

2. Semesterarbeit - Grafiken mit Matplotlib & Python

April 25, 2020

Autor	Patrick Michel
6. Semester	FFHS - Fernfachhochschule Schweiz
Fach	AnPy, Analysis mit Python, BSc INF 2017 Pas, BE1-I, FS20
Dozent	Geuss Markus

1 Inhaltsverzeichnis

- Einführung
- Aufgabenstellung
- Systemanforderungen
- Riemann-Integral
- Sehnentrapezregel
- Tangententrapezregel
- Simpsonsche Regel
- Vermutung & Grenzwert
- Genauigkeit der Approximationen
- Reflexion & Einschätzung
- Verweise

2 Einführung

In dieser Arbeit geht es um das numerische Integrieren und somit Bestimmen von Flächen einer Funktion. Dies wird in einem theoretischen Teil beschrieben sowie exemplarisch mit der Programmiersprache Python umgesetzt.

2.1 Aufgabenstellung

1. Beschreiben Sie den Begriff des bestimmten Integrales.
2. Beschreiben Sie die Sekantentrapezregel, die Tangententrapezregel und die Simpsonsche Regel als Näherungsformel für zur Berechnung bestimmter Integrale.
3. Erstellen Sie drei Python-Funktionen
 - `sekanten_trapez_regel(f, a, b, n)`
 - `tangenten_trapez_regel(f, a, b, n)`

- `simpson_regel(f, a, b, n)`
- zur Approximation des Integrals

$$\int_a^b f(x) dx \quad (1)$$

- dabei steht n für die Anzahl Streifen (bzw Doppelstreifen bei Simpson) in den Approximationen.
4. Verwenden Sie die drei Approximationsregeln zur Bestimmung der folgenden Integrale (Variieren Sie dabei die Anzahl Streifen n):

$$\int_0^\pi \sin(x) dx \quad (2)$$

$$\int_{-1}^1 \sqrt{1-x^2} dx \quad (3)$$

$$\int_0^1 x^2 dx \quad (4)$$

5. Stellen Sie Vermutungen auf, was beim Grenzwert $n \rightarrow \infty$ passiert. 6. Überprüfen Sie Ihre Vermutungen mit Sympy. 7. Vergleichen Sie die Genauigkeit der gemachten Approximationen.

2.2 Systemanforderungen

Folgende Voraussetzungen muss die Jupyter Umgebung erfüllen um alle nachfolgenden Codebeispiele ausführen zu können.

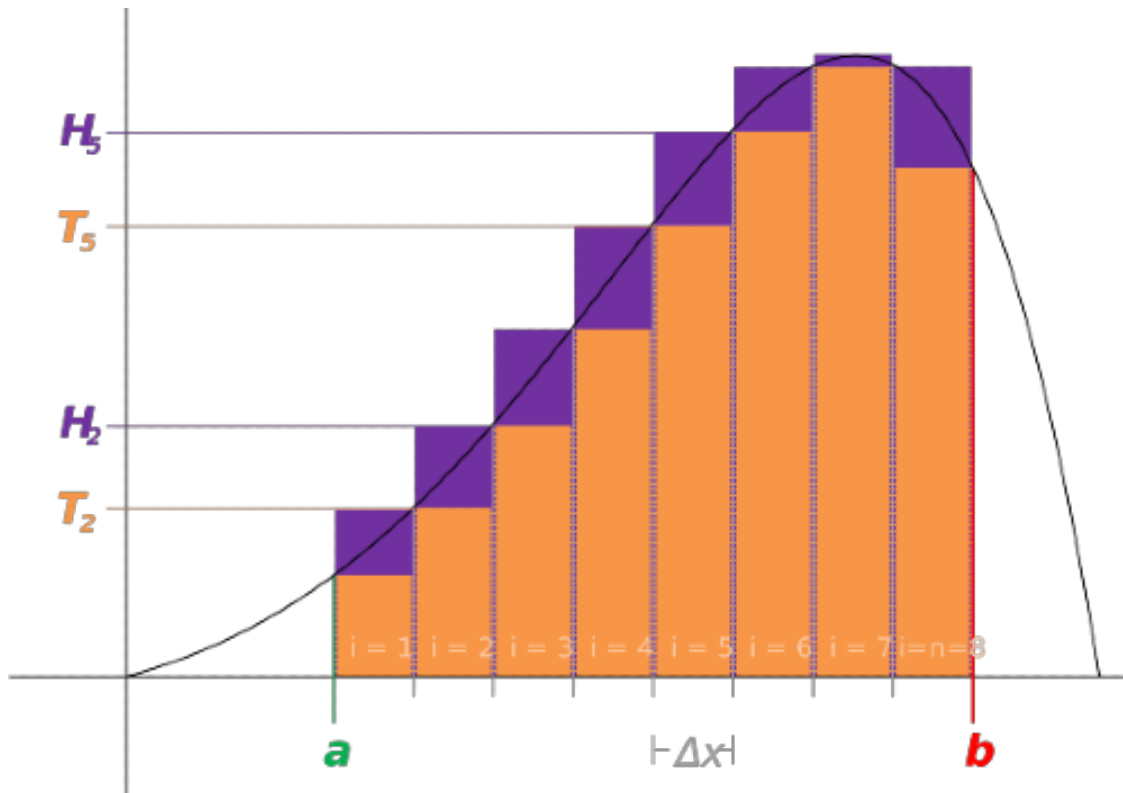
- Python 3.7.x
- Pip Packages
 - Matplotlib
 - Numpy
 - SciPy

Um nachfolgend die Übersicht zu wahren wurden alle nötigen imports in die erste Sektion geholt. Für eine einfachere Handhabung wird der importierte pyplot der alias *plt* vergeben.

```
[302]: from IPython.display import display
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

2.3 Riemann-Integral

Das Riemann Integral ist eine mathematische Methode zur Veranschaulichung des Flächeninhaltes zwischen dem Abschnitt einer x-Achse und dem Graphen einer Funktion.



Das Grundprinzip beruht auf dem Prinzip der Annäherung. Es wird zuerst ein Bereich auf der x-Achse den es zu integrieren gibt definiert. Im obigen Beispiel ist dieser Bereich von $a \rightarrow b$. Danach werden n Punkte welche in einem gleichen Abstand auf dem Funktionsverlauf liegen genommen und aus diesen dann gleich grosse Rechtecke erstellt. Daraus ergibt sich nun eine mehr oder weniger genau Abdeckung der Fläche dieser Funktion. Um die Fläche nun zu bestimmen kann man die Flächen der einzelnen Rechtecke summieren. Je grösser nun n wird also je kleiner die einzelnen Rechtecke werden desto genauer wird das Resultat in Bezug auf die tatsächliche Fläche. Wenn man n nun an ∞ also den Grenzwert annähert kommt schlussendlich der exakte Wert der Fläche heraus. Doch je grösser n ist desto aufwändiger wird natürlich auch die Berechnung für den Computer. Ein Integral von hand zu bestimmen und berechnen ist theoretisch möglich doch macht durch den schnell wachsenden Aufwand keinen Sinn. Daher kann man das gut den Computern überlassen.

2.4 Approximationsregeln & visuelle Überprüfung

In den nachfolgenden Kapiteln kommen nun verschiedene Implementationen von Regeln welche das Bestimmen der Fläche ermöglichen. Auch werden die Resultate geplottet damit eine visuelle Veranschaulichung ermöglicht wird. Doch um zu überprüfen, ob die Implementation korrekt ist wird für die erste Implementation immer die gleiche Integration, die der Fläche eines Halbkreises genommen. Nachfolgend nun die Approximationsregeln zur Verwendung in Python und auch die Logik zur visuellen Darstellung:

```
[312]: # Um die Implementation zu testen ist die Definition einer Funktion nötig
# Als Beispiel nehmen wir die Fläche eines halben Einheitskreises da diese
      ↪ Fläche
```

```

# auch ganz genau durch eine einfachere Formel bestimmt werden kann
# was eine einfache Vergleichbarkeit ermöglicht
# diese Funktion wird auch für die nachfolgenden Implementationen verwendet
appr_circle = [-1, 1, lambda x: np.sqrt(-x**2 + r**2)]

# n-Werte welche für jeden Test verwendet werden
n_to_test = [10, 100, 1000]

```

```

[317]: def test_rules_by_appr(rules, appr, plot=True, n_to_test=n_to_test):
        results = []
        for rule in rules:
            for n in n_to_test:
                A, course_x, course_y = rule[1](appr[2], appr[0], appr[1], n)
                if plot:
                    plt.plot(course_x, course_y, label=f'{rule[0]}: n={n}')
                results.append(A)
        if plot:
            plt.legend()
            plt.show()
        return results

```

```

[318]: # Exakte Fläche eines Halbkreises
A_E = (r**2 * np.pi) / 2

def create_results_table(results, exact=True):
    table = []

    if exact:
        table.append([0, A_E, 'Exakt'])

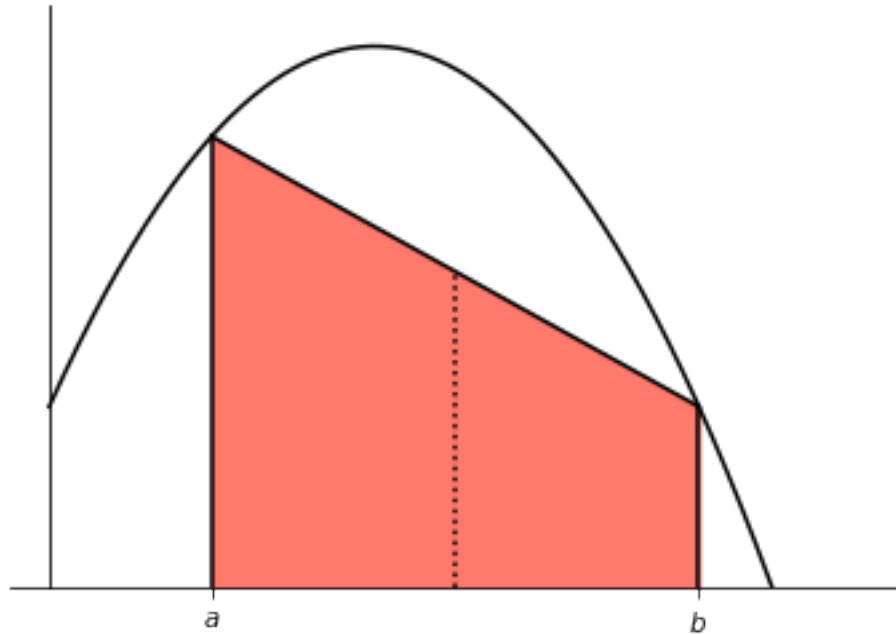
    for i in range(0, len(n_to_test)):
        percent = results[i] / A_E if results[i] < A_E else A_E / results[i]
        table.append([n_to_test[i], results[i], f'{round(percent * 100, 3)}%'])

    return table

```

2.5 Sehnentrapezregel

Bei der Sehnentrapezregel wird die Annäherung durch Trapeze umgesetzt.



Der Flächeninhalt des Trapezes kann durch folgende Formel bestimmt werden:

$$ST_1 = (b - a) * \frac{f(a) + f(b)}{2} \quad (5)$$

Um nun die Fläche genauer zu bestimmen wird wie beim Riemann Integral der Bereich von $a \rightarrow b$ in n Sehnentrapeze aufgeteilt. Um dann die gesamte Fläche zu bestimmen, wird wieder die Fläche jedes einzelnen Sehnentrapezes gerechnet und dann daraus die Summe gebildet. Dies kann je nach Funktion schon eine sehr gute Näherung im Vergleich zu Rechtecken bringen. Nachfolgend nun die allgemeine Sehnentrapezformel:

$$A = \frac{b - a}{n} * \left(\frac{f(x_0)}{2} + f(x_1) + f(x_2) + \dots + \frac{f(x_n)}{2} \right) \quad (6)$$

Nun folgt die Implementierung dieser Formel in Python.

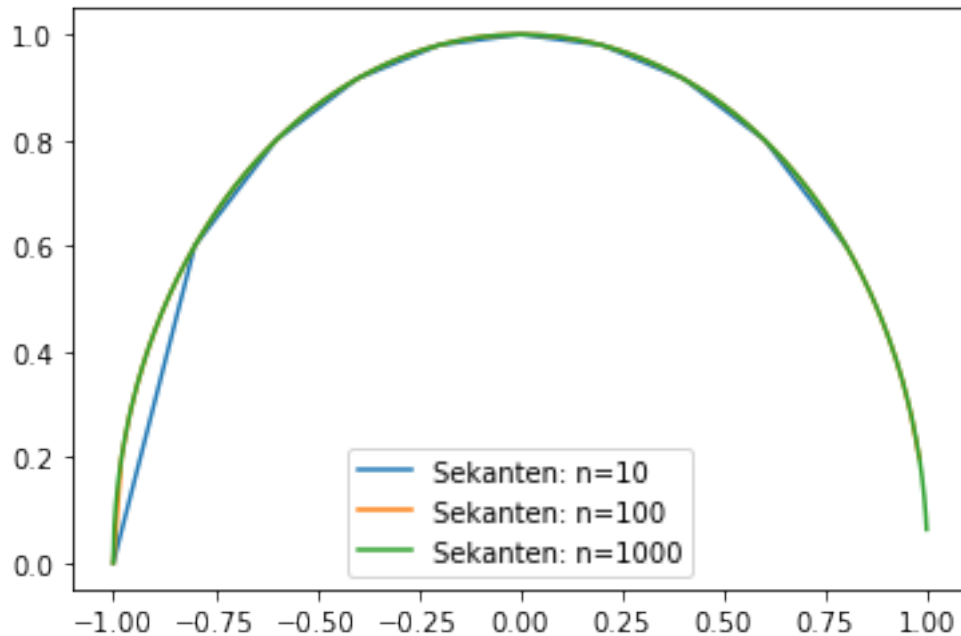
```
[319]: def sekanten_trapez_regel(f, a, b, n):
    A, x, y = 0, [], []
    steps = (b - a) / n
    for i in range(0, n):
        x.append(a + i * steps)
        if i == 0 or i == n:
            y.append(f(x[-1]) / 2)
        else:
            y.append(f(x[-1]))
        A += y[-1] * steps
    pass
    return A, x, y
```

```
[320]: # Dies ist nun ein Beispiel der Implementation
# die Genauigkeit wird anhand der Kreis Approximation berechnet

st_results = test_rules_by_appr(['Sekanten', sekanten_trapez_regel],
    ↪appr_circle)

st_table = create_results_table(st_results)

display(pd.DataFrame(st_table, columns=["n-Wert", "Resultat", "Genauigkeit"]))
```

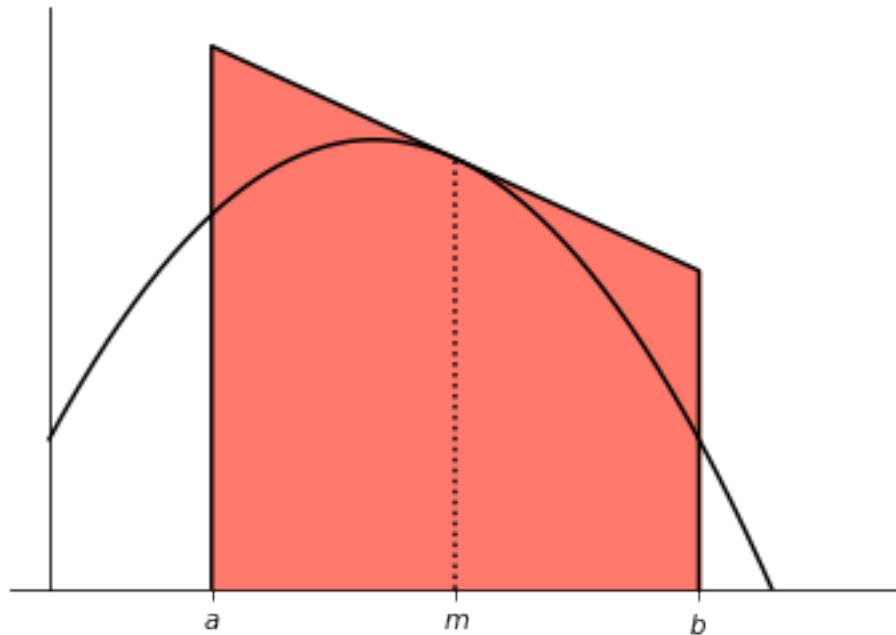


	n-Wert	Resultat	Genauigkeit
0	0	1.570796	Exakt
1	10	1.518524	96.672%
2	100	1.569134	99.894%
3	1000	1.570744	99.997%

Die Obere Grafik zeigt einen Halbkreis welcher integriert wird. Die verschiedenen Farben stellen die verschiedenen Verläufe bei unterschiedlichen n-Werten dar. Bereits schon ab $n = 100$ ist der Verlauf der Integration nicht mehr sichtbar da die Genauigkeit schon bei 99.89% liegt. In der allgemeinen Formel wird x_0 und x_n durch 2 geteilt. Ist n aber genügend gross hat das praktisch keinen Einfluss mehr auf die Genauigkeit.

2.6 Tangententrapezregel

Bei der Tangententrapezregel wird die Annäherung ähnlich wie bei der Sehnentrapezregel durch Trapeze umgesetzt. Allerdings wird hier als obere schräge Seite die Mitte eines Trapezes genommen.



Der Flächeninhalt des Trapezes kann durch folgende Formel bestimmt werden:

$$TT_1 = (b - a) * f\left(\frac{a + b}{2}\right) \quad (7)$$

Um nun die Fläche genauer zu bestimmen wird wieder der Bereich von $a \rightarrow b$ in n Streifen aufgeteilt. Um dann die gesamte Fläche zu bestimmen, wird wieder die Fläche jedes einzelnen Trapezes gerechnet und dann daraus die Summe gebildet. Nachfolgend nun die allgemeine Tangententrapezformel:

$$A = \frac{b - a}{n} * \left(f\left(\frac{x_0 + x_1}{2}\right) + f\left(\frac{x_1 + x_2}{2}\right) + \dots + f\left(\frac{x_{n-1} + x_n}{2}\right) \right) \quad (8)$$

Nun folgt die Implementierung dieser Formel in Python.

```
[321]: def tangenten_trapez_regel(f, a, b, n):  
    A, x, y = 0, [], []  
    steps = (b - a) / n  
    for i in range(0, n):  
        x.append(a + i * steps)  
        y.append(f((x[-1] + x[-1] + steps) / 2))  
        #if i > 0 and i != n:  
            #y.append(f((x[-1] + x[-2]) / 2))
```

```

    #else:
        #y.append(f(x[-1]))
    A += y[-1] * steps
    pass
    return A, x, y

```

```

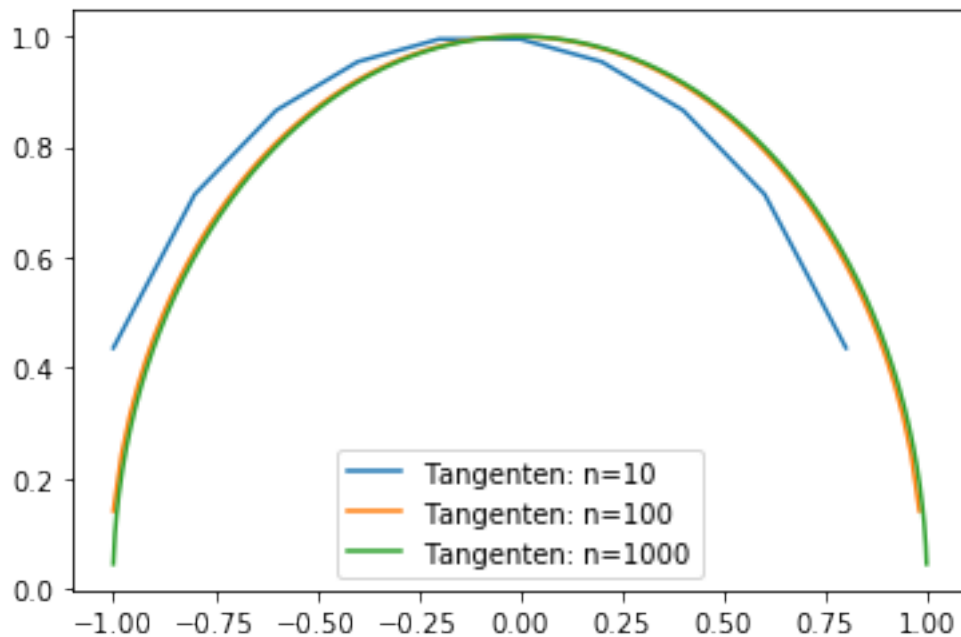
[322]: # Dies ist nun ein Beispiel der Implementation
# die Genauigkeit wird anhand der Kreis Approximation berechnet

tt_results = test_rules_by_appr(['Tangenten', tangenten_trapez_regel],
    ↳appr_circle)

tt_table = create_results_table(tt_results)

display(pd.DataFrame(tt_table, columns=["n-Wert", "Resultat", "Genauigkeit"]))

```



	n-Wert	Resultat	Genauigkeit
0	0	1.570796	Exakt
1	10	1.585994	99.042%
2	100	1.571283	99.969%
3	1000	1.570812	99.999%

Die Obere Grafik zeigt einen Halbkreis welcher integriert wird. Die verschiedenen Farben stellen die verschiedenen Verläufe bei unterschiedlichen n-Werten dar. Was auffällt bei $n = 10$ ist der Verlauf noch extrem verschoben. Doche bereits schon ab $n = 100$ ist der Verlauf der Integration

nicht mehr sichtbar da die Genauigkeit schon bei 99.85% liegt.

2.7 Simpsonsche Regel

Nach den Resultaten von oben sollte die Sehnentrapezregel immer etwas zu wenig und die Tangententrapezregel immer etwas zu viel Fläche angeben. Bei der Simpsonschen Regel ist nun die Idee, dass man diese zwei bekannten Eigenschaften kombiniert. Das heisst man erzeugt ein gewichtetes Mittel. Das Mittel ist gewichtet, da die Tangententrapezregel meist etwas näher am tatsächlichen Wert liegt.

Aus den oben ausgeführten Überlegungen ergibt sich folgende Formel:

$$S_n := \frac{ST_n + 2 * TT_n}{3} \quad (9)$$

Nun folgt die Implementierung dieser Formel in Python.

```
[323]: def simpson_regel(f, a, b, n):
    st_A, st_x, st_y = sekanten_trapez_regel(f, a, b, n)
    tt_A, tt_x, tt_y = tangenten_trapez_regel(f, a, b, n)
    A, x, y = (st_A + 2 * tt_A) / 3, [], []

    for i in range(0, len(st_x)):
        x.append((st_x[i] + 2 * tt_x[i]) / 3)

    for i in range(0, len(st_y)):
        y.append((st_y[i] + 2 * tt_y[i]) / 3)

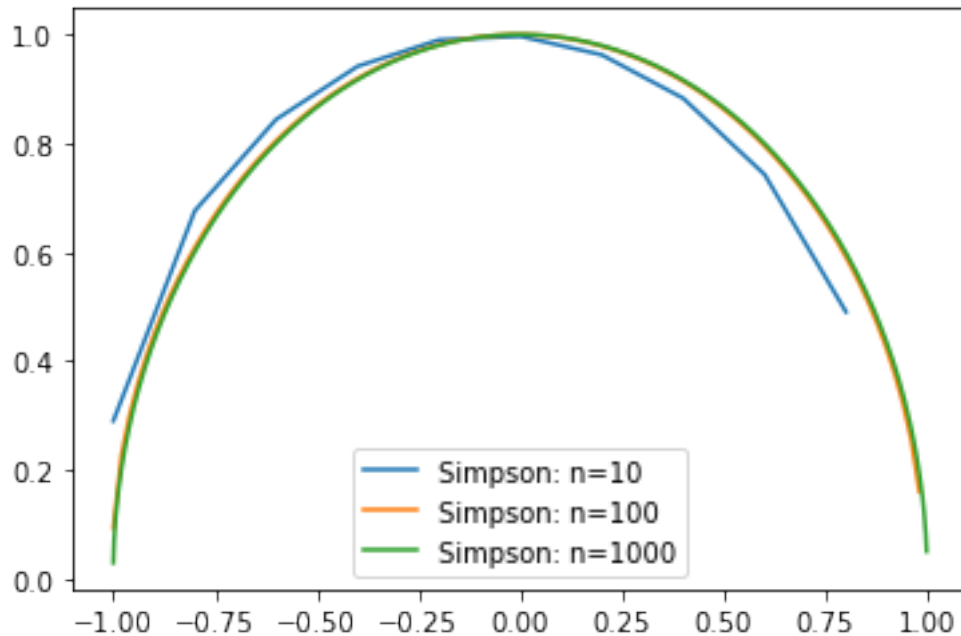
    return A, x, y

[324]: # Dies ist nun ein Beispiel der Implementation
# die Genauigkeit wird anhand der Kreis Approximation berechnet

s_results = test_rules_by_appr(['Simpson', simpson_regel], appr_circle)

s_table = create_results_table(s_results)

display(pd.DataFrame(s_table, columns=["n-Wert", "Resultat", "Genauigkeit"]))
```



	n-Wert	Resultat	Genauigkeit
0	0	1.570796	Exakt
1	10	1.563504	99.536%
2	100	1.570567	99.985%
3	1000	1.570789	100.0%

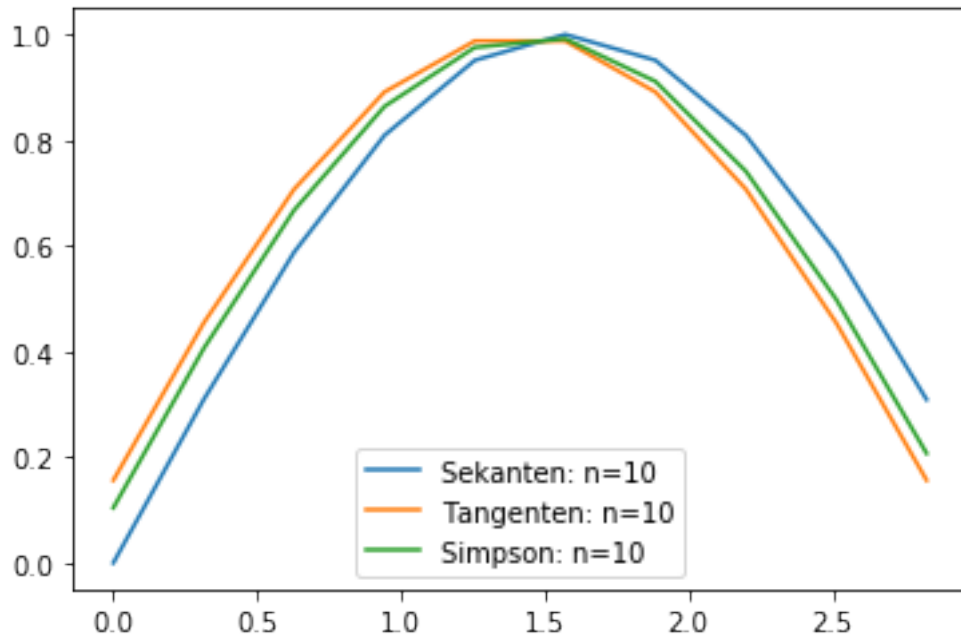
Das Kombinieren der zwei Verfahren hat wie in der Theorie beschrieben funktioniert. Das kombinierte Ergebnis ist von der Genauigkeit in jedem Fall besser als die einzelnen Regeln. Mit dieser Regel ist eine Genauigkeit von 100% bei $n = 100$ gegeben.

2.8 Gegenüberstellung

Um diese verschiedenen Regeln nun vernünftig vergleichen zu können werden diese auf die in der Aufgabenstellung definierten Integrale angewendet.

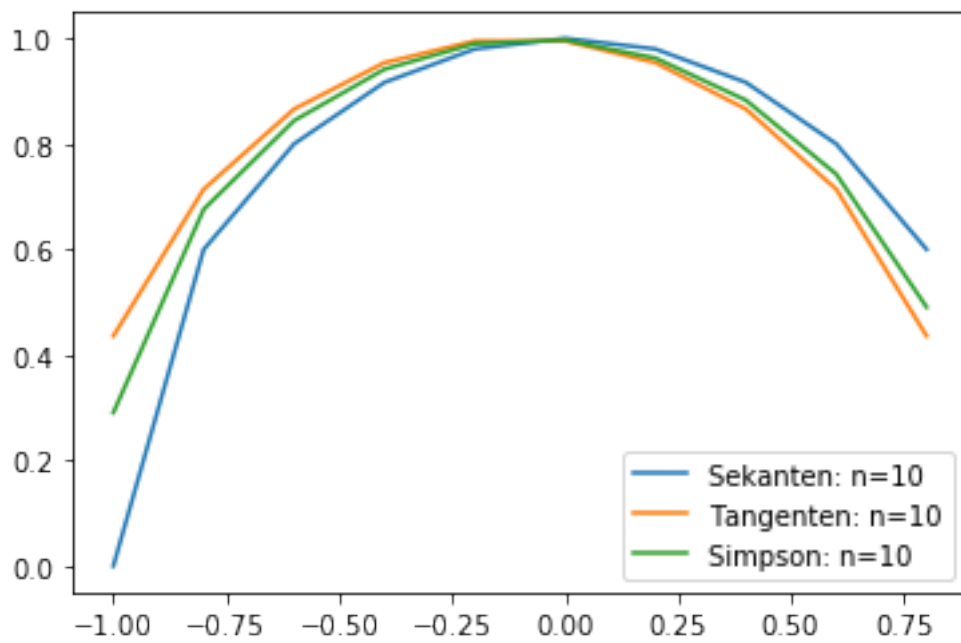
```
[325]: # Integrationen der Aufgabenstellung
# Werden beim Vergleich verwendet
appr_sin = [0, np.pi, lambda x: np.sin(x)]
appr_sqrt = [-1, 1, lambda x: np.sqrt(1 - x**2)]
appr_square = [0, 1, lambda x: x**2]
```

```
[332]: results = test_rules_by_appr([['Sekanten', sekanten_trapez_regel],
                                     ['Tangenten', tangenten_trapez_regel],
                                     ['Simpson', simpson_regel]], appr_sin, True, [10])
display(results)
```

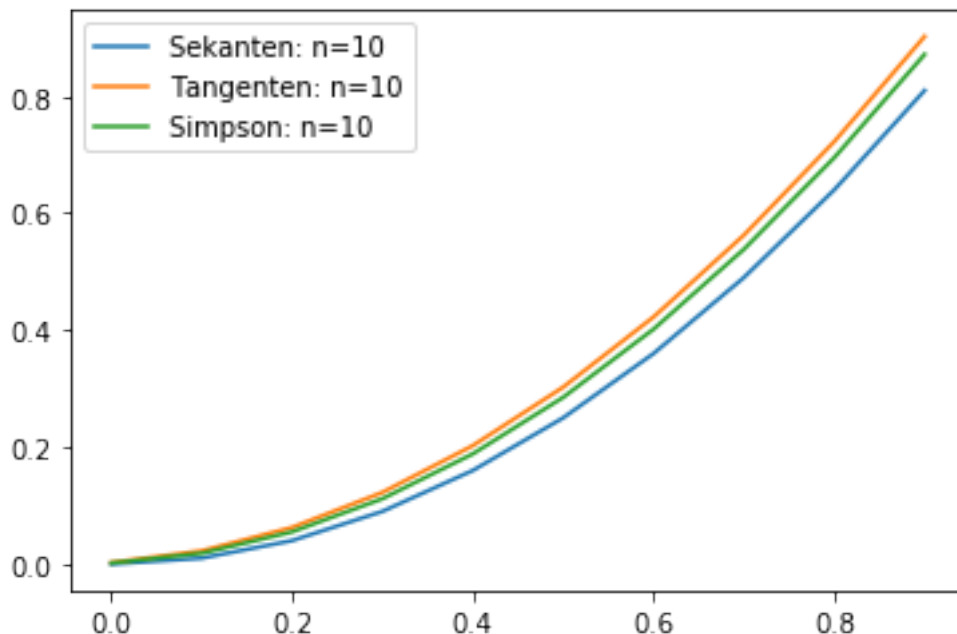


[1.9835235375094544, 2.0082484079079745, 2.0000067844418012]

```
[329]: null = test_rules_by_appr([['Sekanten', sekanten_trapez_regel],
                                   ['Tangenten', tangente_trapez_regel],
                                   ['Simpson', simpson_regel]], appr_sqrt, True, [10])
```



```
[330]: null = test_rules_by_appr([[ 'Sekanten', sekanten_trapez_regel],  
                                  [ 'Tangenten', tangenten_trapez_regel],  
                                  [ 'Simpson', simpson_regel]], appr_square, True, [10])
```



2.9 Reflexion & Einschätzung

Wie oben gezeigt bieten die vorgestellten Pakete oder Bibliotheken extrem viele Möglichkeiten Daten zu visualisieren. Nicht alle Darstellungsformen machen für alle Daten Sinn daher ist es wichtig die Darstellungsform immer auf die Daten und die daraus zu gewinnenden Daten abzustimmen. Matplotlib ist in Kombination mit Numpy extrem effizient und sehr einfach zu handhaben. Meiner persönlichen Meinung dienen diese Pakete in Kombination als eine Alternative für die Programmiersprache R welche sehr ähnliche Möglichkeiten bietet. Auch ist es sehr wertvoll bestimmte Themen noch einmal programmatisch zu visualisieren. Dazu zählen auch die Laufzeitklassen welche durch das in einem oberen Kapitel erstellten Diagramm extrem gut sichtbar gemacht werden konnten. Auch bietet die Programmiersprache Python sehr viele sprachliche Besonderheiten welche es extrem einfach machen mit Listen zu arbeiten. Die Standarddarstellung der Graphen ist nicht immer extrem aussagekräftig. Daher ist es eigentlich immer nötig diese den eigenen Bedürfnissen anzupassen. Dazu gehört das Anpassen der Farben oder auch das Sortieren der Daten damit sie schlussendlich mehr Sinn ergeben.

2.10 Verweise

https://de.wikipedia.org/wiki/Riemannsches_Integral

<https://www.math.ubc.ca/~pwalls/math-python/integration/riemann-sums/>

<https://de.wikipedia.org/wiki/Trapezregel>

<https://www.mathe-seite.de/oberstufe/analysis-tiefere-einblicke/naehrungsverfahren/trapezregel/>

<https://mathepedia.de/Trapezregel.html>

- [1]: <https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet>
- [2]: <https://pythonbasics.org/matplotlib-line-chart/>
- [3]: <https://de.wikipedia.org/wiki/Landau-Symbole>
- [4]: <https://bodo-schoenfeld.de/jupyter-notebook-balkendiagramm-erstellen/>
- [5]: Findeisen, P. Die Charakterisierung der Normalverteilung nach Gauß. Metrika 29, 55–63 (1982). <https://doi.org/10.1007/BF01893364>
- [6]: <https://www.parkmycloud.com/blog/aws-vs-azure-vs-google-cloud-market-share/>

[]: