TECHNISCHE UNIVERSITÄT DRESDEN

Fakultät für Verkehrswissenschaften "'Friedrich List"' Institut für Autombiltechnik Dresden – IAD

Professur für Fahrzeugmechatronik

Einbindungen von Plots mittels PGFPlots, `Matplotlib` und PythonTEX

Tim Häberlein

22. März 2024

Inhaltsverzeichnis

1 Import von mit Matplotlib gespeicherten Figures

Im folgenden Abschnitt werden verschiedene Möglichkeiten gezeigt, wie mit `Matplotlib` erstellte Plots in LaTeX eingebunden werden können. Dabei können grundsätzlich 3 Varianten unterschieden werden, wie die figures mit `Matplotlib` gespeichert werden können:

- pgf
- pdf
- svg oder png

Die erste Variante hat den Vorteil, dass die pgf-Datei direkt als Code in LaTeX eingebunden werden kann. Das bedeutet, dieser wird zur Laufzeit übersetzt und z. B. die Schrift mit verändert. Die Schriftgröße und die Bildbreite müssen jedoch fix eingestellt oder von Hand im exportierten Code abgeändert werden.

Die zweite Variante kann zwar skaliert werden, jedoch wird hier der Text nicht mit skaliert. Das bedeutet, dass selbst bei einer kleinen Änderung (wie z. B. der Schriftart oder des Textlayouts) das Bild neu exportiert werden muss.

Bei der dritten Variante wird das Bild als Vektorgrafik gespeichert und kann somit auch beliebig skaliert werden. Um die Schrift unabhängig zur Laufzeut wie bei pgf in LaTeXzu übersetzten, gibt es die Möglichkeit die .svg-Datei in Inkscape zu importieren und getrennt wieder zu exportieren. Da diese Variante den Aufwand bei weitem übersteigt und sonst ähnlich zu pdf ist, wird im Folgenden nicht näher darauf eingegangen.

1.1 Vorbereitung in LaTeX

Zur Bestimmung der Grafikbreite muss zuerst die Breite des Textes bestimmt werden. In LaTeX sind dazu die Befehle `\textwidth` und/oder `\columnwidth` definiert. Um die Breite auszulesen kann folgender Code in das relevante LaTeX-Dokument eingefügt werden:

```
% Ausgabe von textwidth in der Kommandoeile
\typeout {textwidth ist: \the\textwidth}
```

Quelltext 1: Read textwidth in LaTeX

Der Wert wird dann in der `.log`-Datei ausgegeben. Alternativ kann auch der Wert direkt in das Dokument geschrieben werden:

```
% ausgabe von textwidth im Dokument
Der aktuelle Wert von \textbackslash textwidth ist \the\textwidth .
```

erzeugt den direkt im Dokument: Der aktuelle Wert von \textwidth ist 455.24417pt.

1.2 Vorbereitungen in Python

Um das PGF Backend nutzen zu können, muss zunächst in Python die `Matplotlib`-Bibliothek eingebunden werden:

```
# import matplotlib as mpl and set pgf as backend
import matplotlib as mpl
mpl.use('pgf')

# import pyplot from matplotlib
import matplotlib.pyplot as plt
```

Quelltext 2: import `Matplotlib`

Danach kann der Plot mit `plt.savefig('figure.pgf')` oder `plt.savefig('figure.pdf')` je nach gewünschtem Format gespeichert werden. Die explizite Anweisung `plt.savefig('figure.pdf', backend='pgf')` während des Speichern schaltet das pfg-Backend im Rahmen dieses Aufrufs einmalig frei. Der Schalter `mpl.use('pgf')` nach dem Import wird damit überflüssig.

Mit `rcParams` kann das Verhalten des pfg-Backends konfiguriert werden:

Tabelle 1: `rcParams` für `Matplotlib`-Plots in LaTeX

Parameter	Beschreibung
pfg.preample	spezifische Pakete, die in die Präample aufgenommen werden sollen
pgf.rcfonts	Schriftart
pgf.texsystem	„xelatex“ (voreingestellt), „lualatex“ oder „pdflatex“

Die Parameter können über das plt-Objekt wie folgt angepasst werden. Eine Übergabe spezieller Präambel-Pakete ist aufgrund der späteren Latex-Einbindung unter Verwendung von pgf nicht notwendig, bei pdf jedoch obligatorisch.

```
import matplotlib.pyplot as plt
plt.rcParams.update({
    "pgf.texsystem": "pdflatex",  # use pdflatex backend - usually the case
    "font.family": "serif",  # use serif/main font for text elements
    "text.usetex": True,     # use inline math for ticks
    "pgf.rcfonts": False,    # don't setup fonts from rc parameters
    ## You can change the font size of individual items with:
    # "font.size": 11,
    # "axes.titlesize": 11,
    # "legend.fontsize": 11,
    # "axes.labelsize": 11,
    ## optional preamble setup
    # "pgf.preamble": "\n".join([
    #      r"\usepackage{url}",            # load additional packages
    #      r"\usepackage{unicode-math}",   # unicode math setup
    #      r"\setmainfont{DejaVu Serif}",  # serif font via preamble
    # ])
})
```

Quelltext 3: set rcParams

1.3 Bestimmung der Bildgröße

Zur Bestimmung des Höhen-Breiten-Verhältnisses kann der goldene Schnitt ($\Phi >$) verwendet werden. Das Verhältnis ist rund 1:1,618. Wobei die genaue Formel (Gleichung 1) lautet:

$$\Phi = \frac{a}{b} = \frac{a+b}{a} = \frac{1+\sqrt{5}}{2} \approx 1,6180339887 \tag{1}$$

Mit der Angabe der Zeilenbreite aus Latex (s. Quelltext 1) kann sowohl Höhe und Breite des plots mit dem folgendne Code berechnet werden.

```
def calc_figsize(width_pt, subplots=(1, 1)):
    """Set figure dimensions to sit nicely in our document.

    Args:
        width_pt (float): Document width in points (1 inch = 72.27 points)
```

```
        subplots (tuple): Number of rows and columns of subplots.

    Returns:
        tuple: Figure dimensions in inches.
    """
    ## Variablen
    inches_per_pt = 1 / 72.27
    # Golden ratio to set aesthetic figure height
    golden_ratio = (5**.5 - 1) / 2

    ## Berechnung
    # Figure width in inches
    fig_width = width_pt * inches_per_pt
    # Figure height in inches
    fig_height = fig_width * golden_ratio * (subplots[0] / subplots
    [1])

    ## Rueckgabe
    return fig_width, fig_height
```

Quelltext 4: calculate golden ratio

1.4 Einbindung in LaTeX

Die Einbindung in ein LaTeX-Dokument wird im Abschnitt Abschnitt 2 in Quelltext 6 folgt erfolgen.

1.5 Verweise

Die Informationen wurden folgenden Quellen entnommen:

- Exporting Matplotlib Plots to LaTeX beschreibt die Einbindung anhand eines einfachen Beispiels.
- Matplotlib plots for LaTeX with PGF beschreibt die Einbindung anhand eines einfachen Beispiels und zeigt die Funktion zur Berechnung des goldenen Schnitts.
- offizielle `Matplotlib`-Seite beschreibt die Einbindung anhand eines einfachen Beispiels und zeigt die Funktion zur Berechnung des goldenen Schnitts.

2 Vergleich von PGF und PDF aus Matplotlib

Mit folgendem Code wurde ein Plot erstellt und in beiden Formaten gespeichert. Der Python-Code ist ebenfalls als Jupyter-Notebook verfügbar.

```
from PlotBase import *
fig, ax = plt.subplots(figsize=calc_figsize(0.8*textwidth))
x = np.linspace(-2*np.pi, 2*np.pi, 100)
y = np.sin(x)

ax.plot(x, y, label=r'$f(x) = sin(x)$', color=tudcolors.cddarkblue().
    rgb_values)
ax.set_title('Sinus-Funktion', pad=20)
ax.grid(True)  # Ensure grid is visible

# Adjusting axis labels to be closer to the arrows
# ax.set_xlabel('x', labelpad=10, loc='right')  # Positioning label at
     the end
```

```
# ax.set_ylabel('y', labelpad=25, loc='top', rotation=0)  # Positioning label at the end and rotating

# Using text objects for axis labels
ax.text(1.1 * np.max(x), -0.15, 'x', ha='right', va='center')  # X-Achsenbeschriftung
ax.text(0.3, 1.1 * np.max(y), 'y', ha='left', va='center', rotation=0)  # Y-Achsenbeschriftung

# Add the legend at the top right
plt.legend(loc='upper right')

# Adjusting the tick positions
ax.spines['left'].set_position(('data', 0))
ax.spines['left'].set_color(tudcolors.cdgray().rgb_values)
ax.spines['bottom'].set_position(('data', 0))
ax.spines['bottom'].set_color(tudcolors.cdgray().rgb_values)
ax.spines['right'].set_color('none')
ax.spines['top'].set_color('none')

# Adding arrows
ax.plot((1), (0), ls="", marker=">", ms=10, color=tudcolors.cdgray().rgb_values, transform=ax.get_yaxis_transform(), clip_on=False)
ax.plot((0), (1), ls="", marker="^", ms=10, color=tudcolors.cdgray().rgb_values, transform=ax.get_xaxis_transform(), clip_on=False)

# Adjust tick positions to the left and bottom
ax.yaxis.set_ticks_position('left')
ax.xaxis.set_ticks_position('bottom')
# Die Farbe der Tick-Labels an den Achsen aendern
ax.tick_params(axis='x', colors=tudcolors.cdgray().rgb_values)  # Farbe der X-Achsen-Ticks
ax.tick_params(axis='y', colors=tudcolors.cdgray().rgb_values)  # Farbe der Y-Achsen-Ticks

# Set x ticks at multiples of pi and apply the custom formatter
ax.xaxis.set_major_locator(ticker.MultipleLocator(base=np.pi))
ax.xaxis.set_major_formatter(ticker.FuncFormatter(format_func))

fig.tight_layout()

# Saving the plots
fig.savefig('sinus.pgf')
fig.savefig('sinus.pdf')
```

Quelltext 5: Plot Code

2.1 PGF-Plot

Im folgenden Abschnitt wird der Plot aus Quelltext 5 als PGF-Plot eingebunden und ist in Abbildung 1 dargestellt.

Die Einbindung erfolgt mit dem folgenden Code:

```
\documentclass{article}

\usepackage{pgf}
\pgfplotsset{compat=1.18}

\def\mathdefault#1{#1}
\everymath=\expandafter{\the\everymath\displaystyle}
\makeatletter\@ifpackageloaded{underscore}{}{\usepackage[strings]{underscore}}\makeatother
```

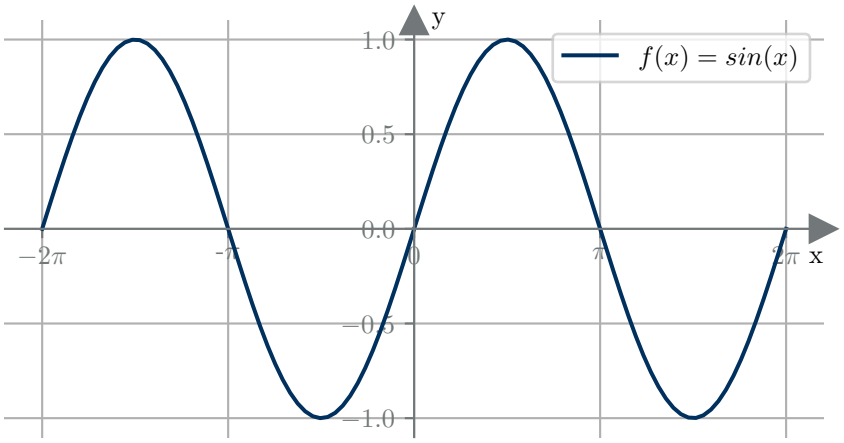


Abbildung 1: Ein PGF-Plot aus `Matplotlib`.

```
9
10 \begin{document}
11     \begin{figure}[htp!]
12         \begin{minipage}{\textwidth}
13             \centering
14             %\resizebox{0.8\textwidth}{!}{\input{sinus.pgf}}
15             \input{sinus.pgf}
16             \caption{Ein PGF-Plot aus \matplotlib.}\label{fig:pgf-plot
                   }
17         \end{minipage}
18     \end{figure}
19 \end{document}
```

Quelltext 6: Einbindung eines pgf-plots

2.2 PDF-Plot

Im folgenden Abschnitt wird der Plot aus Quelltext 5 als PGF-Plot eingebunden und ist in Abbildung 2 dargestellt.

Die Einbindung erfolgt mit dem folgenden Code:

```
1  \begin{figure}[htb]
2      \begin{center}
3          \begin{tikzpicture}
4              \node[inner sep=0pt, outer sep=0pt] (test) at (0,0) {
5                  \includegraphics[clip, width=0.8\textwidth]{sinus}};
6              %\helplines
7          \end{tikzpicture}
8      \end{center}
9      \caption{Ein PDF-Plot aus \matplotlib.}\label{fig:pdf-plot}
10 \end{figure}
```

Quelltext 7: Einbindung eines pdf-plots

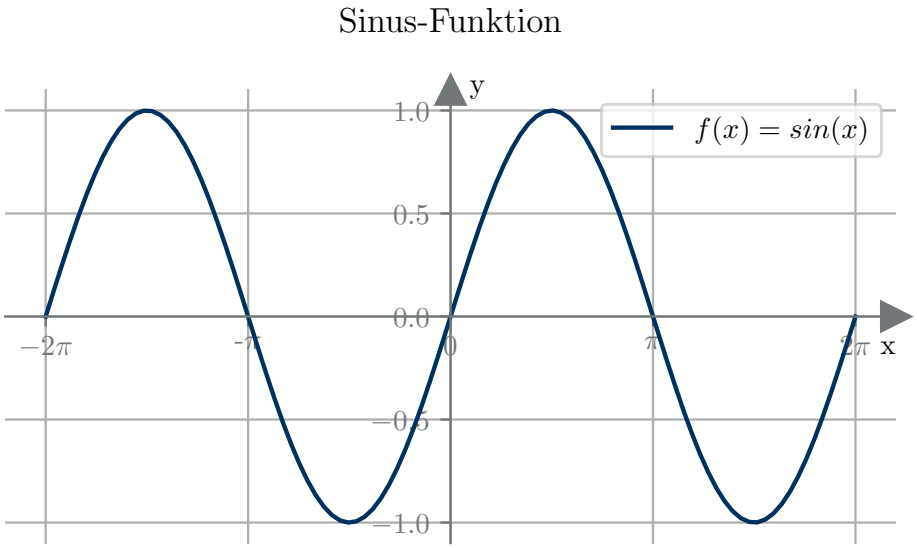


Abbildung 2: Ein PDF-Plot aus `Matplotlib`.

2.3 Vergleich

Der Vergleich der beiden Plots ist noch einmal in Abbildung 3 dargestellt. Es zeigt sich, dass z. B. bei der Änderung der Textbreite – von den ursprünglich für die Berechnung der Grafiken verwendeten 418.25555 Punkten auf 455.24417 Punkte – durch die Verwendung des Schalters `cd=true` der Dokumentenklasse tudscrartcl ein Unterschied entsteht. In der PDF-Variante wird der Text nicht passend skaliert und wirkt dadurch in diesem Fall größer.

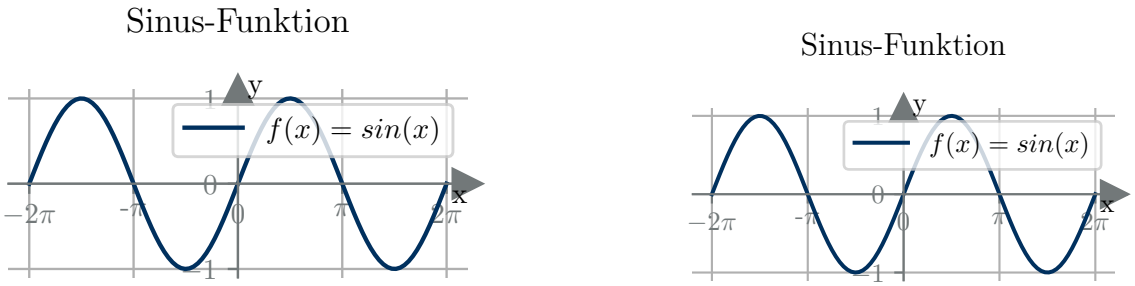


Abbildung 3: Vergleich von PDF (links) und PGF (rechts).

3 Import von direkten PGF-Plots

PGF-Plots ist ein Paket für LaTeXund dient der Erstellung von 2D und 3D Plots. Es ist in der Lage, die Plots direkt in LaTeXzu erstellen und zu rendern und greift dabei auf `TikZ` zurück.

3.1 Vorbereitungen in LaTeX

Um einen PGF-Plot in LaTeXzu erstellen, müssen die notwendigen Pakete eingebunden werden:

```
1 \usepackage{packages}
2
```

```
\usepackage{pgfplots}
\usepackage{pgf}
\pgfplotsset{compat=1.9}
```

Quelltext 8: Einbinden der notwendigen Pakete für PGF-Plots

3.2 Einbindung in LaTeX

Die Einbindung eines PGF-Plots in LaTeXkann über folgenden Code erfolgen:

```
\begin{figure}[htb]
    \centering
    \setlength{\figurewidth}{0.8\columnwidth}% Skalierung der Breite
    \subimport{}{pgf_sinus.tex}
    \caption{Ein direkter PGF-Plot einer Sinus-Funktion.}
    \label{fig:direkter-pgf-plot-einer-sinus-funktion}
\end{figure}
```

Quelltext 9: Einbinden eines PGF-Plots in LaTeX

Dabei kann über den `\setlength{\figurewidth}{0.8\columnwidth}`-Befehl die Breite des Plots skaliert werden, da die erstellte Länge im Rahmen des PGF-Plots (s. Quelltext 10 Zeile 3) verwendet wird.

3.3 PGF-Plot

Folgender Code wurde verwendet, um den Sinus-Plot in Abbildung 4 auf der nächsten Seite zu erstellen:

```
\begin{tikzpicture}
    \begin{axis}[
        width=\figurewidth, height=0.618\figurewidth,
        axis lines=middle,
        axis line style={-latex},
        grid=major, %both
        major grid style={cdgray},
        minor grid style={cdgrey!25},
        title=Sinus-Funktion,
        xlabel={$x$},
        ylabel={$y$},
        ymin=-1, ymax=1, minor y tick num=1,
        domain=-2*pi:2*pi,
        samples=100,
        xtick={-2*pi, -pi, 0, pi, 2*pi},
        xticklabels={$-2\pi$, $-\pi$, $0$, $\pi$, $2\pi$},
        ytick={-1, -0.5, 0, 0.5, 1},
        legend pos=north east,
        ]
        \addplot [cddarkblue, very thick]
            {sin(deg(x))};
        \addlegendentry{$f(x) = \sin(x)$}
    \end{axis}
\end{tikzpicture}
```

Quelltext 10: Code für einen Simus-Plot mit pgf

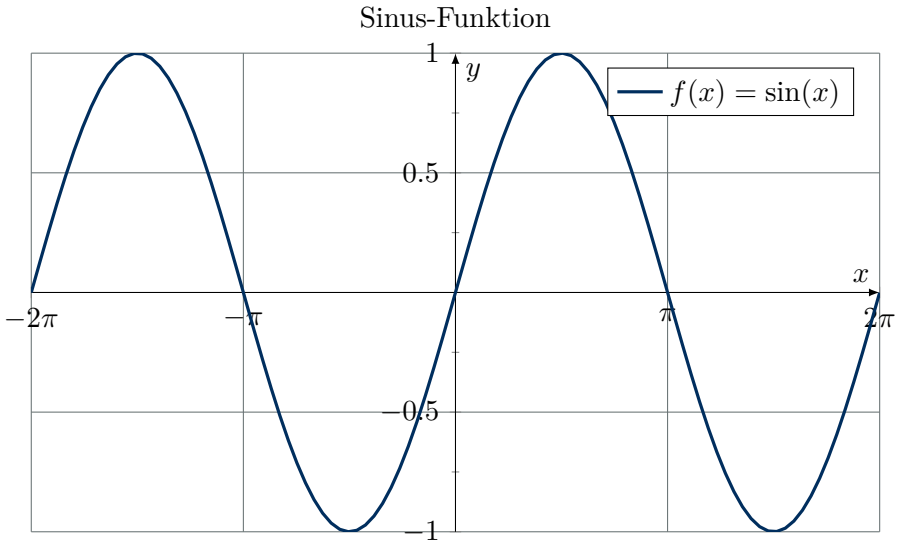


Abbildung 4: Ein direkter PGF-Plot einer Sinus-Funktion.

3.4 Vergleich direkter PGF und `Matplotlib` PGF Plot

Abbildung 5 auf der nächsten Seite zeit den Vergleich der beiden PGF-Varianten. Natürlich ist ein kleiner Unterschied im Design zu erkennen (Pfeilspitzen, Schriftgrößen etc.). Die Schriftgrößen werden in Matplotlib absolut festgesetzt, wären der direkte PGF-Plot die Schriftgrößen des umgebenden Dokuments verwendet. Zusätzlich muss für die Einbindung des PGF-Plots über `Matplotlib` ein neuer Code generiert werden, der die Abbildung skaliert, da alles über fixe Längen erstellt wurde (s. Abschnitt 1.3 auf Seite 4).

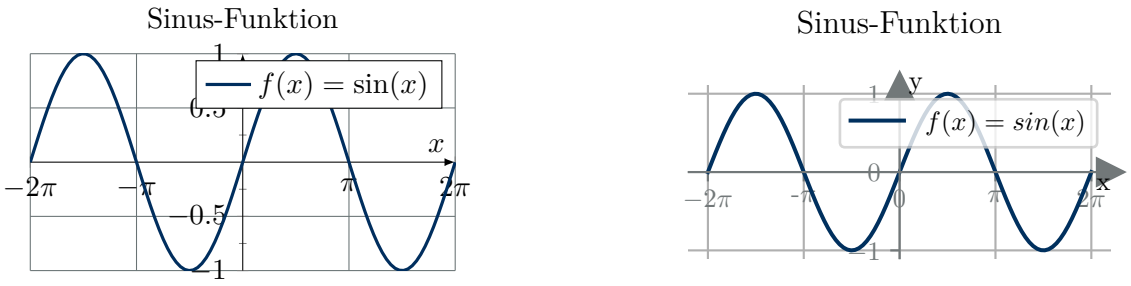


Abbildung 5: Vergleich von GPF direkt (links) und PGF über `Matplotlib` (rechts).

3.5 Verweise

Die Informationen wurden folgenden Quellen entnommen:

- Exporting Matplotlib Plots to LaTeX beschreibt die Einbindung anhand eines einfachen Beispiels.
- CTAN PGFPlots CTAN (Comprehensive TEX Archive Network) Website zu PGFPlots - offizielle Paketdokumentation.
- Grafiken in LATEX mit TikZ und PGFPLOTS Vorstellung von TikZ und PGFPlots anhand einer Präsentation von Patrick Schulz, 2016.

4 Import von PGF-Plots über Matplotlib mittels PythonTeX

PythonTeX ist ein LaTeX-Paket, welches es ermöglicht, Python-Code in LaTeX-Dokumente einzubinden und auszuführen. Dies ermöglicht den direkten Import des in Quelltext 5 aus Abschnitt 2 erzeugten Plots.

4.1 Vorbereitungen in LaTeXfür PythonTeX

PythonTeX wird mit mittels des folgendem Codes importiert:

```
1 \usepackage{pythontex}
```

Quelltext 11: Einbinden von PythonTeX

4.2 Vorbereitungen in Windows und Python für PythonTeX

Für die Verwendung von Python empfiehlt sich immer die Verwendung eines virtuellen Environments (venv) für jedes spezifische Projekt, da so Paket-Abhängigkeiten sauber gehalten werden können und das main-Python-Environment nicht geändert wird. Die Erstellung eines venv erfolgt unter Windows mittels folgendem Befehl: python −m venv /path/to/venv.

Im nächsten Schritt müssen relevante Pakete in dieses venv installiert werden. Dazu wird das venv aktiviert und anschließend mittels des Paketmanagers pip die Pakete in dieses venv installiert.

Die Python-Umgebung mit PythonTeX benötigt im vorliegenden Fall die folgenden Pakete:

pygments für die Syntax-Highlighting und

matplotlib für die Erstellung von Plots.

Die Installation und Einrichtung kann mittels des folgenden Codes erfolgen:

```
1 :: installation python venv
2 python -m venv C:\Users\example_user\IdeaProjects\tex_test\venv
3 :: activate venv
4 C:\Users\example_user\IdeaProjects\tex_test\venv\Scripts\activate.bat
5 :: install python-packages
6 pip install pygments
7 pip install matplotlib
```

Quelltext 12: Installation von **pygments** für PythonTeX

Der Aufruf des PythonTeX Compilers ist in Quelltext 13 im Allgemeinen dargestellt.

```
1 :: Aufruf des pythontex-Compilers im allgemeinen
2 pythontex --interpreter python:<path/to/venv> <tex_filename>
```

Quelltext 13: Aufruf des PythonTeX Compilers

Zu Begin muss jedoch auch das venv aktiviert werden, um den PythonTeX-Compiler korrekt ausführen zu können. Sollte zusätzlich noch ein output-directory mittels des pdflatex-Compilers angegeben sein, so muss auch der PythonTeX-Compiler diesen kennen. In diesem Fall empfiehlt sich die Verwendung eines kleinen Scriptes mittels eines Argumentes (#1) für das LaTeX-File, welches dann den PythonTeX-Compiler aufruft. Für den vorliegenden Fall könnte dieses Script wie folgt aussehen:

```
1 :: Wechsel in das output-directory des pdflatex-Compilers
2 cd C:\Users\s7285521\IdeaProjects\_Diss\tex_test\out
3 :: Aktivierung des venv und Aufruf des pythontex-Compilers
```

```
..\venv\Scripts\activate.bat && pythontex --interpreter python:C:\\
    Users\\s7285521\\IdeaProjects\\_Diss\\tex_test\\venv\\Scripts\\
    python.exe %1
```

Quelltext 14: Script für den Aufruf des PythonTeX-Compilers

Der typische Kompilierungsvorgang sieht dann wie folgt aus: pdflatex - pythontex - pdflatex. Sollte sich im Python-Code nicht geändert haben, genügt danach ein pdflatex Durchgang.

4.3 Einbindung und Verwendung von PythonTeX in LaTeX

In der Regel erfolgt die Einbindung von PythonTeX in LaTeX mittels der `pycode`-Umgebung. Die Verwendung einer Konsolen-Umgebung wird mittels `pyconsole` realisiert.

Die Ausgabe des Quelltext 15 ist darunter dargestellt. Weder die Konsolenumgebung noch die Inline-Konsolen-Umgebung brechen Text auf eine neue Zeile um.

```
\documentclass[class=tudscrartcl, cdfont=false]{standalone}
\usepackage{pythontex}
\begin{document}
    \python-Code kann mittels \lstinline[language=Tex, style=
        latexstyle]|\begin{pyconsole}| und \lstinline[language=Tex,
        style=latexstyle]|\end{pyconsole}|  in \LaTeX{} eingebunden
        und ausgefuehrt werden.
    Dabei sowohl Ein- als auch Ausgaben dargestellt, als ob es sich um
         eine richtige Konsole handeln wuerde.
\begin{pyconsole}
import os
os.getcwd()
\end{pyconsole}
    Der \python-Code innerhalb der \texttt{pyconsole}-Umgebung darf
        dabei nicht eingerueckt werden.
    Mithilfe des \lstinline[language=Tex, style=latexstyle]|\pycon{}|-
        Befehls kann der Output auch inline dargestellt werden.
    Inline dargestellt ist das \python-Workingdirectory: \\ \pycon{os.
        getcwd()}
\end{document}
```

Quelltext 15: PythonTeX-Verwendung

Python-Code kann mittels `\begin{pyconsole}` und `\end{pyconsole}` in LaTeX eingebunden und ausgeführt werden. Dabei werden sowohl Ein- als auch Ausgaben dargestellt, als ob es sich um eine richtige Konsole handeln würde.

```
>>> import os
>>> os.getcwd()
'C:\\Users\\s7285521\\IdeaProjects\\_Diss\\tex_test\\out'
```

Der Python-Code innerhalb der `pyconsole`-Umgebung darf dabei nicht eingerückt werden. Mithilfe des `\pycon{}`-Befehls kann der Output auch inline dargestellt werden. Inline dargestellt ist das Python-Workingdirectory:
'C:\\Users\\s7285521\\IdeaProjects_Diss\\tex_test\\out'

Bei der Verwendung von Variablen ist darauf zu achten, dass diese nur in der jeweiligen Umgebung definiert sind und verwendet werden können. Definieren wir noch eine Variable `a` in einer `pycode`-Umgebung, so ist diese nur innerhalb dieser Umgebung definiert und kann nicht in einer `pyconsole`-Umgebung verwendet werden.

```
\begin{pycode}
a = 3
\end{pycode}
```

```
\begin{pyconsole}
a = 2
a
\end{pyconsole}
```

Quelltext 16: PythonTeX-Verwendung von Variablen

```
>>> a = 2
>>> a
2
```

Nach dem Aufruf des Codes aus Quelltext 16 wird die pyconsole-Umgebung ausgegeben. Da es in der pycode-Umgebung keine Ausgabe (z. B. einen print-Befehl) gibt, wird diese nicht dargestellt. Beide Variablen können wir jetzt in der jeweiligen in-Line-Umgenbung verwenden:

- \pycon{a*2}: 4
- \py{a*2}: 6

4.4 Übergabe von Variablen zwischen LaTeX und Python

Die Übergabe von Variablen zwischen LaTeX und Python erfolgt mittels:

```
\documentclass[class=tudscrartcl, cdfont=false]{standalone}
\usepackage{pythontex}
\setpythontexcontext{textwidth=\the\textwidth, textheight=\the\
    textheight}
    \begin{document}
        \py{pytex.context['textwidth']} und \py{pytex.context.
            textheight} sind dabei die Variablen, die von \LaTeX{} an
            \python uebergeben werden.
        Und können in mittels vorhandener Funktionen in Inches
            umgerechnet werden: \py{pytex.pt_to_in(pytex.context['
            textwidth'])}.
        Der Rückgabetyp ist dann \py{type(pytex.pt_to_in(pytex.context
            ['textwidth']))}.
    \end{document}
```

Quelltext 17: Übergabe von Variablen zwischen LaTeX und Python

455.24417pt und 702.78317pt sind dabei die Variablen, die von LaTeX an Python übergeben werden. Und können in mittels vorhandener Funktionen in Inches umgerechnet werden: 6.299213643282137. Der Rückgabetyp ist dann <class 'float'>.

Dabei muss der Aufruf von \setpythontexcontext{} in der Präambel erfolgen. Die Variablen stehen dann ich allen pycode-Umgebungen zur Verfügung.

4.5 Sessions

Sofern der pycode-Umgebung kein optionales Argument als Session-Name übergeben wird (s. Quelltext 18), laufen alle Umgebungen nacheinander ab und können auf die gleichen Variablen zugreifen. Soll spezifischer Code für alle Sessions ausgeführt werden, so kann dies in der pythontexcustomcode-Umgebung erfolgen (s. Quelltext 19).

```
\begin{pycode}[<session-name>]
    <code>
\end{pycode}
```

Quelltext 18: PythonTeX-Sessions

```
\begin{pythontexcustomcode}{py}
    <code>
\end{pythontexcustomcode}
```

Quelltext 19: PythonTeX-Costum Code Environment

4.6 Working-Directory

Bei der Verwendung von PythonTeX bietet es sich an, dass Working-Directory von Python in das Verzeichnis zu setzen, in dem sich die LaTeX-Datei befindet und nicht im output-directory des pdflatex-Compilers zu arbeiten. So erfolgen imports von diesem Verzeichnis und outputs werden auch dort gespeichert. Gerade bei der Erstellung von Plot mittels `Matplotlib` ist dies von Vorteil, da der Output direkt im Verzeichnis der LaTeX-Datei liegt und nicht erst in das output-directory gewechselt werden muss. Der Wechsel kann dabei innerhalb der `pycode`-Umgebung mittels Python oder durch den LaTeX-Befehl `\setpythontexworkingdir{<path>}` erfolgen. Ein Beispiel ist in Quelltext 20 dargestellt.

```
\documentclass[class=tudscrartcl, cdfont=false]{standalone}
\usepackage{pythontex}
\setpythontexworkingdir{../src/mwe_matplotlib/pythontex}
\begin{document}
    \begin{pyconsole}
    import os
    os.getcwd()
    \end{pyconsole}
\end{document}
```

Quelltext 20: Setzen des Working-Directory für PythonTeX

Da diese Einstellung jedoch nur in der Präambel erfolgen kann, bietet es sich diese Option für komplexere Projekte mit mehreren `.tex`-Dateien nicht an. Der Python-Compiler wird nur einmal aufgerufen und hat dementsprechend dann für alle Files das gleiche Python-Working-directory. In diesem Fall bietet es sich an, das Working-Directory innerhalb der `pycode`-Umgebung zu setzen oder alle Dateien im out-directory von pdflatex zu halten. Für den Fall, dass das Python-Working-directory innerhalt der `pycode`-Umgebung gesetzt werden soll, ist dies in Quelltext 21 dargestellt.

```
\documentclass[class=tudscrartcl, cdfont=false]{standalone}
\usepackage{pythontex}
\begin{document}
    \begin{pyconsole}
import os
os.getcwd()
os.chdir('../src/mwe_matplotlib/pythontex')
os.getcwd()
    \end{pyconsole}
\end{document}
```

Quelltext 21: Setzen des Working-Directory für PythonTeX durch Python

Die Ausgabe sieht dann wie folgt aus:

```
>>> import os
>>> os.getcwd()
'C:\\Users\\s7285521\\IdeaProjects\\_Diss\\tex_test\\out'
>>> os.chdir('../src/mwe_matplotlib/pythontex')
```

```
>>> os.getcwd()
'C:\\Users\\s7285521\\IdeaProjects\\_Diss\\tex_test\\src\\mwe_matplotlib\\pythontex'
```

4.7 Einbindung in LaTeX

Ein Beispiel-Plot mittels `Matplotlib` und PythonTeX ist in Quelltext 22 dargestellt.

4.8 Verweise

Die Informationen wurden folgenden Quellen entnommen:

- PythonTeX: reproducible documents with LaTeX, Python, and more Paper des Autors (Geoffrey M Poore) von 2015 in Computational Science & Discovery, Volume 8, Number 1, DOI: 10.1088/1749-4699/8/1/014010. Übersicht zur Verwendung von PythonTeX.
- GitHub PythonTeX offizielle GitHub-Seite von PythonTeX.
- CTAN PythonTeX offizielle Paketdokumentation.
- Dokumentenautomation mit PythonTEX Vorstellung und Beispiele zu PythonTeX von Karsten Brodmann, 2018.

5 Vergleich PGF und PythonTeX Plots

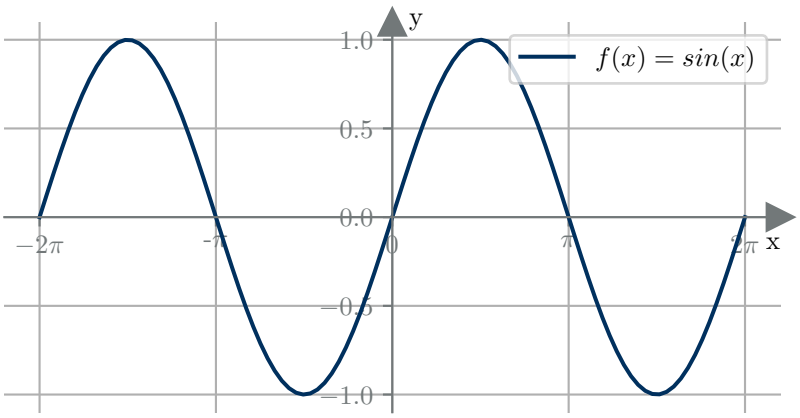
In diesem Abschnitt wird der Vergleich von PGF-Plots und PythonTeX-Plots dargestellt. Im ersten Schritt werden zuvor jedoch noch 2 verschiedene Möglichkeiten vorgestellt, wie ein Integration in LaTeX erfolgen kann.

5.1 Einbindung über PythonTeX und LaTeX

Die Einbindung des PGF-Plots mittels PythonTeX ist in dargestellt Quelltext 22. Dabei wird der Plot mittels `Matplotlib` durch PythonTeX direkt zur Laufzeit erstellt und als PGF-Datei gespeichert. Der Python-Code für die Plot Funktion ist in `PlotBase.py` enthalten. Der Inhalt der Datei ist Quelltext 25 zu entnehmen. Anschließend wird diese Datei in das LaTeX-Dokument eingebunden. Das Ergebnis ist in Abschnitt 5.1 dargestellt.

```
1  \begin{pycode}[plot1]
2  import os, sys
3  os.chdir('../src/mwe_matplotlib/pythontex')
4  sys.path.append(os.getcwd())
5
6  from PlotBase import plot_sinus
7  filename = 'sinus_07.pgf'
8  plot_sinus(0.7*pytex.pt_to_in(pytex.context['textwidth']), filename)
9  #pytex.add_created(filename)
10 \end{pycode}
11 \begin{figure}[htb!]
12     \centering
13     %\setlength{\figurewidth}{0.8\columnwidth}% Skalierung der Breite
14     \subimport{}{sinus_07.pgf}
15     \caption{\pythontex PGF-Plot einer Sinus-Funktion.}
16 \end{figure}
```

Quelltext 22: Einbindung von PGF-Plots mittels PythonTeX



Sofern ein `.pgf`-Datei verwendet wird, besteht das Problem darin, dass beim ersten Durchlauf von *pdflatex* noch keine Datei existiert. Diese wird erst zur Laufzeit von *pythontex* erstellt. Daher muss vorher eine Datei mit dem gleichen Namen und der Endung `.pgf` erstellt werden, damit *pdflatex* diese korrekt einbinden kann. Alternativ kann der export über `.pdf` erfolgen. Durch die Einbindung mittels `\includgraphics` wird ein Platzhalter eingebunden und kann somit kann *pdflatex* ohne Fehler durchlaufen.

5.2 Direkte Einbindung über PythonTeX

Eine weitere Alternative wäre die Einbindung der Grafik direkt in der `pycode`-Umgebung, wie es in Quelltext 23 dargestellt ist. Dieser Code erstellt den Plot direkt zur Laufzeit und bindet diesen ein (s. Abbildung 6).

```
\begin{pycode}[plot2]
import os, sys
os.chdir('../src/mwe_matplotlib/pythontex')
sys.path.append(os.getcwd())

from PlotBase import plot_sinus
scalefactor = 0.9
filename = 'sinus_{}.pgf'.format(scalefactor)
plot_sinus(scalefactor*pytex.pt_to_in(pytex.context['textwidth']),
    filename)
pytex.add_created(filename)
print(r"\begin{figure}")
print(r"\centering")
print(r"\subimport{{}}{{{name}}}".format(name=filename))
print(r"\caption{\pythontex PGF-Plot einer Sinus-Funktion direkt aus \
    pythontex.}")
print(r"\label{fig:pythontex-pgf-plot-einer-sinus-funktion-direkt-aus-
    pythontex}")
print(r"\end{figure}")
\end{pycode}
```

Quelltext 23: direkte Einbindung von PGF-Plots mittels PythonTeX

5.3 Vergleich PythonTeX und PGFPlots

Der Vergleich von PGF-Plots und PythonTeX-Plots ist in Abbildung 7 dargestellt. Dabei wird links der Plot mittels PGFPlots direkt erstellt und rechts der Plot mittels PythonTeX und

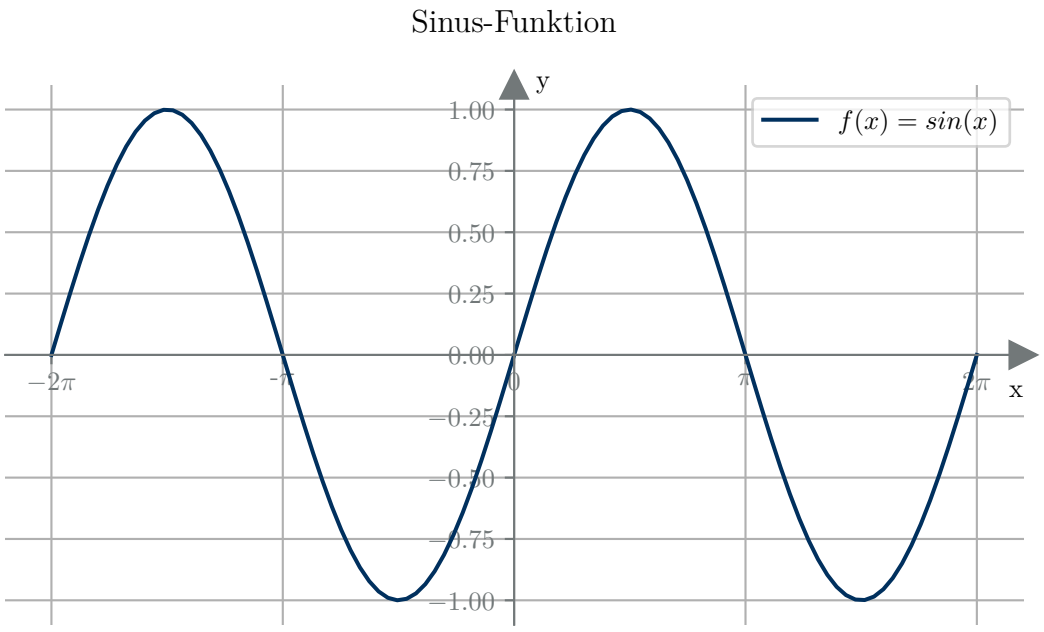


Abbildung 6: PythonTeX PGF-Plot einer Sinus-Funktion direkt aus PythonTeX.

`Matplotlib` erstellt. Unterschiede ergeben sich vorallem in Umfang des Codes und der Art und Weise der Einbindung. Dies ist mit `Matplotlib` und PythonTeX deutlich komplexer und aufwänder. Der eigentliche Umfang zur Erstellung des Plotes ist mit PGFPlots im gewählten Beispiel (Quelltext 10) geringer als mit `Matplotlib` (Quelltext 5). Auch die eigentliche Einbindung in das Dokument fällt geringer aus (Quelltext 24). Weiterhin ist zu erwähnen, dass für unterschiedliche Skalierungen bei PythonTeX mehrere Dateien erstellt werden müssen.

```
\begin{figure}[hb!]
    \begin{minipage}[b]{0.45\textwidth}
        \centering
        \setlength{\figurewidth}{\columnwidth}
        \subimport{../}{pgf_sinus.tex}
    \end{minipage}
    \hfill
    \begin{minipage}[b]{0.45\textwidth}
    \centering
        \begin{pycode}[plot3]
import os, sys
os.chdir('../src/mwe_matplotlib/pythontex')
sys.path.append(os.getcwd())

from PlotBase import plot_sinus
scalefactor = 1
filename = 'sinus_{}.pgf'.format(scalefactor)
plot_sinus(scalefactor*pytex.pt_to_in(pytex.context['textwidth']),
    filename)
pytex.add_created(filename)

print(r"\subimport{{}}{{{name}}}".format(name=filename))
        \end{pycode}
    \end{minipage}
    \caption{Vergleich von GPF direkt (links) und \pythontex (rechts)
        .}\label{fig:pgf-pythontex-comparison}
\end{figure}
```

Quelltext 24: Code zum Vergleich von PGFPlots und PythonTeX

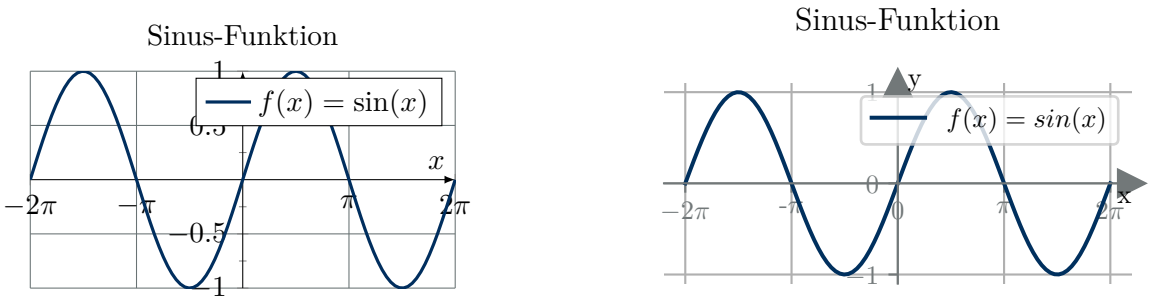


Abbildung 7: Vergleich von GPF direkt (links) und PythonTeX (rechts).

6 Fazit

Die Einbindung von PGF-Plots mittels PythonTeX wurde in Abschnitt 5 auf Seite 14 und Abschnitt 4 auf Seite 10 gezeigt und bietet eine gute Möglichkeit, Python-Code in LaTeX einzubinden. Vor allem bei speziellen Grafiken, die bereits mittels Python erstellt wurden, bietet sich diese Möglichkeit an. Beispielsweise können so Map-Plots oder ähnliches integriert werden, welche über PGFPlots nicht oder nur sehr aufwändig erstellt werden können.

Weiterhin kann PythonTeX für die Integration von Berechnungen oder zur Datenanbindung mittels Datenbankinterfaces aus Python direkt in das LaTeX-Dokument genutzt werden. Dies ermöglicht große Automatisierungsgrade für die Erstellung von Dokumenten. Auch die Integration des Python-Paketes `SymPy` ist ein mächtiges Werkzeug. Es ermöglicht die direkten Einbindung von Symbolrechnungen in LaTeX und ist sogar in PythonTeX mittels spezieller Befehle und Umgebungen integriert.

Die Einbindung von aus Python heraus gespeicherten `Matplotlib`-Plots ist über die Berechnung der Textbreite auch möglich. Diese muss jedoch immer wieder neu erfolgen, sobald sich was am Layout des Dokumentes ändert. Aus diesem Grund ist diese Variante nur bedingt empfehlenswert.

Abschnitt 5 zeigt, dass PGFPlots im Falle des gewählten Beispiels deutlich weniger Aufwand bedeuten und die Integration in LaTeX geringeren Aufwand bedeutet. Diese Methode ist daher zu bevorzugen.

7 Anhang: PlotBase.py

```python
# -*- coding: utf-8 -*-
"""
:author: Tim Häberlein
:version: 1.0
:date: 15.03.2024
:organisation: TU Dresden, FZM
"""

# -------- start import block ---------
import matplotlib as mpl  # import matplotlib as mpl and set pgf as backend
import matplotlib.pyplot as plt  # eigentliche plot-Bibliothek
import matplotlib.ticker as ticker
import numpy as np  # numpy für arrays und so
import copy


# -------- end import block ---------
```

```
18
19
20 # class für tud farben
21
22 class RGBColor(object):
23     """
24     Klasse für die RGB-Farben.
25     """
26
27     def __init__(self, r: int, g: int, b: int):
28         """
29         Constructor for the RGBColor class. Initializes the color
       attributes.
30
31        Args:
32        - r (int): Red component of the color.
33        - g (int): Green component of the color.
34        - b (int): Blue component of the color.
35        """
36         self.r = r
37         self.g = g
38         self.b = b
39         self.items = [self.r, self.g, self.b]
40         self.index = 0
41
42     def __iter__(self):
43         #return tuple({self.r, self.g, self.b})
44         return iter(self.items)
45
46     def __next__(self):
47         if self.index < len(self.items):
48             result = self.items[self.index]
49             self.index += 1
50             return result
51         else:
52             raise StopIteration
53
54     @property
55     def rgb_values(self) -> tuple:
56         return self.r / 255, self.g / 255, self.b / 255
57
58     def __repr__(self):
59         return "RGBColor ({}, {}, {})".format(self.r, self.g, self.b)
60
61
62 class TUDColors(object):
63     """
64     Klasse für die Farben der TU Dresden.
65         # HKS 41 0 / 48 / 94  #00305d - cddarkblue
66         # Cyan     0 / 158 / 224 #009de0 - not defined
67         # HKS 92 114 / 120 / 121 #717778 - cdgray
68         # HKS 44 0 / 106 / 179 #006ab2 - cdblue
69         # HKS 33 147 / 16 / 126  #93107d - cdpurple
70         # HKS 36 84 / 55 / 138 #54368a - cdindigo
71         # HKS 65 106 / 176 / 35 #69af22 - cdgreen
72         # HKS 57 0 / 125 / 64 #007d3f - cddarkgreen
73         # HKS 07 238 / 127 / 0 #ee7f00 - cdorange
74     """
75
76     def __init__(self):
77         """
78         Constructor for the TUDColors class. Initializes the color
       attributes.
```

```
79          Each attribute represents RGB values of the color.
80          """
81          self._cddarkblue = RGBColor(0, 48, 94)
82          self._cdgray = RGBColor(114, 120, 121)
83          self._cdblue = RGBColor(0, 106, 179)
84          self._cdpurple = RGBColor(147, 16, 126)
85          self._cdindigo = RGBColor(84, 55, 138)
86          self._cdgreen = RGBColor(106, 176, 35)
87          self._cddarkgreen = RGBColor(0, 125, 64)
88          self._cdorange = RGBColor(238, 127, 0)
89
90      @staticmethod
91      def shade_color(color: RGBColor, shade) -> RGBColor:
92          """
93          Method to shade a color by a given factor.
94
95          Args:
96          - color: Tuple with RGB values.
97          - shade: Factor to shade the color.
98
99          Returns:
100         - Tuple with RGB values of the shaded color.
101         """
102         white = (255, 255, 255)
103         shade = 1 - shade
104         new_values = (int(c + (white[i] - c) * shade) for i, c in
    enumerate(color))
105         # new_values = [int(c / shade) for c in color]
106         return copy.copy(RGBColor(*new_values))
107
108     def cddarkblue(self, shade=100) -> RGBColor:
109         """
110         Property that returns the RGB values of the color 'cddarkblue'
     shaded by a given factor.
111
112         Args:
113         - shade: Factor to shade the color. Default is 100 (no shading
    ).
114
115         Returns:
116         - Tuple with RGB values of the shaded color.
117         """
118         shade = shade / 100
119         return self.shade_color(self._cddarkblue, shade)
120
121     def cdgray(self, shade=100) -> RGBColor:
122         """
123         Property that returns the RGB values of the color 'cdgray'
    shaded by a given factor.
124
125         Args:
126         - shade: Factor to shade the color. Default is 100 (no shading
    ).
127
128         Returns:
129         - Tuple with RGB values of the shaded color.
130         """
131         shade = shade / 100
132         return self.shade_color(self._cdgray, shade)
133
134     def cdblue(self, shade=100) -> RGBColor:
135         """
136         Property that returns the RGB values of the color 'cdblue'
```

```
        shaded by a given factor.
137
138             Args:
139             - shade: Factor to shade the color. Default is 100 (no shading
        ).
140
141             Returns:
142             - Tuple with RGB values of the shaded color.
143             """
144             shade = shade / 100
145             return self.shade_color(self._cdblue, shade)
146
147         def cdpurple(self, shade=100) -> RGBColor:
148             """
149             Property that returns the RGB values of the color 'cdpurple'
        shaded by a given factor.
150
151             Args:
152             - shade: Factor to shade the color. Default is 100 (no shading
        ).
153
154             Returns:
155             - Tuple with RGB values of the shaded color.
156             """
157             shade = shade / 100
158             return self.shade_color(self._cdpurple, shade)
159
160         def cdindigo(self, shade=100) -> RGBColor:
161             """
162             Property that returns the RGB values of the color 'cdindigo'
        shaded by a given factor.
163
164             Args:
165             - shade: Factor to shade the color. Default is 100 (no shading
        ).
166
167             Returns:
168             - Tuple with RGB values of the shaded color.
169             """
170             shade = shade / 100
171             return self.shade_color(self._cdindigo, shade)
172
173         def cdgreen(self, shade=100) -> RGBColor:
174             """
175             Property that returns the RGB values of the color 'cdgreen'
        shaded by a given factor.
176
177             Args:
178             - shade: Factor to shade the color. Default is 100 (no shading
        ).
179
180             Returns:
181             - Tuple with RGB values of the shaded color.
182             """
183             shade = shade / 100
184             return self.shade_color(self._cdgreen, shade)
185
186         def cddarkgreen(self, shade=100) -> RGBColor:
187             """
188             Property that returns the RGB values of the color 'cddarkgreen
        ' shaded by a given factor.
189
190             Args:
```

```
191         - shade: Factor to shade the color. Default is 100 (no shading
    ).
192 
193         Returns:
194         - Tuple with RGB values of the shaded color.
195         """
196         shade = shade / 100
197         return self.shade_color(self._cddarkgreen, shade)
198 
199     def cdorange(self, shade=100) -> RGBColor:
200         """
201         Property that returns the RGB values of the color 'cdorange'
    shaded by a given factor.
202 
203         Args:
204         - shade: Factor to shade the color. Default is 100 (no shading
    ).
205 
206         Returns:
207         - Tuple with RGB values of the shaded color.
208         """
209         shade = shade / 100
210         return self.shade_color(self._cdorange, shade)
211 
212 
213 # make settings for the plot
214 mpl.use('pgf')
215 plt.rcParams.update({
216     "pgf.texsystem": "pdflatex",  # use pdflatex backend - usually the
     case
217     "font.family": "serif",  # use serif/main font for text elements
218     "text.usetex": True,  # use inline math for ticks
219     "pgf.rcfonts": False,  # don't setup fonts from rc parameters
220     ## You can change the font size of individual items with:
221     # "font.size": 11,
222     # "axes.titlesize": 11,
223     # "legend.fontsize": 11,
224     # "axes.labelsize": 11,
225 })
226 
227 
228 # Set the width of the document
229 def plot_sinus(figurewidth_in,  filename):
230     """
231     Plots the sinus function and saves it as a pgf and pdf file.
232 
233     Args:
234     - figurewidth_in (float): Width of the figure in inches.
235     - filename (str): Name of the file to save the plot to.
236     """
237     fig, ax = plt.subplots(figsize=(figurewidth_in, 0.618*
    figurewidth_in))
238     x = np.linspace(-2*np.pi, 2*np.pi, 100)
239     y = np.sin(x)
240 
241     ax.plot(x, y, label=r'$f(x) = sin(x)$', color=tudcolors.cddarkblue
    ().rgb_values)
242     ax.set_title('Sinus-Funktion', pad=20)
243     ax.grid(True)  # Ensure grid is visible
244 
245     # Adjusting axis labels to be closer to the arrows
246     # ax.set_xlabel('x', labelpad=10, loc='right')  # Positioning
    label at the end
```

```
247     # ax.set_ylabel('y', labelpad=25, loc='top', rotation=0)  #
        Positioning label at the end and rotating
248
249     # Verwenden von Text-Objekten für Achsenbeschriftungen
250     ax.text(1.1 * np.max(x), -0.15, 'x', ha='right', va='center')  # X
        -Achsenbeschriftung
251     ax.text(0.3, 1.1 * np.max(y), 'y', ha='left', va='center',
        rotation=0)  # Y-Achsenbeschriftung
252
253     # Hinzufügen der Legende oben rechts
254     plt.legend(loc='upper right')
255
256     # Adjusting the tick positions
257     ax.spines['left'].set_position(('data', 0))
258     ax.spines['left'].set_color(tudcolors.cdgray().rgb_values)
259     ax.spines['bottom'].set_position(('data', 0))
260     ax.spines['bottom'].set_color(tudcolors.cdgray().rgb_values)
261     ax.spines['right'].set_color('none')
262     ax.spines['top'].set_color('none')
263
264     # Adding arrows
265     ax.plot((1), (0), ls="", marker=">", ms=10, color=tudcolors.cdgray
        ().rgb_values, transform=ax.get_yaxis_transform(), clip_on=False)
266     ax.plot((0), (1), ls="", marker="^", ms=10, color=tudcolors.cdgray
        ().rgb_values, transform=ax.get_xaxis_transform(), clip_on=False)
267
268     # Adjust tick positions to the left and bottom
269     ax.yaxis.set_ticks_position('left')
270     ax.xaxis.set_ticks_position('bottom')
271     # Die Farbe der Tick-Labels an den Achsen ändern
272     ax.tick_params(axis='x', colors=tudcolors.cdgray().rgb_values)  #
        Farbe der X-Achsen-Ticks
273     ax.tick_params(axis='y', colors=tudcolors.cdgray().rgb_values)  #
        Farbe der Y-Achsen-Ticks
274
275     # Set x ticks at multiples of pi and apply the custom formatter
276     ax.xaxis.set_major_locator(ticker.MultipleLocator(base=np.pi))
277     # Define a function to format the x-axis labels as multiples of pi
278     def format_func(value, tick_number):
279         N = int(np.round(value / np.pi))
280         if N == 0:
281             return "0"
282         elif N == 1:
283             return r"$\pi$"
284         elif N == -1:
285             return r"-$\pi$"
286         else:
287             return r"${0}\pi$".format(N)
288     ax.xaxis.set_major_formatter(ticker.FuncFormatter(format_func))
289
290     fig.tight_layout()
291
292     # Speichern der Plots
293     #filename = '{name}.pdf'.format(name=filename)
294     fig.savefig(filename, bbox_inches='tight')
295
296
297 # Function to calc figure size
298 def calc_figsize(width_pt, subplots=(1, 1)):
299     """Set figure dimensions to sit nicely in our document.
300
301     Args:
302         width_pt (float): Document width in points (1 inch = 72.27
```

```
    points)
303         subplots (tuple): Number of rows and columns of subplots.
304
305     Returns:
306         tuple: Figure dimensions in inches.
307     """
308     ## Variablen
309     inches_per_pt = 1 / 72.27
310     # Golden ratio to set aesthetic figure height
311     golden_ratio = (5 ** .5 - 1) / 2
312
313     ## Berechnung
314     # Figure width in inches
315     fig_width = width_pt * inches_per_pt
316     # Figure height in inches
317     fig_height = fig_width * golden_ratio * (subplots[0] / subplots
    [1])
318
319     ## Rueckgabe
320     return fig_width, fig_height
321
322
323 def cmyk_zu_rgb(c: int, m: int, y: int, k: int):
324     """
325     Konvertiere CMYK Farbwerte in RGB.
326
327     Args:
328     - c, m, y, k: CMYK-Werte im Bereich [0, 1].
329
330     Returns:
331     - Eine Tuple mit (R, G, B) Werten im Bereich [0, 255].
332     """
333     r = 255 * (1 - c) * (1 - k)
334     g = 255 * (1 - m) * (1 - k)
335     b = 255 * (1 - y) * (1 - k)
336
337     return int(r), int(g), int(b)
338
339
340 tudcolors = TUDColors()
341
342
343 if __name__ == '__main__':
344     x = calc_figsize(418.25555)
345     print(x)
346     rgb = cmyk_zu_rgb(0.1, 0, 0.05, 0.65)
347     print("RGB:", rgb)
348
349     tudcolors = TUDColors()
350     cddarkblue50 = tudcolors.cddarkblue(50)
351     cddarkblue = tudcolors.cddarkblue()
```

Quelltext 25: Inhalt der PlotBase.py