vom Verbraucher aufgenommenen Energie E im Zeitintervall $[t_1, t_2]$. Die so berechnete mittlere Leistung $\bar{p}$ führt zum selben Energieverbrauch E wie der kontinuierliche Lastgang $p(t)$, denn

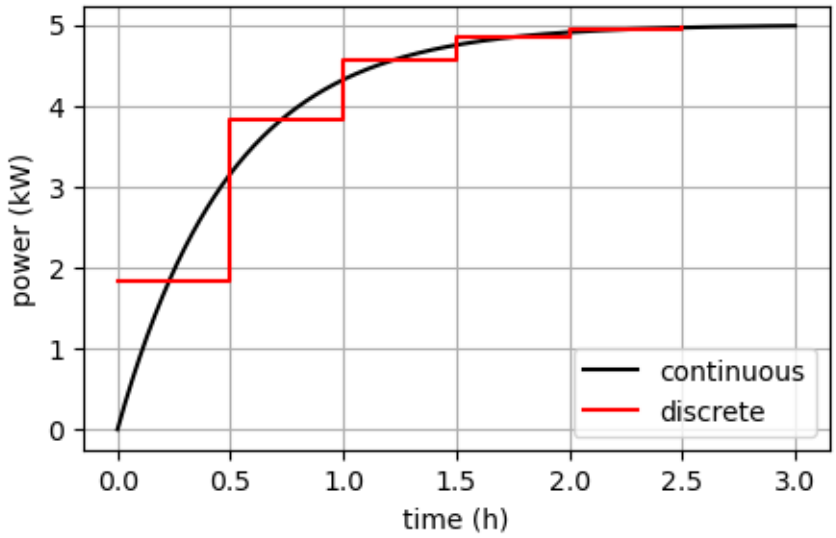
$$\bar{p}(t_2 - t_1) = E.$$

Bei der Optimierung von Lasten in einem Energienetzwerk werden typischer Weise keine kontinuierlichen Lastgänge, sondern stückweise konstanten Lastgänge optimiert.

Im folgenden Beispiel verwenden wir eine Abtastzeit von $\Delta t = 0.5$ Stunden:

```
dt = 0.5  # sampling time in hours
t_discrete = np.arange(start=0, stop=3, step=dt)  # discrete times in hours
print(f"{t_discrete=}")

p_discrete = []
for t_value in t_discrete:
    t_1 = t_value
    t_2 = t_value + dt
    p_mean = 5/dt * (dt + 1/2 *(np.exp(-2*t_2) - np.exp(-2*t_1)))
    p_discrete.append(p_mean)

# t_discrete = list(t_discrete) + [3]
# p_discrete = p_discrete + [p[-1]]

plt.figure(figsize=(5, 3))
plt.plot(t, p, 'k-', label='continuous')
plt.step(t_discrete, p_discrete, where='post', color='r', label='discrete')
plt.xlabel('time (h)')
plt.ylabel('power (kW)')
plt.legend()
plt.grid(True)
```

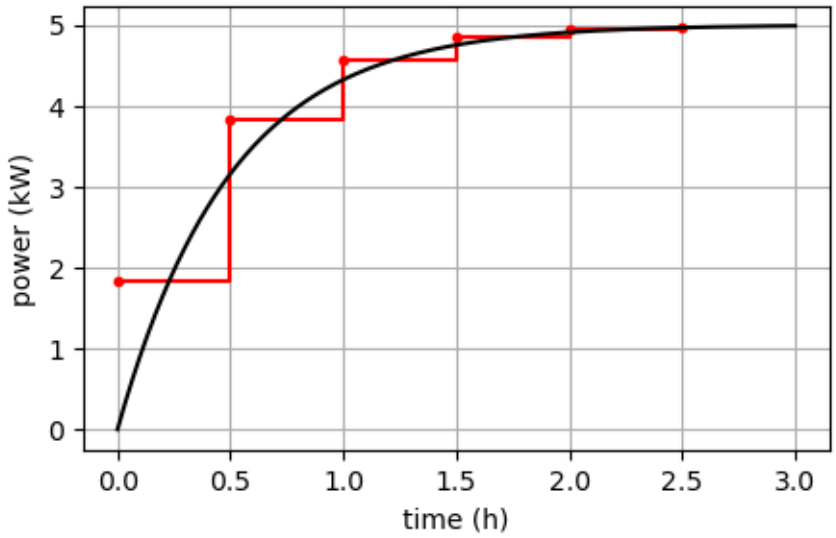
```
t_discrete=array([0. , 0.5, 1. , 1.5, 2. , 2.5])
```



Das selbe mit einem DataFrame: Zum Arbeiten mit Zeitreihen eignen sich in Python die pandas DataFrames sehr gut. Wir erstellen ein DataFrame mit den Zeitpunkten und den konstanten Leistungswerten als Spalten.

```
import pandas as pd

df = pd.DataFrame({'time': t_discrete, 'power':p_discrete})
display(df)

# plot:
ax = df.plot(x='time', y='power', marker ='.',
             drawstyle='steps-post', figsize=(5, 3), legend=False,
             color='r', xlabel='time (h)', ylabel='power (kW)')
ax.plot(t, p, 'k-')
ax.grid(True)
```

	time	power
0	0.0	1.839397
1	0.5	3.837279
2	1.0	4.572259
3	1.5	4.842643
4	2.0	4.942112
5	2.5	4.978704




```
# We check, if the total energies over the complete time interval [0, 3] are equal:
E_continuous = 5*(3 + 1/2 *(np.exp(-2*3) - np.exp(-2*0)))
E_discrete = np.sum(p_discrete)*dt

print(f"{E_continuous = } kWh")
print(f"{E_discrete   = } kWh")
```

```
E_continuous = 12.506196880441667 kWh
E_discrete   = 12.506196880441665 kWh
```

7.3. Resampling

Das Ändern des Sampling-Intervalls wird als Resampling bezeichnet und wird am einfachsten mit dem DateFrame-Methode `resample` durchgeführt. Dazu muss das DataFrame einen Index haben, der die Zeitpunkte als `datetime`-Objekte enthält.

```
datetime_vector = pd.Timestamp('2023-12-02 14:00:00') + pd.to_timedelta(t_discrete, unit='h')
print(datetime_vector)
```

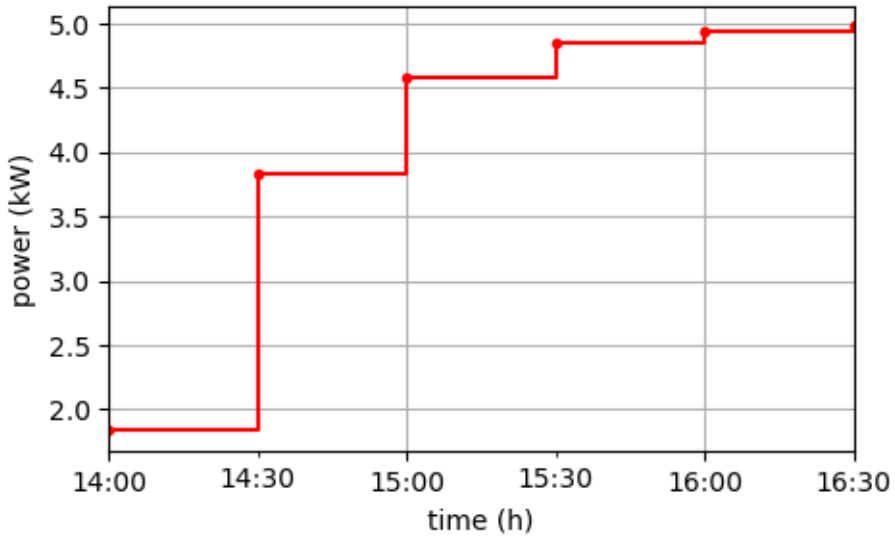
```
DatetimeIndex(['2023-12-02 14:00:00', '2023-12-02 14:30:00',
               '2023-12-02 15:00:00', '2023-12-02 15:30:00',
               '2023-12-02 16:00:00', '2023-12-02 16:30:00'],
              dtype='datetime64[ns]', freq=None)
```

```
df_30Min = pd.DataFrame(p_discrete, index=datetime_vector, columns=['power'])
df_30Min
```

	power
2023-12-02 14:00:00	1.839397
2023-12-02 14:30:00	3.837279
2023-12-02 15:00:00	4.572259
2023-12-02 15:30:00	4.842643

	power
2023-12-02 16:00:00	4.942112
2023-12-02 16:30:00	4.978704

```
df_30Min.plot(figsize=(5, 3), drawstyle='steps-post', grid=True, marker ='.',
              legend=False, color='r', xlabel='time (h)', ylabel='power (kW)');
```




```
# check if energy is conserved:
df_30Min.sum()*dt
```

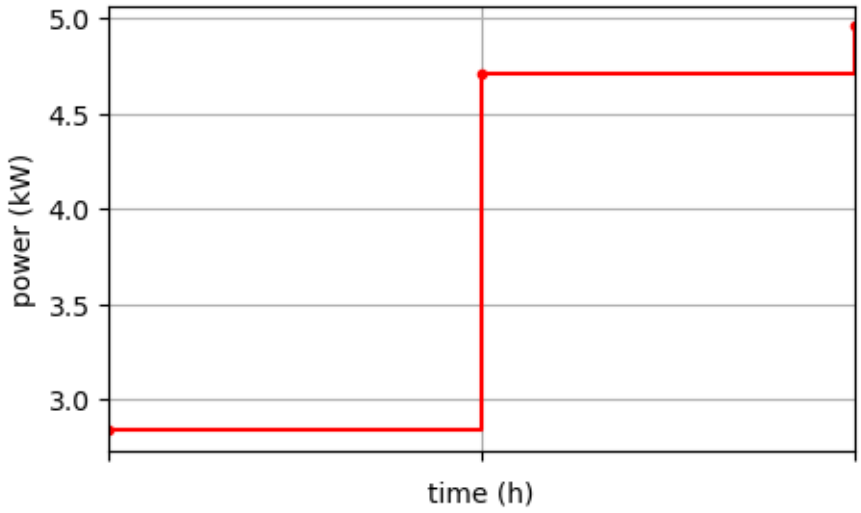
```
power    12.506197
dtype: float64
```

7.3.1. Downsampling

```
# downsampling from 30 min to 1 h
df_1H = df_30Min.resample(rule='1H').mean()  # mean to keep the energy constant!
df_1H
```

	power
2023-12-02 14:00:00	2.838338
2023-12-02 15:00:00	4.707451
2023-12-02 16:00:00	4.960408

```
df_1H.plot(figsize=(5, 3), drawstyle='steps-post', grid=True, marker ='.',
           legend=False, color='r', xlabel='time (h)', ylabel='power (kW)');
```




```
# check if energy is conserved:
dt = 1  # h
df_1H.sum()*dt
```

```
power    12.506197
dtype: float64
```

7.3.2. Upsampling

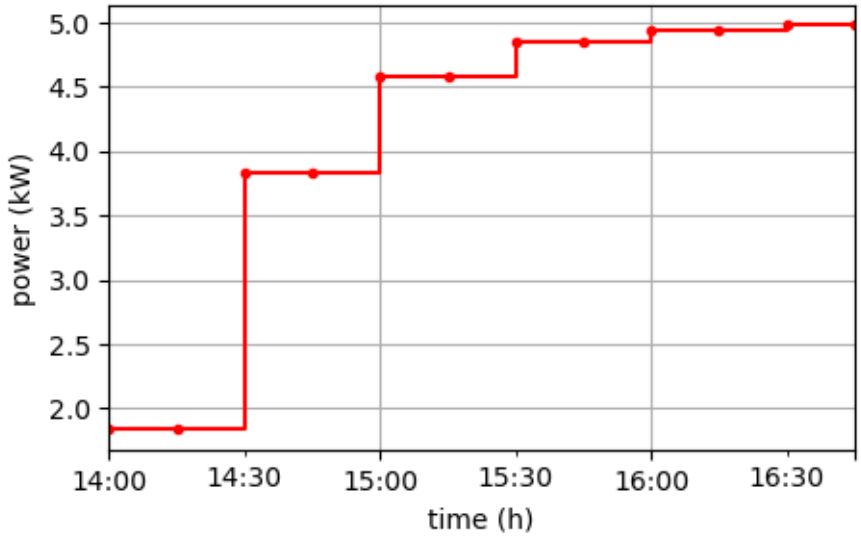
```
# upsampling from 30 min to 15 min
df_15Min = df_30Min.resample(rule='15Min').ffill()
df_15Min
```

	power
2023-12-02 14:00:00	1.839397
2023-12-02 14:15:00	1.839397
2023-12-02 14:30:00	3.837279
2023-12-02 14:45:00	3.837279
2023-12-02 15:00:00	4.572259
2023-12-02 15:15:00	4.572259
2023-12-02 15:30:00	4.842643
2023-12-02 15:45:00	4.842643
2023-12-02 16:00:00	4.942112
2023-12-02 16:15:00	4.942112
2023-12-02 16:30:00	4.978704

```
# Add a value at the end of the time series!
df_15Min.loc['2023-12-02 16:45:00', 'power'] = 4.978704
df_15Min
```

	power
2023-12-02 14:00:00	1.839397
2023-12-02 14:15:00	1.839397
2023-12-02 14:30:00	3.837279
2023-12-02 14:45:00	3.837279
2023-12-02 15:00:00	4.572259
2023-12-02 15:15:00	4.572259
2023-12-02 15:30:00	4.842643
2023-12-02 15:45:00	4.842643
2023-12-02 16:00:00	4.942112
2023-12-02 16:15:00	4.942112
2023-12-02 16:30:00	4.978704
2023-12-02 16:45:00	4.978704

```
df_15Min.plot(figsize=(5, 3), drawstyle='steps-post', grid=True, marker ='.',
              legend=False, color='r', xlabel='time (h)', ylabel='power (kW)');
```




```
# check if energy is conserved:
dt = 0.25  # h
df_15Min.sum()*dt
```

```
power    12.506197
dtype: float64
```

7.4. Übung

Der Lastgang des Verbrauchers A ist in 15-Minuten-Schritten durch die folgenden kW-Werte gegeben: 30, 12, 10, 45, 50, 20, 40, 10. Der Lastgang des Verbrauchers B ist in 30-Minuten-Schritten durch die folgenden kW-Werte gegeben: 0, 70, 0, 90.

1. Erstellen Sie ein DataFrame, das den Gesamtlastgang der beiden Verbraucher in 15-Minuten-Schritten enthält.
2. Wie groß ist die maximale, absolute Leistungsänderung des Gesamtlastgangs?

Hinweis: Verwenden Sie die pandas-Funktion `pd.date_range` um einen DataFrame-Index mit gewünschten Zeitpunkten zu erstellen.

8. Ladeflexibilität

8.1. Laden eines E-Mobils

8.1.1. Problemstellung

Sie kommen mittags um 12 Uhr mit ihrem E-Mobil nach Hause und haben nun 8 Stunden Zeit, es wieder voll zu laden, da Sie abends um 20 Uhr eine längere Reise beginnen. Ihr E-Mobil hat eine Batterie mit einer Kapazität von 60 kWh, der aktuelle SoC (State of Charge) beträgt 40 %. Ihre Ladestation hat eine maximale Ladeleistung von 11 kW. Ihr Energieversorger bietet Ihnen einen zeitabhängigen Stromtarif mit den folgenden, stündlichen Energiepreisen in Cent/kWh für die kommenden acht Stunden an: 10, 8, 12, 13, 11, 10, 13, 14.

Fragen: Mit welchen, stündlich konstanten Ladeleistungen p_j sollen Sie Ihr E-Mobil laden, so dass Sie die Ladekosten minimieren? Wie hoch sind die minimalen Ladekosten? Wann ist Ihr E-Mobil voll geladen?

Vorgehen:

1. Definieren Sie die Zeitpunkte, Zeitperioden und ihre Indizierung.
2. Erstellen Sie einen Plot der Energiepreise über die acht Stunden.
3. Modellieren Sie das Energienetzwerk und fixieren Sie die Pfeilrichtung der Kanten.
4. Basierend auf dieser Vorzeichenkonvention formulieren Sie das Optimierungsproblem:
    1. Definieren Sie die Entscheidungsvariablen inklusive Einheiten, Datentyp und Schranken.
    2. Formulieren Sie die Zielfunktion.
    3. Formulieren Sie die die Nebenbedingungen.
5. Modellieren Sie das LP mit Pyomo. Warum ist es sinnvoll, die Entscheidungsvariablen mit Indizes $0, 1, 2, ...$ anstatt mit Zeitpunkten t oder Ähnlichem zu indizieren?
6. Lösen Sie das LP mit verschiedenen Solvern.
7. Wieviel Kosten sparen Sie sich mit den zeitabhängigen Energiepreisen im Vergleich zum zugehörigen mittleren Energiepreis.
8. Stellen Sie die Ergebnisse grafisch dar, beantworten Sie die Fragen, und interpretieren Sie die Ergebnisse.

8.1.2. Zeiten und Daten

Zeitpunkte, Zeitperioden und ihre Indizierung:

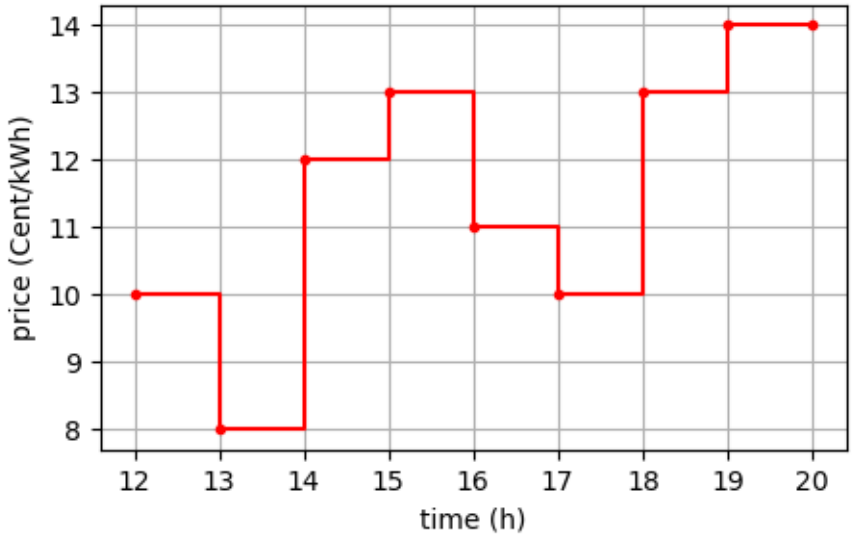
- Samplingintervall $\Delta t = 1$ h
- Zeitpunkte $t = 12, 13, \dots, 20$ Uhr
- Indizes der Zeitpunkte $i = 0, 1, 2, \dots, 8$
- Indizes der Zeitperioden $j = 0, 1, 2, \dots, 7$

Daten:

- Preise c_j in Cent/kWh für die Zeitperioden $j = 0, 1, 2, \dots, 7$

```
import numpy as np
import matplotlib.pyplot as plt
import pyomo.environ as pyo
```

```
dt = 1  # h
times = np.arange(start=12, stop=20 + 1, step=dt)  # h, 12, 13, ..., 19, 20
time_indices = range(len(times))
period_indices = range(len(times) - 1)

prices  = np.array([10, 8, 12, 13, 11, 10, 13, 14])
prices_ = np.array([10, 8, 12, 13, 11, 10, 13, 14, 14])  # append last price for step-plotting

plt.figure(figsize=(5, 3))
plt.step(times, prices_, where='post', color='red', marker='.')
plt.xlabel('time (h)')
plt.ylabel('price (Cent/kWh)')
plt.grid()
```



8.1.3. Energienetzwerk

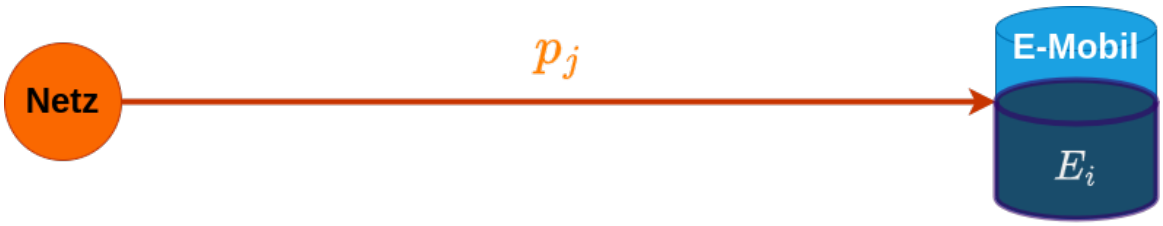


Abbildung 8.1.: Einfachstes Netzwerk

8.1.4. Modellierung

Entscheidungsvariablen:

- p_j Ladeleistung in kW während der Zeitperioden mit Indizes $j = 0, 1, 2, \ldots, 7$. Schranken: $0 \leq p_j \leq 11$ kW.

- E_i Energie der Batterie in kWh zu den Zeitpunkten mit Indizes $i = 0, 1, 2, \ldots, 8$. Schranken: $0 \leq E_i \leq 60$ kWh.

Zielfunktion: Minimiere die Ladekosten in Cent:

$$\min \sum_{j=0}^{7} c_j p_j \Delta t$$

Nebenbedingungen:

- Die Energie der Batterie zu Beginn: $E_0 = 0,40 \cdot 60 = 24$ kWh.
- Die Energie der Batterie am Ende: $E_8 = 60$ kWh.
- Die Energie der Batterie am nächsten Zeitpunkt ergibt sich aus der Energie der Batterie zum aktuellen Zeitpunkt und der Ladeleistung in der dazwischenliegenden Zeitperiode mal der Periodendauer:

  $$E_{i+1} = E_i + p_i \Delta t \quad \forall i = 0, 1, 2, \ldots, 7$$

  Dabei wird angenommen, dass das Laden verlustfrei erfolgt.

8.1.5. Implementierung

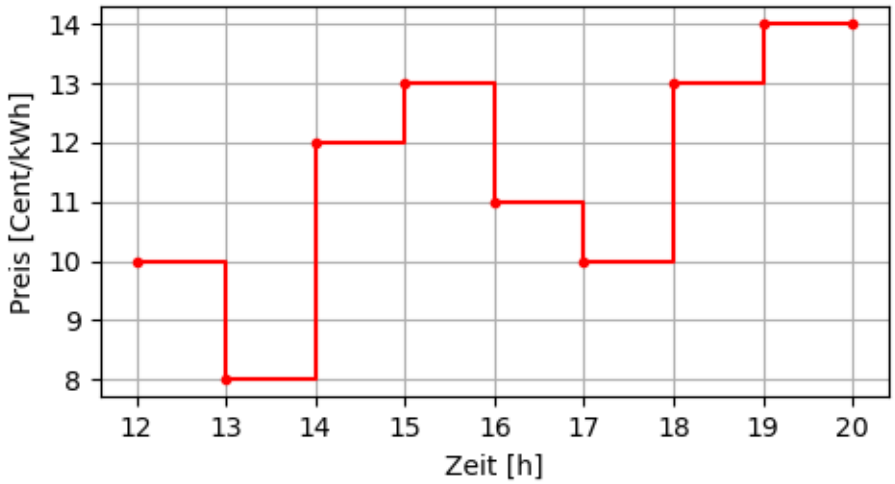
```
model = pyo.ConcreteModel()

model.I = pyo.Set(initialize=time_indices)
model.J = pyo.Set(initialize=period_indices)

model.E = pyo.Var(model.I, bounds=(0.0, 60.0))
model.p = pyo.Var(model.J, bounds=(0.0, 11.0))

model.cost = pyo.Objective(expr=sum(prices[j]*model.p[j]*dt for j in model.J),
                           sense=pyo.minimize)

model.initial_energy = pyo.Constraint(expr = model.E[0] == 24.0)
model.final_energy = pyo.Constraint(expr = model.E[8] == 60.0)

@model.Constraint(model.I)
def charging(model, i):
    if i < 8:
        return model.E[i + 1] == model.E[i] + model.p[i]*dt
    else:
        return pyo.Constraint.Skip

# model.pprint()
```

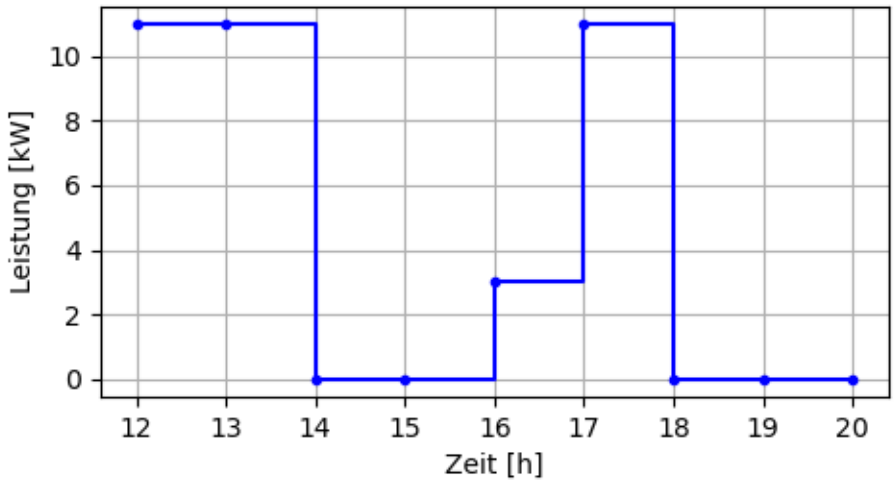
```
solver = pyo.SolverFactory('cbc')
# solver = pyo.SolverFactory('glpk')
# solver = pyo.SolverFactory('appsi_highs')
# solver = pyo.SolverFactory('gurobi')

results = solver.solve(model, tee=False)
print(f"status = {results.solver.status}")

print(f"minimal cost = {pyo.value(model.cost)/100.0:.2f} EUR")
```

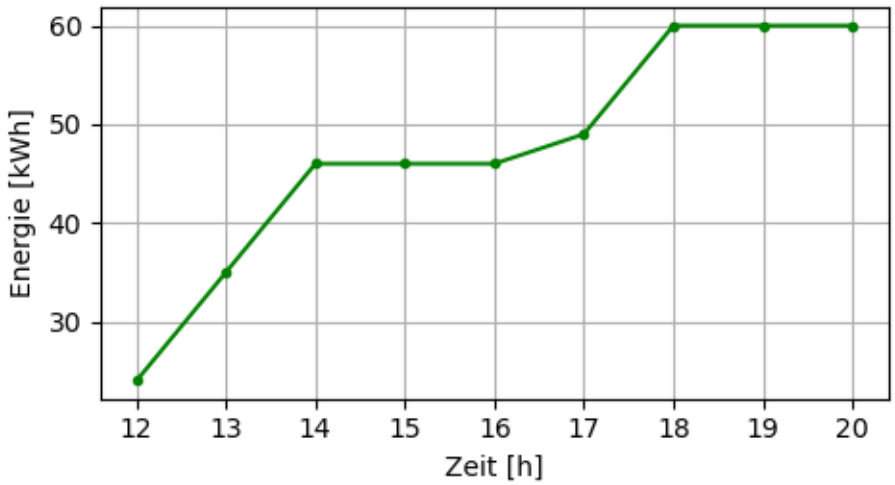
```
status = ok
minimal cost = 3.41 EUR
```

8.1.6. Ergebnisse

```
E_sol_dict = model.E.extract_values()
E_sol = [E_sol_dict[i] for i in time_indices]

p_sol_dict = model.p.extract_values()
p_sol = [p_sol_dict[j] for j in period_indices]
p_sol_ = np.concatenate( (p_sol, [p_sol[-1]]) )

plt.figure(figsize=(5, 8))
plt.subplot(3, 1, 1)
plt.step(times, prices_, where='post', marker='.', color='red')
plt.xlabel('Zeit [h]')
plt.ylabel('Preis [Cent/kWh]')
plt.grid(True)

plt.subplot(3, 1, 2)
plt.step(times, p_sol_, where='post', marker='.', color='blue')
plt.xlabel('Zeit [h]')
plt.ylabel('Leistung [kW]')
plt.grid(True)

plt.subplot(3, 1, 3)
plt.plot(times, E_sol, marker='.', color='green')
plt.xlabel('Zeit [h]')
plt.ylabel('Energie [kWh]')
plt.grid(True)

plt.tight_layout()
```







Das E-Mobil ist bereits um 18 Uhr vollgeladen.

```
price_mean = prices.mean()  # Cent/kWh
print(f"mean price = {price_mean:.2f} Cent/kWh")
```

```
mean price = 11.38 Cent/kWh
```

```python
cost_mean_price = price_mean*(60 - 24)/100  # EUR
cost_prices = pyo.value(model.cost)/100.0   # EUR

print(f"cost with varying prices = {cost_prices:.2f} EUR")
print(f"cost with constant price = {cost_mean_price:.2f} EUR")

print(f"absolute saving = {cost_mean_price - cost_prices:.2f} EUR")
print(f"relative saving = {(cost_mean_price - cost_prices)/cost_mean_price*100:.2f} %")
```

```
cost with varying prices = 3.41 EUR
cost with constant price = 4.09 EUR
absolute saving = 0.68 EUR
relative saving = 16.73 %
```

8.1.7. Interpretation

Das Ladeproblem kann auch mit dem "Valley/Water Filling" Algorithmus gelöst werden. Dabei werden zuerst die Stunden mit den niedrigsten Preisen in maximalem Umfang zum Laden verwendet.

8.2. Übung: Laden inkl. fixen Lasten

8.2.1. Problemstellung

1. Fügen Sie der Problemstellung folgende Features hinzu, und gehen Sie wieder in denselben Schritten (Modellieren des Energienetzwerks, Formulierung und Implementierung des LP) vor:
   - Ihr Haushalt hat zusätzlich zum Laden des E-Mobils einen Verbrauch (engl. demand), der durch folgenden Lastgang in kW gegeben ist: 3, 4, 2, 1, 5, 6, 5, 5.
   - Ihre PV-Anlage hat folgende stündliche Erzeugung in kW: 5, 5, 4, 3, 2, 1, 1, 0. Sie können überschüssige Energie in das Netz einspeisen.
2. Vergleichen Sie in der um demand und PV erweiterten Problemstellung die minimalen Kosten bzgl. der zeitabhängigen Energiepreise mit den Kosten bzgl. des mittleren Energiepreises.

9. Verlustfreie Batterie

9.1. Stromkosten minimieren

9.1.1. Problemstellung

Sie kaufen sich einen stationären Batteriespeicher für Ihr Haus und haben eine PV-Anlage auf dem Dach, die Sie mit dem Speicher verbinden. Sie möchten den Speicher so nutzen, dass Sie Ihre Stromkosten minimieren.

1. Definieren Sie die Zeitpunkte, Zeitperioden und ihre Indizierung.
2. Erstellen Sie einen Plot der PV-Erzeugung und des Stromverbrauchs über den Tag.
3. Modellieren Sie das Energienetzwerk und fixieren Sie die Pfeilrichtung der Kanten.
4. Basierend auf dieser Vorzeichenkonvention formulieren Sie das Optimierungsproblem:
    1. Definieren Sie die Entscheidungsvariablen inklusive Einheiten, Datentyp und Schranken.
    2. Formulieren Sie die Zielfunktion.
    3. Formulieren Sie die die Nebenbedingungen.
5. Modellieren Sie das LP mit Pyomo. Warum ist es sinnvoll, die Entscheidungsvariablen mit Indizes $0, 1, 2, ...$ anstatt mit Zeitpunkten t oder Ähnlichem zu indizieren?
6. Lösen Sie das LP mit verschiedenen Solvern, und vergleichen Sie die Lösungen.
7. Bestimmen Sie für jede Lösung den Autarkiegrad und den Eigenverbrauchsanteil, und vergleichen Sie diese zum Situation ohne Batteriespeicher.

9.1.2. Daten

Batteriespeicher:

- Kapazität: 10 kWh
- maximale Lade- und Entladeleistung: 5 kW
- keine Selbstentladung. keine Lade- oder Entladeverluste
- Anfangsladung: 5 kWh
- Endladung: 5 kWh

Stromtarif:

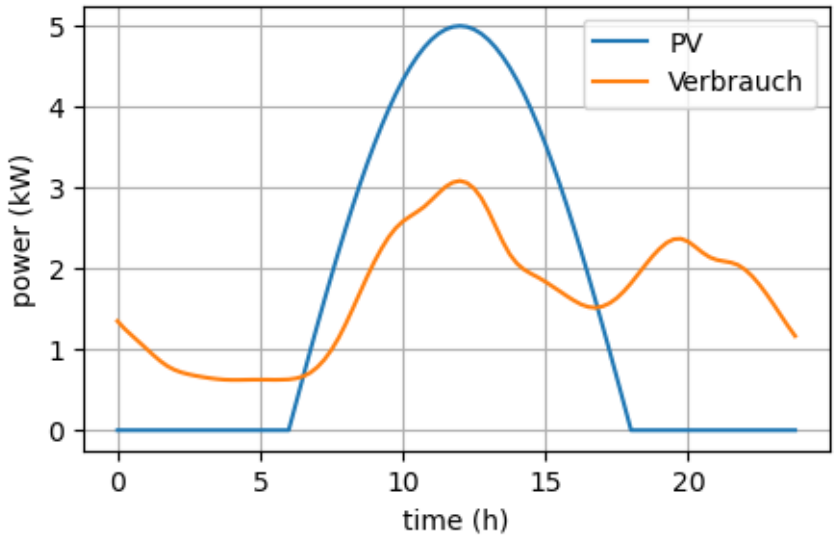
- Bezugspreis: 0.20 EUR/kWh
- Einspeisevergütung: 0.12 EUR/kWh

Ertrag der PV-Anlage und Stromverbrauch des Haushalts in kW:

```
import numpy as np
import matplotlib.pyplot as plt
```

```
dt = 0.25  # h
times = np.arange(start=0, stop=24 + dt, step=dt)  # timestamps: 0, 0.25, 0.5, ..., 23.75, 24
periods = np.arange(start=0, stop=24, step=dt)     # start times of periods:  0, 0.25, 0.5, ..., 23.75
pv_power = np.zeros_like(periods)
pv_power[6*4:6*4 + 12*4] = 5*np.sin(2*np.pi/(6*4) * periods[:12*4] )  # PV power in kW

# source: https://www.bdew.de/energie/standardlastprofile-strom/
demand = np.array([87.7, 81.5, 76.2, 71.0, 65.8, 60.5, 55.6, 51.5, 48.5, 46.4, 44.9, 43.7, 42.7,
```

```
                    41.8, 41.1, 40.6, 40.4, 40.4, 40.5, 40.6, 40.6, 40.6, 40.5, 40.6, 40.8, 41.6,
                    43.2, 46.1, 50.6, 56.7, 64.6, 74.2, 85.5, 97.9, 110.7, 123.4, 135.3, 145.9,
                    155.1, 162.4, 167.8, 171.8, 175.5, 179.6, 184.9, 190.5, 195.7, 199.1, 200.4,
                    198.8, 194.3, 186.7, 176.1, 163.9, 151.7, 141.3, 134.0, 129.1, 125.7, 122.7,
                    119.3, 115.5, 111.7, 107.8, 104.2, 101.1, 99.1, 98.4, 99.5, 102.1, 106.2,
                    111.7, 118.2, 125.5, 132.8, 139.8, 145.9, 150.7, 153.5, 153.9, 151.6, 147.4,
                    142.8, 139.1, 136.9, 135.8, 134.8, 132.8, 129.0, 123.7, 117.0, 109.3, 101.0,
                    92.3, 83.7, 75.7])
demand = demand/(np.sum(demand)*dt)*40  # demand load in kW, scaled to 40 kWh energy demand

plt.figure(figsize=(5, 3))
plt.plot(periods, pv_power, label='PV')
plt.plot(periods, demand, label='Verbrauch')
plt.xlabel('time (h)')
plt.ylabel('power (kW)')
plt.legend()
plt.grid()
```



9.1.3. Überschlagsrechnung

```
# the demand energy is slightly greater than the produced PV energy:
print(f"demand energy: {np.sum(demand)*dt:.2f} kWh")
print(f"PV energy:     {np.sum(pv_power)*dt:.2f} kWh")
```

```
demand energy: 40.00 kWh
PV energy:     38.18 kWh
```

```
# rough estimate, more precisely a lower bound, for the cost:
# (demand energy in kWh - PV energy in kWh) * 0.2 EUR/kWh
cost_rough = (np.sum(demand)*dt - np.sum(pv_power)*dt)*0.2
print(f"rough estimate of cost = {cost_rough:.2f} EUR")
```

```
rough estimate of cost = 0.36 EUR
```

9.1.4. Energienetzwerk

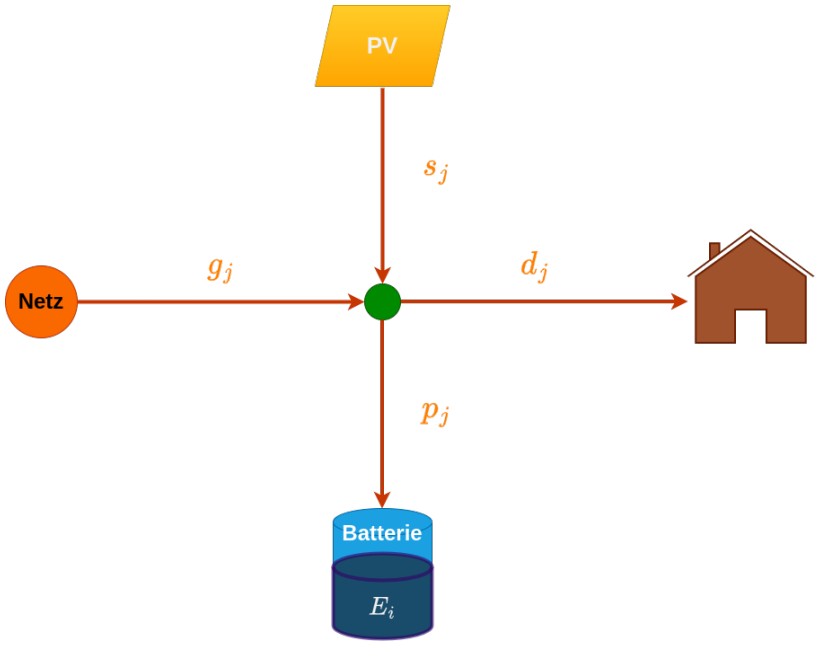


Abbildung 9.1.: Energienetzwerk

9.1.5. Modellierung

Samping-Intervall: $\Delta t = 0.25$ Stunden

Entscheidungsvariablen:

- p_j (Ent-)Ladeleistung in kW während der Zeitperioden mit Indizes $j = 0, 1, 2, \dots, 95$. Schranken: $-5 \leq p_j \leq 5$ kW.
- E_i Energie der Batterie in kWh zu den Zeitpunkten mit Indizes $i = 0, 1, 2, \dots, 96$. Schranken: $0 \leq E_i \leq 10$ kWh.

Zielfunktion: Minimiere die aggregierten Netzkosten in EUR:

$$\min \sum_{j=0}^{95} c(g_j \Delta t)$$

Nebenbedingungen:

- Die Energie der Batterie zu Beginn: $E_0 = 5$ kWh.
- Die Energie der Batterie am Ende: $E_{96} = 5$ kWh.
- Die Energie der Batterie am nächsten Zeitpunkt ergibt sich aus der Energie der Batterie zum aktuellen Zeitpunkt und der Ladeleistung in der dazwischenliegenden Zeitperiode:

$$E_{i+1} = E_i + p_i \Delta t \quad \forall i = 0, 1, 2, \dots, 95$$

- Netzwerkgleichungen:

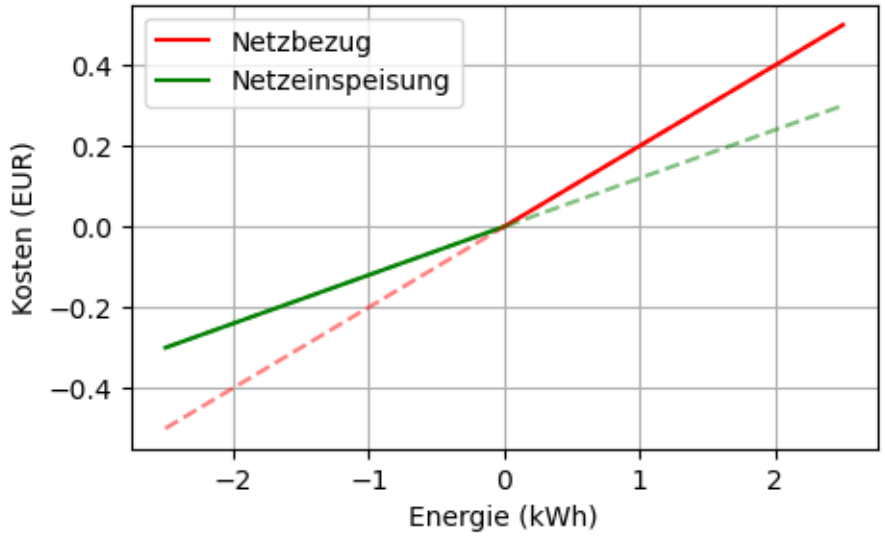
$$g_j + s_j = d_j + p_j \quad \forall j = 0, 1, 2, \dots, 95.$$

Modellierungstrick: Die Zielfunktion $\sum_{j=0}^{95} c(g_j \Delta t)$ ist nicht-linear, genauer: die Funktion c ist konvex und stückweise linear:

$$c(g_j \Delta t) = \begin{cases} 0.2 g_j \Delta t & \text{falls } g_j \geq 0 \\ 0.12 g_j \Delta t & \text{falls } g_j < 0 \end{cases}$$

In der folgenden Abbildung betrachten wir der Einfachheit halber nur eine Zeitperiode:

```
g_pos = np.linspace(  0, 10)
g_neg = np.linspace(-10,  0)
g_all = np.linspace(-10, 10)

c_pos = 0.20  # Preis für Netzbezug in EUR/kWh
c_neg = 0.12  # Preis für Einspeisevergütung in EUR/kWh

plt.figure(figsize=(5, 3))
plt.plot(g_pos*dt, c_pos*g_pos*dt, 'r', label="Netzbezug")
plt.plot(g_neg*dt, c_neg*g_neg*dt, 'g', label="Netzeinspeisung")
plt.plot(g_all*dt, c_pos*g_all*dt, '--r', alpha=0.5)
plt.plot(g_all*dt, c_neg*g_all*dt, '--g', alpha=0.5)
plt.xlabel("Energie (kWh)")
plt.ylabel("Kosten (EUR)")
plt.legend()
plt.grid()
```



Die **Minimierung** einer solchen Zielfunktion lässt sich in einem LP mit zusätzlichen Variablen k_j und zusätzlichen Nebenbedingungen umsetzen:

$$\begin{aligned} \text{min.} \quad & \sum_{j=0}^{95} k_j \\ \text{s. t.} \quad & 0.2 g_j \Delta t \leq k_j \quad \forall j = 0, 1, 2, \ldots, 95 \\ & 0.12 g_j \Delta t \leq k_j \quad \forall j = 0, 1, 2, \ldots, 95 \end{aligned}$$

9.1.6. Implementierung

```
import pyomo.environ as pyo
```

```
n = len(periods)                 # 96
time_indices = np.arange(n + 1)  # 0, 1, ..., 95, 96
period_indices = range(n)        # 0, 1, ..., 95
```

```
model = pyo.ConcreteModel()

model.I = pyo.Set(initialize=time_indices)
model.J = pyo.Set(initialize=period_indices)

model.E = pyo.Var(model.I, bounds=(0.0, 10.0))
model.p = pyo.Var(model.J, bounds=(-5.0, 5.0))
model.g = pyo.Var(model.J, domain = pyo.Reals)
model.k = pyo.Var(model.J, domain = pyo.Reals)

model.cost = pyo.Objective(expr=sum(model.k[j] for j in model.J),
                           sense=pyo.minimize)

model.initial_energy = pyo.Constraint(expr = model.E[0] == 5.0)
model.final_energy = pyo.Constraint(expr = model.E[n] == 5.0)

@model.Constraint(model.J)
def cost_buy(model, j):
    return 0.2*model.g[j]*dt <= model.k[j]

@model.Constraint(model.J)
def cost_sell(model, j):
    return 0.12*model.g[j]*dt <= model.k[j]

@model.Constraint(model.J)
def node(model, j):
    return model.g[j] == model.p[j] +  demand[j] - pv_power[j]

@model.Constraint(model.I)
def charging(model, i):
    if i < n:
        return model.E[i + 1] == model.E[i] + model.p[i] * dt
    else:
        return pyo.Constraint.Skip

# model.pprint()
```

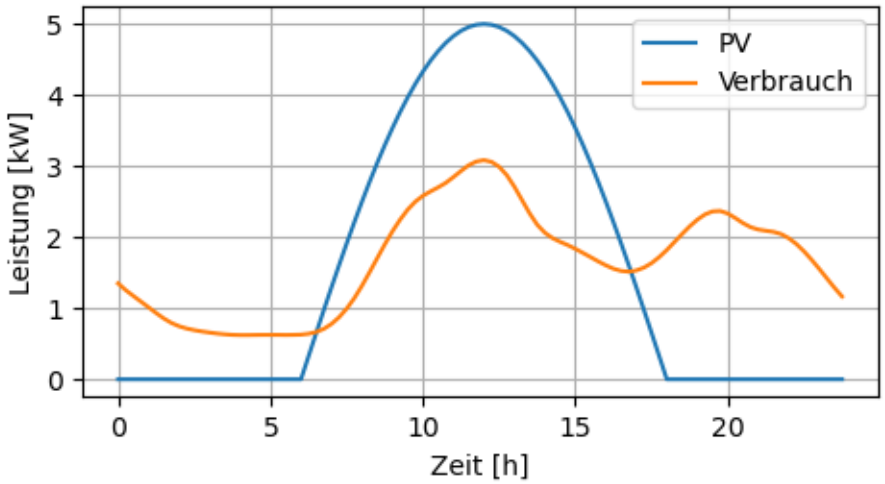
```
solver = pyo.SolverFactory('cbc')
solver = pyo.SolverFactory('glpk')
# solver = pyo.SolverFactory('appsi_highs')
# solver = pyo.SolverFactory('gurobi')

results = solver.solve(model, tee=False)
print(f"status = {results.solver.status}")

print(f"minimal cost = {pyo.value(model.cost):.2f} EUR")
```

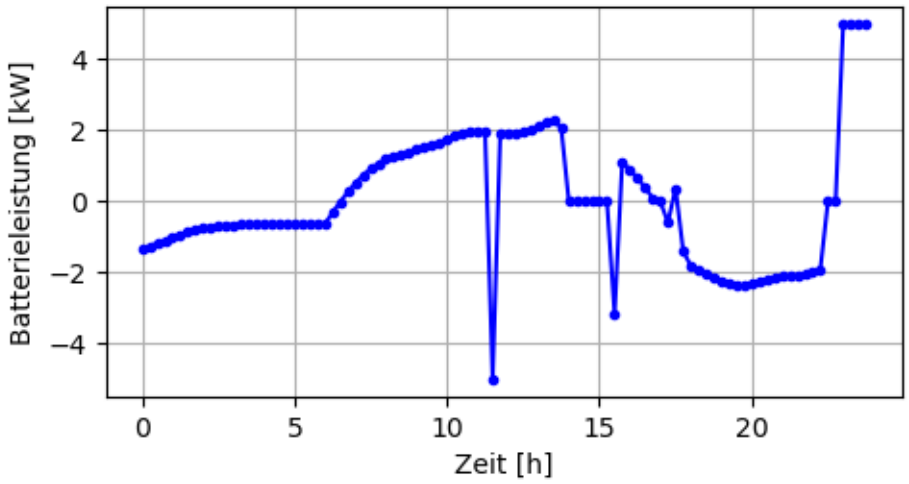
```
status = ok
minimal cost = 0.83 EUR
```

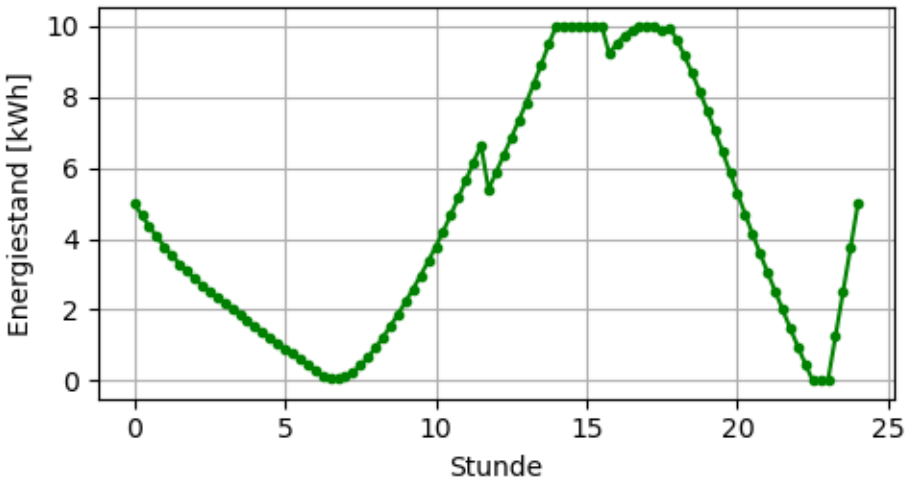
9.1.7. Ergebnisse

Hinweis: Vergleichen Sie die Lösungen von verschiedenen Solvern!

```
E_sol_dict = model.E.extract_values()
E_sol = [E_sol_dict[i] for i in time_indices]

p_sol_dict = model.p.extract_values()
p_sol = [p_sol_dict[j] for j in period_indices]

plt.figure(figsize=(5, 8))
plt.subplot(3, 1, 1)
plt.plot(periods, pv_power, label='PV')
plt.plot(periods, demand, label='Verbrauch')
plt.xlabel('Zeit [h]')
plt.ylabel('Leistung [kW]')
plt.legend()
plt.grid(True)

plt.subplot(3, 1, 2)
plt.plot(periods, p_sol, marker='.', color='blue')
plt.xlabel('Zeit [h]')
plt.ylabel('Batterieleistung [kW]')
plt.grid(True)

plt.subplot(3, 1, 3)
plt.plot(times, E_sol, marker='.', color='green')
plt.xlabel('Stunde')
plt.ylabel('Energiestand [kWh]')
plt.grid(True)

plt.tight_layout()
```

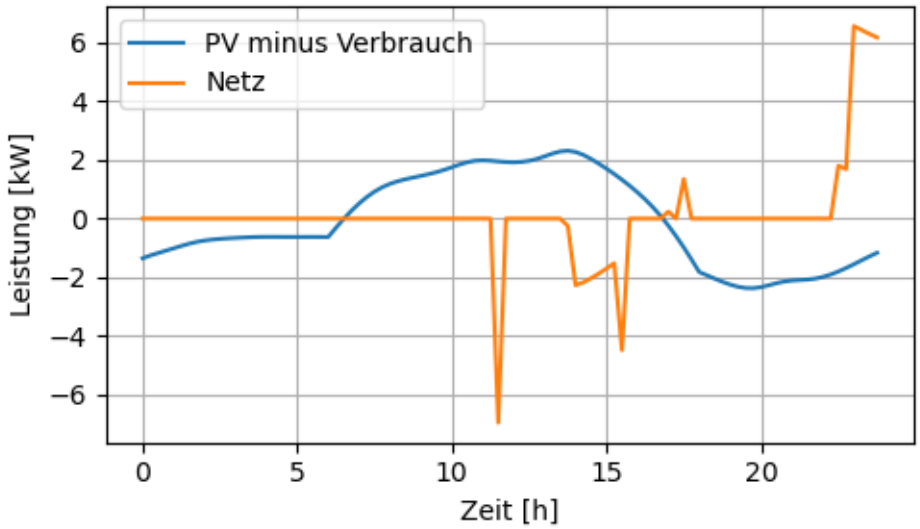







```
g_sol_dict = model.g.extract_values()
g_sol = [g_sol_dict[j] for j in period_indices]
```

```
plt.figure(figsize=(5, 3))
plt.plot(periods, pv_power - demand, label='PV minus Verbrauch')
plt.plot(periods, g_sol, label='Netz')
plt.xlabel('Zeit [h]')
plt.ylabel('Leistung [kW]')
plt.legend()
plt.grid(True)
plt.tight_layout()
```




```
# Autarkiegrad und Eigenverbrauchsanteil:

feed_in = -np.minimum(g_sol, 0) # Einspeiseleistung in kW
Eigenverbrauch = np.sum(pv_power*dt) - np.sum(feed_in*dt)
print(f"Eigenverbrauch = {Eigenverbrauch:.1f} kWh")

PV_Ertrag = np.sum(pv_power*dt)
Eigenverbrauchsanteil = Eigenverbrauch/PV_Ertrag
print(f"Eigenverbrauchsanteil = {Eigenverbrauchsanteil*100:.1f} %")

Gesamtverbrauch = np.sum(demand*dt)
Autarkiegrad = Eigenverbrauch/Gesamtverbrauch
print(f"Autarkiegrad = {Autarkiegrad*100:.1f} %")
```

```
Eigenverbrauch = 32.4 kWh
Eigenverbrauchsanteil = 84.8 %
Autarkiegrad = 80.9 %
```

```
# Zum Vergleich: Autarkiegrad und Eigenverbrauchsanteil ohne Speicher:

Eigenverbrauch = np.sum(np.minimum(pv_power, demand)*dt)
print(f"Eigenverbrauch = {Eigenverbrauch:.1f} kWh")

Eigenverbrauchsanteil = Eigenverbrauch/PV_Ertrag
print(f"Eigenverbrauchsanteil = {Eigenverbrauchsanteil*100:.1f} %")
```

```
Autarkiegrad = Eigenverbrauch/Gesamtverbrauch
print(f"Autarkiegrad = {Autarkiegrad*100:.1f} %")
```

```
Eigenverbrauch = 22.4 kWh
Eigenverbrauchsanteil = 58.7 %
Autarkiegrad = 56.1 %
```

Interpretation:

- Die verschiedenen Solver liefern wie zu erwarten dieselben minimalen Kosten. Die Lösungen unterscheiden sich jedoch in den Werten der Entscheidungsvariablen. Dies liegt daran, dass die Solver unterschiedliche Algorithmen verwenden, um die Lösung zu finden.
- Eingenverbrauch, Eingenverbrauchsanteil und Autarkiegrad sind bei allen Lösungen gleich.

9.2. Übung: Eigenverbrauch maximieren

9.2.1. Problemstellung

1. Begründen Sie mathematisch, warum das Maximieren des Eigenverbrauchs äquivalent zum Maximieren des Eigenverbrauchsanteils und des Autarkiegrads ist.
2. Modellieren Sie ein Optimierungsproblem, das den Eigenverbrauch maximiert.
3. Vergleichen Sie Ihre Lösung mit der Lösung des Problems, das die Netzkosten minimiert. Konnten Sie den Eigenverbrauch erhöhen?

Hinweis: Lesen Sie bei Bedarf den Abschnitt Eigenverbrauchsanteil und Autarkiegrad.

10. Batterieverluste

10.1. Problemstellung

Reale Batterien haben oft nicht zu vernachlässigende Lade- und Entladeverluste. Wir modellieren im Folgenden eine Batterie mit 90 % Ladewirkungsgrad und 90 % Entladewirkungsgrad. Die Batterie habe eine Kapazität von 50 kWh und maximal 20 kW Lade- und Entladeleistung. Die Batterie sei zu Beginn und am Ende zu 50 % geladen.

Wir betrachten einen Zeitraum von 8 Stunden in Viertelstundenschritten und verwenden die Batterie, um bei gegebenem Lastprofil und gegebenen zeitabhängigen Energiepreisen

1. die Energiekosten zu minimieren.
2. die Spitzenlast gegenüber dem Netz zu minimieren.

Wir modellieren und implementieren beide Optimierungsprobleme und untersuchen folgende Frage: Wann führt eine Relaxierung des MILP- zu einem LP-Optimierungsproblems ebenfalls zu einer optimalen Lösung?

10.2. Energienetzwerk

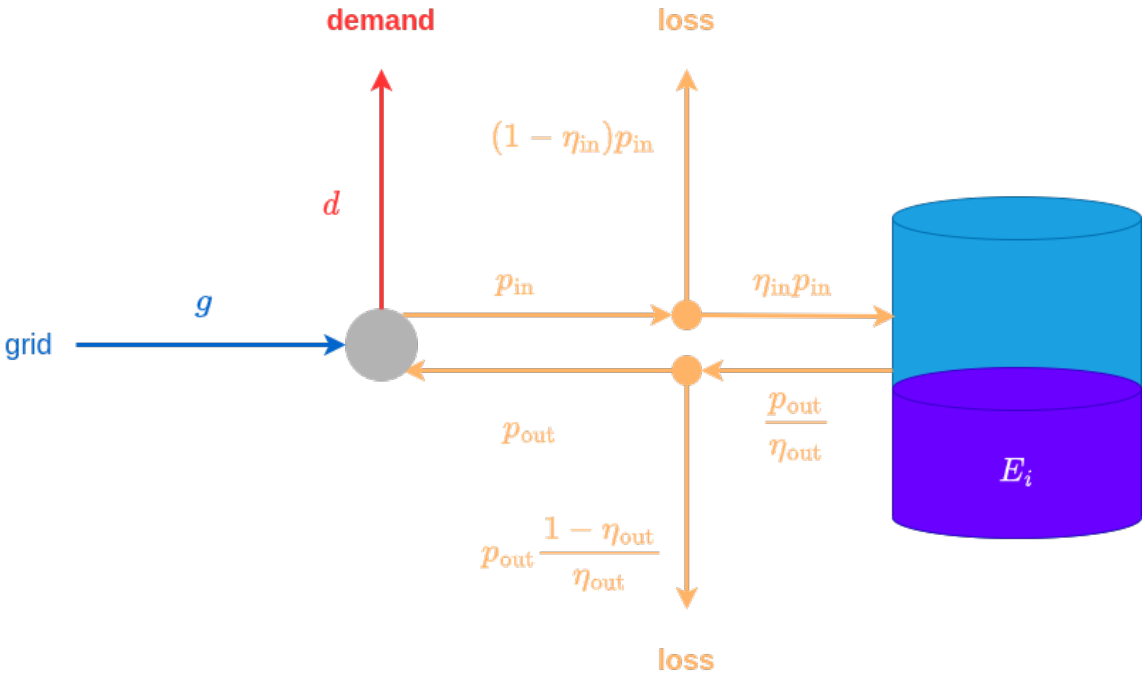
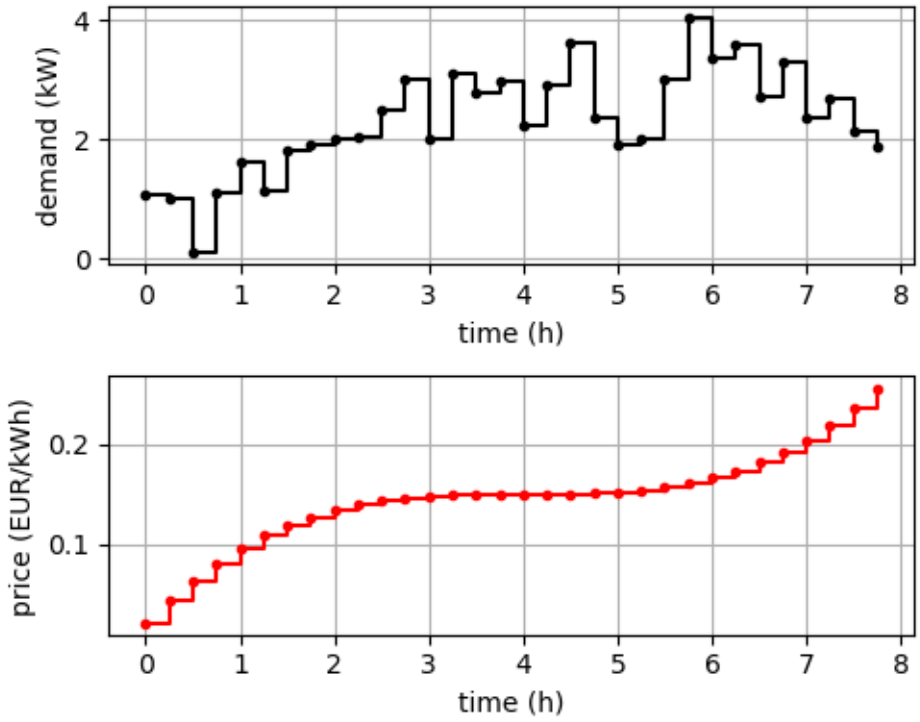


Abbildung 10.1.: Batterieverluste

10.3. Daten

```
import numpy as np
import matplotlib.pyplot as plt
import pyomo.environ as pyo
```

```
# data:

dt = 0.25  # h
times = np.arange(start=0, stop=8 + dt, step=dt)
periods = np.arange(start=0, stop=8, step=dt)

np.random.seed(10)
noise = np.random.normal(loc=0, scale=0.5, size=len(periods))
demand = 2 + 0.2*periods - 0.1*(periods - 4)**2 + noise

# case: use peak power objective, the following extra demand entry, gurobi and relax=True
# demand[15] = 100

price = 0.15 + 0.002*(periods - 4)**3

plt.figure(figsize=(5, 4))
plt.subplot(2, 1, 1)
plt.step(periods, demand, where='post', color='black', marker='.')
plt.xlabel('time (h)')
plt.ylabel('demand (kW)')
plt.grid()

plt.subplot(2, 1, 2)
plt.step(periods, price, where='post', color='r', marker='.')
plt.xlabel('time (h)')
plt.ylabel('price (EUR/kWh)')
plt.grid()

plt.tight_layout()
```



10.4. Modellierung

Daten:

- Samping-Intervall: $\Delta t = 0.25$ Stunden
- Anzahl der Zeitperioden: $n = 32$
- Ladewirkungsgrad: $\eta_{\text{in}} = 0.9$
- Entladewirkungsgrad: $\eta_{\text{out}} = 0.9$
- Batteriekapazität: $E_{\max} = 50$ kWh
- Maximale (Ent-)Ladeleistung: $p_{\max} = 20$ kW
- Anfangsenergie: $E_{\text{start}} = 25$ kWh
- Endenergie: $E_{\text{end}} = 25$ kWh
- Verbrauch (demand): d_j während der Zeitperioden j
- Energiepreise: c_j während der Zeitperioden j

Entscheidungsvariablen:

- $p_{\text{in},j}$: Ladeleistung in kW während der Zeitperioden j, $0 \leq p_{\text{in},j} \leq p_{\max}$
- $p_{\text{out},j}$: Entladeleistung in kW während der Zeitperioden j, $0 \leq p_{\text{out},j} \leq p_{\max}$
- E_i: Energieinhalt der Batterie in kWh zu den Zeitpunkten i
- g_j: Netzleistung in kW während der Zeitperioden j
- $b_{\text{in},j}$: binäre Variable, die angibt, ob die Batterie in der Zeitperiode j geladen wird ($b_{\text{in},j} = 1$) oder nicht ($b_{\text{in},j} = 0$)
- $b_{\text{out},j}$: binäre Variable, die angibt, ob die Batterie in der Zeitperiode j entladen wird ($b_{\text{out},j} = 1$) oder nicht ($b_{\text{out},j} = 0$)

Zielfunktionen:

1. Minimiere die Netzkosten $\sum_{j=0}^{n-1} c_j g_j \Delta t$ in EUR
2. Minimiere die Spitzenlast m in kW mit $g_j \leq m$ für alle $j = 0, 1, 2, \ldots, n-1$

Nebenbedingungen:

- Die Energie der Batterie zu Beginn: $E_0 = E_{\text{start}}$
- Die Energie der Batterie am Ende: $E_n = E_{\text{end}}$
- zeitliche Änderung der Energie der Batterie:

$$E_{i+1} = E_i + (\eta_{\text{in}} p_{\text{in},i} - \frac{p_{\text{out},i}}{\eta_{\text{out}}})\Delta t \quad \forall i = 0, 1, 2, \ldots, n-1$$

- Kein gleichzeitiges Laden und Entladen der Batterie:
  - $p_{\text{in},j} \leq p_{\max} b_{\text{in},j}$ für alle $j = 0, 1, 2, \ldots, n-1$
  - $p_{\text{out},j} \leq p_{\max} b_{\text{out},j}$ für alle $j = 0, 1, 2, \ldots, n-1$
  - $b_{\text{in},j} + b_{\text{out},j} \leq 1$ für alle $j = 0, 1, 2, \ldots, n-1$
- Netzwerkgleichungen:

$$g_j + p_{\text{out},j} = d_j + p_{\text{in},j} \quad \forall j = 0, 1, 2, \ldots, n-1$$

10.5. Implementierung

```
n = len(periods)
time_indices   = range(n + 1)  # 0, 1, ..., n - 1, n
period_indices = range(n)      # 0, 1, ..., n - 1

eta_in  = 0.9    # efficiency of charging
eta_out = 0.9    # efficiency of discharging
E_max   = 50.0   # kWh, maximum energy level
E_start = 25.0   # kWh, starting energy level
E_end   = 25.0   # kWh, final energy level
p_max   = 20.0   # kW, maximum (dis-)charging power

objective = "cost"
objective = "peak power"

relax = False
# relax = True
```

```
model = pyo.ConcreteModel()

model.I = pyo.Set(initialize=time_indices)
model.J = pyo.Set(initialize=period_indices)

model.E = pyo.Var(model.I, bounds=(0.0, E_max))
model.p_in  = pyo.Var(model.J, bounds=(0.0, p_max))
model.p_out = pyo.Var(model.J, bounds=(0.0, p_max))
model.g = pyo.Var(model.J, domain=pyo.Reals)
if not relax:
    model.b_in  = pyo.Var(model.J, domain=pyo.Binary)
    model.b_out = pyo.Var(model.J, domain=pyo.Binary)

if objective == "cost":
    model.obj = pyo.Objective(expr=
        sum(price[j]*model.g[j]*dt for j in model.J),
        sense=pyo.minimize)
elif objective == "peak power":
```

```
    model.m = pyo.Var(domain=pyo.NonNegativeReals)
    model.obj = pyo.Objective(expr=model.m, sense=pyo.minimize)
    @model.Constraint(model.J)
    def peak_power(model, j):
        return model.g[j] <= model.m

@model.Constraint(model.J)
def node(model, j):
    return model.g[j] + model.p_out[j] == model.p_in[j] +  demand[j]

model.initial_energy = pyo.Constraint(expr = model.E[0] == E_start)
model.final_energy   = pyo.Constraint(expr = model.E[n] == E_end)

@model.Constraint(model.I)
def energy_update(model, i):
    if i < n:
        return model.E[i + 1] == model.E[i] + (eta_in*model.p_in[i] -
                                 model.p_out[i]/eta_out)*dt
    else:
        return pyo.Constraint.Skip

if not relax:
    @model.Constraint(model.J)
    def charging(model, j):
        return model.p_in[j] <= model.b_in[j]*p_max

    @model.Constraint(model.J)
    def discharging(model, j):
        return model.p_out[j] <= model.b_out[j]*p_max

    @model.Constraint(model.J)
    def only_one(model, j):
        return model.b_in[j] + model.b_out[j] <= 1
```

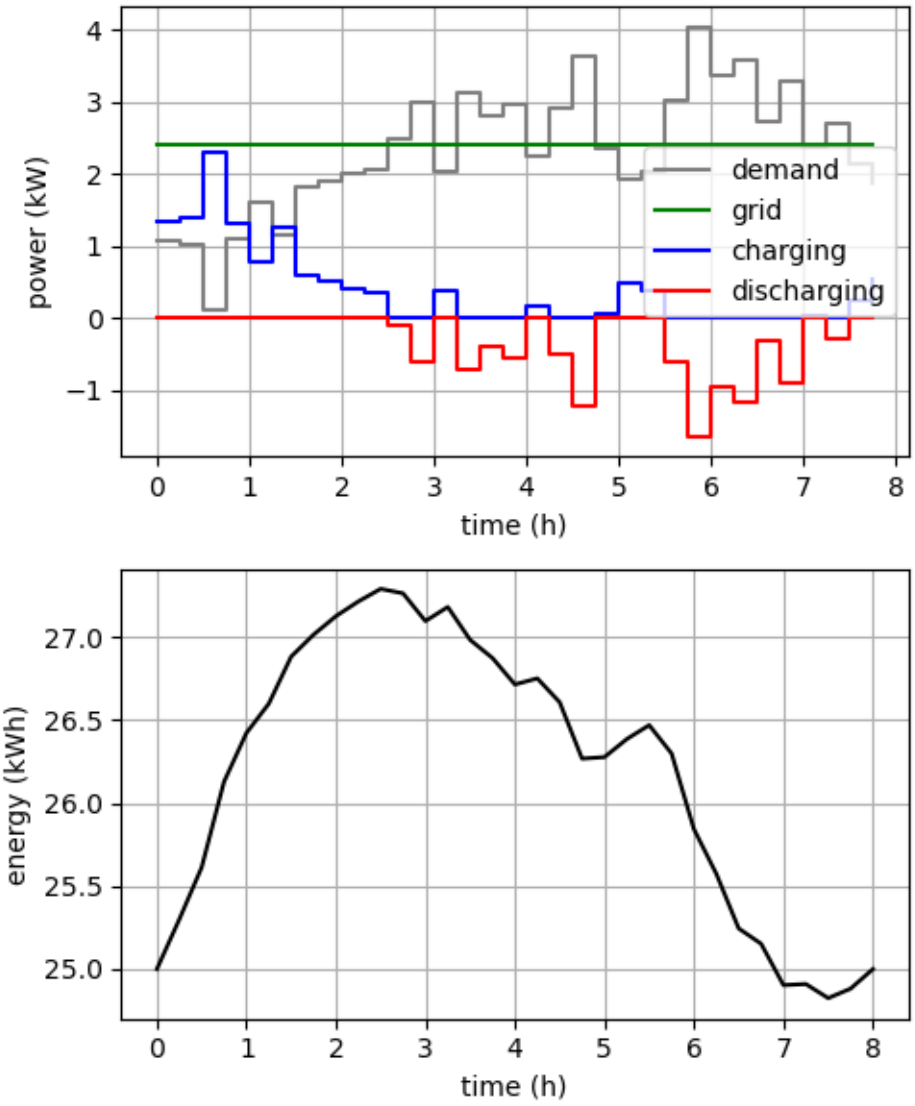
```
solver = pyo.SolverFactory('cbc')
# solver = pyo.SolverFactory('glpk')
# solver = pyo.SolverFactory('appsi_highs')
# solver = pyo.SolverFactory('gurobi')

results = solver.solve(model, tee=False)
print(f"status = {results.solver.status}")

if objective == "cost":
    print(f"minimal cost = {pyo.value(model.obj):.2f} EUR")
elif objective == "peak power":
    print(f"minimal peak power = {pyo.value(model.obj):.2f} kW")
```

```
status = ok
minimal peak power = 2.39 kW
```

10.6. Ergebnisse

```
E_sol_dict = model.E.extract_values()
E_sol = np.array([E_sol_dict[i] for i in time_indices])

g_sol_dict = model.g.extract_values()
g_sol = np.array([g_sol_dict[j] for j in period_indices])

p_in_sol_dict = model.p_in.extract_values()
p_in_sol = np.array([p_in_sol_dict[j] for j in period_indices])

p_out_sol_dict = model.p_out.extract_values()
p_out_sol = np.array([p_out_sol_dict[j] for j in period_indices])

# non concurrent (dis-)charging check: if the sum of the products
# of p_in_sol and p_out_sol is zero, then there is no time period
# where both are positive.
check = np.sum(p_in_sol*p_out_sol)
print(f"non concurrent (dis-)charging check: {check:.6f}")

plt.figure(figsize=(5, 6))
plt.subplot(2, 1, 1)
plt.step(periods, demand, where='post', color='grey', label='demand')
plt.step(periods, g_sol, where='post', color='green', label='grid')
plt.step(periods, p_in_sol, where='post', color='b', label='charging')
plt.step(periods, -p_out_sol, where='post', color='r', label='discharging')
plt.xlabel('time (h)')
plt.ylabel('power (kW)')
plt.legend()
plt.grid()

plt.subplot(2, 1, 2)
plt.plot(times, E_sol, color='black')
plt.xlabel('time (h)')
plt.ylabel('energy (kWh)')
plt.grid()

plt.tight_layout()
```

```
non concurrent (dis-)charging check: 0.000000
```



Antwort zur Frage “Wann führt eine Relaxierung des MILP- zu einem LP-Optimierungsproblems ebenfalls zu einer optimalen Lösung?”: Bei der Minimierung der Energiekosten, aber nicht immer bei der Minimierung der Spitzenlast, siehe `# case: [...]` im Abschnitt Daten.

11. Verschiebbare Lasten

11.1. Waschmaschinenstart

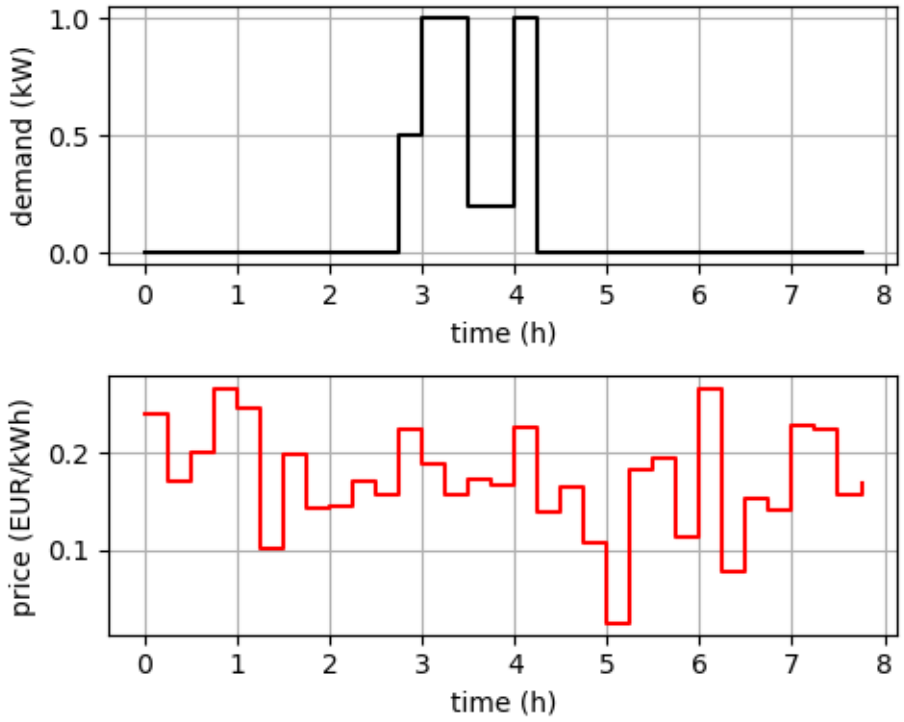
11.1.1. Problemstellung

Sie haben einen zeitabhängingen Energiepreis c_j und wollen den dazu kostenoptimalen Startzeitpunkt für Ihre Waschmaschine finden, sodass der Waschvorgang in den kommenden acht Stunden beendet ist. Der Lastgang des Waschvorgangs und die Preise sind durch die folgenden Daten in Viertelstundenschritten gegeben:

```
import numpy as np
import matplotlib.pyplot as plt
import pyomo.environ as pyo
```

```
# time:
T  = 8.0   # h
dt = 0.25  # h
n = int(T/dt)  # number of time periods
times = np.arange(start=0, stop=T + dt, step=dt)
periods = np.arange(start=0, stop=T, step=dt)
time_indices   = range(n + 1)  # 0, 1, ..., n - 1, n
period_indices = range(n)      # 0, 1, ..., n - 1

# load profile:
load_profile = np.array([0.5, 1.0, 1.0, 0.2, 0.2, 1.0])  # kW
# duration of the load profile in number of time periods:
m = len(load_profile)

# example demand for a given start time:
load_start = 11  # index of starting time of load profile
demand = np.zeros_like(periods)
demand[load_start:load_start + m] = load_profile

# prices:
np.random.seed(0)
noise = np.random.normal(loc=0, scale=0.05, size=n)
price = .15 + 0.2*noise**2 + noise

# plot:
plt.figure(figsize=(5, 4))
plt.subplot(2, 1, 1)
plt.step(periods, demand, where='post', color='black')
plt.xlabel('time (h)')
plt.ylabel('demand (kW)')
plt.grid(True)

plt.subplot(2, 1, 2)
plt.step(periods, price, where='post', color='red')
plt.xlabel('time (h)')
plt.ylabel('price (EUR/kWh)')
```

```
plt.grid(True)

plt.tight_layout()
```



11.1.2. Modellierung

Daten:

- Samping-Intervall: $\Delta t = 0.25$ Stunden
- Anzahl der Zeitperioden: $n = 8$ Stunden / $\Delta t = 32$
- Lastgang der Waschmaschine: l_k in kW für $k = 0, \dots, m-1$
- Energiepreise: c_j in EUR/kWh

Entscheidungsvariablen:

- p_j: Leistungsaufnahme der Waschmaschine in kW während der Zeitperioden $j = 0, 1, \dots, n-1$
- s_i: binäre Variablen für $i = 0, 1, \dots, n-m$, die angeben, dass die Waschmaschine zum Zeitpunkt i gestartet wird, falls $s_i = 1$.

Zielfunktion: $\min \sum_{j=0}^{n-1} c_j p_j \Delta t$

Nebenbedingungen:

- $\sum_{i=0}^{n-m} s_i = 1$: Die Waschmaschine wird genau einmal gestartet.
- $p_j = \sum_{k=0}^{m-1} l_k s_{j-k}$ für alle $j = 0, 1, \dots, n-1$, falls $j - k \leq n - m$: Falls die Waschmaschine zum Zeitpunkt $j - k$ gestartet wurde, dann hat sie in der Zeitperiode j die Leistungsaufnahme l_k.

11.1.3. Implementierung

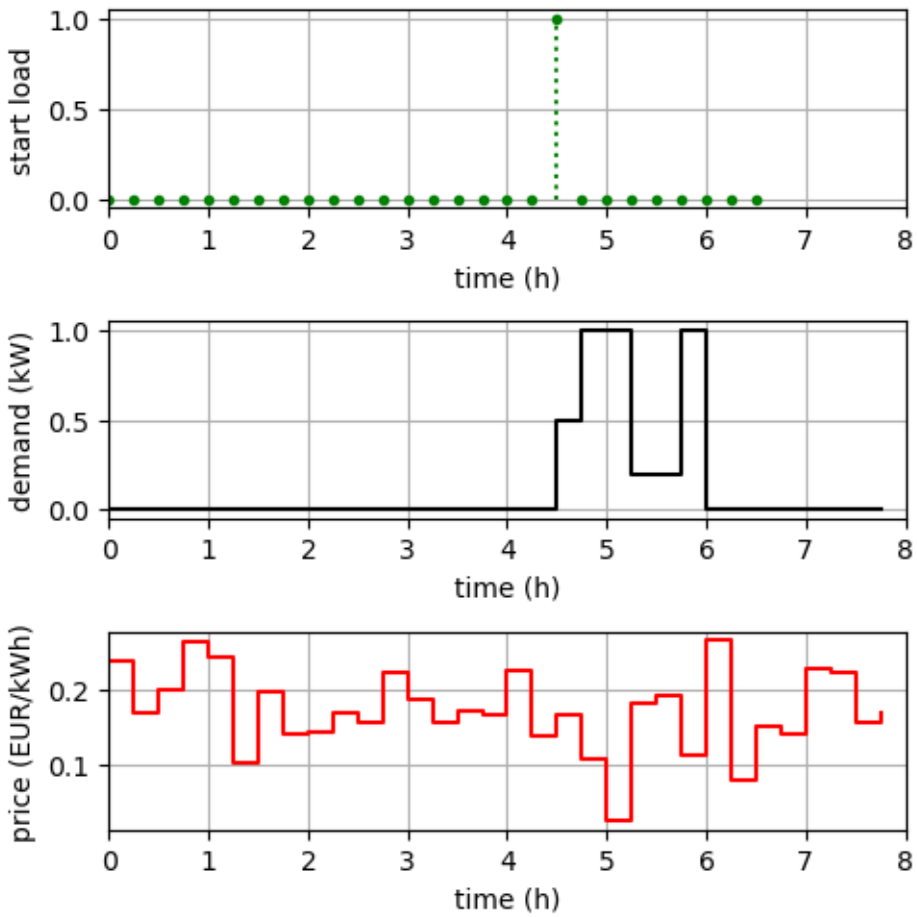
```
model = pyo.ConcreteModel()

model.J = pyo.Set(initialize=period_indices)
# set of time points where the load profile can be started:
model.S = pyo.Set(initialize=time_indices[:-m])

model.p = pyo.Var(model.J, domain=pyo.NonNegativeReals)  # power in kW
model.s = pyo.Var(model.S, domain=pyo.Binary)  # start load at time/period

model.cost = pyo.Objective(expr=sum(price[j]*model.p[j]*dt
                for j in model.J), sense=pyo.minimize)

model.one_start = pyo.Constraint(expr=sum(model.s[j] for j in model.S) == 1)

@model.Constraint(model.J)
def demand_constraint(model, j):
    return model.p[j] == sum(load_profile[k]*model.s[j - k]
                             for k in range(m)
                             if j - k in model.S)
```

```
solver = pyo.SolverFactory('cbc')
# solver = pyo.SolverFactory('glpk')
# solver = pyo.SolverFactory('appsi_highs')
# solver = pyo.SolverFactory('gurobi')

results = solver.solve(model, tee=False)
print(f"status = {results.solver.status}")
print(f"minimal cost = {pyo.value(model.cost):.2f} EUR")
```

```
status = ok
minimal cost = 0.10 EUR
```

```
p_sol_dict = model.p.extract_values()
p_sol = np.array([p_sol_dict[j] for j in period_indices])

s_sol_dict = model.s.extract_values()
s_sol = np.array([s_sol_dict[j] for j in time_indices[:-m]])

plt.figure(figsize=(5, 5))
plt.subplot(3, 1, 1)
plt.stem(times[:-m], s_sol,basefmt=' ',
         markerfmt='.', linefmt='g:')
plt.xlim(0, T)
plt.xlabel('time (h)')
plt.ylabel('start load')
plt.grid(True)

plt.subplot(3, 1, 2)
plt.step(periods, p_sol, where='post', color='black')
plt.xlim(0, T)
plt.xlabel('time (h)')
plt.ylabel('demand (kW)')
plt.grid(True)
```

```
plt.subplot(3, 1, 3)
plt.step(periods, price, where='post', color='red')
plt.xlim(0, T)
plt.xlabel('time (h)')
plt.ylabel('price (EUR/kWh)')
plt.grid(True)

plt.tight_layout()
```



11.2. Übung: Power Tracking

11.2.1. Problemstellung

Gegeben ist die Ertragskurve einer PV-Anlage mit 5 kWp über einen Tag. Sie wollen 20 Gabelstapler-Austauschbatterien so laden, dass die Gesamtladekurve möglichst der PV-Ertragskurve folgt. Die Ladekurve einer Batterie und der PV-Ertrag sind durch die Daten unten angegeben.

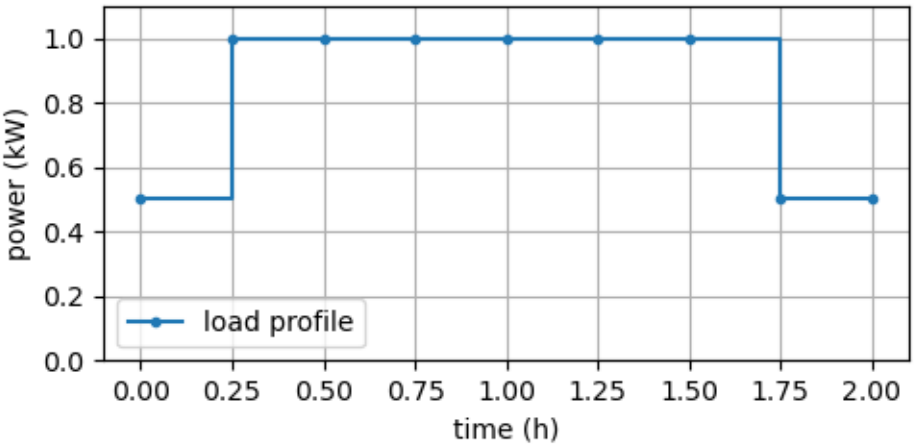
Das Folgen einer vorgegebenen Leistungskurve wird auch als *Power Tracking* bezeichnet. Dabei wird der Unterschied zwischen der vorgegebenen Kurve und der nachfahrenden Kurve minimiert. Wir quantifizieren

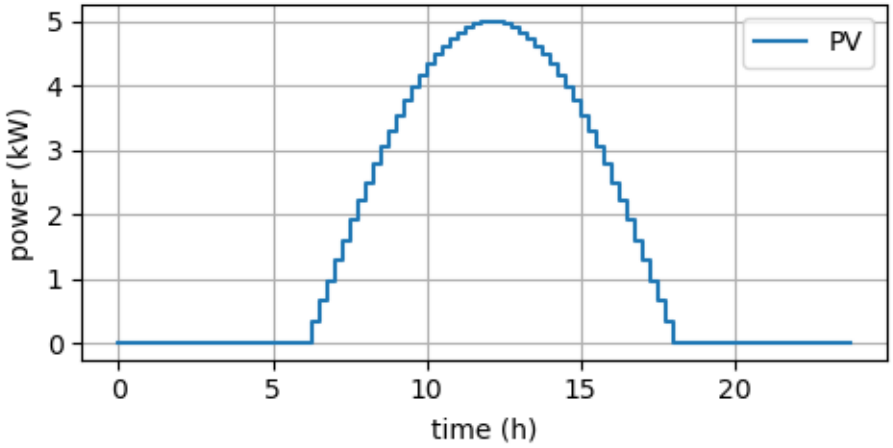
den Unterschied in diesem Beispiel durch die Summe der Beträge der Differenzen zwischen den beiden Kurven:

$$\sum_{j=0}^{n-1} |p_{\mathrm{pv},j} - p_{\mathrm{load},j}|$$

Andere Möglichkeiten, den Unterschied zu quantifizieren sind die Summe der Quadrate der Differenzen oder die maximale absolute Differenz, siehe den Abschnitt Tracking.

```
# sampling interval:
dt = 0.25  # h

# load profile for charging a forklift
load_profile = np.array([0.5, 1, 1, 1, 1, 1, 1, 0.5])  # kW
m = len(load_profile)

# PV power:
T = 24     # h
n = int(T/dt)  # number of time periods
times = np.arange(start=0, stop=T + dt, step=dt)  # timestamps: 0, 0.25, 0.5, ..., 23.75, 24
periods = np.arange(start=0, stop=T, step=dt)     # start times of periods:  0, 0.25, 0.5, ..., 23.75
pv_power = np.zeros_like(periods)
pv_power[6*4:6*4 + 12*4] = 5*np.sin(2*np.pi/(6*4) * periods[:12*4] )  # PV power in kW

plt.figure(figsize=(5, 5))
plt.subplot(2, 1, 1)
load_profile_ = np.hstack((load_profile, load_profile[-1]))
plt.step(np.arange(m + 1)*dt, load_profile_, where='post',
         marker='.', label='load profile')
plt.ylim(0, 1.1*load_profile.max())
plt.xlabel('time (h)')
plt.ylabel('power (kW)')
plt.legend()
plt.grid()

plt.subplot(2, 1, 2)
plt.step(periods, pv_power, where='post', label='PV')
plt.xlabel('time (h)')
plt.ylabel('power (kW)')
plt.legend()
plt.grid()

plt.tight_layout()
```


power (kW)
1.0
0.8
0.6
0.4
0.2
0.0
load profile
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
time (h)


power (kW)
5
4
3
2
1
0
PV
0
5
10
15
20
time (h)

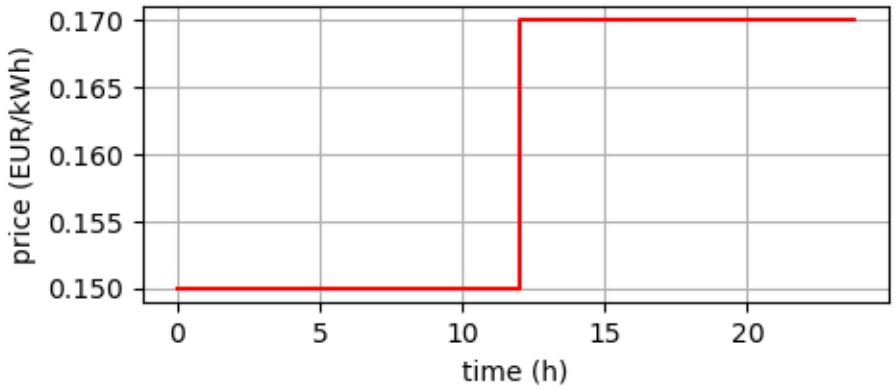
12. Thermischer Speicher

12.1. Problemstellung

Sie wollen Ihren 150 Liter Warmwasserboiler aus dem Abschnitt Thermische Speicher mit den zusätzlichen, unten angegebenen Parametern kostenoptimal betreiben. In den kommenden 24 Stunden soll er von 25 °C auf 70 °C aufgeheizt werden. Der Strompreis beträgt in den ersten 12 Stunden 0,15 €/kWh und in den zweiten 12 Stunden 0,17 €/kWh. Wir bestimmen die optimale Ladeleistung in Viertelstundenintervallen.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
# time:
T  = 24.0  # h
dt = 0.25  # h
n = int(T/dt)  # number of time periods
times = np.arange(start=0, stop=T + dt, step=dt)
periods = np.arange(start=0, stop=T, step=dt)
time_indices   = range(n + 1)  # 0, 1, ..., n - 1, n
period_indices = range(n)      # 0, 1, ..., n - 1

# thermal storage:
c = 0.175       # kWh/K
k = 0.0012      # kW/K
T_env = 15      # °C
E_max = c*75    # kWh, 75 °C
p_max = 1.0     # kW
E_start = c*20  # kWh, 20 °c
E_end = c*70    # kWh, 70 °C

# prices in EUR/kWh:
price = np.zeros(n)
price[0:n//2] = 0.15
price[n//2:n] = 0.17

plt.figure(figsize=(5, 2))
plt.step(periods, price, where='post', color='r')
plt.xlabel('time (h)')
plt.ylabel('price (EUR/kWh)')
plt.grid()
```



12.2. Modellierung

Daten:

- c_j: Strompreis in €/kWh während der Zeitperioden $j = 0, 1, \dots, n-1$
- Parameter des Warmwasserboilers, vgl.Code Code oben und Abschnitt Thermische Speicher.

Entscheidungsvariablen:

- p_j: elektrische Leistungsaufnahme des Warmwasserboilers in kW während der Zeitperioden $j = 0, 1, \dots, n-1$ mit $0 \leq p_j \leq p_{\max}$.
- E_i: Energieinhalt des Wassers zu den Zeitpunkten $i = 0, 1, \dots, n$ mit $0 \leq E_i \leq E_{\max}$.

Zielfunktion: $\min \sum_{j=0}^{n-1} c_j p_j \Delta t$

Nebenbedingungen:

- $E_0 = c \cdot 20$ °C
- $E_n = c \cdot 70$ °C
- $E_{i+1} = E_i e^{-\frac{k}{c}\Delta t} + \frac{c}{k}(1 - e^{-\frac{k}{c}\Delta t})(p_i + kT_{\text{env}})$ für $i = 0, 1, \dots, n-1$

12.3. Implementierung

```
import pyomo.environ as pyo
```

```
model = pyo.ConcreteModel()

model.I = pyo.Set(initialize=time_indices)
model.J = pyo.Set(initialize=period_indices)

model.E = pyo.Var(model.I, bounds=(0.0, E_max))
model.p = pyo.Var(model.J, bounds=(0.0, p_max))

model.obj = pyo.Objective(expr=
    sum(price[j]*model.p[j]*dt for j in model.J),
    sense=pyo.minimize)

model.initial_energy = pyo.Constraint(expr = model.E[0] == E_start)
model.final_energy   = pyo.Constraint(expr = model.E[n] == E_end)
```

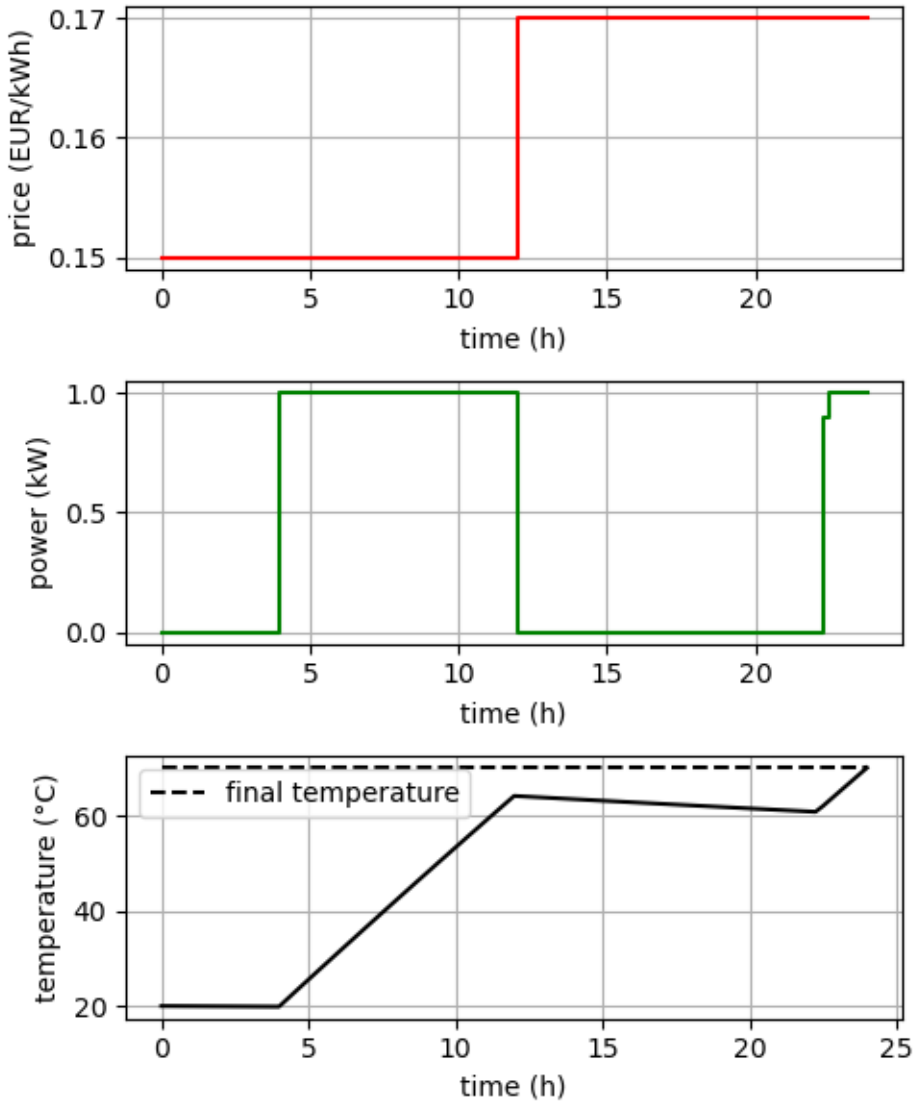
```
@model.Constraint(model.I)
def energy_update(model, i):
    if i < n:
        return model.E[i + 1] == model.E[i]*np.exp(-k/c*dt) + \
        c/k*(1 - np.exp(-k/c*dt))*(model.p[i] + k*T_env)
    else:
        return pyo.Constraint.Skip
```

```
solver = pyo.SolverFactory('cbc')
# solver = pyo.SolverFactory('glpk')
# solver = pyo.SolverFactory('appsi_highs')
# solver = pyo.SolverFactory('gurobi')

results = solver.solve(model, tee=False)
print(f"status = {results.solver.status}")
print(f"minimal cost = {pyo.value(model.obj):.2f} EUR")
```

```
status = ok
minimal cost = 1.49 EUR
```

12.4. Ergebnisse

```
E_sol_dict = model.E.extract_values()
E_sol = np.array([E_sol_dict[i] for i in time_indices])

p_sol_dict = model.p.extract_values()
p_sol = np.array([p_sol_dict[j] for j in period_indices])

plt.figure(figsize=(5, 6))
plt.subplot(3, 1, 1)
plt.step(periods, price, where='post', color='r')
plt.xlabel('time (h)')
plt.ylabel('price (EUR/kWh)')
plt.grid()

plt.subplot(3, 1, 2)
plt.step(periods, p_sol, where='post', color='green')
plt.xlabel('time (h)')
plt.ylabel('power (kW)')
plt.grid()

plt.subplot(3, 1, 3)
plt.plot(times, E_sol/c, color='black')
plt.hlines(y=E_end/c, xmin=0, xmax=T, color='black',
           linestyle='--', label='final temperature')
plt.xlabel('time (h)')
plt.ylabel('temperature (°C)')
plt.legend()
plt.grid()

plt.tight_layout()
```


price (EUR/kWh)
0.17
0.16
0.15
0
5
10
15
20
time (h)
power (kW)
1.0
0.5
0.0
0
5
10
15
20
time (h)
temperature (°C)
final temperature
60
40
20
0
5
10
15
20
25
time (h)

13. Transportproblem

13.1. Fabriken zu Lagerhäusern

13.1.1. Problemstellung

Das Unternehmen EiD (Erbsen in Dosen) produziert Dosenerbsen in seinen drei Fabriken F1, F2 und F3. Die produzierte Ware wird in die vier Lagerhäuser L1, L2, L3 und L4 transportiert. Die Produktionsmengen der Fabriken, die Bedarfsmengen der Lagerhäuser und die Kosten des Transports pro LKW-Ladung sind in der Parametertabelle Transportproblem_Beispiel.xlsx angeführt. Beachten Sie, dass die Summe der Produktionsmengen gleich der Summe der Bedarfsmengen ist.

Wir lösen das Transportproblem und stellen die Lösung dar.

Quelle: Hillier, Lieberman: Introduction to Operations Research. 10th edition, 2015. p. 319 ff.

13.1.2. Daten

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import pyomo.environ as pyo
```

```
# read data with pandas from excel file:
df = pd.read_excel("daten/Transportproblem_Beispiel.xlsx", index_col=0)
df
```

	L1	L2	L3	L4	Produktion
F1	464	513	654	867	75.0
F2	352	416	690	791	125.0
F3	995	682	388	685	100.0
Bedarf	80	65	70	85	NaN

13.1.3. Modellierung

Wir stellen die Daten des Transportproblems in einem ungerichteten Graphen dar, in dem die Fabriken und Lagerhäuser Knoten sind und jede Fabrik mit jedem Lagerhaus durch eine Kante verbunden ist. Die Kanten haben dabei die Transportkosten pro Mengeneinheit als Gewicht:

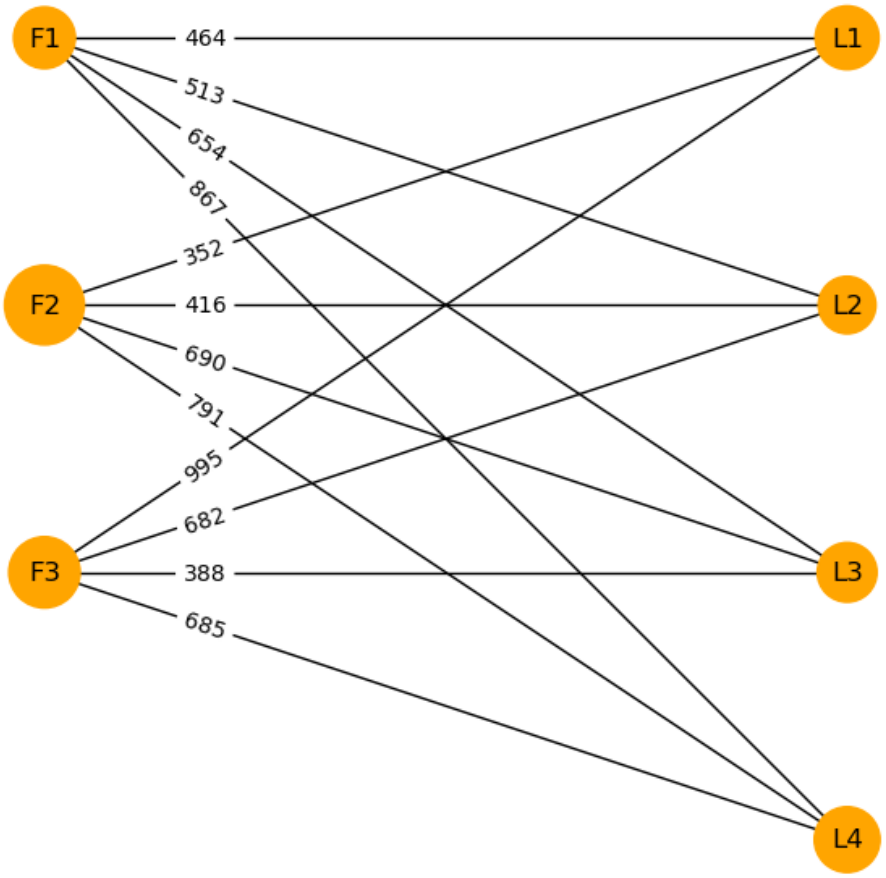


Abbildung 13.1.: Graph des Transportproblems

Für die Optimierung erhält jede Kante eine Flussvariable, die die transportierte Menge über die Kante angibt. Die Zielfunktion ist die Summe der Transportkosten pro Mengeneinheit multipliziert mit der zugehörigen transportierten Menge. Die Nebenbedingungen fixieren die Bedarfsmengen der Lagerhäuser und die Produktionsmengen der Fabriken.

13.1.4. Implementierung

```
# make a list of the sources' names:
sources = df.index[:-1].tolist()
sources
```

```
['F1', 'F2', 'F3']
```

```
# make a list of the targets' names:
targets = df.columns[:-1].tolist()
targets
```

```
['L1', 'L2', 'L3', 'L4']
```

```
# make a dictionary of the targets' demands:
demand = df.loc['Bedarf', targets].to_dict()
demand
```

```
{'L1': 80.0, 'L2': 65.0, 'L3': 70.0, 'L4': 85.0}
```

```
# make a dictionary of the sources' production quantities:
production = df.loc[sources, "Produktion"].to_dict()
production
```

```
{'F1': 75.0, 'F2': 125.0, 'F3': 100.0}
```

```
# costs per unit as dataframe:
c = df.loc[sources, targets]
c
```

	L1	L2	L3	L4
F1	464	513	654	867
F2	352	416	690	791
F3	995	682	388	685

```
model = pyo.ConcreteModel()

model.S = pyo.Set(initialize=sources)  # set of sources (plants)
model.T = pyo.Set(initialize=targets)  # set of targets (warehouses)

if True:  # use integrality constraint:
    print("using integrality constraint")
    model.x = pyo.Var(model.S, model.T, domain=pyo.NonNegativeIntegers)
else:     # relax the integrality constraint:
    print("relaxing integrality constraint")
    model.x = pyo.Var(model.S, model.T, domain=pyo.NonNegativeReals)

model.cost = pyo.Objective(expr=
    sum(model.x[s, t]*c.loc[s, t] for s in model.S for t in model.T),
    sense=pyo.minimize)

@model.Constraint(model.S)
def demand_constraint(model, s):
    return sum(model.x[s, t] for t in model.T) == production[s]

@model.Constraint(model.T)
def production_constraint(model, t):
    return sum(model.x[s, t] for s in model.S) == demand[t]

# model.pprint()
```

```
using integrality constraint
```

```
solver = pyo.SolverFactory('cbc')
# solver = pyo.SolverFactory('glpk')
# solver = pyo.SolverFactory('appsi_highs')
# solver = pyo.SolverFactory('gurobi')

results = solver.solve(model, tee=False)
print(f"status = {results.solver.status}")

print(f"minimal cost = {pyo.value(model.cost):.2f} EUR")
```

```
status = ok
minimal cost = 152535.00 EUR
```

13.1.5. Ergebnisse

```
x_sol_dict = model.x.extract_values()
x_sol_dict
```

```
{('F1', 'L1'): 0.0,
 ('F1', 'L2'): 20.0,
 ('F1', 'L3'): 0.0,
 ('F1', 'L4'): 55.0,
 ('F2', 'L1'): 80.0,
 ('F2', 'L2'): 45.0,
 ('F2', 'L3'): 0.0,
 ('F2', 'L4'): 0.0,
 ('F3', 'L1'): 0.0,
 ('F3', 'L2'): 0.0,
 ('F3', 'L3'): 70.0,
 ('F3', 'L4'): 30.0}
```

```
x_sol_df = pd.DataFrame(index=sources, columns=targets)
for item in x_sol_dict:
    x_sol_df.loc[item[0], item[1]] = x_sol_dict[item]
x_sol_df
```

	L1	L2	L3	L4
F1	0.0	20.0	0.0	55.0
F2	80.0	45.0	0.0	0.0
F3	0.0	0.0	70.0	30.0

13.2. Übung: Northern Airplane

13.2.1. Problemstellung

Die (erfundene) Northern Airplane Company baut Verkehrsflugzeuge. Ein wichtiger Schritt im Produktionsprozess ist die Herstellung und Installation des Düsentriebwerks. Die Produktion der Düsentriebwerke muss für die nächsten vier Monate geplant werden. Die vertraglich vereinbarten Liefermengen, die maximalen Produktionsmengen und die Produktions- und Lagerkosten pro Triebwerk in Mio. EUR sind für die kommenden 4 Monate in der folgenden Tabelle angeführt:

Monat	Liefermenge	max. Produktionsmenge	Produktionskosten pro Stück	Lagerkosten pro Stück
1	10	25	1.08	0.015
2	15	35	1.11	0.015
3	25	30	1.10	0.015
4	20	10	1.13	0.015

Aufgrund der Schwankungen der Produktionskosten kann es sich lohnen, einige der Triebwerke einen oder mehrere Monate vor ihrem geplanten Einbau zu fertigen. Der Nachteil dabei ist, dass die Lagerung dieser Triebwerke zusätzliche Kosten mit sich bringt.

Es soll der gesamtkostenoptimale Produktionszeitplan über die vier Monate bestimmt werden. Aufgaben:

1. Modellieren Sie in einer Exceldatei das Optimierungsproblem als Transportproblem:
   - Verwenden Sie als Quellen die Produktionsmonate und als Ziele die Liefermonate.
   - Überprüfen Sie, ob die Summe der Produktionsmengen gleich der Summe der Liefermengen ist, und fügen Sie bei Bedarf einen Dummy-Knoten ein.
   - Verwenden Sie die "Big M Methode" (d. h. hier sehr große Stückkosten), um Verwendung von bestimmten Kanten zu vermeiden.
2. Implementieren Sie das Transportproblem.

14. Zuordnungsproblem

14.1. Karftwerksstandorte

14.1.1. Problemstellung

Drei Kraftwerkstypen (KT) mit der selben Nennleistung können an 4 verschiedenen Standorten (SO) errichtet werden. Von jedem Kraftwerkstyp soll genau ein Kraftwerk errichtet werden. An jedem Standort kann nur ein Kraftwerk errichtet werden. Die Betriebskosten unterscheiden sich je nach Kraftwerkstyp und Standort gemäß der folgenden Tabelle:

KT,SO	SO1	SO2	SO3	SO4
KT1	13	16	12	11
KT2	15	-	13	20
KT3	5	7	10	6

Der Kraftwerkstyp 2 ist für den Standort 2 ungeeignet. Welcher Kraftwerkstyp soll an welchem Standort errichtet werden, sodass die Gesamtkosten minimal sind?

Um dieses Problem als Zuordnungsproblem zu formulieren, müssen wir einen Dummy-Kraftwerkstyp KT4 einführen. Zudem werden extrem hohe Betriebskosten M (das ist der Trick der Big-M-Methode) für Kraftwerkstyp 2 am Standort 2 angesetzt, um diese Zuordnung in der optimalen Lösung zu verhindern. Das resultierende Zuordnungsproblem hat dann die Kostentabelle

KT,SO	SO1	SO2	SO3	SO4
KT1	13	16	12	11
KT2	15	M	13	20
KT3	5	7	10	6
KT4	0	0	0	0

Quelle: Hillier, Lieberman: Introduction to Operations Research. 10th edition, 2015. p. 348 ff.

14.1.2. Daten

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import pyomo.environ as pyo
```

Siehe Zuordnungsproblem_Beispiel.xlsx

```
# read cost table data from file:
df = pd.read_excel("daten/Zuordnungsproblem_Beispiel.xlsx", index_col=0)
df
```

	SO1	SO2	SO3	SO4
KT1	13	16	12	11
KT2	15	99999	13	20
KT3	5	7	10	6
KT4	0	0	0	0

14.1.3. Modellierung

Wir stellen die Daten des Zuordnungsproblems (analog zum Transportproblem) in einem ungerichteten Graphen dar, in dem jede Quelle (Kraftwerkstypen) mit jedem Ziel (Standorte) verbunden ist. Die Kantenkosten sind die Kosten der Zuordnung:

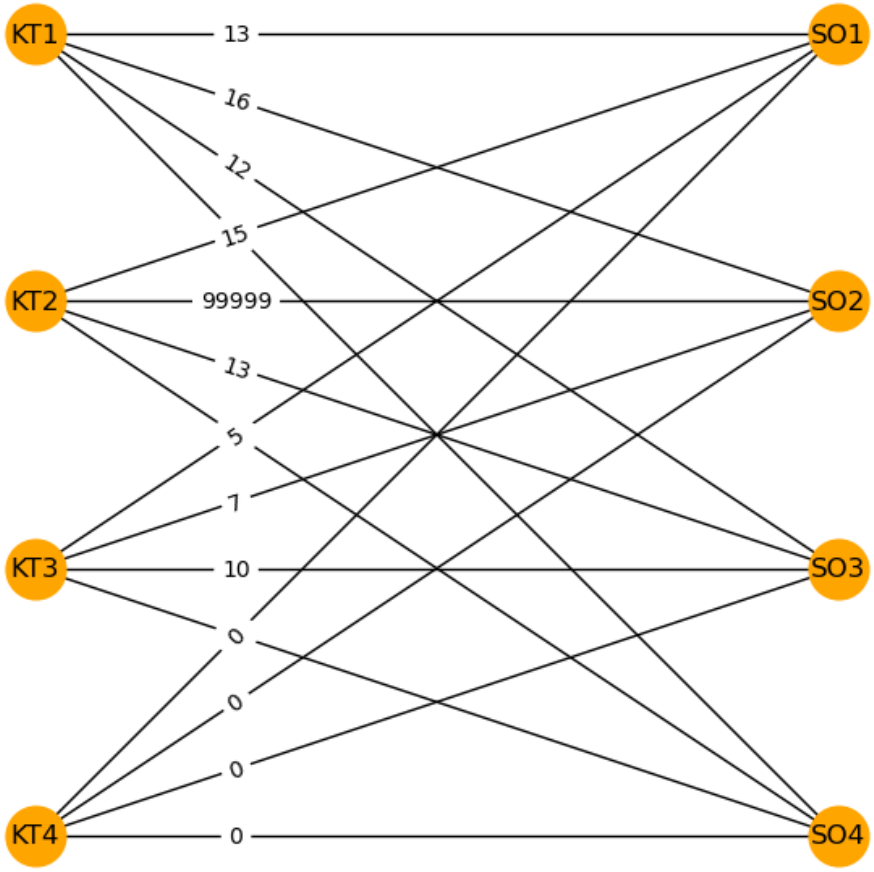


Abbildung 14.1.: Graph des Zuordnungsproblems

Für die Optimierung erhält jede Kante eine binäre Flussvariable, die angibt, ob die Quelle dem Ziel zugeordnet wird oder nicht. Die Zielfunktion ist die Summe der Zuordnungskosten der Kante multipliziert mit der zugehörigen binären Variablen. Die Nebenbedingungen fixieren, dass jede Quelle genau einem Ziel zugewiesen wird und jedes Ziel genau eine Quelle erhält.

14.1.4. Implementierung

```
# index and column names as lists:
sources = df.index.tolist()
targets = df.columns.tolist()
c = df.loc[sources, targets]
```

```
model = pyo.ConcreteModel()

model.S = pyo.Set(initialize=sources)  # set of sources (plants)
model.T = pyo.Set(initialize=targets)  # set of targets (warehouses)

if True:  # use integrality constraint:
    print("using integrality constraint")
    model.x = pyo.Var(model.S, model.T, domain=pyo.Binary)
else:     # relax the integrality constraint:
    print("relaxing integrality constraint")
    model.x = pyo.Var(model.S, model.T, domain=pyo.NonNegativeReals)

model.cost = pyo.Objective(expr=
    sum(model.x[s, t]*c.loc[s, t] for s in model.S for t in model.T),
    sense=pyo.minimize)

@model.Constraint(model.S)
def demand_constraint(model, s):
    return sum(model.x[s, t] for t in model.T) == 1

@model.Constraint(model.T)
def production_constraint(model, t):
    return sum(model.x[s, t] for s in model.S) == 1

# model.pprint()
```

```
using integrality constraint
```

```
solver = pyo.SolverFactory('cbc')
# solver = pyo.SolverFactory('glpk')
# solver = pyo.SolverFactory('appsi_highs')
# solver = pyo.SolverFactory('gurobi')

results = solver.solve(model, tee=False)
print(f"status = {results.solver.status}")

print(f"minimal cost = {pyo.value(model.cost):.2f} EUR")
```

```
status = ok
minimal cost = 29.00 EUR
```

14.1.5. Ergebnisse

```
x_sol_dict = model.x.extract_values()
x_sol_dict
```

```
{('KT1', 'SO1'): 0.0,
 ('KT1', 'SO2'): 0.0,
 ('KT1', 'SO3'): 0.0,
 ('KT1', 'SO4'): 1.0,
 ('KT2', 'SO1'): 0.0,
 ('KT2', 'SO2'): 0.0,
 ('KT2', 'SO3'): 1.0,
 ('KT2', 'SO4'): 0.0,
 ('KT3', 'SO1'): 1.0,
 ('KT3', 'SO2'): 0.0,
 ('KT3', 'SO3'): 0.0,
 ('KT3', 'SO4'): 0.0,
 ('KT4', 'SO1'): 0.0,
 ('KT4', 'SO2'): 1.0,
 ('KT4', 'SO3'): 0.0,
 ('KT4', 'SO4'): 0.0}
```

```
x_sol_df = pd.DataFrame(index=sources, columns=targets)
for item in x_sol_dict:
    x_sol_df.loc[item[0], item[1]] = x_sol_dict[item]
x_sol_df
```

	SO1	SO2	SO3	SO4
KT1	0.0	0.0	0.0	1.0
KT2	0.0	0.0	1.0	0.0
KT3	1.0	0.0	0.0	0.0
KT4	0.0	1.0	0.0	0.0

15. Umladeproblem

15.1. Erbsen in Dosen

15.1.1. Problemstellung

Wir betrachten wieder das Unternehmen EiD (Erbsen in Dosen), das Dosenerbsen in drei Fabriken F1, F2 und F3 produziert. Die produzierte Ware wird zu vier Lagerhäusern L1, L2, L3 und L4 transportiert. Das Unternehmen kann Kosten einsparen, indem es seinen eigenen Fuhrpark aufgibt und stattdessen Spediteure für den Transport der Erbsenkonserven einsetzt. Da kein einziges Speditionsunternehmen das gesamte Gebiet mit allen Konservenfabriken und Lagerhäusern bedient, müssen viele der Sendungen mindestens einmal auf einen anderen LKW umgeladen werden. Die möglichen Routen von einer Fabrik zu einem Lagerhaus können über andere Fabriken, vier Umschlagspunkte und andere Fabriken gehen.

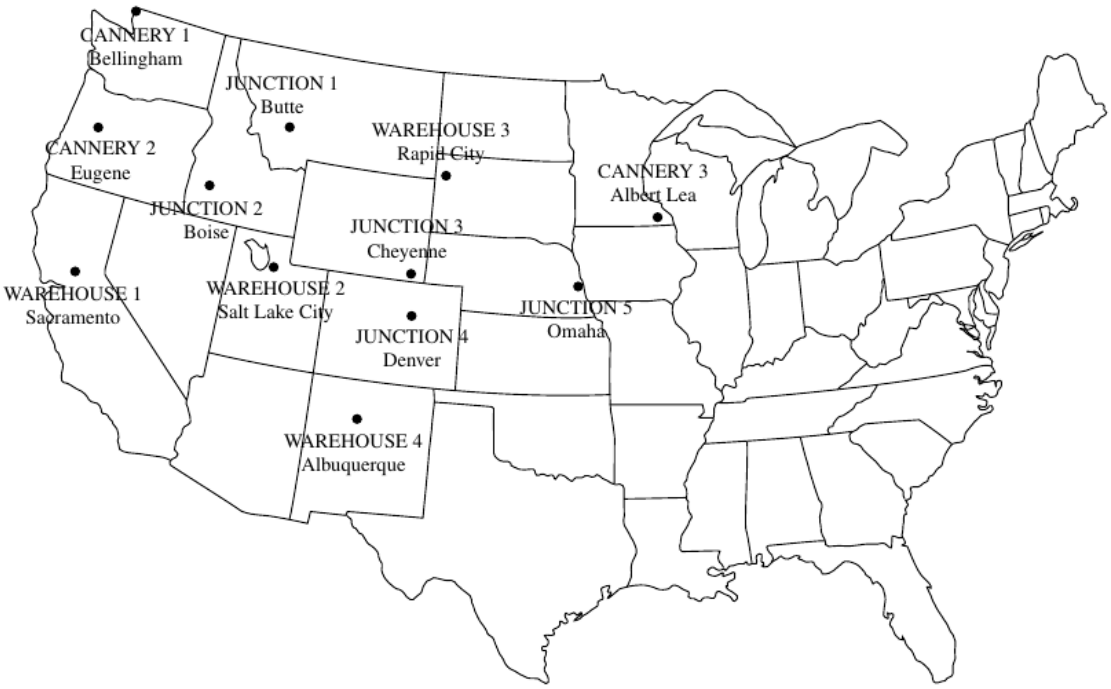


Abbildung 15.1.: Beispiel eines Umladeproblems. *Quelle:* Hillier, Lieberman: Introduction to Operations Research. 10th edition, 2015. Web Chapter 23-1, p. 23-4

Die Produktionsmengen der Fabriken, die Bedarfsmengen der Lagerhäuser und die Kosten des Transports pro LKW-Ladung sind in der Parametertabelle Umladeproblem_Beispiel.xlsx angeführt.

Gesucht sind die Routen und deren Warenmengen, sodass die Gesamtkosten minimal sind. Wir lösen dieses Umladeproblem und stellen die Lösung dar.

Quelle: Hillier, Lieberman: Introduction to Operations Research. 10th edition, 2015. Web Chapter 23-1, p. 23-3 ff.

15.1.2. Daten

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import pyomo.environ as pyo
```

```
# read cost table data from file:
df = pd.read_excel("daten/Umladeproblem_Beispiel.xlsx", index_col=0)
df
```

from\to	S1	S2	S3	J1	J2	J3	J4	J5	T1	T2	T3	T4	production
S1	NaN	146.0	NaN	324.0	286.0	NaN	NaN	NaN	452.0	505.0	NaN	871.0	75.0
S2	146.0	NaN	NaN	373.0	212.0	570.0	609.0	NaN	335.0	407.0	688.0	784.0	125.0
S3	NaN	NaN	NaN	658.0	NaN	405.0	419.0	158.0	NaN	685.0	359.0	673.0	100.0
J1	322.0	371.0	656.0	NaN	262.0	398.0	430.0	NaN	503.0	234.0	329.0	NaN	NaN
J2	284.0	210.0	NaN	262.0	NaN	406.0	421.0	644.0	305.0	207.0	464.0	558.0	NaN
J3	NaN	569.0	403.0	398.0	406.0	NaN	81.0	272.0	597.0	253.0	171.0	282.0	NaN
J4	NaN	608.0	418.0	431.0	422.0	81.0	NaN	287.0	613.0	280.0	236.0	229.0	NaN
J5	NaN	NaN	158.0	NaN	647.0	274.0	288.0	NaN	831.0	501.0	293.0	482.0	NaN
T1	453.0	336.0	NaN	505.0	307.0	599.0	615.0	831.0	NaN	359.0	706.0	587.0	NaN
T2	505.0	407.0	683.0	235.0	208.0	254.0	281.0	500.0	357.0	NaN	362.0	341.0	NaN
T3	NaN	687.0	357.0	329.0	464.0	171.0	236.0	290.0	705.0	362.0	NaN	457.0	NaN
T4	868.0	781.0	670.0	NaN	558.0	282.0	229.0	480.0	587.0	340.0	457.0	NaN	NaN
demand	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	80.0	65.0	70.0	85.0	NaN

15.1.3. Modellierung

Wir stellen die Daten des Umladeproblems in einem gerichteten Graphen dar, in dem die Fabriken (Quellen, sources), Umschlagspunkte (junctions) und Lagerhäuser (Ziele, targets) Knoten sind und die gerichteten Kanten die möglichen Transportwege angeben. Die Kanten haben dabei die Transportkosten pro Mengeneinheit als Gewicht:

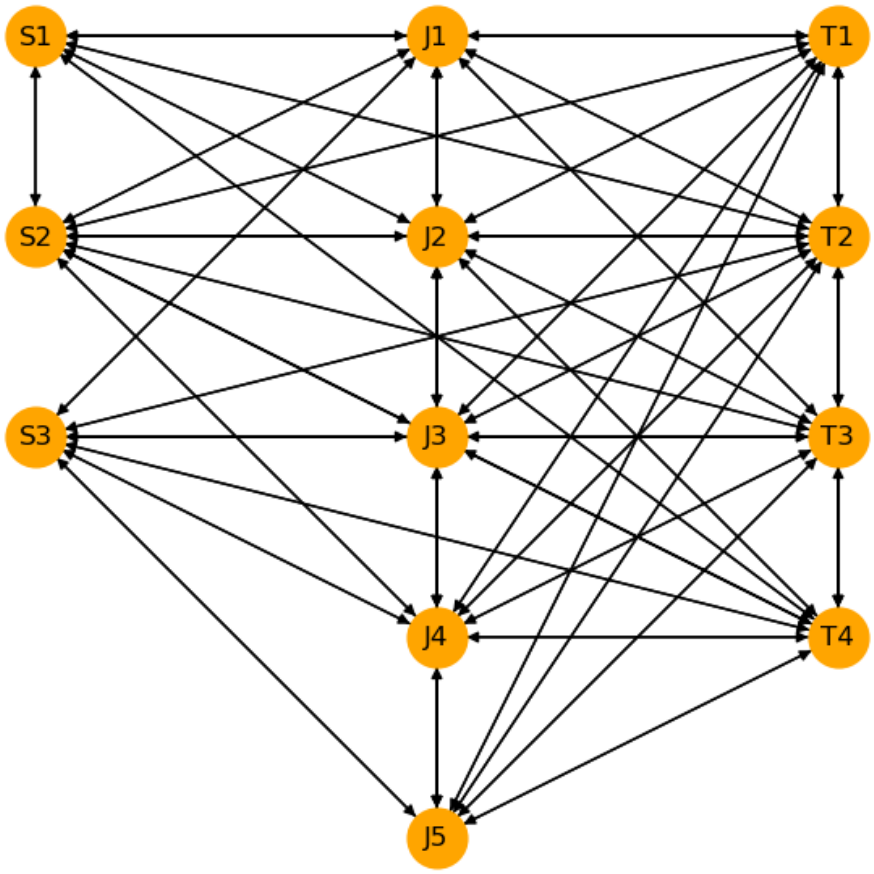


Abbildung 15.2.: Graph des Umladeproblems

Für die Optimierung erhält jede Kante eine Flussvariable, die die transportierte Menge über die Kante angibt. Die Zielfunktion ist die Summe der Transportkosten pro Mengeneinheit multipliziert mit der zugehörigen transportierten Menge. Die Nebenbedingungen fixieren die Bedarfsmengen der Lagerhäuser, die Produktionsmengen der Fabriken und die Mengenbilanzen der Umschlagspunkte.

15.1.4. Implementierung

```
sources = df.index[:3].tolist()
junctions = df.index[3:8].tolist()
targets = df.index[8:12].tolist()
nodes = sources + junctions + targets

demand = df.loc['demand', targets].to_dict()
production = df.loc[sources, "production"].to_dict()

# costs per unit as pandas data frame:
c = df.loc[nodes, nodes]  # includes NaN entries
```

```
# print(c)

# edges as list of tuples:
edges = [(n, m) for n in nodes for m in nodes if not np.isnan(c.loc[n, m])]
# print(edges)
```

```
model = pyo.ConcreteModel()

model.S = pyo.Set(initialize=sources)    # set of sources
model.J = pyo.Set(initialize=junctions)  # set of junctions
model.T = pyo.Set(initialize=targets)    # set of targets
model.N = pyo.Set(initialize=nodes)      # set of nodes
model.E = pyo.Set(initialize=edges)      # set of edges

if True:  # use integrality constraint:
    print("using integrality constraint")
    model.x = pyo.Var(model.E, domain=pyo.NonNegativeIntegers)
else:     # relax the integrality constraint:
    print("relaxing integrality constraint")
    model.x = pyo.Var(model.E, domain=pyo.NonNegativeReals)

model.cost = pyo.Objective(expr=
    sum(model.x[n, m]*c.loc[n, m] for (n, m) in model.E),
    sense=pyo.minimize)

@model.Constraint(model.S)
def production_constraint(model, s):
    return sum(model.x[s, n] for n in model.N if (s, n) in model.E) - \
           sum(model.x[n, s] for n in model.N if (n, s) in model.E) == \
           production[s]

@model.Constraint(model.T)
def demand_constraint(model, t):
    return sum(model.x[n, t] for n in model.N if (n, t) in model.E) - \
           sum(model.x[t, n] for n in model.N if (t, n) in model.E) == \
           demand[t]

@model.Constraint(model.J)
def junction_constraint(model, j):
    return sum(model.x[j, n] for n in model.N if (j, n) in model.E) - \
           sum(model.x[n, j] for n in model.N if (n, j) in model.E) == 0

# model.pprint()
```

```
using integrality constraint
```

```
solver = pyo.SolverFactory('cbc')
# solver = pyo.SolverFactory('glpk')
# solver = pyo.SolverFactory('appsi_highs')
# solver = pyo.SolverFactory('gurobi')

results = solver.solve(model, tee=False)
print(f"status = {results.solver.status}")

print(f"minimal cost = {pyo.value(model.cost):.2f} EUR")
```

```
status = ok
minimal cost = 145175.00 EUR
```

15.1.5. Ergebnisse

```python
x_sol_dict = model.x.extract_values()
# x_sol_dict

# formatted print out of solution:
for edge in edges:
    if x_sol_dict[edge] > 0:
        print(f"flow on edge {edge} = {x_sol_dict[edge]}.")
```

```
flow on edge ('S1', 'J2') = 75.0.
flow on edge ('S2', 'T1') = 80.0.
flow on edge ('S2', 'T2') = 45.0.
flow on edge ('S3', 'J5') = 30.0.
flow on edge ('S3', 'T3') = 70.0.
flow on edge ('J2', 'T2') = 75.0.
flow on edge ('J5', 'T4') = 30.0.
flow on edge ('T2', 'T4') = 55.0.
```

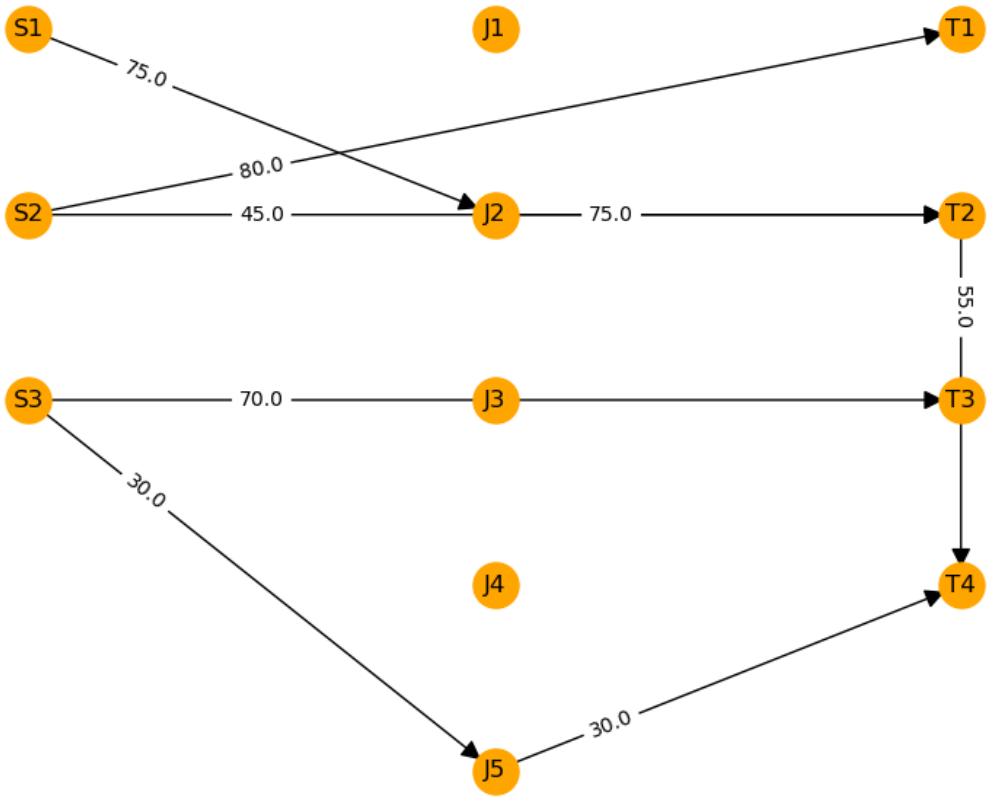


Abbildung 15.3.: Optimale Lösung des Umladeproblems

```
x_sol_df = pd.DataFrame(index=nodes, columns=nodes)
for item in x_sol_dict:
    x_sol_df.loc[item[0], item[1]] = x_sol_dict[item]
x_sol_df
```

	S1	S2	S3	J1	J2	J3	J4	J5	T1	T2	T3	T4
S1	NaN	0.0	NaN	0.0	75.0	NaN	NaN	NaN	0.0	0.0	NaN	0.0
S2	0.0	NaN	NaN	0.0	0.0	0.0	0.0	NaN	80.0	45.0	0.0	0.0
S3	NaN	NaN	NaN	0.0	NaN	0.0	0.0	30.0	NaN	0.0	70.0	0.0
J1	0.0	0.0	0.0	NaN	0.0	0.0	0.0	NaN	0.0	0.0	0.0	NaN
J2	0.0	0.0	NaN	0.0	NaN	0.0	0.0	0.0	0.0	75.0	0.0	0.0
J3	NaN	0.0	0.0	0.0	0.0	NaN	0.0	0.0	0.0	0.0	0.0	0.0
J4	NaN	0.0	0.0	0.0	0.0	0.0	NaN	0.0	0.0	0.0	0.0	0.0
J5	NaN	NaN	0.0	NaN	0.0	0.0	0.0	NaN	0.0	0.0	0.0	30.0
T1	0.0	0.0	NaN	0.0	0.0	0.0	0.0	0.0	NaN	0.0	0.0	0.0
T2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	NaN	0.0	55.0
T3	NaN	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	NaN	0.0
T4	0.0	0.0	0.0	NaN	0.0	0.0	0.0	0.0	0.0	0.0	0.0	NaN

```
# compare with the problem data:
df
```

from\to	S1	S2	S3	J1	J2	J3	J4	J5	T1	T2	T3	T4	production
S1	NaN	146.0	NaN	324.0	286.0	NaN	NaN	NaN	452.0	505.0	NaN	871.0	75.0
S2	146.0	NaN	NaN	373.0	212.0	570.0	609.0	NaN	335.0	407.0	688.0	784.0	125.0
S3	NaN	NaN	NaN	658.0	NaN	405.0	419.0	158.0	NaN	685.0	359.0	673.0	100.0
J1	322.0	371.0	656.0	NaN	262.0	398.0	430.0	NaN	503.0	234.0	329.0	NaN	NaN
J2	284.0	210.0	NaN	262.0	NaN	406.0	421.0	644.0	305.0	207.0	464.0	558.0	NaN
J3	NaN	569.0	403.0	398.0	406.0	NaN	81.0	272.0	597.0	253.0	171.0	282.0	NaN
J4	NaN	608.0	418.0	431.0	422.0	81.0	NaN	287.0	613.0	280.0	236.0	229.0	NaN
J5	NaN	NaN	158.0	NaN	647.0	274.0	288.0	NaN	831.0	501.0	293.0	482.0	NaN
T1	453.0	336.0	NaN	505.0	307.0	599.0	615.0	831.0	NaN	359.0	706.0	587.0	NaN
T2	505.0	407.0	683.0	235.0	208.0	254.0	281.0	500.0	357.0	NaN	362.0	341.0	NaN
T3	NaN	687.0	357.0	329.0	464.0	171.0	236.0	290.0	705.0	362.0	NaN	457.0	NaN
T4	868.0	781.0	670.0	NaN	558.0	282.0	229.0	480.0	587.0	340.0	457.0	NaN	NaN
demand	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	80.0	65.0	70.0	85.0	NaN

16. Maximaler Fluss

16.1. Wasserversorgung

16.1.1. Problemstellung

Wir betrachten ein System von Aquädukten, die Wasser von drei Flüssen (Knoten F1, F2 und F3) in eine Großstadt (Knoten T) transportieren. Die anderen Knoten sind Verbindungspunkte im Wasserversorgungssystem der Stadt. Jeder Aquädukt hat eine gewisse Transportkapazität in Tausend m^3 Wasser pro Tag. Die Stadt möchte einen Durchflussplan bestimmen, der den Wasserdurchfluss zur Stadt maximiert, siehe Abbildung Abbildung 16.1.

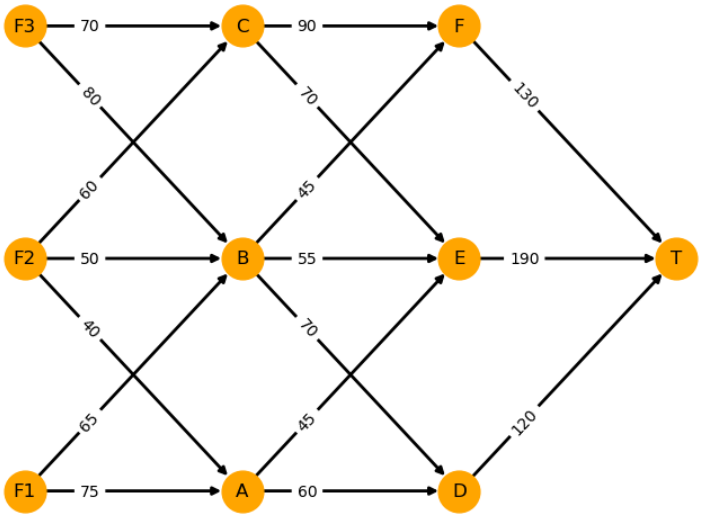


Abbildung 16.1.: Wasserversorgung

Quelle: Hillier, Lieberman: Introduction to Operations Research. 10th edition, 2015.Hillier, Lieberman: Introduction to Operations Research. 10th edition, 2015. Problem 10.5-3, Seite 428 f.

16.1.2. Modellierung

Um dieses Problem als ein Maximum Flow Problem zu formulieren, fügen wir dem gerichteten Graphen noch eine Quelle S hinzu, die mit den Knoten F1, F2 und F3 verbunden ist. Als Kantenkapazitäten verwenden wir die gesamten Wassermengen der Flüsse, siehe Abbildung Abbildung 16.2.

Wir formulieren das Maximum Flow Problem auf dem Graphen mit Kantenmenge E und Knotenmenge N:

- Entscheidungsvariablen: Fluss x_e durch Kante $e \in E$
- Zielfunktion: Maximiere $x_{D,T} + x_{E,T} + x_{F,T}$ oder äquivalent $x_{S,F1} + x_{S,F2} + x_{S,F3}$
- Nebenbedingungen:
  - Flussbilanz für jeden Knoten außer Quelle und Senke

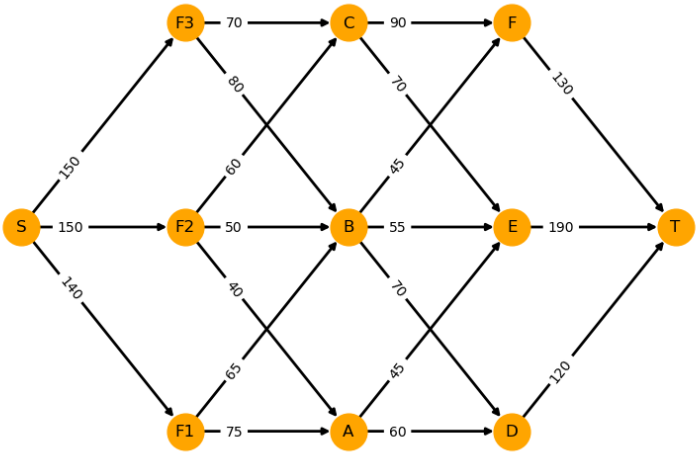


Abbildung 16.2.: Wasserversorgung mit Quelle

– Kapazitätsbeschränkungen für jede Kante

16.1.3. Daten

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import pyomo.environ as pyo
```

```
capacity = {
    ("S", "F1"): 140,
    ("S", "F2"): 150,
    ("S", "F3"): 150,
    ("F1", "A"): 75,
    ("F1", "B"): 65,
    ("F2", "A"): 40,
    ("F2", "B"): 50,
    ("F2", "C"): 60,
    ("F3", "B"): 80,
    ("F3", "C"): 70,
    ("A",  "D"): 60,
    ("A",  "E"): 45,
    ("B",  "D"): 70,
    ("B",  "E"): 55,
    ("B",  "F"): 45,
    ("C",  "E"): 70,
    ("C",  "F"): 90,
    ("D",  "T"): 120,
    ("E",  "T"): 190,
    ("F",  "T"): 130}
```

16.1.4. Implementierung

```
nodes = ["S", "F1", "F2", "F3", "A", "B", "C", "D", "E", "F", "T"]
junctions  = ["F1", "F2", "F3", "A", "B", "C", "D", "E", "F"]
edges = list(capacity.keys())
edges
```

```
[('S', 'F1'),
 ('S', 'F2'),
 ('S', 'F3'),
 ('F1', 'A'),
 ('F1', 'B'),
 ('F2', 'A'),
 ('F2', 'B'),
 ('F2', 'C'),
 ('F3', 'B'),
 ('F3', 'C'),
 ('A', 'D'),
 ('A', 'E'),
 ('B', 'D'),
 ('B', 'E'),
 ('B', 'F'),
 ('C', 'E'),
 ('C', 'F'),
 ('D', 'T'),
 ('E', 'T'),
 ('F', 'T')]
```

```
model = pyo.ConcreteModel()

model.N = pyo.Set(initialize=nodes)      # set of nodes
model.J = pyo.Set(initialize=junctions)  # set of junctions
model.E = pyo.Set(initialize=edges)      # set of edges

model.x = pyo.Var(model.E, domain=pyo.NonNegativeReals)

@model.Constraint(model.E)
def flow_upper_bound(model, *e):
    # Note: The asterisk in *e is needed to
    # pass the input argument e as a tuple
    return model.x[e] <= capacity[e]

model.flow = pyo.Objective(expr=
    sum(model.x["S", n] for n in ["F1", "F2", "F3"]),
    sense=pyo.maximize)

@model.Constraint(model.J)
def node_constraint(model, j):
    return sum(model.x[j, n] for n in model.N if (j, n) in model.E) - \
           sum(model.x[n, j] for n in model.N if (n, j) in model.E) == 0

# model.pprint()
```

```
solver = pyo.SolverFactory('cbc')
# solver = pyo.SolverFactory('glpk')
# solver = pyo.SolverFactory('appsi_highs')
# solver = pyo.SolverFactory('gurobi')

results = solver.solve(model, tee=False)
print(f"status = {results.solver.status}")

print(f"maximum flow = {pyo.value(model.flow):.2f} qm/d")
```

```
status = ok
maximum flow = 395.00 qm/d
```

16.1.5. Ergebnisse

```
x_sol_dict = model.x.extract_values()
# x_sol_dict

# formatted print out of solution:
for edge in edges:
    print(f"flow on edge {edge} = {x_sol_dict[edge]:.2f} qm/d")
```

```
flow on edge ('S', 'F1') = 140.00 qm/d
flow on edge ('S', 'F2') = 130.00 qm/d
flow on edge ('S', 'F3') = 125.00 qm/d
flow on edge ('F1', 'A') = 75.00 qm/d
flow on edge ('F1', 'B') = 65.00 qm/d
flow on edge ('F2', 'A') = 20.00 qm/d
flow on edge ('F2', 'B') = 50.00 qm/d
flow on edge ('F2', 'C') = 60.00 qm/d
flow on edge ('F3', 'B') = 55.00 qm/d
flow on edge ('F3', 'C') = 70.00 qm/d
flow on edge ('A', 'D') = 50.00 qm/d
flow on edge ('A', 'E') = 45.00 qm/d
flow on edge ('B', 'D') = 70.00 qm/d
flow on edge ('B', 'E') = 55.00 qm/d
flow on edge ('B', 'F') = 45.00 qm/d
flow on edge ('C', 'E') = 70.00 qm/d
flow on edge ('C', 'F') = 60.00 qm/d
flow on edge ('D', 'T') = 120.00 qm/d
flow on edge ('E', 'T') = 170.00 qm/d
flow on edge ('F', 'T') = 105.00 qm/d
```

17. Kürzester Weg

17.1. LP-Formulierung

17.1.1. Problemstellung

Durch die Daten unten ist der ungerichtete Graph der Abbildung Abbildung 17.1 gegeben, dessen Kantengewichte die Strecken zwischen den Endknoten der Kanten angegeben hat. Wir bestimmen mittels der LP-Formulierung die kürzesten Wege vom Konten S zu allen anderen Knoten.

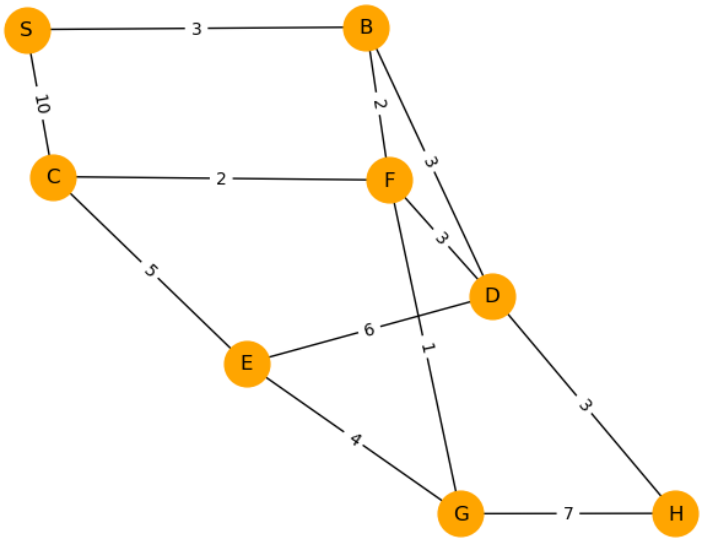


Abbildung 17.1.: Ungerichteter Graph

17.1.2. Daten

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import pyomo.environ as pyo
```

```
weight = {
    ('S', 'B'): 3,
    ('S', 'C'): 10,
    ('B', 'D'): 3,
    ('D', 'E'): 6,
    ('D', 'F'): 3,
    ('E', 'C'): 5,
```

```
    ('B', 'F'): 2,
    ('C', 'F'): 2,
    ('E', 'G'): 4,
    ('G', 'H'): 7,
    ('F', 'G'): 1,
    ('D', 'H'): 3}
```

17.1.3. Modellierung

Wie im Abschnitt Kürzester Weg beschrieben formulieren wir den ungerichteten Graphen um zu einem äquivalenten gerichteten Graphen mit den Kantengewichten w_{ij} auf den Kanten zwischen den Knoten i und j. Die Abbildung Abbildung 17.2 zeigt den zugehörigen gerichteten Graphen.

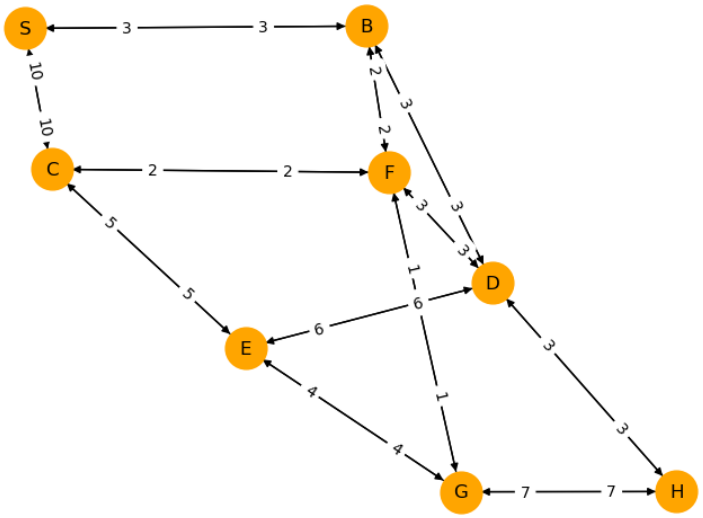


Abbildung 17.2.: Gerichter Graph

17.1.4. Implementierung

```
nodes = ['S', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
weight_reverse = {(j, i): v for (i, j), v in weight.items()}
weight_directed = weight | weight_reverse # union of the two dictionaries
edges =  list(weight_directed.keys())

ext_flow = {node: -1 for node in nodes}
ext_flow['S'] = len(nodes) - 1
```

```
model = pyo.ConcreteModel()

model.N = pyo.Set(initialize=nodes)      # set of nodes
model.E = pyo.Set(initialize=edges)      # set of edges

model.x = pyo.Var(model.E, domain=pyo.NonNegativeReals)
```

```
model.obj = pyo.Objective(expr=
    sum(weight_directed[i, j]*model.x[i, j] for (i, j) in edges),
    sense=pyo.minimize)

@model.Constraint(model.N)
def node_balance(model, i):
    return sum(model.x[i, n] for n in model.N if (i, n) in model.E) - \
           sum(model.x[n, i] for n in model.N if (n, i) in model.E) == \
           ext_flow[i]
```

```
solver = pyo.SolverFactory('cbc')
# solver = pyo.SolverFactory('glpk')
# solver = pyo.SolverFactory('appsi_highs')
# solver = pyo.SolverFactory('gurobi')

results = solver.solve(model, tee=False)
print(f"status = {results.solver.status}")

print(f"minimum objective value = {pyo.value(model.obj):.2f}")
```

```
status = ok
minimum objective value = 46.00
```

17.1.5. Ergebnisse

```
x_sol_dict = model.x.extract_values()
# x_sol_dict

# formatted print out of solution:
for edge in edges:
    if x_sol_dict[edge] > 0:
        print(f"flow on edge {edge} = {x_sol_dict[edge]:.2f}")
```

```
flow on edge ('S', 'B') = 7.00
flow on edge ('B', 'D') = 2.00
flow on edge ('B', 'F') = 4.00
flow on edge ('F', 'G') = 2.00
flow on edge ('D', 'H') = 1.00
flow on edge ('F', 'C') = 1.00
flow on edge ('G', 'E') = 1.00
```

In Abbildung Abbildung 17.3 ist die Lösung des Problems dargestellt.

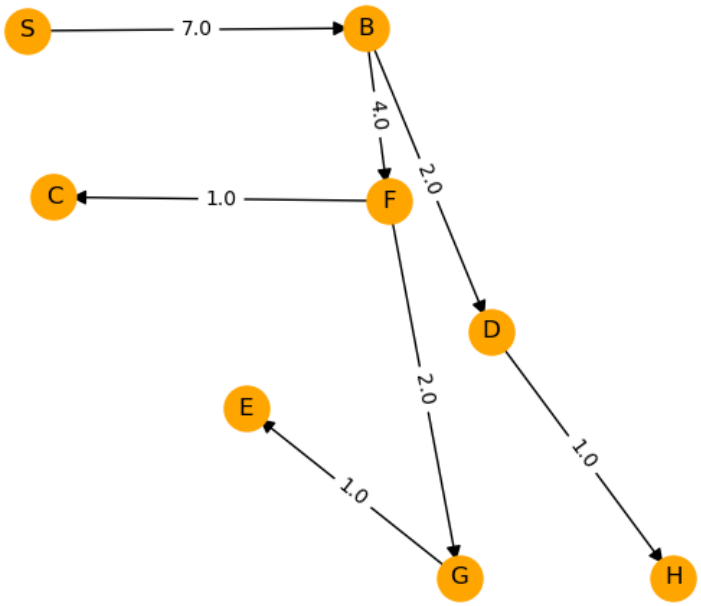


Abbildung 17.3.: Lösung

Teil IV.

Aufgaben

18. Aufgaben 1

Die Aufgaben müssen in ILIAS als lauffähige und dokumentierte Jupyter Notebooks inkl. Datenfiles abgegeben werden. LP-Modellierungen können als eingescannte PDFs abgegeben werden.

18.1. Aufgabe 1: Pyomo (10 Punkte)

Lösen Sie folgendes LP mit Pyomo, d. h.:

1. Bestimmen Sie die Entscheidungsvariablen inklusive Einheiten, Datentyp und Schranken.
2. Formulieren Sie die Zielfunktion und die Nebenbedingungen.
3. Modellieren Sie das LP mit Pyomo.
4. Lösen Sie das LP mit verschiedenen Solvern, und geben Sie die Lösung formatiert aus.
5. Interpretieren Sie die Lösung. Ist sie plausibel? Ändern Sie z. B. die Daten, um Ihre Lösung an einfachen Spezialfällen zu überprüfen.

Schichtenproblem: In einem Servicezentrum werden in den folgenden 4-Stunden-Schichten mindestens die angegebenen Arbeitskräfte benötigt:

Schicht	Arbeitskräfte min.
1 - 5 Uhr	2
5 - 9 Uhr	4
9 - 13 Uhr	7
13 - 17 Uhr	5
17 - 21 Uhr	2
21 - 1 Uhr	2

Die Arbeitszeit einer Arbeitskraft innerhalb von 24 Stunden beträgt 8 Stunden am Stück. Gesucht ist ein 24-Stunden-Schichtplan, der möglichst wenige Arbeitskräfte erfordert.

- Implementieren Sie das LP in Pyomo
    1. mit einzeln generierten Variablen, vgl. das Beispiel Butter und Eiscreme.
    2. mit einer Indexmenge, die die Variablen indiziert, vgl. das Beispiel Biomasse(heiz)kraftwerk.
- Stellen Sie die optimale Lösung grafisch dar.
- Liefern unterschiedliche Solver dieselbe Lösung?

19. Aufgaben 2

Die Aufgaben müssen in ILIAS als lauffähige und dokumentierte Jupyter Notebooks inkl. Datenfiles abgegeben werden. LP-Modellierungen können als eingescannte PDFs abgegeben werden.

19.1. Aufgabe 1: Zeitreihen (4 Punkte)

1. Gehen Sie auf die Webseite PVGIS, wählen Sie in der Landkarte einen sinnvollen Standort aus, und laden Sie im CSV-Format die PV Power für das Jahr 2020 für eine 5 kWp Anlage herunter. Laden Sie die Datei mit Hilfe des Codes unten in ein DataFrame, und stellen Sie die PV Power inkl. Achsenbeschriftung grafisch dar. Erklären Sie den angegebenen Code.
2. Registrieren Sie sich auf der Webseite Entsoe, und laden Sie anschließend unter `Load` die Last `Total Load - Day Ahead / Actual` für das Jahr 2020 für ein (zu dem von Ihnen gewählten PVGIS-Standort passendes) Land im CSV-Format herunter. Laden Sie die Datei mit Hilfe des Codes unten in ein DataFrame, und stellen Sie die tatsächliche (actual) Last grafisch inkl. Achsenbeschriftung dar. Erklären Sie den angegebenen Code. Die Entsoe-Seite hat ein hilfreiches Glossar.
3. Skalieren sie den tatsächlichen (actual) Jahreslastgang auf einen typischen elektrischen Jahresgesamtverbrauch eines Haushalts.
4. Stellen Sie die PV-Leistung und die Last in einem gemeinsamen Diagramm dar.
5. Berechnen Sie für den Tag 2020-08-01
    1. den *Eigenverbrauchsanteil* der PV-Anlage. Dieser gibt an, wie viel der selbst produzierten Energie man selbst verbraucht hat.
    2. den *Autarkiegrad* der PV-Anlage. Dieser gibt an, wie viel der benötigten Energie von der eigenen PV-Anlage gedeckt wurde.

```
import numpy as np
import pandas as pd

path = "/home/kr/lehre/effsys/2_en/notes/daten/"
```

```
# PV data:
file = "Timeseries_47.245_9.642_SA2_5kWp_crystSi_14_40deg_-6deg_2020_2020.csv"
my_parser = lambda date: pd.to_datetime(date, format='%Y%m%d:%H%M')
# cf. https://docs.python.org/3/library/datetime.html#strftime-and-strptime-format-codes
pv = pd.read_csv(path + file, skiprows=10, header=0, index_col=0,
                 parse_dates=True, date_parser=my_parser, nrows=8795-11)
pv.index = pv.index- pd.Timedelta(minutes=10)
pv.head(3)  # column P: PV system power (W)
```

time	P	G(i)	H_sun	T2m	WS10m	Int
2020-01-01 00:00:00	0.0	0.0	0.0	-9.57	1.66	0.0
2020-01-01 01:00:00	0.0	0.0	0.0	-9.65	1.59	0.0
2020-01-01 02:00:00	0.0	0.0	0.0	-10.11	1.59	0.0

```
# price data:
file = "Total Load - Day Ahead _ Actual_202001010000-202101010000.csv"
my_parser = lambda date: pd.to_datetime(date[:16], format='%d.%m.%Y %H:%M')
load = pd.read_csv(path + file, index_col=0, date_parser=my_parser)
# rename columns "Day-ahead Total Load Forecast [MW] - BZN|AT" and "Actual Total Load [MW] - BZN|AT":
load.columns = ['forecast', 'actual']
load.index.name = 'time'
load.head(3)
```

time	forecast	actual
2020-01-01 00:00:00	5752.0	5967.0
2020-01-01 01:00:00	5467.0	5768.0
2020-01-01 02:00:00	5360.0	5592.0

19.2. Aufgabe 2: E-Mobil-Flotte (6 Punkte)

Sie betreiben eine kleine E-Mobil-Flotte mit zwei E-Mobilen und folgenden Spezifikationen:

- E-Mobil A: 200 kWh Batterie, maximale Ladeleistung 75 kW, Verbrauch 1.2 kWh/km
- E-Mobil B: 300 kWh Batterie, maximale Ladeleistung 125 kW, Verbrauch 1.4 kWh/km

Die E-Mobile können in Ihrem Depot an einer Ladestation mit zwei Ladepunkten geladen werden. Die Ladestation hat in Summe maximal 150 kW Ladeleistung.

Die E-Mobile sind morgens um 8:00 Uhr zu 80 % geladen und sollen bis Mitternacht wieder auf mindestens 60 % geladen werden. Die E-Mobile absolvieren folgende Fahrten, die alle im Depot beginnen und enden:

- E-Mobil A:
  - von 10 Uhr bis 12 Uhr, 120 km
  - von 16 Uhr bis 18 Uhr, 80 km
- E-Mobil B:
  - von 9 Uhr bis 13 Uhr, 150 km
  - von 15 Uhr bis 20 Uhr, 90 km

In den Zeiten, in denen die E-Mobile nicht fahren, können sie im Depot geladen werden. Sie modellieren das Laden verlustfrei und wollen einen optimalen Ladeplan in Stundenschritten für die E-Mobile erstellen, der die Spitzenladeleistung der Ladestation minimiert. Verwenden Sie für die Modellierung die folgenden Variablen:

- a_j: Ladeleistung in kW des E-Mobils A in der Stunde mit Index j
- b_j: Ladeleistung in kW des E-Mobils B in der Stunde mit Index j

Die Minimierung der Spitzenleistung kann durch Minimierung einer zusätzlichen Variablen $m > 0$ in kW in Kombination mit folgenden zusätzlichen Nebenbedingung modelliert werden:

$$m \geq a_j + b_j \quad \forall j.$$

1. Definieren Sie die Zeitpunkte, Zeitperioden und ihre Indizierung.
2. Modellieren Sie das Energienetzwerk und fixieren Sie die Pfeilrichtung der Kanten.
3. Basierend auf dieser Vorzeichenkonvention formulieren Sie das Optimierungsproblem:
   1. Definieren Sie die Entscheidungsvariablen inklusive Einheiten, Datentyp und Schranken.
   2. Formulieren Sie die Zielfunktion.
   3. Formulieren Sie die die Nebenbedingungen.

4. Modellieren Sie das LP mit Pyomo, und lösen Sie es mit verschiedenen Solvern.
5. Stellen Sie die zeitlichen Verläufe der E-Mobil-Ladeleistungen, der E-Mobil-Ladestände und des Lastgangs der Ladestation grafisch dar, und interpretieren Sie diese.

20. Aufgaben 3

Die Aufgaben müssen in ILIAS als lauffähige und dokumentierte Jupyter Notebooks inkl. Datenfiles abgegeben werden. LP-Modellierungen können als eingescannte PDFs abgegeben werden.

20.1. Aufgabe 1: Arbitrage (6 Punkte)

Sie verwenden eine Batterie, um am Energiemarkt Arbitrage zu betreiben. Arbitrage bedeutet in diesem Zusammenhang, dass Sie die Batterie zu Zeiten mit tiefen Preisen laden und zu Zeiten mit hohen Preisen entladen. Danach hat die Batterie wieder den anfänglichen Ladestand. Sie fragen sich, wie groß der Preisunterschied sein muss, damit Sie bei gegebenen Lade- und Entladeverlusten einen Gewinn erzielen.

1. Beantworten Sie die Frage allgemein, indem Sie eine Formel herleiten, die folgende Parameter enthält:

   - η_{in}, η_{out}: Lade- und Entladewirkungsgrad
   - c_{low}: niedriger Preis
   - c_{high}: hoher Preis

   Erläutern Sie die Formel in Worten.

2. Überprüfen Sie Ihre Formel durch folgende Simulation: $\eta_{\text{in}} = \eta_{\text{out}} = 0.9$, $c_{\text{low}} = 0.1$. Fixieren Sie die restlichen Parameter für eine Batterie Ihrer Wahl. Wählen Sie 50 Werte für c_{high} zwischen 0.1 EUR/kWh und 0.15 EUR/kWh, und berechnen Sie für jeden Wert von c_{high} über ein LP den optimalen Gewinn über zwei Zeitperioden mit den Marktpreisen c_{low} und c_{high}. Bestimmen Sie, ab welchem der Werte von c_{high} Sie Gewinn machen, und vergleichen Sie dies mit Ihrer Formel.

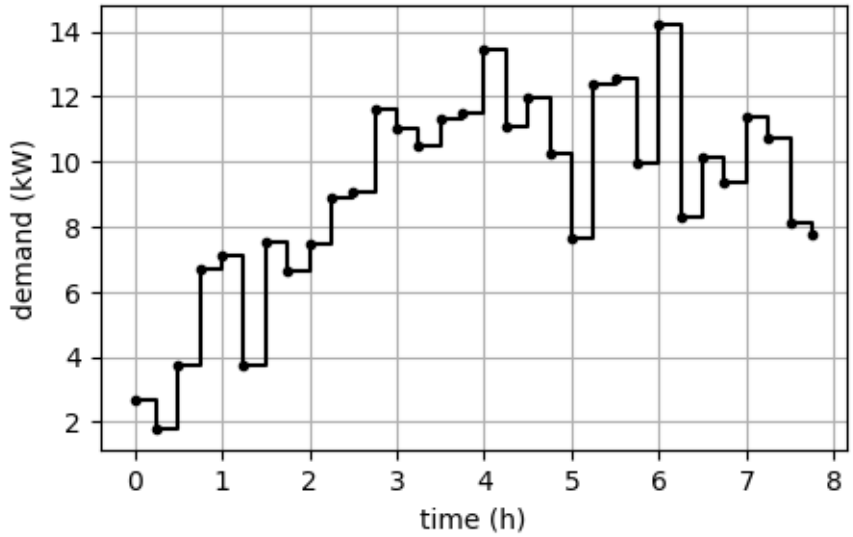
20.2. Aufgabe 2: Peak Power Variation (4 Punkte)

20.2.0.1. Problemstellung

Wir betrachten eine ähnliche Situation wie im Beispiel Batterieverluste, nur dass die Batterie kleiner und der Verbrauch größer ist:

```
import numpy as np
import matplotlib.pyplot as plt

# time:
dt = 0.25  # h
times = np.arange(start=0, stop=8 + dt, step=dt)
periods = np.arange(start=0, stop=8, step=dt)
n = len(periods)
time_indices   = range(n + 1)  # 0, 1, ..., n - 1, n
period_indices = range(n)      # 0, 1, ..., n - 1

# demand:
np.random.seed(0)
noise = np.random.normal(loc=0, scale=1.5, size=len(periods))
demand = 8 + 0.8*periods - 0.5*(periods - 4)**2 + noise

plt.figure(figsize=(5, 3))
```

```
plt.step(periods, demand, where='post', color='black', marker='.')
plt.xlabel('time (h)')
plt.ylabel('demand (kW)')
plt.grid()

# battery:
eta_in  = 0.9    # efficiency of charging
eta_out = 0.9    # efficiency of discharging
E_max   = 20.0   # kWh, maximum energy level
E_start = 15.0   # kWh, starting energy level
E_end   = 15.0   # kWh, final energy level
p_max   = 10.0   # kW, maximum (dis-)charging power
```



1. Minimieren Sie mit dem Code aus Beispiel Batterieverluste die Spitzenlast (Peak Power), und stellen Sie die resultierende Netzlast grafisch dar. In der Grafik sehen Sie, dass die Netzlast teils große Schwankungen und Sprünge aufweist.
2. Um diese zu reduzieren, modellieren und implementieren Sie das Optimierungsproblem, das die maximale Variation minimiert, daher der Name "Peak Power Variation", siehe Dynamische Probleme: Variation. Stellen Sie die resultierende Netzlast wieder grafisch dar, und vergleichen Sie die Grafik mit jener der "Peak Power" Minimierung. Untersuchen Sie, ob man die Minimierung der "Peak Power Variation" relaxieren kann.

21. Aufgaben 4

Die Aufgaben müssen in ILIAS als lauffähige und dokumentierte Jupyter Notebooks inkl. Datenfiles abgegeben werden. LP-Modellierungen können als eingescannte PDFs abgegeben werden.

21.1. Aufgabe 1: Demand Side Management (10 Punkte)

21.1.1. Problemstellung

Im Demand Side Management (DSM) werden unter anderem die Lasten von Verbrauchern direkt oder indirekt gesteuert, um z. B. ihre Gesamtlastspitze zu reduzieren oder ihre Gesamtlast einer nicht flexiblen (z. B. erneuerbaren) Erzeugung anzupassen.

- Im *centralized DSM* werden die Lasten aller Verbraucher zentral gesteuert, indem ein großes Optimierungsproblem gelöst wird, das alle Flexibilitäten der Verbraucher berücksichtigt. Dazu müssen dem zentralen Controller alle Daten der Verbraucher bekannt sein!
- Im *decentralized DSM* werden die Lasten der Verbraucher dezentral durch Anreizsignale (z. B. zeitabhängige Preise) gesteuert, und jeder Verbraucher löst sein eigenes Optimierungsproblem, das nur seine eigene Flexibilität berücksichtigt. Wenn als Anreizsignal Preise verwendet werden, dann minimiert jeder Verbraucher seine Kosten. Dazu müssen nur die Verbraucher ihre eigenen Daten kennen!

Literaturhinweis: P. Palensky and D. Dietrich, "Demand Side Management: Demand Response, Intelligent Energy Systems, and Smart Loads," IEEE Transactions on Industrial Informatics, vol. 7, no. 3, pp. 381–388, Aug. 2011, doi: 10.1109/TII.2011.2158841.

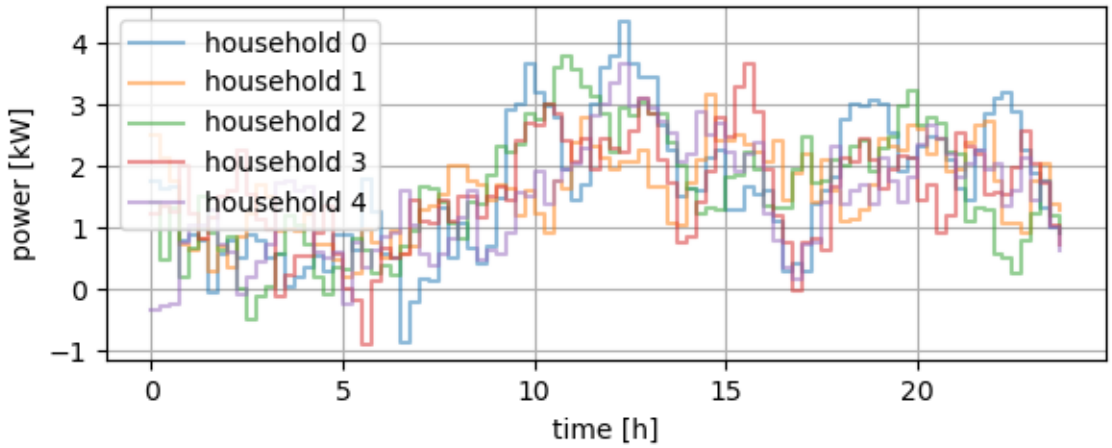
Wir betrachten fünf Verbraucher in Viertelstundenschritten über einen Tag. Der fixierte Verbrauch ist im Code unten angegeben. Die Flexibilität für die Verbraucher wird jeweils durch die gleiche, verlustfreie Batterie gegeben, deren Parameter unten angeführt sind. Die Gesamtlast der Verbraucher soll über den Tag möglichst der nicht flexiblen, angegeben PV-Erzeugung angepasst werden. Dabei wird die maximale absolute Abweichung zwischen Last und Erzeugung als Bewertung der Anpassung verwendet.

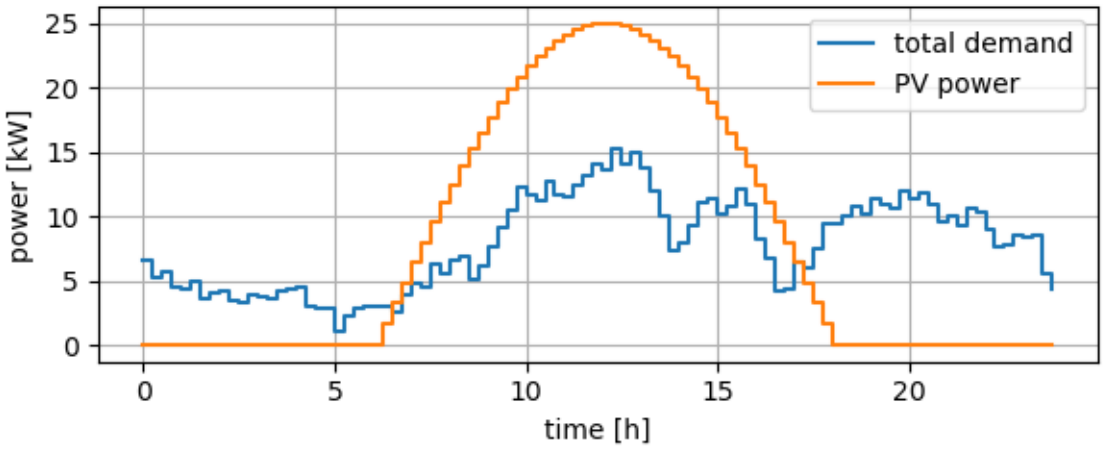
1. Lösen Sie das centralised DSM-Problem: Modellierung, Implementierung und Darstellung der Lösung. *Hinweis:* Verwenden Sie doppelt indizierte Variablen, um die Lasten der Verbraucher zu modellieren, vgl. Aufgaben 2: Implementierungsvariante
2. Definieren Sie ein zeitabhängiges Preissignal, das den Verbrauchern für den betrachteten Tag im Vorhinein geschickt wird und das indirekt die Lasten der Verbraucher so steuert, dass die Gesamtlast möglichst der PV-Erzeugung angepasst wird. Die Preise müssen nicht dem Tarif der Verbraucher entsprechen sondern dienen als Pseudopreise nur der Steuerung. Lösen Sie das decentralized DSM-Problem: Modellierung, Implementierung und Darstellung der Lösung. Vergleichen Sie die Lösungen des centralized und decentralized DSM-Problems.
3. (*) Überlegen Sie sich ein anderes Anreizsignal, das die Lasten der Verbraucher evtl. besser steuert als das Preissignal.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
dt = 0.25  # h
times = np.arange(start=0, stop=24 + dt, step=dt)  # timestamps: 0, 0.25, 0.5, ..., 23.75, 24
periods = np.arange(start=0, stop=24, step=dt)     # start times of periods:  0, 0.25, 0.5, ..., 23.75

# source: https://www.bdew.de/energie/standardlastprofile-strom/
```

```
demand = np.array([87.7, 81.5, 76.2, 71.0, 65.8, 60.5, 55.6, 51.5, 48.5, 46.4, 44.9, 43.7, 42.7,
                   41.8, 41.1, 40.6, 40.4, 40.4, 40.5, 40.6, 40.6, 40.6, 40.5, 40.6, 40.8, 41.6,
                   43.2, 46.1, 50.6, 56.7, 64.6, 74.2, 85.5, 97.9, 110.7, 123.4, 135.3, 145.9,
                   155.1, 162.4, 167.8, 171.8, 175.5, 179.6, 184.9, 190.5, 195.7, 199.1, 200.4,
                   198.8, 194.3, 186.7, 176.1, 163.9, 151.7, 141.3, 134.0, 129.1, 125.7, 122.7,
                   119.3, 115.5, 111.7, 107.8, 104.2, 101.1, 99.1, 98.4, 99.5, 102.1, 106.2,
                   111.7, 118.2, 125.5, 132.8, 139.8, 145.9, 150.7, 153.5, 153.9, 151.6, 147.4,
                   142.8, 139.1, 136.9, 135.8, 134.8, 132.8, 129.0, 123.7, 117.0, 109.3, 101.0,
                   92.3, 83.7, 75.7])
demand = demand/(np.sum(demand)*dt)*40  # demand load in kW, scaled to 40 kWh energy demand

# households:
num_of_households = 5  # number of households
# household load profiles:
np.random.seed(7)
demand_hh = np.zeros((num_of_households, len(periods)))
for h in range(num_of_households):
    noise = np.random.normal(loc=0, scale=0.4, size=len(periods))
    weights = np.exp(-.5*periods[:4])
    demand_hh[h] = demand + np.convolve(noise, weights, mode='same')

# power generation:
pv_power = np.zeros_like(periods)
pv_power[6*4:6*4 + 12*4] = 25*np.sin(2*np.pi/(6*4) * periods[:12*4] )  # PV power in kW

plt.figure(figsize=(6, 5))
plt.subplot(2, 1, 1)
plt.step(periods, demand_hh.T, where='post', alpha=0.5)
plt.xlabel('time [h]')
plt.ylabel('power [kW]')
plt.legend([f'household {h}' for h in range(num_of_households)])
plt.grid()

plt.subplot(2, 1, 2)
plt.step(periods, demand_hh.sum(axis=0), where='post', label='total demand')
plt.step(periods, pv_power, where='post', label='PV power')
plt.xlabel('time [h]')
plt.ylabel('power [kW]')
plt.legend()
plt.grid()

plt.tight_layout()
```






```
print(f"total demand energy = {demand_hh.sum()*dt:.2f} kWh")
print(f"total PV energy     = {pv_power.sum()*dt:.2f} kWh")
```

```
total demand energy = 189.37 kWh
total PV energy     = 190.92 kWh
```

```
# household battery parameters:
E_max   = 20.0   # kWh, maximum energy level
E_start = 10.0   # kWh, starting energy level
E_end   = 10.0   # kWh, final energy level
p_max   = 10.0   # kW, maximum (dis-)charging power
```

22. Aufgaben 5

Die Aufgaben müssen in ILIAS als lauffähige und dokumentierte Jupyter Notebooks inkl. Datenfiles abgegeben werden. LP-Modellierungen können als eingescannte PDFs abgegeben werden.

22.1. Aufgabe 1: Produktionszeitplan (6 Punkte)

Die XY GmbH muss zwei Produkte in ausreichender Menge herstellen, um die vertraglich vereinbarten Verkäufe in jedem der nächsten drei Monate zu erfüllen. Die beiden Produkte verwenden die gleichen Produktionsanlagen, und jedes Stück beider Produkte erfordert die gleiche Menge an Produktionskapazität. Die verfügbaren Produktions- und Lagereinrichtungen ändern sich von Monat zu Monat, so dass die Produktionskapazitäten, Produktionsstückkosten und Lagerhaltungsstückkosten von Monat zu Monat variieren. Daher kann es sich lohnen, eines oder beide Produkte manchmal im Überschuss zu produzieren und bis zum Bedarf zu lagern.

Monat	maximale Stückzahl gesamt		Produkt 1 / Produkt 2			
				Produktionskosten pro Stück		Lagerkosten pro Stück
	R	Ü	Verkäufe	R	Ü	
1	10	3	5/3	15/16	18/20	1/2
2	8	2	3/5	17/15	20/18	2/1
3	10	3	4/4	19/17	22/22	

Abbildung 22.1.: Transportproblem der XY GmbH

Für jeden der drei Monate gibt die zweite Spalte der obigen Tabelle die maximale Anzahl von Stücken der beiden Produkte insgesamt an, die in der regulären Zeit (R) und mit Überstunden (Ü) produziert werden können. Für jedes der beiden Produkte geben die nachfolgenden Spalten

- die Anzahl der Stücke an, die für die vertraglich vereinbarten Verkäufe benötigt werden
- die Kosten (in Tausend EUR) pro Stück in regulärer Zeit
- die Kosten (in Tausend EUR) pro Stück pro Überstunde
- die Kosten (in Tausend EUR) pro Stück der Lagerung bis in den nächsten Monat

Die Zahlen für die beiden Produkte sind jeweils durch einen Schrägstrich getrennt, wobei die erste Zahl für Produkt 1 und die zweite Zahl für Produkt 2 gilt.

1. Formulieren Sie dieses Problem als ein Transportproblem, indem Sie eine passende Parametertabelle erstellen.
2. Bestimmen Sie eine optimale Lösung und stellen Sie diese dar.

22.2. Aufgabe 2: Lagenschwimmen-Staffel (4 Punkte)

Der Trainer eines Schwimmteams muss Schwimmer für die Olympischen Spiele in eine 200 Yards Lagenschwimmen-Staffel einteilen. Da die meisten seiner besten Schwimmer in mehr als einem Schwimmstil sehr schnell sind, ist nicht klar, welcher Schwimmer jedem der vier Schwimmstile zugewiesen werden soll. Die besten Zeiten der fünf schnellsten Schwimmer sind in der folgenden Tabelle angegeben.

Schwimmstil	Carl	Chris	David	Tony	Ken
Rücken	37.7	32.9	33.8	37.0	35.4
Brust	43.4	33.1	42.2	34.7	41.8
Schmetterling	33.3	28.5	38.9	30.4	33.6
Freistil	29.2	26.4	29.6	28.5	31.1

Der Trainer möchte herausfinden, welche vier Schwimmer er den vier verschiedenen Schwimmstilen zuordnen soll, um die Summe der entsprechenden Bestzeiten zu minimieren und so eine optimale Aufstellung zu erhalten.

Implementieren Sie dieses Zuordungsproblem inklusive Darstellung der Lösung.

Literaturverzeichnis

[BC14] Antimo Barbato und Antonio Capone. "Optimization Models and Methods for Demand-Side Management of Residential Users: A Survey". en. In: *Energies* 7.9 (Sep. 2014), S. 5787–5824. DOI: 10.3390/en7095787. URL: https://www.mdpi.com/1996-1073/7/9/5787 (besucht am 24. 01. 2020).

[Ber98] Dimitri P. Bertsekas. *Network Optimization: Continuous And Discrete Models*. Englisch. Belmont, Mass: Athena Scientific, 1998. ISBN: 978-1-886529-02-1. URL: http://web.mit.edu/dimitrib/www/net.html.

[BJS10] Mokhtar S. Bazaraa, John J. Jarvis und Hanif D. Sherali. *Linear Programming and Network Flows*. Englisch. 4. Hoboken, N.J: Wiley, Jan. 2010. ISBN: 978-0-470-46272-0. URL: https://onlinelibrary.wiley.com/doi/book/10.1002/9780471703778.

[BV18] Stephen Boyd und Lieven Vandenberghe. *Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares*. Englisch. 1. Aufl. Cambridge, UK ; New York, NY: Cambridge University Press, Juni 2018. ISBN: 978-1-316-51896-0. URL: https://web.stanford.edu/~boyd/vmls/.

[EK10] David Easley und Jon Kleinberg. *Networks, Crowds, and Markets: Reasoning about a Highly Connected World*. Englisch. New York: Cambridge University Press, Sep. 2010. ISBN: 978-0-521-19533-1. URL: https://www.cambridge.org/core/books/networks-crowds-and-markets/A70C7855A3003FE1079C25F8397AF641#.

[Ham06] Horst W. Hamacher. *Lineare Optimierung und Netzwerkoptimierung: Zweisprachige Ausgabe Deutsch Englisch*. Deutsch, Englisch. 2., verb. Aufl. 2006. Wiesbaden: Vieweg+Teubner Verlag, Apr. 2006. ISBN: 978-3-8348-0185-2. URL: https://link.springer.com/book/10.1007/978-3-8348-9031-3.

[HL20] Frederick Hillier und Gerald Lieberman. *Introduction to Operations Research*. Englisch. 11th edition. New York, NY: McGraw-Hill Education Ltd, 2020. ISBN: 978-1-260-57587-3. URL: https://www.mheducation.com/highered/product/introduction-operations-research-hillier-lieberman/M9781259872990.html.

[JB08] Paul A. Jensen und Jonathan F. Bard. *Operations Research Models and Methods*. Englisch. 1. Aufl. Hoboken, N.J. : Great Britain: John Wiley & Sons, 2008. ISBN: 978-0-471-38004-7. URL: https://www.wiley.com/en-us/Operations+Research+Models+and+Methods-p-9780471380047.

[Kni07] Gunter Knieps. *Netzökonomie*. Deutsch. 2007. Aufl. Wiesbaden: Gabler Verlag, Apr. 2007. ISBN: 978-3-8349-0107-1. URL: https://link.springer.com/book/10.1007/978-3-8349-9231-4.

[LL19] Svein Linge und Hans Petter Langtangen. *Programming for Computations - Python: A Gentle Introduction to Numerical Simulations with Python 3.6*. Englisch. 2nd ed. 2020 Edition. Cham: Springer, Nov. 2019. ISBN: 978-3-030-16876-6. URL: https://link.springer.com/book/10.1007/978-3-030-16877-3.

[Nic+22] Stefan Nickel u. a. *Operations Research*. de. 3. Aufl. Berlin, Heidelberg: Springer, 2022. ISBN: 978-3-662-65345-6. DOI: 10.1007/978-3-662-65346-3. URL: https://link.springer.com/10.1007/978-3-662-65346-3 (besucht am 25. 10. 2023).

[PLB15] Markos Papageorgiou, Marion Leibold und Martin Buss. *Optimierung: Statische, dynamische, stochastische Verfahren für die Anwendung*. de. 4. Aufl. Springer Vieweg, 2015. ISBN: 978-3-662-46935-4. DOI: 10.1007/978-3-662-46936-1. URL: https://www.springer.com/de/book/9783662469354 (besucht am 10. 12. 2020).

[SC17] Ramteen Sioshansi und Antonio J. Conejo. *Optimization in Engineering: Models and Algorithms*. Englisch. 1st ed. 2017 Edition. New York, NY: Springer, Juli 2017. ISBN: 978-3-319-56767-9. URL: https://link.springer.com/book/10.1007/978-3-319-56769-3.

[Sch16] Wolfgang Schellong. *Analyse und Optimierung von Energieverbundsystemen*. Deutsch. 1. Aufl. 2016. Berlin Heidelberg: Springer Vieweg, Juli 2016. ISBN: 978-3-662-48527-9. URL: https://link.springer.com/book/10.1007/978-3-662-49463-9.

[SG12] Gerard Sierksma und Diptesh Ghosh. *Networks in Action: Text and Computer Exercises in Network Optimization*. Englisch. 2010. Aufl. Erscheinungsort nicht ermittelbar: Springer, Feb. 2012. ISBN: 978-1-4614-2543-4. URL: https://www.amazon.de/Networks-Action-Optimization-International-Operations/dp/1441955127/.

[SM13] Leena Suhl und Taïeb Mellouli. *Optimierungssysteme: Modelle, Verfahren, Software, Anwendungen*. de. 3. Aufl. Springer-Lehrbuch. Gabler Verlag, 2013. ISBN: 978-3-642-38936-8. DOI: 10.1007/978-3-642-38937-5. URL: https://www.springer.com/de/book/9783642389368 (besucht am 31. 10. 2019).

[SZ15] Gerard Sierksma und Yori Zwols. *Linear and Integer Optimization: Theory and Practice*. en. 3. Aufl. Google-Books-ID: KsH1CwAAQBAJ. CRC Press, Mai 2015. ISBN: 978-1-4987-4312-9. URL: https://www.amazon.de/Linear-Integer-Optimization-Practice-Mathematics/dp/1498710166/.

[TW15] Volker Turau und Christoph Weyer. *Algorithmische Graphentheorie*. DT. 4th edition. Berlin, Boston: De Gruyter, 2015. ISBN: 978-3-11-041727-2. DOI: 10.1515/9783110417326. URL: https://www.degruyter.com/view/product/455161?format=B (besucht am 05. 08. 2019).

[Wil13] H. Paul Williams. *Model Building in Mathematical Programming 5e*. Englisch. 5. Hoboken, N.J: Wiley, Feb. 2013. ISBN: 978-1-118-44333-0. URL: https://www.wiley.com/en-us/Model+Building+in+Mathematical+Programming%2C+5th+Edition-p-9781118443330.