Numerische Mathematik - Projektteil 1

1 Trapezregel und Gauß-Quadratur im Vergleich

Ziel dieses Projekts ist es asymptotisch exponentiell abfallende Funktionen auf dem unbeschränkten Intervall $[0, +\infty)$ numerisch zu integrieren. Dazu untersuchen wir zwei mögliche Strategien:

(a) Abschneiden des unbeschränkten Intervalls und Anwendung einer Quadraturformel für ein beschränktes Intervall $[0, T]$ für $T > 0$

(b) Anwendung einer Gauß-Quadratur für die Gewichtsfunktion $w(x) = \exp(-x)$

a)

Sei $f : [0, +\infty) \to \mathbb{R}$ eine beschränkte Funktion, deren erste und zweite Ableitung ebenso beschränkt sind. Desweiteren definieren wir für eine Menge $M \subset [0, +\infty)$ das gewichtete Integral von f mit

$$Q_M(f) := \int_M f(x) \exp(-x) dx.$$

Wir approximieren das Integral $Q_{[0,\infty)}(f)$ indem wir für ein $T > 0$ das Integral auf dem beschränkten Intervall $Q_{[0,T]}$ durch die summierte Trapezregel approximieren. Wir definieren

$$Q_{T,h}(f) := \frac{h}{2} \left(\exp(-0) f(0) + 2 \sum_{j=1}^{N-1} \exp(-jh) f(jh) + \exp(-T) f(T) \right)$$

und wollen nun eine Abschätzung des Fehlers herleiten.
Mit der Dreiecksungleichung erhalten wir:

$$|Q_{[0,\infty)}(f) - Q_{T,h}(f)| \leq |Q_{[0,\infty)}(f) - Q_{[0,T)}(f)| + |Q_{[0,T)}(f) - Q_{T,h}(f)|$$

Der erste Term lässt sich direkt berechnen:

$$|Q_{[0,\infty)}(f) - Q_{[0,T)}(f)| = \int_T^\infty \exp(-x) f(x) dx = \exp(-T)$$

Dank Satz 4.8. aus dem Vorlesungsskript wissen wir:

$$|Q_{[0,T)}(f) - Q_{T,h}(f)| \leq \frac{T}{12} h^2 \sup_{x \in [0,T]} \{\exp(-x)(f''(x) - 2f'(x) + f(x))\}$$

Also erhalten wir mit $C_2 := \frac{\sup_{x \in [0,T]} \{\exp(-x)(f''(x) - 2f'(x) + f(x))\}}{12}$:

$$|Q_{[0,\infty)}(f) - Q_{T,h}(f)| \leq \exp(-T) + C_2 T h^2$$

b)

Nun gilt es die theoretischen Resultate aus Aufgabe a) zu implementieren und durch geeignete numerische Beispiele zu testen. Wir haben uns für die Programmiersprache Python entschieden.

```
def trapez (f,N,T):
    x = np.linspace(0,T,N)
    summe = sum([f(x[i])*np.exp(-x[i]) for i in range(1,N-1)])
    trapez = 2*summe
    trapez += f(x[0])+f(x[N-1])*np.exp(-x[N-1])
    trapez *= (T/(2*(N-1)))
    return trapez
```

Wir betrachten die Funktion $f : x \mapsto \sin(x)$ mit

$$\int_0^\infty \sin(x)\exp(-x)dx = \frac{1}{2}$$

In Abbildung 1 sieht man den Fehler in Abhängigkeit von h anhand einigen ausgewählten, festen Werten für T. In Abbildung 2 ist der Fehler abhängig von T mit einigen festen Werten für h zu sehen.
Jetzt wollen wir das optimale h in Abhängigkeit von T finden, das abhängig von der Anzahl der Funktionsauswertungen den geringsten Fehler liefert. In Abbildung 1 sieht man, dass bei festem T irgendwann ein Punkt erreicht ist, ab dem der Fehler abhängig von T dominiert und eine weitere Verkleinerung der Teil-Intervalllänge h keine merkbare Verbesserung des Fehlers herbeiführen würde.
Ebenso verhält es sich in Abbildung 2, wobei hier ab einem gewissen Punkt eine weitere Vergrößerung von T, sogar zu einer Verschlechterung des Fehlers führen würde. Daher scheint es am sinnvollsten, h so zu wählen, dass sich die Fehler $T\epsilon_h$ und ϵ_T die Waage erhalten. Lösen wir nun nach h auf, erhalten wir: $h = \frac{\exp(-T)}{T}$.
Damit erhalten wir folgende Konvergenzplots:

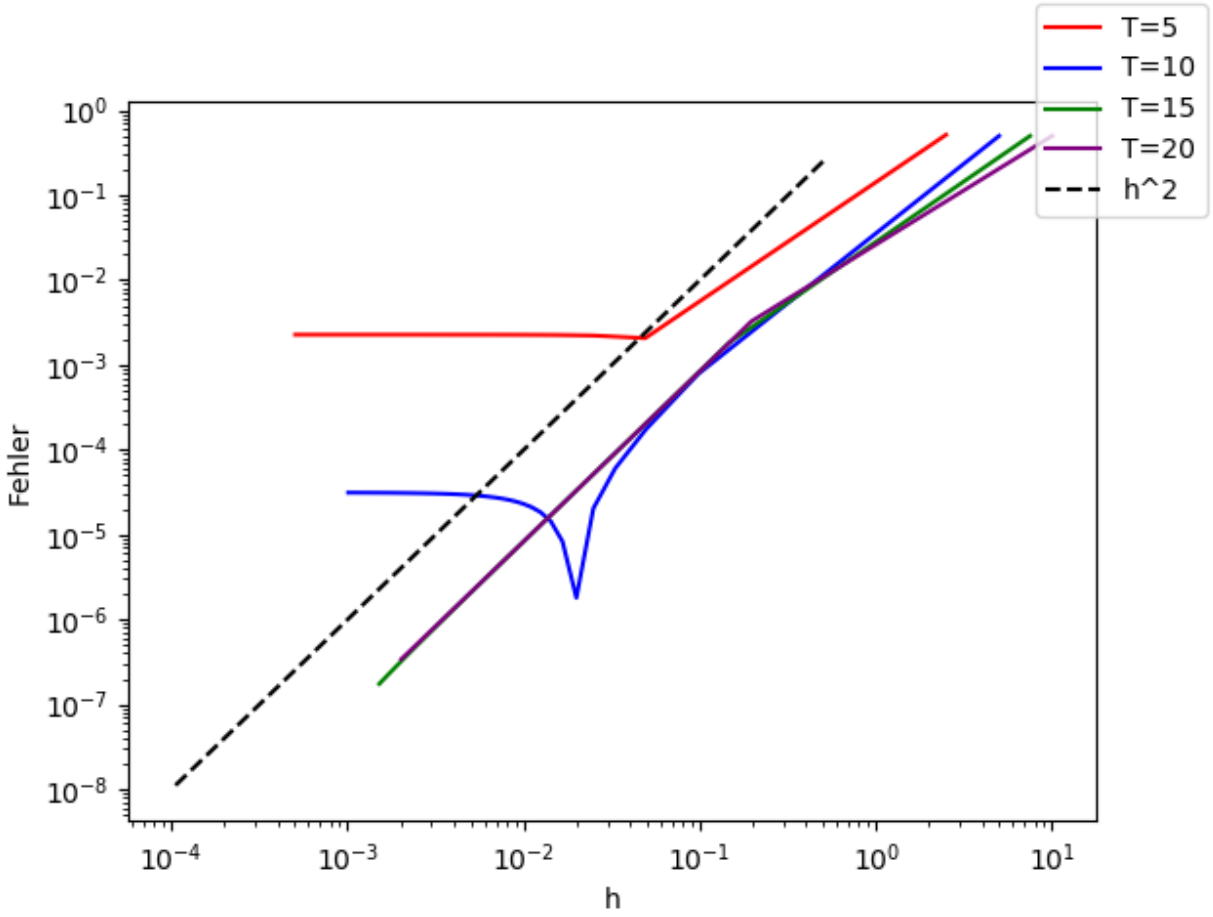


Abbildung 1: T fest, h variabel

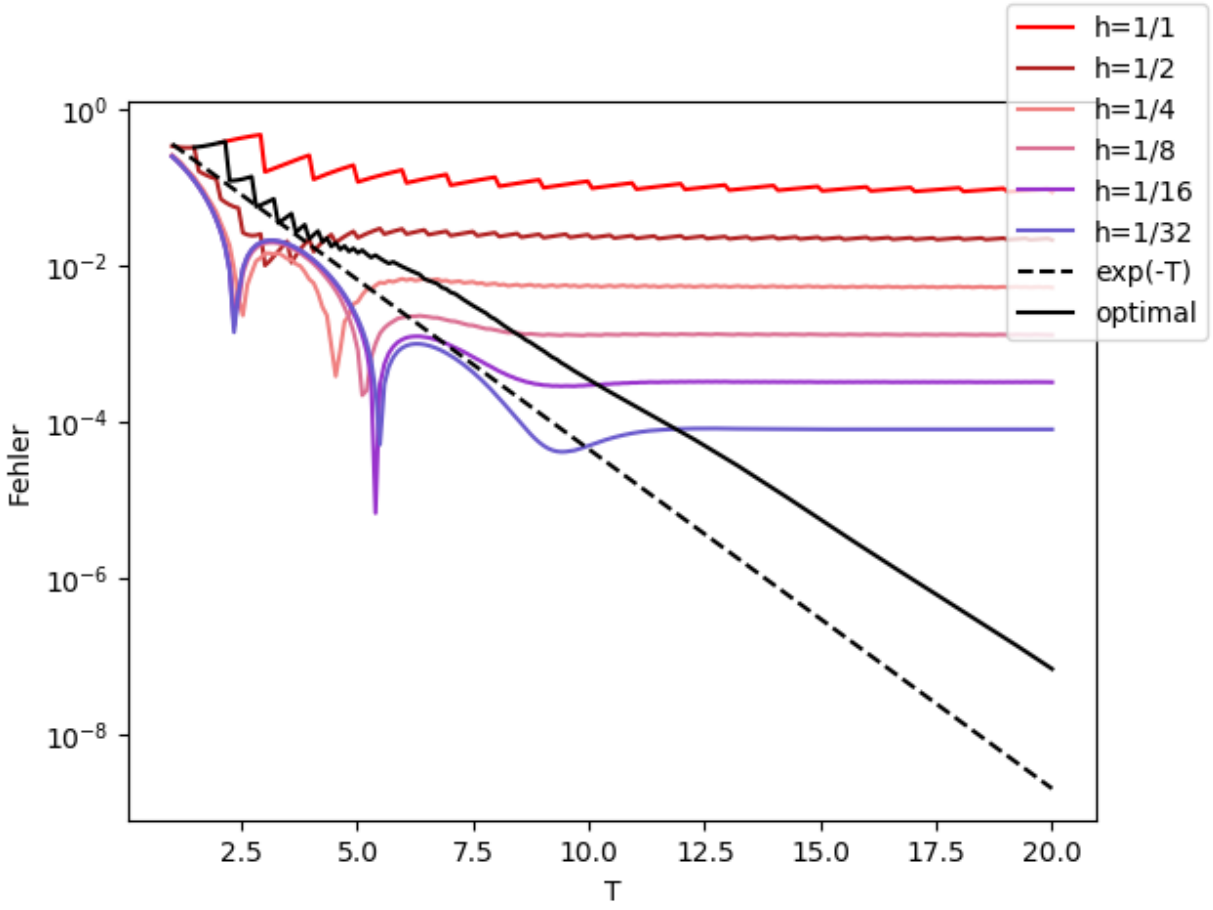


Abbildung 2: h fest, T variabel

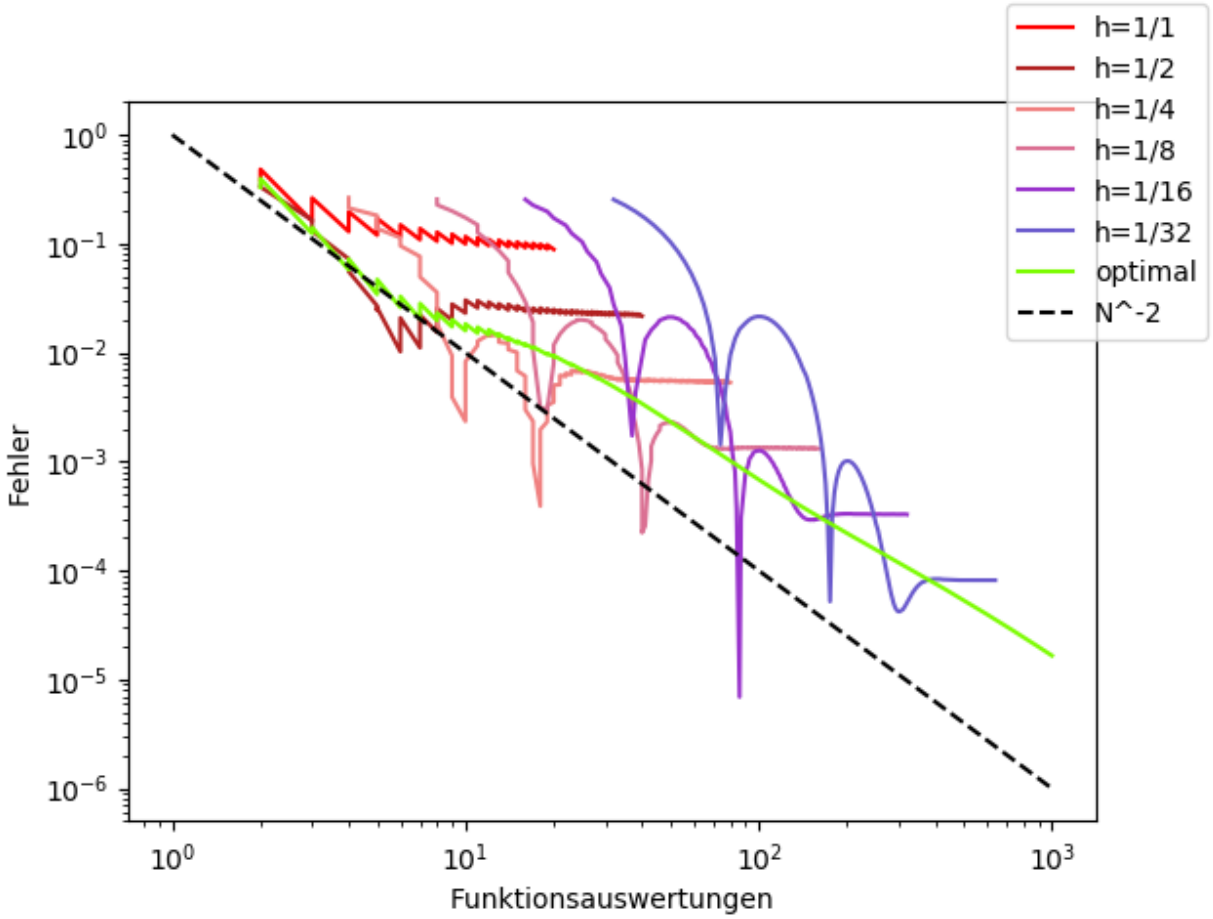


Abbildung 3: Vergleich: feste h gegen optimales h

c)

Um uns als nächstes der zweiten möglichen Strategie zuwenden zu können, müssen wir vorerst mit etwas theoretischer Vorarbeit starten, um die benötigten Quadraturpunkte und Quadraturgewichte für die Gaussquadratur berechnen zu können. In Übungsaufgabe 39 wurde gezeigt, dass die Polynome p_j gegeben durch die Dreitermrekursion

$$p_0(x) = 1, p_1(x) = x - 1, p_{n+1}(x) = (x - 2n - 1)p_n(x) - n^2 P_{n-1}(x)$$

die Orthogonalpolynome zur Gewichtsfunktion $\omega(x) = \exp(x)$ sind. Wir zeigen noch, dass die Nullstellen von p_n genau die Eigenwerte der Matrix

$$T_n = \begin{pmatrix} 1 & -1 & & \\ -1 & 3 & \ddots & \\ & \ddots & \ddots & -n+1 \\ & & -n+1 & 2n-1 \end{pmatrix}$$

sind und dass die zugehörigen Gauß-Gewichte durch die Quadrate der ersten Einträgen der zugehörigen normierten Eigenvektoren sind.

$$\begin{pmatrix} 1 & -1 & & \\ -1 & 3 & \ddots & \\ & \ddots & \ddots & -n+1 \\ & & -n+1 & 2n-1 \end{pmatrix} \cdot \begin{pmatrix} \tau_0 p_0(x_k) \\ \tau_1 p_1(x_k) \\ \vdots \\ \tau_{n-1} p_{n-1}(x_k) \end{pmatrix} =$$

$$\begin{pmatrix} \tau_0 - \tau_1 p_1(x_k) \\ \vdots \\ -(i-1)\tau_{i-2} p_{i-2}(x_x) + (2i-1)\tau_{i-1} p_{i-1}(x_k) + (-i)\tau_i p_i(x_k) \\ \vdots \\ -(n-1)\tau_{n-2} p_{n-2}(x_x) + (2n-1)\tau_{n-1} p_{n-1}(x_k) \end{pmatrix}$$

Betrachten wir nun die i-te Zeile für $1 < i < n$ und gehen mit p_i in die Rekursion

$$\begin{aligned} &= \cdots + (-i)\tau_i[(x_k - 2(i-1) - 1)p_{i-1}(x_k) - (i-1)^2 p_{i-2}(x_k)] \\ &= \cdots + (-i)\tau_i[(x_k - (2i-1))p_{i-1}(x_k) - (i-1)^2 p_{i-2}(x_k)] \end{aligned}$$

Durch Einsetzen erhalten wir $(-i)\tau_i = \tau_{i-1}$ und $\tau_{i-1}(i-1)^2 = -(i-1)\tau_{i-2}$ und somit ergibt sich nur noch $x_k \tau_{i-1} p_{i-1}(x_k)$ in der i-ten Zeile. In der ersten Zeile erhält man durch Einsetzen wiederum $x_k \tau_0$.
Wir wissen, dass x_k die Nullstellen von p_n sind, somit können wir in der letzten Zeile die Terme $-n\tau_n p_n(x_k) + n\tau_n p_n(x_k)$ hinzufügen:

$$-(n-1)\tau_{n-2} p_{n-2}(x_x) + (2n-1)\tau_{n-1} p_{n-1}(x_k) - n\tau_n p_n(x_k) + n\tau_n p_n(x_k)$$

Durch das Ergebnis von vorhin und mit dem Wissen, dass $p_n(x_k) = 0$, erhalten wir für die letzte Zeile $x_k \tau_{n-1} p_{n-1}(x_k)$. Somit haben wir gezeigt, dass x_k Eigenwerte von T_n sind.
Im zweiten Teil zeigen wir, dass die Quadraturgewichte durch $w_{k,n} = \frac{1}{\|\nu_{k,n}\|^2}$ gegeben sind. $w_{k,n}$ sind die

Gewichte der Gauss-Quadratur zur Gewichtsfunktion e^{-x}.

$$\sum_{j=0}^{n-1}\sum_{l=0}^{n} w_{l,n}\tau_j^2 p_j(x_k)p_j(x_l) = \sum_{j=0}^{n-1}\left(\sum_{l=0}^{n} w_{l,n}p_j(x_l)\right)\tau_j^2 p_j(x_k) =$$

$$\sum_{j=0}^{n-1}\left(\int_0^\infty e^{-x}p_j(x)dx\right)\tau_j^2 p_j(x_k) = \sum_{j=0}^{n-1}\left(\int_0^\infty e^{-x}p_j(x)p_0(x)dx\right)\tau_j^2 p_j(x_k)$$

Wir wissen, dass $p_j(x)$ untereinander orthogonal zur Gewichtsfunktion e^{-x} sind. Somit sind alle Summanden $= 0$, ausser wenn $j = 0$. Das Integral von $\int_0^\infty e^{-x}dx$ beträgt 1.

$$\sum_{j=0}^{n-1}\left(\int_0^\infty e^{-x}p_j(x)p_0(x)dx\right)\tau_j^2 p_j(x_k) = 1 \cdot \tau_0^2 p_0(x_k) = 1$$

Gleichzeitig erhalten wir unter der Ausnutzung, dass die Eigenvektoren einer symmetrischen Matrix bezüglich der euklidischen Skalarprodukts orthogonal aufeinander stehen:

$$\sum_{j=0}^{n-1}\sum_{l=0}^{n} w_{l,n}\tau_j^2 p_j(x_k)p_j(x_l) = \sum_{l=0}^{n} w_{l,n}\sum_{j=0}^{n-1}\tau_j p_j(x_k)\tau_j p_j(x_l) =$$

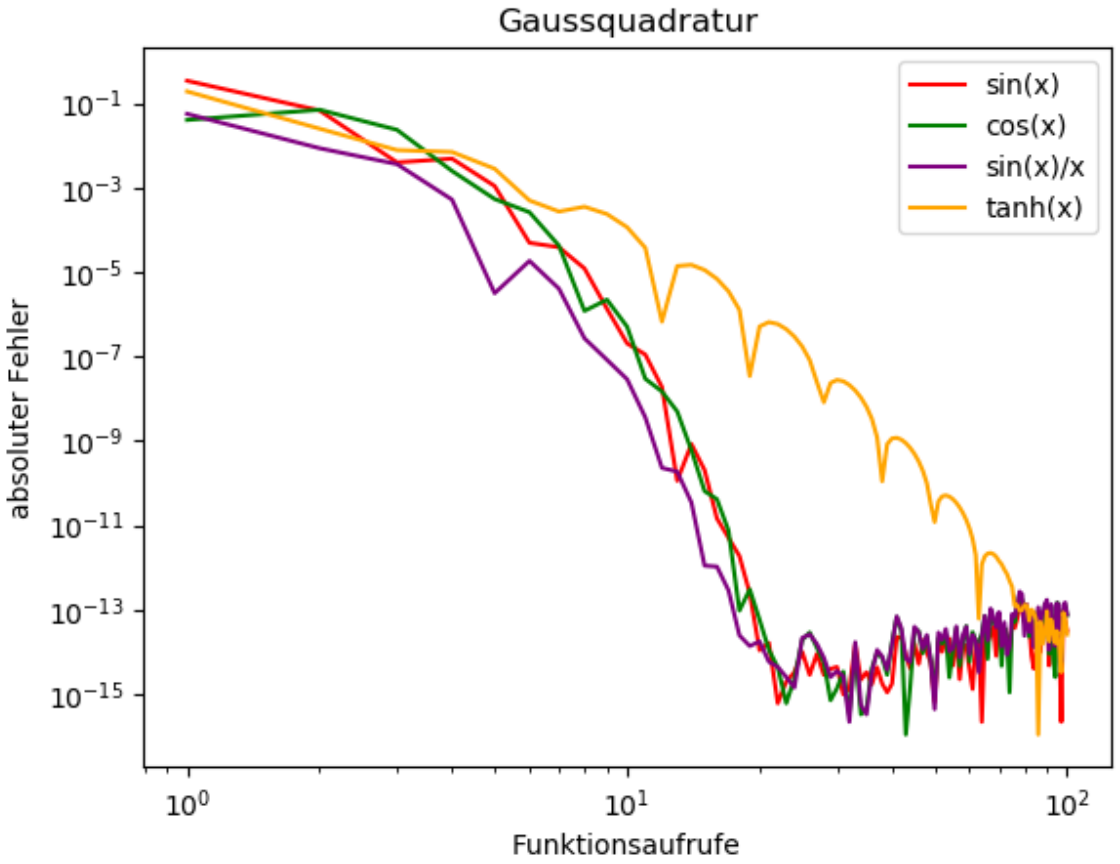
$$\sum_{l=0}^{n} w_{l,n}\sum_{j=0}^{n-1}\nu_{k,n}^\top\nu_{l,n} = w_{k,n}\nu_{k,n}^\top\nu_{k,n} = w_{k,n}\|\nu_{k,n}\|^2$$

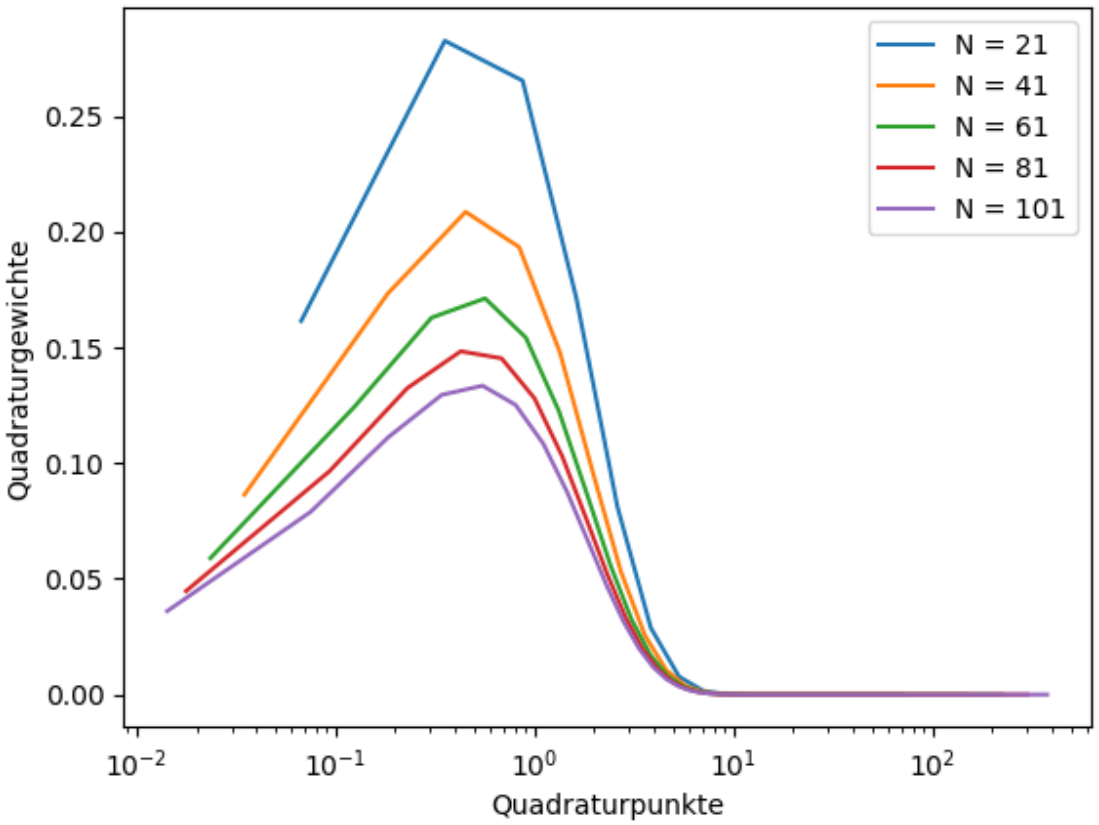
Damit erhalten wir: $w_{k,n} = \frac{1}{\|\nu_{k,n}\|^2}$

d)

Nachdem wir Satz 4.23 hinsichtlich der Gewichtsfunktion $w(x) = \exp(-x)$ gezeigt haben, können wir somit die Gewichte (α_j) der Gauß-Quadratur berechnen. Mithilfe der Funktion numpy.linalg.eig lassen sich die Eigenwerte und die dazugehörigen normierten Eigenvektoren der Matrix T_n berechnen. Da der erste Eintrag der normierten Eigenvektoren bereits $\frac{1}{\|\nu_{k,n}\|}$ beträgt, müssen wir ihn nur noch quadrieren um das gewünschte Gewicht zu erhalten.
Aus Satz 4.17 des Vorlesungsskripts wissen wir, dass die Quadraturpunkte durch die Nullstellen des orthogonalen Polynoms p_n gegeben sind. Mit dem Ergebnis der vorigen Aufgabe wissen wir, dass die Nullstellen von p_n genau die Eigenwerte der Matrix T_n sind. Damit haben wir alles, was wir für die Implementierung der Gaussquadratur benötigen.

```
def gauss(f,N):
    T = np.diag([2*i+1 for i in range(N)])
    T -= np.diag([i for i in range(1,N)],1)
    T -= np.diag([i for i in range(1,N)],-1)
    values, vectors = np.linalg.eig(T)
    alphas = [(vectors[0][i])**2 for i in range(N)]
    summe = sum([alphas[i]*f(values[i]) for i in range(N)])
    return summe, alphas, values
```

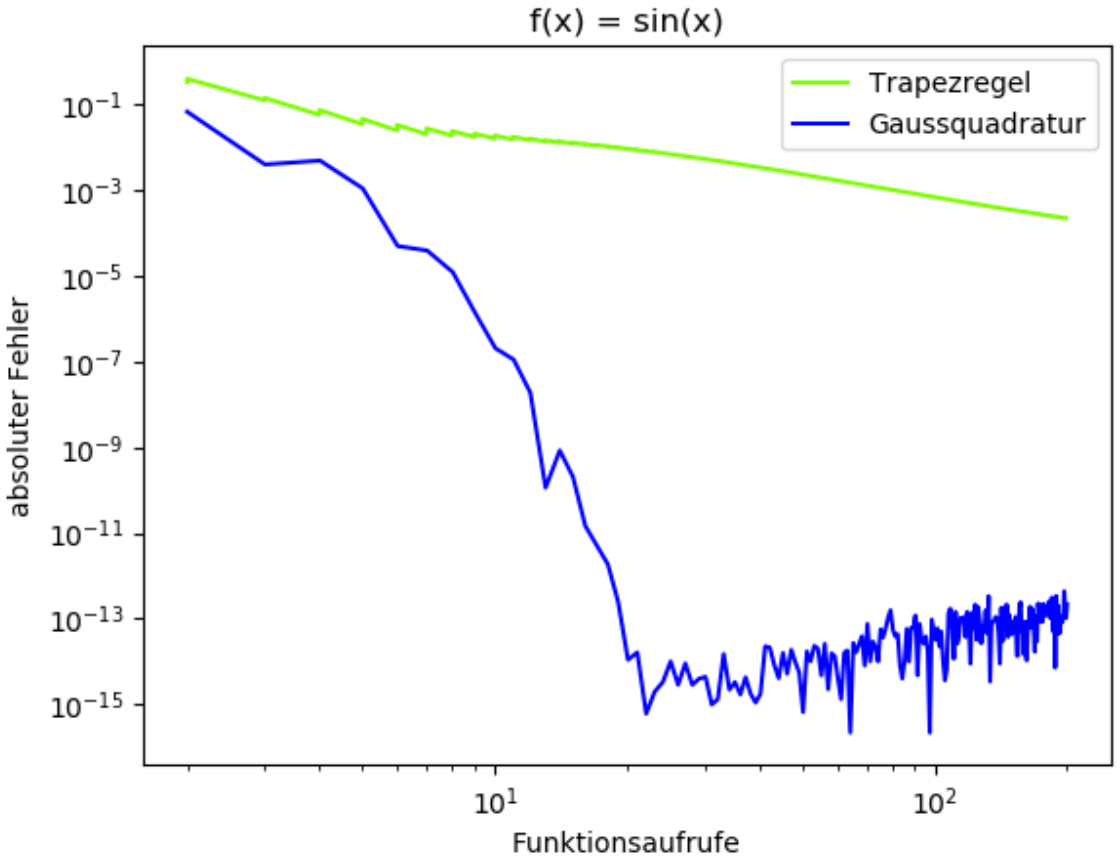

Gaussquadratur
sin(x)
cos(x)
sin(x)/x
tanh(x)
absoluter Fehler
10^{-1}
10^{-3}
10^{-5}
10^{-7}
10^{-9}
10^{-11}
10^{-13}
10^{-15}
10^{0}
10^{1}
10^{2}
Funktionsaufrufe

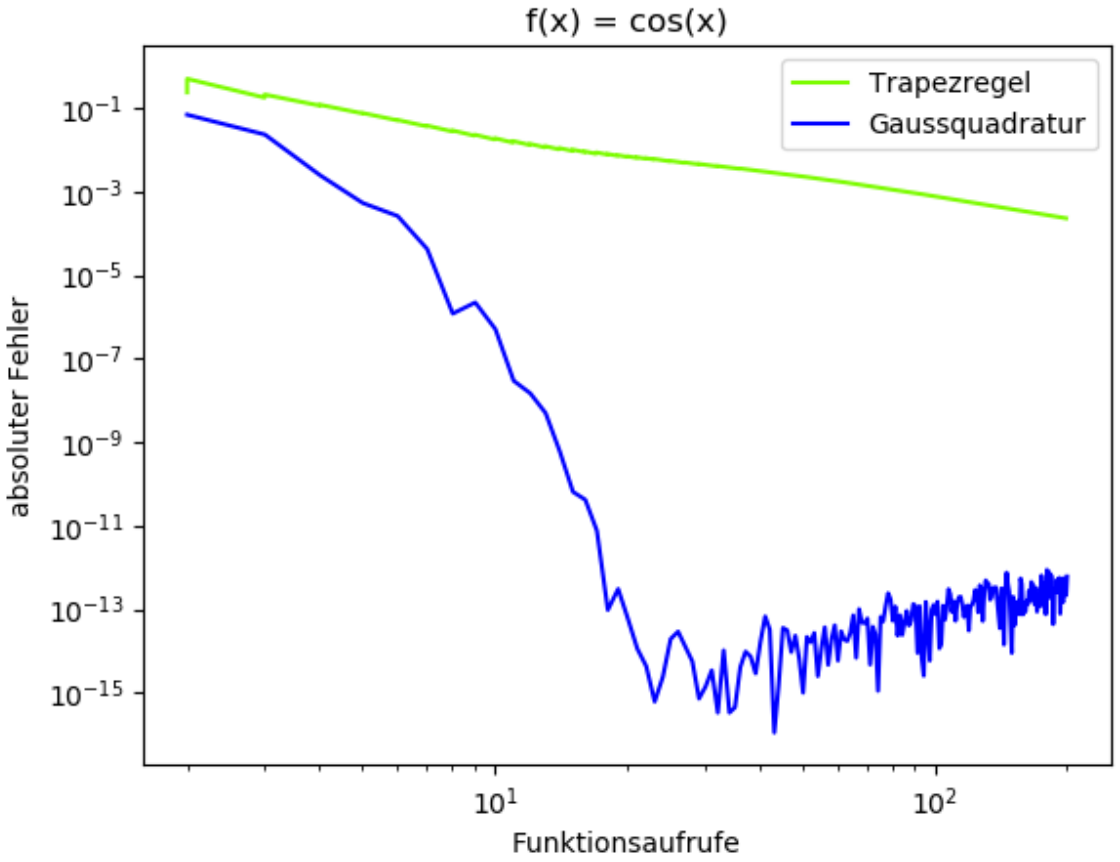


Im ersten Plot sehen wir den Fehler der Quadratur anhand ausgewählter trigonometrischer Funktionen. In 3 von 4 Fällen sind wir bereits nach 20 Funktionsaufrufe im Bereich der Maschinengenauigkeit, was den Fehler betrifft, lediglich beim Tangens Hyperbolicus schwächelt die Gaussquadratur ein wenig und bentötigt ungefähr 100 Funktionsaufrufe um mit dem Fehler in die Nähe der Maschinengenauigkeit zu kommen.
Im zweiten Plot werden die Stützstellen gegen ihre zugehörigen Gewichte geplottet. Man erkennt gut, dass Quadraturpunkte rund um 1 am höchsten gewichtet werden und für größere werdende Quadraturpunkte konvergieren die zugehörigen Gewichte schnell gegen Null. Wie man anschaulich erwarten würde, tragen Werte jenseits von 10 kaum mehr etwas zum Integral bei und werden dementsprechend verschwindend gering gewichtet. Auch interessant zu beobachten ist, dass die kleinste Nullstelle bei steigendem N immer näher an 0 herankommt.

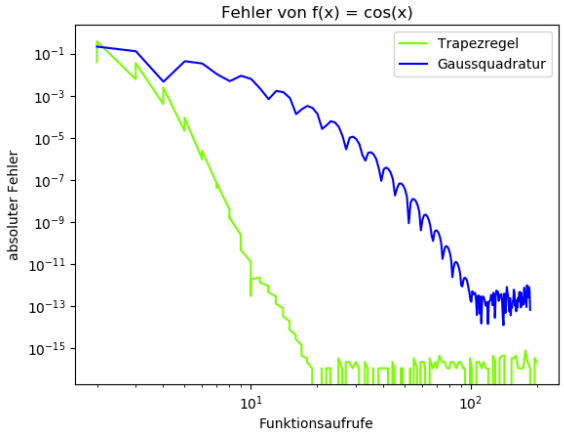
e)

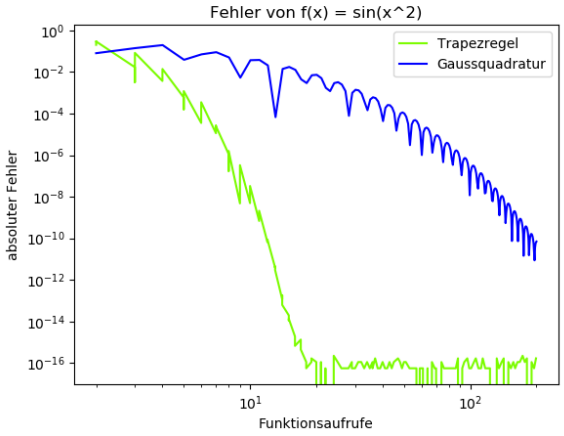
Schlußendlich stellt sich die Frage, welcher der beiden Ansätze nun der effizientere sei. Dazu betrachten wir die Kovergenzplots beider Quadraturen anhand geeigneter Beispiele:

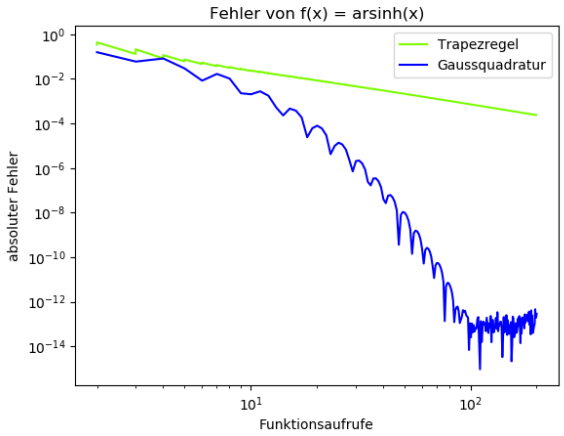

f(x) = sin(x)
Trapezregel
Gaussquadratur
absoluter Fehler
10^{-1}
10^{-3}
10^{-5}
10^{-7}
10^{-9}
10^{-11}
10^{-13}
10^{-15}
10^{1}
10^{2}
Funktionsaufrufe

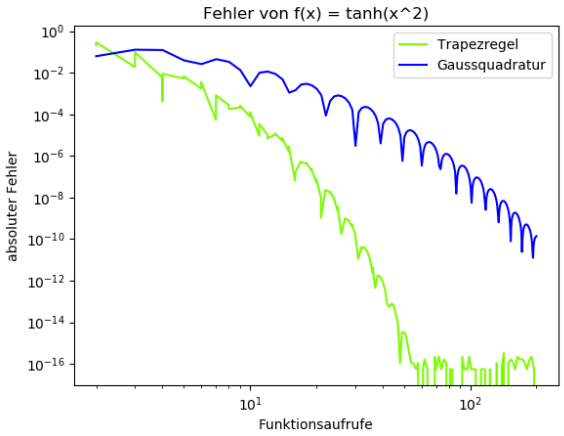


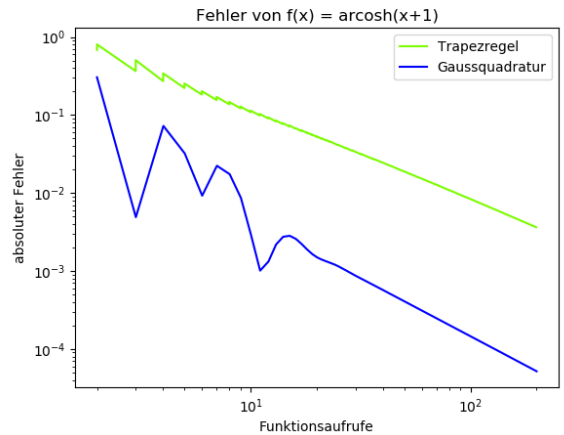
Hier erkennt man, dass die Gauß-Quadratur ein deutlich schnelleres Konvergenzverhalten hat, als die summierte Trapezregel. Nach bereits ca. 20 Funktionsaufrufen sind wir im Bereich der Maschinengenauigkeit angelangt, ab dem natürlich keine weitere Verbesserung mehr möglich ist.
Wenn man nun als Gewichtsfunktion $\exp(-x^2)$ wählt, liefert uns die Gaussquadratur bei den meisten Funktionen eine nur unwesentlich langsamere Konvergenz, allerdings mit Ausnahmen. Bei der Sinus- und Cosinus-Funktion schneidet die Trapezregel wesentlich besser als erwartet ab und erreicht bereits nach ca. 20 Funktionsauswertungen einen Fehler in der Größenordnung der Maschinengenauigkeit. Beim Areasinh sehen wir allerdings wieder das zu erwartende Konvergenzverhalten der Trapezregel, klar geschlagen von der Gaussquadratur. Bei $f : x \mapsto \tanh(x^2)$ gewinnt allerdings wieder die Trapezregel. Noch interessanter wird die Geschichte bei $f : x \mapsto \operatorname{arcosh}(x + 1)$: Hier konvergieren beide Quadraturformeln mit der selben Geschwindigkeit, die Gaussquadratur ist lediglich um einen Faktor von ca. 100 besser. Bei der Funktion $f : x \mapsto \sin(\frac{1}{x})$ an der Stelle 0 gleich 0 gesetzt scheitern jedoch beide Formeln kläglich. Die letzten beiden Resultate lassen sich allerdings erklären, wenn man die Ableitung der Funktion in einer Umgebung von 0 betrachtet und feststellt, dass sie dort unbeschränkt ist.

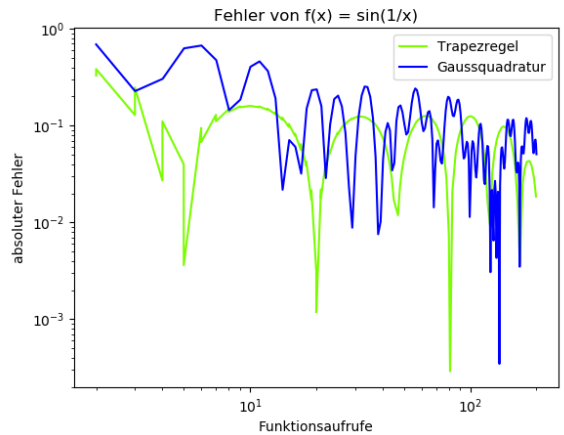












2 Summierte Quadraturformeln und Extrapolation

Sei $f \in C([a,b])$ und $Q(f) := \int_a^b f(x)dx$. Zur numerischen Approximation von $Q(f)$ seien $x_0, \ldots, x_n$ äquidistant verteilte Quadraturknoten in $[a,b]$ mit $a = x_0 < x_1 < \cdots < x_n = b$.

Es werden im weiteren Verlauf der Ausarbeitung, verschiedene summierte Quadraturformeln und deren Zusammenhänge untersucht.

2.1 Die summierte Trapez-Regel

Die Menge aller zulässigen Abstände zwischen beliebig äquidistanten Quadraturknoten, bezeichnen wir mit $H := \left\{ \frac{b-a}{n} : n \in \mathbb{N} \right\}$. Somit, können wir, für fixes $h \in H$, diese $x_0, \ldots, x_n$ durch $x_i := a + hi$, $i = 1, \ldots, n$, charakterisieren. Man verifiziert unmittelbar, dass

$$x_0 = a,\, x_n = b, \quad \forall\, i = 1, \ldots, n : x_i - x_{i-1} = h.$$

Wir beginnen damit, die ersten paar abgeschlossenen Newton-Cotes-Formeln auszurechnen. Diese ergeben sich bekanntlich aus dem lagrange'schen Interpolationspolynom $p_f \in \Pi_n$ von f.

$$Q^{(n)}(f) = \int_a^b p_f(x)dx = \int_a^b \sum_{i=0}^{n} f(x_i)L_i(x)dx = \sum_{i=0}^{n} f(x_i) \int_a^b L_i(x)dx$$

$$L_i(x) := \prod_{\substack{j=0 \\ j \neq i}}^{n} \frac{x - x_j}{x_i - x_j}, \quad i = 0, \ldots, n$$

Nachdem die Lagrange Basis Polynome L_j, $j = 0, \ldots, n$, umständlich zu integrieren sind, wird hier nicht per Hand, sondern mit dem Python-Paket SymPy gearbeitet.

```
# returns i-th lagrange basis polynomial at t via interpolation points x
def L(i, t, x):
    n = len(x) + 1
    dummy = 1

    for j in range(n+1):
        if j != i:
            dummy *= (t - x[j])/(x[i] - x[j])

    return dummy

# calculates closed newton-cotes formula of order n on interval [a, b]
def Q(n):
    a, b = sp.symbols('a b')
    t = sp.Symbol('t')

    h = (b - a)/n
    x = [a + h*i for i in range(n+1)]

    # y_i = f(x_i)
    y = sp.IndexedBase('y')
    y = [y[i] for i in range(n+1)]

    dummy = 0

    for i in range(n+1):
```

```
        weight = sp.integrate(L(i, t, x), (t, a, b))
        weight = sp.factor(weight)

        dummy += weight * y[i]

    return sp.factor(dummy)
```

Damit errechnet man jeweils die Trapez-, Simpson- und Mile-Regel.

$$
\begin{aligned}
Q^{(1)}(f) &= \frac{b-a}{2}\left(f(a)+f(b)\right) \\
Q^{(2)}(f) &= \frac{b-a}{6}\left(f(a)+4f\left(\frac{a+b}{2}\right)+f(b)\right) \\
Q^{(4)}(f) &= \frac{b-a}{90}\left(7f(a)+32f\left(\frac{3a+b}{4}\right)+12f\left(\frac{a+b}{2}\right)+32f\left(\frac{a+3b}{4}\right)+7f(b)\right)
\end{aligned}
$$

Jetzt können auch die jeweils summierten Quadraturformeln bestimmt werden. Wir setzen uns vorerst nur mit der Summierten Trapez-Regel $Q_{\cdot}^{(1)}(f) : H \to \mathbb{K}$ auseinander.

$$
\begin{aligned}
Q_h^{(1)}(f) &= \sum_{i=1}^{n} Q^{(1)}(f|_{[x_{i-1},x_i]}) \\
&= \sum_{i=1}^{n} \frac{x_i - x_{i-1}}{2}\left(f(x_{i-1})+f(x_i)\right) \\
&= \frac{h}{2}\left(\sum_{i=1}^{n} f(x_{i-1}) + \sum_{i=1}^{n} f(x_i)\right) \\
&= \frac{h}{2}\left(f(x_0) + \sum_{i=2}^{n} f(x_{i-1}) + \sum_{i=1}^{n-1} f(x_i) + f(x_n)\right) \\
&= \frac{h}{2}\left(f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right)
\end{aligned}
$$

Wir wollen nun den Quadraturfehler der summierten Trapez-Regel genauer zu untersuchen.

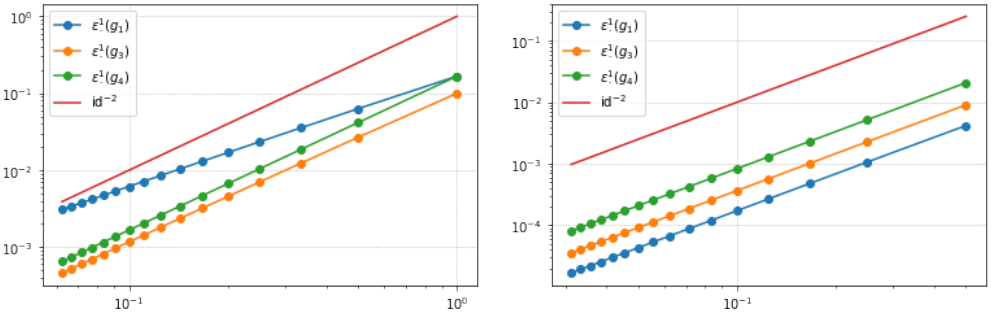
$$
\epsilon_h^{(1)} := \left| Q - Q_h^{(1)} \right|
$$

Dazu definieren wir uns eine Folge von Testfunktionen $g_k(x) := \sqrt{x}^k$, $k \in \mathbb{N}$. Deren Stammfunktionen sind offensichtlich $(\int g_k)(x) = \frac{2}{k+2}\sqrt{x}^{k+2}$. Die summierte Trapezregel $Q_{\cdot}^{(1)}$ zu implementieren, ist ebenso schnell geschafft. Damit können wir auch schon den Quadraturfehler $\epsilon_{\cdot}^{(1)}(g_k)$, explizit berechnen lassen.

```
# integrates f on [a, b] via trapezium rule with equidistant nodes of distance h
def Q_1(h, f, a, b):
    n = (b-a)/h
    n = int(n)
    x = np.linspace(a, b, n+1)
```

```
    S_11 = f(a)
    S_12 = np.sum(f(x[1:n]))
    S_13 = f(b)

    return h/2 * (S_11 + 2*S_12 + S_13)
```

Dieser wurde in Abbildung 4 auf $\left\{\frac{b-a}{n} : n = 1, \ldots, 16\right\} \subset H$, für $k = 1, 3, 4$, auf beiden Achsen logarithmisch, geplottet. Dabei werden g_0 und g_2 sind nicht abgebildet, weil sie, als Lineare Funktionen, offensichtlich sofort, durch $Q^{(1)}$ bzw. $Q_h^{(1)}$, exakt berechnet werden. Das macht sie für die folgende Analyse uninteressant. Auch g_k, $k \geq 5$, werden ausgelassen, da sie, bezüglich Konvergenzverhalten, keine markanten Unterschiede zu g_4 aufweisen. id^{-2} ist dabei eine Referenzfunktion für das Konvergenzverhalten.



(a) $a = 0$, $b = 1$ (b) $a = \frac{1}{2}$, $b = 1$

Abbildung 4: Konvergenzverhalten von $\epsilon^{(1)}_{\cdot}(g_k)$, $k = 1, 3, 4$ auf $[a, b]$

Es scheinen also g_3 und g_4, quadratisch zu konvergieren, d.h. $\epsilon_h^{(1)}(g_k) = \mathcal{O}\left(h^2\right)$, $h \to \infty$, $k = 3, 4$. Abbildung 4 ist auch zu entnehmen, dass $\epsilon_h^{(1)}(g_1)$ langsamer konvergiert. Es stellt sich heraus, dass dieses Phänomen auch bereits bei $a \approx 0$ auftritt.

Um das zu erklären, betrachten wir g_1 im Intervall $[0, 1]$. Der Quadraturfehler $\epsilon_h^{(1)}(g_1)$ dieser Funktion wird dazu auf n Intervalle der Länge h aufgeteilt. Man beachte, dass darauf die normale Trapez-Regel $Q^{(1)}$ gilt.

$$[0, 1] = \bigcup_{i=0}^{n-1} [hi, h(i+1)] \tag{1}$$

Der erste Fehler $\epsilon_h^{(1)}(g_1|_{[0,h]})$ muss direkt über die Stammfunktion von g_1 bestimmt werden.

$$\begin{aligned}\epsilon_h^{(1)}(g_k|_{[0,h]}) &= \left|Q(g_1|_{[0,h]}) - Q^{(1)}(g_1|_{[0,h]})\right| \\ &= \frac{2}{1+2}\sqrt{x}^{1+2}\Big|_0^h - \frac{h-0}{2}\left(\sqrt{0}^1 + \sqrt{h}^1\right) = \frac{2}{3}\sqrt{h}^3 - \frac{1}{2}\sqrt{h}^3 = \frac{1}{6}\sqrt{h}^3\end{aligned}$$

Für die anderen Fehler $\epsilon_h^{(1)}(g_1|_{[hi,h(i+1)]})$, braucht man die zweiten Ableitungen g_1''. Dabei ist $i = 1, \ldots, n-1$.

$$\frac{d^2}{dx^2} g_1(x) = \frac{d^2}{d^2x} x^{\frac{1}{2}} = \frac{1}{2}\frac{d}{dx} x^{-\frac{1}{2}} = -\frac{1}{4} x^{-\frac{3}{2}} = -\frac{1}{4\sqrt{x}^3}$$

Nachdem $|g_1''|$ monoton fällt, muss $\left\|g_1''|_{[hi,h(i+1)]}\right\|_\infty = |g_1''(hi)| = \frac{1}{4\sqrt{hi}^3}$. Somit erhalten wir eine Fehlerabschätzung mit $\exists\, \xi \in (hi, h(i+1))$:

$$\begin{aligned}\epsilon_h^{(1)}(g_k|_{[hi,h(i+1)]}) &= \left|Q(g_1|_{[hi,h(i+1)]}) - Q^{(1)}(g_1|_{[hi,h(i+1)]})\right| \\ &= \left|-\frac{(h(i+1)-hi)^3}{12} g_1''|_{[hi,h(i+1)]}(\xi)\right| \leq \frac{h^3}{12}\left\|g_1''|_{[hi,h(i+1)]}\right\|_\infty = \frac{h^3}{12}\frac{1}{4\sqrt{hi}^3} = \frac{\sqrt{h}^3}{48\sqrt{i}^3}\end{aligned}$$

Wir setzen die einzelnen zum Fehler, laut (1), auf das gesamte Intervall $[0,1]$ zusammen.

$$\begin{aligned}\epsilon_h^{(1)}(g_1) = \left|Q(g_1) - Q_h^{(1)}(g_1)\right| &= \left|\sum_{i=0}^{n-1} Q(g_1|_{[hi,h(i+1)]}) - \sum_{i=0}^{n-1} Q^{(1)}(g_1|_{[hi,h(i+1)]})\right| \\ &\leq \sum_{i=0}^{n-1}\left|Q(g_1|_{[hi,h(i+1)]}) - Q^{(1)}(g_1|_{[hi,h(i+1)]})\right| = \epsilon_h^{(1)}(g_1|_{[0,h]}) + \sum_{i=1}^{n-1}\epsilon_h^{(1)}(g_1|_{[hi,h(i+1)]}) \\ &\leq \frac{1}{6}\sqrt{h}^3 + \sum_{i=1}^{n-1}\frac{\sqrt{h}^3}{48\sqrt{i}^3} = \underbrace{\frac{1}{6}\left(1 + \frac{1}{8}\sum_{i=1}^{n-1}\frac{1}{i^{\frac{3}{2}}}\right)}_{=:C}\sqrt{h}^3\end{aligned}$$

Weil $C < \infty$, und die Abschätzung für kleines h scharf ist bekommt man also tatsächlich keine quadratische Konvergenz.

$$\epsilon_h^{(1)}(g_1) \approx \mathcal{O}\left(\sqrt{h}^3\right), \quad h \to 0$$

2.2 Die summierte Simpson- und Mile-Regel

Es wird, ohne Beweis, vorausgesetzt, dass die summierte Trapez-Regel $Q_\cdot^{(1)}(f)$, eine asymptotische Entwicklung besitzt.

$$Q_h^{(1)}(f) = Q(f) + \sum_{i=1}^{r} a_i h^{2i} + a_{r+1}(h) \quad \text{mit } a_{r+1}(h) = \mathcal{O}\left(h^{2r+2}\right) \text{ für } h \to 0. \tag{2}$$

Wir wollen nämlich eine lineare Richardson-Extrapolation, mit der Romberg-Folge $(h_k)_{k\in\mathbb{N}_0}$, auf die summierte Trapez-Regel $Q_\cdot^{(1)}(f)$ anwenden.

$$h_0 \in H, \quad h_k := \frac{h_0}{2^k}, \quad n_k := \frac{b-a}{h_k}, \quad k \in \mathbb{N}_0$$

Diese Art von Folge, wo das vorherige Glied ganzzahlig geteilt wird, wird sich als überaus nützlich erweisen. Das lässt nämlich eine Wiederverwertung von Funktionsauswertungen an den vorherigen Stützstellen zu.

Man verifiziert unmittelbar die nützlichen Eigenschaften $\forall\, k, \ell \in \mathbb{N}_0$:

$$\frac{h_{k+\ell}}{h_k} = \frac{1}{2^\ell}, \quad n_k = 2^k n_0.$$

Für $i = 0, \ldots, n_0$ und $j = 0, \ldots, n_1$, brauchen $Q^{(1)}_{h_0}$ und $Q^{(1)}_{h_1}$ noch die Quadraturknoten $x_i := a + h_0 i$ und $y_i := a + h_1 j$. Durch Einsetzen und elementares Nachrechnen, ergeben sich die Eigenschaften $\forall\, i = 1, \ldots, n_0$:

$$y_{2i-1} = \frac{x_{i-1} + x_i}{2}, \quad y_{2i} = x_i.$$

Es wird, für die Extrapolation, ein Ausschnitt des Neville-Aitken Schemas benutzt.

$$\begin{array}{lcl} Q^{(1)}_{h_0}(f) = a_{0,0} & & \\ & \searrow & \\ Q^{(1)}_{h_1}(f) = a_{1,0} & \to & a_{0,1} = a_{0,0} + \frac{a_{0,0} - a_{1,0}}{\left(\frac{h_1}{h_0}\right)^2 - 1} \end{array}$$

Man erkennt, dass eine lineare Extrapolation der summierten Trapez-Regel $Q^{(1)}_{\cdot}(f)$, genau in der summierten Simpson-Regel $Q^{(2)}_{\cdot}(f)$ resultiert.

$$\begin{aligned} a_{0,1} &= Q^{(1)}_{h_0}(f) + \frac{Q^{(1)}_{h_0}(f) - Q^{(1)}_{h_1}(f)}{\left(\frac{1}{2}\right)^2 - 1} \\ &= \frac{1}{3}\left(-Q^{(1)}_{h_0}(f) + 4Q^{(1)}_{h_1}(f)\right) \\ &= \frac{1}{3}\left(-\frac{h_0}{2}\left(f(a) + 2\sum_{i=1}^{n_0-1} f(x_i) + f(b)\right) + 4\frac{h_1}{2}\left(f(a) + 2\sum_{i=1}^{n_1-1} f(y_i) + f(b)\right)\right) \\ &= \frac{1}{3}\left(-\frac{h_0}{2}\left(f(a) + 2\sum_{i=1}^{n_0-1} f(x_i) + f(b)\right) + \frac{2h_0}{2}\left(f(a) + 2\sum_{i=1}^{2n_0-1} f(y_i) + f(b)\right)\right) \\ &= \frac{h_0}{3}\left(-\frac{1}{2}\left(f(a) + 2\sum_{i=1}^{n_0-1} f(x_i) + f(b)\right) + \left(f(a) + 2\left(\sum_{i=1}^{n_0} f(y_{2i-1}) + \sum_{i=1}^{n_0-1} f(y_{2i})\right) + f(b)\right)\right) \\ &= \frac{h_0}{3}\left(-\frac{1}{2}\left(f(a) + 2\sum_{i=1}^{n_0-1} f(x_i) + f(b)\right) + \left(f(a) + 2\left(\sum_{i=1}^{n_0} f\left(\frac{x_{i-1} - x_i}{2}\right) + \sum_{i=1}^{n_0-1} f(x_i)\right) + f(b)\right)\right) \\ &= \frac{h_0}{6}\left(f(a) + 4\sum_{i=1}^{n_0} f\left(\frac{x_{i-1} - x_i}{2}\right) + 2\sum_{i=1}^{n_0-1} f(x_i) + f(b)\right) \\ &= Q^{(2)}_{h_0}(f) \end{aligned}$$

Man könnte erwarten, dass eine quadratische Extrapolation zur summierten 3/8-Regel $Q^{(3)}_{\cdot}(f)$ führt. Aufgrund der Wahl der Folge $(h_k)_{k \in \mathbb{N}_0}$, ist die summierte Mile-Regel, $Q^{(4)}_{\cdot}(f)$, aber ein plausiblerer Kandidat. Um diese Vermutungen zu überprüfen, befleißigen wir uns abermals am Neville-Aitken Schema.

$$\begin{array}{lcllcl} Q^{(1)}_{h_0}(f) = a_{0,0} & & & & & \\ & \searrow & & & & \\ Q^{(1)}_{h_1}(f) = a_{1,0} & \to & a_{0,1} = a_{0,0} + \frac{a_{0,0} - a_{1,0}}{\left(\frac{h_1}{h_0}\right)^2 - 1} & & & \\ & \searrow & & \searrow & & \\ Q^{(1)}_{h_2}(f) = a_{2,0} & \to & a_{1,1} = a_{1,0} + \frac{a_{1,0} - a_{2,0}}{\left(\frac{h_2}{h_1}\right)^2 - 1} & \to & a_{0,2} = a_{0,1} + \frac{a_{0,1} - a_{1,1}}{\left(\frac{h_2}{h_0}\right)^2 - 1} \end{array}$$

Es ist ja, durch die lineare Extrapolation, bereits bekannt, dass $a_{0,1} = Q^{(2)}_{h_0}(f)$. Man kann aber sogar, durch die selbe Rechnung, nur mit geshifteten $h_0 \mapsto h_1 \mapsto h_2$, auch auf $a_{1,1} = Q^{(2)}_{h_1}(f)$ kommen.

Durch nochmaliges Einsetzen und elementares Nachrechnen, ergeben sich wiederum $\forall\, i = 1, \ldots, n_0$:

$$\frac{y_{2i-2} + y_{2i-1}}{2} = \frac{3x_{i-1} + x_i}{4}, \quad \frac{y_{2i-1} + y_{2i}}{2} = \frac{x_{i-1} + 3x_i}{4}.$$

Die Annahme, dass eine quadratische Extrapolation der summierten Trapez-Regel $Q^{(1)}_{\cdot}(f)$ die summierte Mile-Regel $Q^{(4)}_{\cdot}(f)$ liefert, wird hiermit bestätigt.

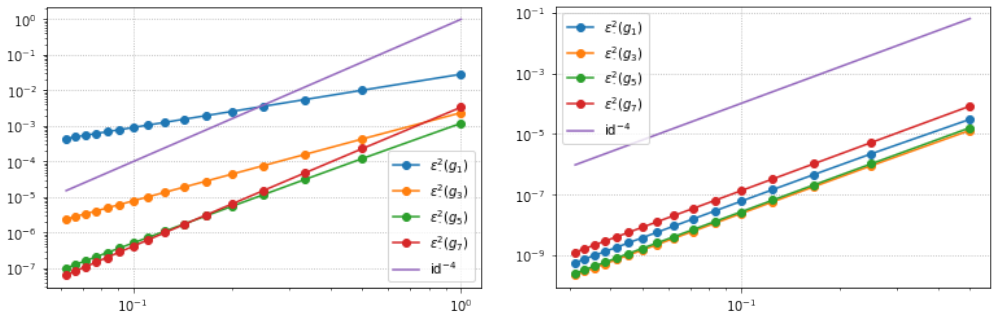
$a_{0,2}$

$$
\begin{aligned}
&= Q^{(2)}_{h_0}(f) + \frac{Q^{(2)}_{h_0}(f) - Q^{(2)}_{h_1}(f)}{\left(\frac{1}{2^2}\right)^2 - 1} \\
&= \frac{1}{15}\left(-Q^{(2)}_{h_0}(f) + 16 Q^{(2)}_{h_1}(f)\right) \\
&= \frac{1}{15}\Bigg(- \frac{h_0}{6}\left(f(a) + 2\sum_{i=1}^{n_0-1} f(x_i) + 4\sum_{i=1}^{n_0} f\left(\frac{x_{i-1}+x_i}{2}\right) + f(b)\right) \\
&+ 16\frac{h_1}{6}\left(f(a) + 2\sum_{i=1}^{n_1-1} f(y_i) + 4\sum_{i=1}^{n_1} f\left(\frac{y_{i-1}+y_i}{2}\right) + f(b)\right)\Bigg) \\
&= \frac{1}{15}\Bigg(- \frac{h_0}{6}\left(f(a) + 2\sum_{i=1}^{n_0-1} f(x_i) + 4\sum_{i=1}^{n_0} f\left(\frac{x_{i-1}+x_i}{2}\right) + f(b)\right) \\
&+ \frac{8h_0}{6}\left(f(a) + 2\sum_{i=1}^{2n_0-1} f(y_i) + 4\sum_{i=1}^{2n_0} f\left(\frac{y_{i-1}+y_i}{2}\right) + f(b)\right)\Bigg) \\
&= \frac{h_0}{90}\Bigg(- \left(f(a) + 2\sum_{i=1}^{n_0-1} f(x_i) + 4\sum_{i=1}^{n_0} f\left(\frac{x_{i-1}+x_i}{2}\right) + f(b)\right) \\
&+ 8\left(f(a) + 2\left(\sum_{i=1}^{n_0} f(y_{2i-1}) + \sum_{i=1}^{n_0-1} f(y_{2i})\right) + 4\left(\sum_{i=1}^{n_0} f\left(\frac{y_{2i-2}+y_{2i-1}}{2}\right) + \sum_{i=1}^{n_0} f\left(\frac{y_{2i-1}+y_{2i}}{2}\right)\right) + f(b)\right)\Bigg) \\
&= \frac{h_0}{90}\Bigg(- \left(f(a) + 2\sum_{i=1}^{n_0-1} f(x_i) + 4\sum_{i=1}^{n_0} f\left(\frac{x_{i-1}+x_i}{2}\right) + f(b)\right) \\
&+ 8\left(f(a) + 2\left(\sum_{i=1}^{n_0} f\left(\frac{x_{i-1}-x_i}{2}\right) + \sum_{i=1}^{n_0-1} f(x_i)\right) + 4\left(\sum_{i=1}^{n_0} f\left(\frac{3x_{i-1}+x_i}{4}\right) + \sum_{i=1}^{n_0} f\left(\frac{x_{i-1}+3x_i}{4}\right)\right) + f(b)\right)\Bigg) \\
&= \frac{h_0}{90}\Bigg(- \left(f(a) + 2\sum_{i=1}^{n_0-1} f(x_i) + 4\sum_{i=1}^{n_0} f\left(\frac{x_{i-1}+x_i}{2}\right) + f(b)\right) \\
&+ 8\left(f(a) + 2\left(\sum_{i=1}^{n_0} f\left(\frac{x_{i-1}-x_i}{2}\right) + \sum_{i=1}^{n_0-1} f(x_i)\right) + 4\left(\sum_{i=1}^{n_0} f\left(\frac{3x_{i-1}+x_i}{4}\right) + \sum_{i=1}^{n_0} f\left(\frac{x_{i-1}+3x_i}{4}\right)\right) + f(b)\right)\Bigg) \\
&= \frac{h_0}{90}\left(7f(a) + 32\sum_{i=1}^{n_0} f\left(\frac{3x_{i-1}+x_i}{4}\right) + 12\sum_{i=1}^{n_0} f\left(\frac{x_{i-1}+x_i}{2}\right) + 14\sum_{i=1}^{n_0-1} f(x_i) + 32\sum_{i=1}^{n_0} f\left(\frac{x_{i-1}+3x_i}{4}\right) + 7f(b)\right) \\
&= Q^{(4)}_{h_0}(f)
\end{aligned}
$$

Uns interessiert natürlich wieder das Konvergenzverhalten der eben extrapolierten Regeln. Dazu implementieren wir diesmal die summierte Simpson-Regel $Q_\cdot^{(2)}$ und Mile-Regel $Q_\cdot^{(4)}$.

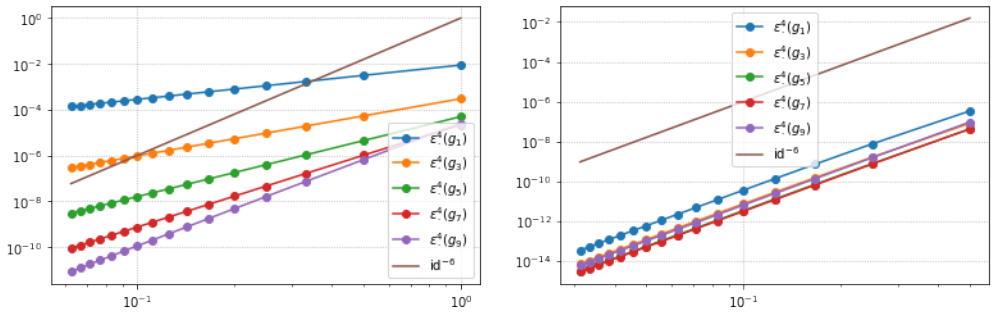
```
# integrates f on [a, b] via simpson's rule with equidistant nodes of distance h
def Q_2(h, f, a, b):
    n = (b-a)/h
    n = int(n)
    x = np.linspace(a, b, n+1)

    S_11 = f(a)
    S_12 = np.sum(f(x[1:n]))
    S_13 = f(b)
    S_2  = np.sum(f((x[0:n] + x[1:n+1])/2))

    return h/6 * (S_11 + 4*S_2 + 2*S_12 + S_13)

# integrates f on [a, b] via mile's rule with equidistant nodes of distance h
def Q_4(h, f, a, b):
    n = (b-a)/h
    n = int(n)
    x = np.linspace(a, b, n+1)

    S_11 = f(a)
    S_12 = np.sum(f(x[1:n]))
    S_13 = f(b)
    S_2  = np.sum(f((x[0:n] + x[1:n+1])/2))
    S_41 = np.sum(f((3*x[0:n] + x[1:n+1])/4))
    S_42 = np.sum(f((x[0:n] + 3*x[1:n+1])/4))

    return h/90 * (7*S_11 + 32*S_41 + 12*S_2 + 14*S_12 + 32*S_42 + 7*S_13)
```

Um die Quadraturfehler der summierten Simpson-Regel $\epsilon_h^{(2)}$ und Mile-Regel $\epsilon_h^{(4)}$ zu untersuchen, wurden wieder Konvergenzplots, analog zu Abbildung 4 erstellt.



(a) $a = 0$, $b = 1$ (b) $a = \frac{1}{2}$, $b = 1$

Abbildung 5: Konvergenzverhalten von $\epsilon_\cdot^{(2)}(g_k)$, $k = 1, 3, 5, 7$ auf $[a, b]$



(a) $a = 0$, $b = 1$ (b) $a = \frac{1}{2}$, $b = 1$

Abbildung 6: Konvergenzverhalten von $\epsilon_{\cdot}^{(4)}(g_k)$, $k = 1, 3, 5, 7, 9$ auf $[a, b]$

Es stellt sich wieder die berechtigte Frage, wie es zu diesen Ergebnissen kommen kann, und ob es einen Zusammenhang zum Quadraturfehler der summierten Trapezregel $\epsilon_h^{(1)}$ geben könnte. Die Antwort wird im nächsten Abschnitt gleich ganz allgemein beantwortet.

Um den unpassenden Quadraturfehler der summierten Simpson-Regel $\epsilon_{\cdot}^{(2)}(g_k)$, $k = 1, 3, 5$, zu erklären, kann man analog zum vorherigen Abschnitt vorgehen. Dazu eignet sich wieder die Partition (1) und, etwas andere, Restglieddarstellung der Simpson-Regel auf den einzelnen Teilintervallen.

$$\forall\, i = 1, \ldots, n-1 : \exists\, \xi \in (hi, h(i+1)) : Q(f) - Q^{(2)}(f) = -\frac{(h(i+1) - hi)^5}{2880} f^{(4)}(\xi) = -\frac{h^5}{2880} f^{(4)}(\xi)$$

2.3 Höhere Quadraturformeln

Zuletzt, werden wir noch eine allgemeine Richardson-Extrapolation der summierten Trapez-Regel $Q_{\cdot}^{(1)}$ formulieren und implementerien.

$$\begin{array}{lccccc} Q_{h_0}^{(1)}(f) = a_{0,0} & & & & & \\ & \searrow & & & & \\ Q_{h_1}^{(1)}(f) = a_{1,0} & \to & a_{0,1} & & & \\ & \searrow & & \searrow & & \\ Q_{h_2}^{(1)}(f) = a_{2,0} & \to & a_{1,1} & \to & a_{0,2} & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{array} \tag{3}$$

$$a_{j,0} = Q_{h_j}^{(1)}(f), \qquad j \in \mathbb{N}_0 \tag{4}$$

$$a_{j,i} = a_{j,i-1} + \frac{a_{i,i-1} - a_{j+1,i-1}}{\left(\frac{h_{j+i}}{h_j}\right)^2 - 1}, \qquad i \in \mathbb{N}, \quad j \in \mathbb{N}_0 \tag{5}$$

Die wohl einfachste bzw. naheliegendste Implementierung der Richardson Extrapolation ist wahrscheinlich rekursiv. Um Rekursionszweige zu sparen, kann man Hilfsvariablen einführen, anstatt die Rekursionsformel in einem Aufwaschen auswerten zu lassen.

```python
# inefficient implementation (with recursion)

def richardson_extrapolation_1(h_0, f, a, b, r):
    return recursion(h_0, f, a, b, 0, r)

def recursion(h_0, f, a, b, j, i):
    h_j = h_0/2**j

    if i == 0:
        return Q_1(h_j, f, a, b)

    else:
        tmp_1 = recursion(h_0, f, a, b, j,   i-1)
        tmp_2 = recursion(h_0, f, a, b, j+1, i-1)

        return tmp_1 + (tmp_1 - tmp_2)/(1/2**(i*2) - 1)
```

Trotz Hilfsvariablen, ist die obere eine miserable Implementierung. Sie ist deswegen so schlecht, weil die Einträge der rechten Spalte, also die Rekursionsbasis (4), vom Neville-Aitken Schema (3), jedes Mal separat berechnet werden.

Weitaus sparsamer, was Funktionsauswertungen angeht, kann man mit Loops davonkommen. Dazu wird bei $a_{0,0} = Q^{(1)}_{h_0}(f)$, oben rechts in (3), gestartet und, mit dem gespeicherten Ergebnis, der darunterliegenden Eintrag $a_{1,0} = Q^{(1)}_{h_1}(f)$ berechnet.

$$
\begin{aligned}
Q^{(1)}_{h_1}(f) &= \frac{h_1}{2}\left(f(a) + 2\sum_{i=1}^{n_1-1} f(y_i) + f(b)\right) \\
&= \frac{h_0}{4}\left(f(a) + 2\sum_{i=1}^{2n_0-1} f(y_i) + f(b)\right) \\
&= \frac{h_0}{4}\left(f(a) + 2\left(\sum_{i=1}^{n_0-1} f(y_{2i}) + \sum_{i=1}^{n_0} f(y_{2i-1})\right) f(b)\right) \\
&= \frac{h_0}{4}\left(f(a) + 2\left(\sum_{i=1}^{n_0-1} f(x_i) + \sum_{i=1}^{n_0} f\left(\frac{x_{i-1} - x_i}{2}\right)\right) + f(b)\right) \\
&= \frac{1}{2}\left(\frac{h_0}{2}\left(f(a) + 2\sum_{i=1}^{n_0-1} f(x_i) + f(b)\right) + h_0\sum_{i=1}^{n_0} f\left(\frac{x_{i-1} - x_i}{2}\right)\right) \\
&= \frac{1}{2}\left(Q^{(1)}_{h_0}(f) + h_0\sum_{i=1}^{n_0} f\left(\frac{x_{i-1} - x_i}{2}\right)\right)
\end{aligned}
$$

Das wird so lange fortgesetzt, bis hinreichend viele Einträge berechnet wurden. Den Rest darf dann aber der Rekursionsschritt (5) erledigen.

```python
# efficient implementation (with loops)

def richardson_extrapolation_2(h_0, f, a, b, r):
    A = np.zeros((r+1, r+1))
```

```
    A[0,0] = Q_1(h_0, f, a, b)

    for j in range(r):
        h_j = h_0/2**j
        n_j = int((b-a)/h_j)
        x = np.linspace(a, b, n_j+1)

        A[j+1, 0] = 1/2 * (A[j, 0] + h_j * np.sum(f((x[0:n_j] + x[1:n_j+1])/2)))

    for i in range(1, r+1):
        for j in range(r+1-i):
            A[j, i] = A[j, i-1] + (A[j, i-1] - A[j+1, i-1])/(1/2**(i*2) - 1)

    return A[0, -1]
```

Ein drittes und letztes Mal, wollen wir über das Konvergenzverhalten bescheid wissen. Dazu verifizieren wir die Voraussetzungen des Satzes zum Extrapolationsfehler.

Die asymptotische Entwicklung (2) der summierten Trapez-Regel $Q_{\cdot}^{(1)}(f) : H \to \mathbb{K}$ lässt sich in folgende Form umschreiben.

$$Q_h^{(1)}(f) = Q(f) + \sum_{i=1}^{r} a_i h^{iq} + a_{r+1}(h) h^{(r+1)q}, \quad h > 0, \quad a_{r+1}(h) = a_{r+1} + \mathcal{O}(1), \quad h \to 0$$

Für unsere Folge $(h_k)_{k\in\mathbb{N}_0}$ aus $\mathbb{R}_{>0}^{\mathbb{N}_0}$ gilt mit $\rho \in [\frac{1}{2}, 1)$ auch $0 < \frac{h_{k+1}}{h_k} \leq \rho < 1$. Sei dann noch $p_r^{(k)} \in \Pi_r$ das interpolierende Polynom mit

$$p_r^{(k)}\left(h_{k+j}^q\right) = Q_{h_{k+j}}^{(1)}(f), \quad j = 0, \ldots, r.$$

Somit erhält man für den Extrapolationsfehler, mit $Q = Q_0^{(1)}$ und $a_{k,r} = p_r^{(k)}(0)$, folgendes Ergebnis. Dabei entspricht r wieder dem Extrapolationsgrad und k ist der Index unser obigen Folge $(h_k)_{k\in\mathbb{N}_0}$.

$$Q(f) - a_{k,r} = \mathcal{O}\left(h_k^{2r+2}\right), \quad k \to \infty \tag{6}$$

(6) vereinigt die, in Abbildung 4 bis 6 aufgetretenen, Konvergenzverhalten der summierten Trapez- bis Mile-Regel. Das wird in der folgenden Abbildung 7 nochmals veranschaulicht. Die Testfunktion $f : x \mapsto \sin \pi x$ hat dabei offensichtlich die Stammfunktion $(\int f)(x) = -\frac{1}{\pi} \cos \pi x$ und die Steigungen entsprechen genau den Potenzen von (6).

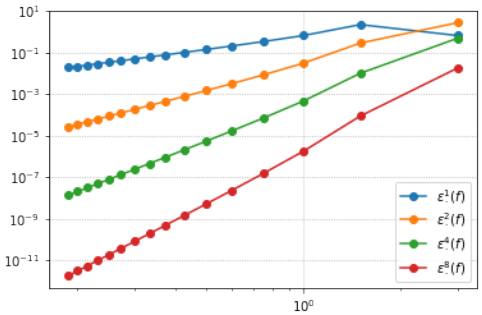


Abbildung 7: Konvergenzverhalten von $\epsilon_{\cdot}^{(m)}(f)$, $m = 2^r$, $r = 0, \ldots, 3$, auf $[0, 1]$

Zuletzt vergleichen wir noch die diversen Quadraturen $Q_h^{(m)}$ bezüglich der Anzahl der benötigten Funktionsauswertungen $\#Q_h^{(m)}$.

Zunächst einmal, kann man für die summierte Trapez-, Simpson, und Mile-Regel unmittelbar ablesen, dass $\#Q_h^{(m)} = nm + 1$, $m = 1, 2, 4$.

Bei der ineffizienten Extrapolation, werden die Ergebnisse von $Q_{h_{j-1}}^{(1)}$ nicht für die Berechnung von $Q_{h_j}^{(1)}$ wieder-verwertet. Bei einem Extrapolationsgrad von r, sind also dementsprechend viele Funktionsauswertungen zu erwarten.

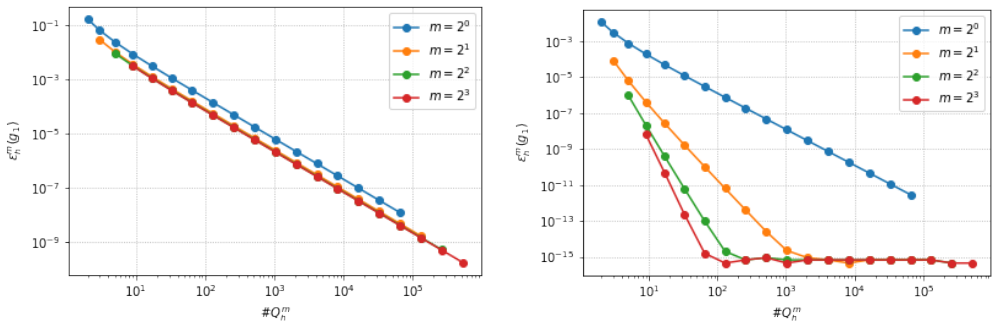
$$\#Q_{h_0}^{(2^r)} = \sum_{j=0}^{r} \#Q_{h_j}^{(1)} = \sum_{j=0}^{r} n_j + 1 = n_0 \sum_{j=0}^{r} 2^j + \sum_{j=0}^{r} 1 = n_0(2^{r+1} - 1) + r + 1$$

Die effizientere Implementierung der Extrapolation, liegt hingegen deutlich im Vorteil. Die summierte Trapez-, Simpson, und Mile-Regel passen hier offensichtlich, mit $m(r) = 2^r$, ebenfalls ins Schema.

$$\#Q_{h_0}^{(2^r)} = \#Q_{h_0}^{(1)} + \sum_{j=1}^{r} n_{j-1} = n_0 + 1 + \sum_{j=0}^{r-1} n_j = n_0 + 1 + n_0 \sum_{j=0}^{r-1} 2^j = n_0 + 1 + n_0(2^r - 1) = n_0 2^r + 1$$

Wenn wir uns nun von der ursprünglichen Grundmenge H der summierten Quadraturformeln $Q_{\cdot}^{(m)}(f)$ auf die Romberg-Folge $\{h_k : k \in \mathbb{N}_0\}$ einschränken, so erhalten wir, laut (6), sogar exponentielle Konvergenz mit der Basis 2. Das hat natürlich seinen Preis. Zwar können die Ergebnisse der Funktionsauswertungen durch diese Folge, wie wir gesehen haben, wieder verwertet werden. Das ändert aber nichts an der Tatsache, dass die Gesamtanzahl der Funktionsauswertungen immens in die Höhe getrieben wird.

In Abbildung 8 wurde der Extrapolationsfehler $\epsilon_{\cdot}^{(m)}(g_1)$, von $g_1 : [a, b] \to \mathbb{K}$, gegen die jeweilige Anzahl der Funktionsauswertungen $\#Q_{\cdot}^{(m)}$, $m = 2^r$, $r = 0, \ldots, 3$, auf beiden Achsen logarithmisch geplottet. Dabei laufen die Funktionen nun jeweils in $\{h_k : k = 0, \ldots, 16\}$.



(a) $a = 0$, $b = 1$ (b) $a = 1$, $b = 2$

Abbildung 8: $\epsilon_{\cdot}^{(m)}(g_1)$, $g_1 : [a, b] \to \mathbb{K}$, vs. $\#Q_{\cdot}^{(m)}$, $m = 2^r$, $r = 0, \ldots, 3$

Man erkennt, dass ein stumpfes Erhöhen der Anzahl der Quadraturknoten $n + 1$, aufgrund der damit steigenden Anzahl der Funktionsauswertungen $\#Q_h^{(m)} = nm + 1$, nicht immer sinnvoll ist.

Um eine bessere Approximation von $Q(f)$ zu erhalten, kann, für eine hinreichend glatte Funktionen f, eine höhere Quadraturformel bzw. ein höherer Extrapolationsgrad r, die billigere Wahl sein. Das wird durch Abbildung 8 und der Potenz in (6) bestätigt.

Sonderfälle, wie $g_1 : [0, 1] \to \mathbb{K}$, sind aber nicht auszuschließen. Dass sich dabei, in Abbildung 8, die Graphen verschieben, ist, aufgrund der Konstruktion der Romberg-Folge $(h_k)_{k\in\mathbb{N}_0}$ und dem Verhalten von $\#Q_h^{(m)}$, unmittelbar klar. Schließlich steigen beide, in k bzw. r, exponentiell, zur Basis 2.

Zuletzt wurden noch beide Extrapolationsmethoden, teuer und billig, implementiert und anhand des nicht trivialen Beispiels $\frac{\sqrt{\pi}}{2}\mathrm{erf}\,(1) = \int_0^1 \exp(-x^2)dx$ getestet. Die jeweiligen Approximationen v_i, und Rechenzeiten t_i (in Sekunden), einer Extrapolation des Grades r, sind, für beide Implementierungen $i = 1, 2$, der Tabelle 1 zu entnehmen.

r	v_1	t_1	v_2	t_2
0	0.68393972058572	0.00011710000035	0.68393972058572	0.00004950000039
1	0.74718042890951	0.00010560000010	0.74718042890951	0.00009030000001
2	0.74683370984975	0.00045020000107	0.74683370984975	0.00019090000023
3	0.74682401848228	0.00045250000039	0.74682401848228	0.00023819999842
4	0.74682413309509	0.00107159999970	0.74682413309509	0.00022480000007
5	0.74682413281224	0.00287879999996	0.74682413281224	0.00038150000000
6	0.74682413281243	0.00366840000061	0.74682413281243	0.00047970000014
7	0.74682413281243	0.00556239999969	0.74682413281243	0.00037729999895
8	0.74682413281243	0.01505440000074	0.74682413281243	0.00065469999936
9	0.74682413281243	0.03084239999953	0.74682413281243	0.00042319999920
10	0.74682413281243	0.03871509999954	0.74682413281243	0.00046709999879
11	0.74682413281243	0.09163629999966	0.74682413281243	0.00052610000057
12	0.74682413281243	0.16136100000040	0.74682413281243	0.00059430000147
13	0.74682413281243	0.32625279999957	0.74682413281243	0.00069510000139
14	0.74682413281243	0.64237740000135	0.74682413281243	0.00083020000056
15	0.74682413281243	1.38510730000053	0.74682413281243	0.00098760000037
16	0.74682413281243	2.70271429999957	0.74682413281243	0.00156319999951
17	0.74682413281243	6.02343980000114	0.74682413281243	0.00252279999950
18	0.74682413281243	13.81374770000002	0.74682413281243	0.00591520000125
19	0.74682413281243	32.06360120000136	0.74682413281243	0.01299840000138
20	0.74682413281243	75.79247419999956	0.74682413281243	0.02835090000008

Tabelle 1: Ergebnisse v_i und Rechenzeiten t_i der Extrapolationsimplementierungen $i = 1, 2$

Tabelle 1 soll veranschaulichen, wie viel Sinn es machen kann, Funktionsauswertungen zu sparen, um ein möglichst genaues Ergebnis v zu erhalten, wenn die Rechenzeit t kurz sein soll.

2.4 Nochmals die summierte Simpson- und Mile-Regel

Der Vollständigkeit halber, berechnen wir noch die vorher benutzte summierte Simpson-Regel $Q_\cdot^{(2)}$ und summierte Mile-Regel $Q_\cdot^{(4)}$.

$$
\begin{aligned}
Q_h^{(2)}(f) &= \sum_{i=1}^{n} Q^{(2)}(f|_{[x_{i-1},x_i]}) \\
&= \sum_{i=1}^{n} \frac{x_i - x_{i-1}}{6} \left(f(x_{i-1}) + 4f\left(\frac{x_i + x_{i-1}}{2}\right) + f(x_i) \right) \\
&= \frac{h}{6} \left(\sum_{i=1}^{n} f(x_{i-1}) + \sum_{i=1}^{n} 4f\left(\frac{x_i + x_{i-1}}{2}\right) + \sum_{i=1}^{n} f(x_i) \right) \\
&= \frac{1}{3} \left(\frac{h}{2} \left(\sum_{i=1}^{n} f(x_{i-1}) + \sum_{i=1}^{n} f(x_i) \right) + 2h \sum_{i=1}^{n} f\left(\frac{x_{i-1} + x_i}{2}\right) \right) \\
&= \frac{1}{3} \left(Q_h^{(1)}(f) + 2h \sum_{i=1}^{n} f\left(\frac{x_{i-1} + x_i}{2}\right) \right)
\end{aligned}
$$

$$
\begin{aligned}
Q_h^{(2)}(f) &= \frac{1}{3} \left(\frac{h}{2} \left(f(a) + 2\sum_{i=1}^{n-1} f(x_i) + f(b) \right) + 2h \sum_{i=1}^{n} f\left(\frac{x_{i-1} + x_i}{2}\right) \right) \\
&= \frac{h}{6} \left(f(a) + 4\sum_{i=1}^{n} f\left(\frac{x_{i-1} + x_i}{2}\right) + 2\sum_{i=1}^{n-1} f(x_i) + f(b) \right)
\end{aligned}
$$

$$
\begin{aligned}
&Q_h^{(4)}(f) \\
&= \sum_{i=1}^{n} Q^{(4)}(f|_{[x_{i-1},x_i]}) \\
&= \sum_{i=1}^{n} \frac{x_i - x_{i-1}}{90} \left(7f(x_{i-1}) + 32f\left(\frac{3x_{i-1} + x_i}{4}\right) + 12f\left(\frac{x_{i-1} + x_i}{2}\right) + 32f\left(\frac{x_{i-1} + 3x_i}{4}\right) + 7f(x_i) \right) \\
&= \frac{h}{90} \left(7\sum_{i=1}^{n} f(x_{i-1}) + 32\sum_{i=1}^{n} f\left(\frac{3x_{i-1} + x_i}{4}\right) + 12\sum_{i=1}^{n} f\left(\frac{x_{i-1} + x_i}{2}\right) + 32\sum_{i=1}^{n} f\left(\frac{x_{i-1} + 3x_i}{4}\right) + 7\sum_{i=1}^{n} f(x_i) \right) \\
&= \frac{1}{15} \left(\frac{7h}{6} \left(\sum_{i=1}^{n} f(x_{i-1}) + 4\sum_{i=1}^{n} f\left(\frac{x_{i-1} + x_i}{2}\right) + \sum_{i=1}^{n} f(x_i) \right) \right. \\
&\left. + \frac{h}{6} \left(32\sum_{i=1}^{n} f\left(\frac{3x_{i-1} + x_i}{4}\right) - 16\sum_{i=1}^{n} f\left(\frac{x_{i-1} + x_i}{2}\right) + 32\sum_{i=1}^{n} f\left(\frac{x_{i-1} + 3x_i}{4}\right) \right) \right) \\
&= \frac{1}{15} \left(7Q_h^{(2)}(f) \right. \\
&\left. + \frac{h}{6} \left(32\sum_{i=1}^{n} f\left(\frac{3x_{i-1} + x_i}{4}\right) - 16\sum_{i=1}^{n} f\left(\frac{x_{i-1} + x_i}{2}\right) + 32\sum_{i=1}^{n} f\left(\frac{x_{i-1} + 3x_i}{4}\right) \right) \right)
\end{aligned}
$$

$$
\begin{aligned}
&Q_h^{(4)}(f) \\
&= \frac{1}{15}\left(\frac{h}{6}\left(f(a) + 4\sum_{i=1}^{n} f\left(\frac{x_{i-1}+x_i}{2}\right) + 2\sum_{i=1}^{n-1} f(x_i) + f(b)\right)\right. \\
&\left. + \frac{h}{6}\left(32\sum_{i=1}^{n} f\left(\frac{3x_{i-1}+x_i}{4}\right) - 16\sum_{i=1}^{n} f\left(\frac{x_{i-1}+x_i}{2}\right) + 32\sum_{i=1}^{n} f\left(\frac{x_{i-1}+3x_i}{4}\right)\right)\right) \\
&= \frac{h}{90}\left(7f(a) + 32\sum_{i=1}^{n} f\left(\frac{3x_{i-1}+x_i}{4}\right) + 12\sum_{i=1}^{n} f\left(\frac{x_{i-1}+x_i}{2}\right) + 14\sum_{i=1}^{n-1} f(x_i) + 32\sum_{i=1}^{n} f\left(\frac{x_{i-1}+3x_i}{4}\right) + 7f(b)\right)
\end{aligned}
$$

Die Zwischenergebnisse reflektieren das, was beim Neville-Aitken Schema (3) passiert, wenn man die Diagonale herunterrutscht.

3 Quadraturformeln im $\mathbb{R}^2$

3.1 Projektbeschreibung

Ziel dieses Projekts ist es, das Integral von Funktionen $f : \Omega \to \mathbb{R}$ für ein gegebenes Gebiet $\Omega \subset \mathbb{R}^2$ zu approximieren. Eine mögliche Strategie besteht darin, das Gebiet in disjunkte einfache Teilgebiete $T_i, i \in I$ für eine Indexmenge I zu zerlegen, sodass

$$\Omega = \sum_{i \in I} T_i.$$

Nun konstruiert man Quadraturformeln für die einfacheren Teilgebiete T_i und summiert über die gebietweisen Integrale. Eine häufige Wahl für T_i sind Dreiecke, deshalb konstruieren wir zunächst Integrationsregeln für das Einheitsdreieck. Sei $\hat{Q} := (0,1) \times (0,1)$ und $\hat{T}$ das offene Dreieck mit den Eckpunkten $(0,0), (1,0), (0,1)$. Sei weiters

$$\Psi : \begin{cases} \mathbb{R}^2 \to \mathbb{R}^2 \\ (x,y) \mapsto (x, (1-x)y). \end{cases}$$

3.2 Vorüberlegungen

Zuerst stellen wir fest, dass die Abbildung Ψ ein Diffeomorphismus von $\mathbb{R}^2$ nach $\mathbb{R}^2$ ist: Wir betrachten die Funktionalmatrix $\mathrm{d}\Psi = \begin{pmatrix} 1 & 0 \\ -y & 1-x \end{pmatrix}$ und erhalten aus der Stetigkeit der partiellen Ableitungen $\Psi \in \mathrm{C}^1$. Andererseits ist $\Psi^{-1} : (x,y) \mapsto \left(x, \frac{y}{1-x}\right)$ die Inverse von Ψ. Aus $\mathrm{d}\Psi^{-1} = \begin{pmatrix} 1 & 0 \\ \frac{y}{1-x^2} & \frac{1}{1-x} \end{pmatrix}$ schließen wir analog $\Psi^{-1} \in \mathrm{C}^1$.
Für ein beliebiges $(x,y) \in \hat{Q}$ gibt es positive Zahlen ϵ_x und ϵ_y mit $x = 1 - \epsilon_x$ und $y = 1 - \epsilon_y$, damit gilt $\Psi(x,y) = (1 - \epsilon_x, \epsilon_x - \epsilon_x\epsilon_y)$. Ein Punkt im ersten Quadranten liegt genau dann in $\hat{T}$, wenn die Summe seiner Komponenten kleiner als 1 ist. Es gilt $(1-\epsilon_x) + (\epsilon_x - \epsilon_x\epsilon_y) = 1 - \epsilon_x\epsilon_y < 1$ und damit $\Psi(\hat{Q}) \subseteq \hat{T}$.
Sei umgekehrt $(x,y) \in \hat{T}$, dann gilt $x + y < 1$. Dieser Punkt hat unter Ψ das Urbild $\left(x, \frac{y}{1-x}\right)$. Aus $\frac{y}{1-x} < 1 \Leftrightarrow x + y < 1$ folgt $\Psi(\hat{Q}) \supseteq \hat{T}$ und somit insgesamt $\Psi(\hat{Q}) = \hat{T}$.
Also ist Ψ auch ein Diffeomorphismus von $\hat{Q}$ nach $\hat{T}$.

3.3 Quadraturformeln auf $\hat{Q}$ und $\hat{T}$

Für $N, M \in \mathbb{N}$ seien $Q_N = \sum_{j=0}^{n(N)} \alpha_j f(x_j)$ und $Q_M = \sum_{j=0}^{n(M)} \beta_j f(y_j)$ zwei Quadraturformeln der Ordnung $N+1$ bzw. $M+1$ auf dem Einheitsintervall. Wir definieren daraus auf $\hat{Q}$ eine Quadraturformel durch

$$Q_{\hat{Q}} := \sum_{i=0}^{n(N)} \sum_{j=0}^{n(M)} \alpha_i \beta_j f(x_i, y_j).$$

$\Pi_{N,M}$ sei der Raum aller Linearkombinationen von Polynomen der Form $p_{ij} : (x, y) \mapsto x^i y^j$ mit $i \leq N, j \leq M$. Durch die Quadraturformel $Q_{\hat{Q}}$ werden alle Polynome aus $\Pi_{N,M}$ exakt integriert, denn es gilt für beliebiges $r \in \Pi_{N,M}$

$$\begin{aligned} Q_{\hat{Q}}(r) &= \sum_{i=0}^{n(N)} \alpha_i \sum_{j=0}^{n(M)} \beta_j r(x_i, y_j) \overset{(1)}{=} \sum_{i=0}^{n(N)} \alpha_i \int_0^1 r(x_i, y)\, dy \\ &= \int_0^1 \sum_{i=0}^{n(N)} \alpha_i r(x_i, y)\, dy \overset{(2)}{=} \int_0^1 \int_0^1 r(x, y)\, dx dy, \end{aligned}$$

wobei (1) [(2)] gilt, da $r(x_i, y)$ $[r(x, y)]$ als Funktion in Abhängigkeit von y $[x]$ ein Polynom vom Grad $\leq M$ $[\leq N]$ ist und daher durch Q_M $[Q_N]$ exakt integriert wird.

Mithilfe von $Q_{\hat{Q}}$ können wir nun auch eine Quadraturformel auf dem Einheitsdreieck $\hat{T}$ konstruieren: Wir definieren

$$Q_{\hat{T}}(f) := Q_{\hat{Q}}((f \circ \Psi)|\det(\mathrm{d}\Psi)|)$$

und erhalten für f mit $(f \circ \Psi)|\det(\mathrm{d}\Psi)| \in \Pi_{N,M}$ unter Verwendung der bereits gezeigten Eigenschaften von Ψ und der Transformationsformel

$$Q_{\hat{T}}(f) = \int_0^1 \int_0^1 (f \circ \Psi)(x, y) |\det(\mathrm{d}\Psi \begin{pmatrix} x \\ y \end{pmatrix})|\, dx\, dy = \int_{\hat{Q}} (f \circ \Psi)|\det(\mathrm{d}\Psi)|\, d\lambda^2 = \int_{\Psi(\hat{Q})=\hat{T}} f\, d\lambda^2.$$

Die Polynome f, die durch $Q_{\hat{T}}$ exakt integriert werden, sind genau jene, die sich als Linearkombination von $p_{ij} : (x, y) \mapsto x^i y^j$ mit $i + j + 1 \leq N, j \leq M$ darstellen lassen. Für diese Basispolynome gilt nämlich

$$\begin{aligned} (p_{ij} \circ \Psi) \begin{pmatrix} x \\ y \end{pmatrix} |\det(\mathrm{d}\Psi \begin{pmatrix} x \\ y \end{pmatrix})| &= x^i (y - xy)^j |1 - x| \\ &= x^i (y^j + \cdots \pm x^j y^j)(1 - x) \\ &= x^i y^j + \cdots \pm x^{i+j} y^j - x^{i+1} y^j + \cdots \pm x^{i+j+1} y^j. \end{aligned}$$

3.3.1 Gaußquadraturen

Um Quadraturformeln auf dem Einheitsquadrat bzw. auf dem Einheitsdreieck anzugeben, die aus dem Produkt von Gauß-Quadraturen entstehen, müssen zunächst die Gauß-Quadraturen zur Gewichtsfunktion $\omega \equiv 1$ auf dem Einheitsintervall bestimmt werden.

Die zugehörigen Quadraturknoten auf dem Intervall $[-1,1]$ lassen sich mithilfe der Orthogonalpolynome bestimmen, die durch die Rekursion

$$L_0(x) = 1, \quad L_1(x) = x, \quad L_{n+1}(x) = xL_n(x) - \frac{n^2}{4n^2-1}L_{n-1}(x), \quad n \in \mathbb{N}$$

gegeben sind.
Gemäß Satz 4.23 des Skripts sind die Nullstellen des n-ten Orthogonalpolynoms L_n (und somit die n Quadraturknoten der Gauß-Quadratur Q_{n-1}) genau die Eigenwerte der Matrix

$$\begin{pmatrix} \beta_0 & \gamma_1 & & & \\ \gamma_1 & \beta_1 & \gamma_2 & & \\ & \gamma_2 & \ddots & \ddots & \\ & & \ddots & \ddots & \gamma_{n-1} \\ & & & \gamma_{n-1} & \beta_{n-1} \end{pmatrix}, \text{ wobei in unserem Fall für alle } m \in \mathbb{N}_0 \text{ gilt } \beta_m = 0, \gamma_m = \sqrt{\tfrac{m^2}{4m^2-1}}.$$

Die affin-lineare Abbildung $\Phi : [-1,1] \to [0,1] : \zeta \mapsto \frac{1}{2}(1+\zeta)$ liefert die entsprechenden Quadraturknoten auf dem Einheitsintervall.
Die zugehörigen Gewichte α_j werden ebenso wie in Satz 4.23 berechnet, wobei für normierte Eigenvektoren aus $\int_0^1 \omega(x) = 1$ folgt, dass

$$\alpha_j = ((v_j)_1)^2, \quad j = 0, \ldots, n.$$

Listing 1: Berechnung von Knoten und Gewichten für die Gauß-Quadratur

```
def nodesnweights(n):

    gamma = np.array([np.sqrt((i+1)**2/(4*(i+1)**2-1)) for i in range(n-1)])
    T = np.zeros((n,n)) + np.diag(gamma,1) + np.diag(gamma,-1)
    [vals,vecs] = np.linalg.eig(T)  #vecs sind bereits normiert

    vals = phi(vals)
    alpha = np.array([(vecs[0][i])**2 for i in range(n)])
    return [vals,alpha]
```

Die aus dem Produkt von solchen Gauß-Quadraturen entstehenden Quadraturformeln $Q_{\hat{Q}}$ und $Q_{\hat{T}}$ kann man nun analog zur Beschreibung von oben implementieren:

Listing 2: Implementierung von $Q_{\hat{Q}}$

```
def gaussQ(f,n):
    [x,a] = nodesnweights(n+1)
    sum = 0
    for i in range(n+1):
        for j in range(n+1):
            sum += a[i]*a[j]*f(x[i],x[j])
    return sum
```

Listing 3: Implementierung von $Q_{\hat{T}}$

```
def gaussUT(f,n):
    def z(x,y):
        return f(x,(1-x)*y)*(1-x)
    return gaussQ(z,n)
```

3.3.2 Quadraturen auf $\hat{T}$ mithilfe von Lagrange-Polynomen

Alternativ dazu konstruieren wir uns Quadraturformeln $Q_n(f) := \int_{\hat{T}} p_f \, d\lambda^2$ auf dem Einheitsdreieck, wobei wir das interpolierende Polynom mit den in Übungsaufgabe 22 berechneten Lagrange-Polynomen darstellen können. Wie im eindimensionalen Fall erhalten wir also eine Quadraturformel der Form $Q_n(f) = \sum_{j=0}^{n} \alpha_j f(x_j, y_j)$ mit Quadraturgewichten $\alpha_j = \int_{\hat{T}} L_j(x, y) \, d\lambda^2$.

Listing 4: Quadraturformeln erster und zweiter Ordnung auf $\hat{\mathrm{T}}$ mit Lagrange-Polynomen

```
def interp1UT(f):
    z = [f(0,0), f(1,0), f(0,1)]
    return z[0]*(1/6) + z[1]*(1/6) + z[2]*(1/6)

def interp2UT(f):
    z = [f(0,0), f(1,0), f(0,1), f(1/2,1/2), f(1/2,0), f(0,1/2)]
    return z[0]*0 + z[1]*0 + z[2]*0 + z[3]*(1/6) + z[4]*(1/6) + z[5]*(1/6)
```

3.4 Quadraturformeln auf beliebigen Dreiecken

Nun können wir die bisher konstruierten Quadraturformeln auf dem Einheitsdreieck auch für ein beliebiges Dreieck T verallgemeinern. Für ein solches Dreieck (gegeben durch die 3 affin unabhängigen Eckpunkte x, y und z) definieren wir uns die affine Abbildung

$$\pi_T : \hat{T} \to T : a \mapsto A_T(a) + x, \qquad \text{wobei } A_T = \begin{pmatrix} z_1 - x_1 & y_1 - x_1 \\ z_2 - x_2 & y_2 - x_2 \end{pmatrix}.$$

Man kann leicht nachprüfen, dass diese Abbildung die drei Eckpunkte des Einheitsdreiecks auf x, y und z abbildet und somit $\pi_T(\hat{T}) = T$ leistet.
Da A regulär ist, ist π ein Diffeomorphismus und es gilt mit der Transformationsformel

$$\int_T f \, d\lambda^2 = \int_{\pi_T(\hat{T})} f \, d\lambda^2 = \int_{\hat{T}} (f \circ \pi_T) |\det(\mathrm{d}\pi_T)| \, d\lambda^2.$$

Wir definieren also für eine auf dem Einheitsdreieck gegebene Quadraturformel $Q_{\hat{T}}$ die neue Quadraturformel auf T als

$$Q_T(f) := Q_{\hat{T}}((f \circ \pi_T)|\det(\mathrm{d}\pi_T)|) = Q_{\hat{T}}((f \circ \pi_T)|\det(A_T)|).$$

Listing 5: Gauß-Quadratur auf beliebigem Dreieck T

```
def gaussT(f,n,T):
    def w(x,y):
        A = np.array([ [T[2][0]-T[0][0], T[1][0]-T[0][0]],
                       [T[2][1]-T[0][1], T[1][1]-T[0][1]] ])
        l = A@np.array([x,y]) + np.array([T[0][0],T[0][1]])
        det = abs(A[0][0]*A[1][1] - A[0][1]*A[1][0])
        return f(l[0],l[1])*det
    return gaussUT(w,n)
```

Listing 6: Quadratur mit Lagrange-Polynomen auf beliebigem Dreieck T

```
def interp1T(f,T):
    def w(x,y):
        A = np.array([ [T[2][0]-T[0][0], T[1][0]-T[0][0]],
                       [T[2][1]-T[0][1], T[1][1]-T[0][1]] ])
        l = A@np.array([x,y]) + np.array([T[0][0],T[0][1]])
        det = abs(A[0][0]*A[1][1] - A[0][1]*A[1][0])
        return f(l[0],l[1])*det
    return interp1UT(w)


def interp2T(f,T):
    def w(x,y):
        A = np.array([ [T[2][0]-T[0][0], T[1][0]-T[0][0]],
                       [T[2][1]-T[0][1], T[1][1]-T[0][1]] ])
        l = A@np.array([x,y]) + np.array([T[0][0],T[0][1]])
        det = abs(A[0][0]*A[1][1] - A[0][1]*A[1][0])
        return f(l[0],l[1])*det
    return interp2UT(w)
```

3.5 Quadraturformeln auf einem beliebigen Gebiet mithilfe von Triangulierung

Jetzt sind wir fast am Ziel angelangt; wir betrachten nun ein Gebiet $\Omega \subset \mathbb{R}^2$ und eine Funktion $f : \Omega \to \mathbb{R}$. Unsere bisherige Arbeit liefert uns die Werkzeuge, um das Integral $\int_\Omega f \, d\lambda^2$ näherungsweise zu berechnen. Hiezu betrachten wir disjunkte Dreiecke $T_i, i \in I$ mit $\sum_{i\in I} T_i \subseteq \Omega$. Es stellt sich die Frage, wie gut die Approximation

$$\int_\Omega f \, d\lambda^2 \approx \sum_{i\in I} Q_{T_i}(f)$$

in Abhängigkeit von der Feinheit der Triangulierung ist.

Zuerst betrachten wir das Integral der Funktion $f : \mathbb{R}^2 \to \mathbb{R}, (x, y) \mapsto \exp(\frac{x+y}{x-y})$ über dem Viereck mit den Eckpunkten $(0, -1), (0, -2), (2, 0), (1, 0)$, das wir mithilfe der Transformationsformel exakt berechnen können. Wir verwenden das Python-Package dmsh, um Triangulierungen zu generieren, wobei wir jeweils Dreiecke mit den Kantenlängen $h = (\frac{2}{3})^i, i = 1, ..., 10$ wählen. Dabei integrieren wir auf zwei Arten: Mit der aus dem Produkt von Gauß-Quadraturen konstruierten Formel Q_T mit Ordnung 4 und mit der mithilfe von Lagrange-Interpolation konstruierten Quadraturformel 2. Ordnung. In diesem Test erweist sich letztere als mindestens ebenbürtig und liefert für $i \geq 7$ sogar einen besseren Schätzwert für das Integral. Mithilfe einer Vergleichsgerade können wir auf einem doppelt logarithmischen Plot ablesen, dass beide Formeln ungefähr Konvergenzordnung 4 haben (Abb. 1).

Als nächstes testen wir unsere Formeln, indem wir die Funktion $g : \mathbb{R}^2 \to \mathbb{R}, (x, y) \mapsto \sin(x^2 + y^2)$ über den Einheitskreis integrieren. Auch hier lässt sich der exakte Wert sofort über die Transformationsformel bestimmen. Aus dem Konvergenzplot (Abb. 2) schließen wir auf Konvergenzordnung 2. Der Fehler im ersten Beispiel war vergleichsweise klein und kann auf die Fehler der Quadraturformeln zurückgeführt werden; hier hingegen wird die Abhängigkeit von der Triangulierung deutlich, die ein rundes Gebiet nicht vollständig überdecken kann. Der dadurch entstandene Fehler ist so groß, dass er die Ungenauigkeit der Quadraturformeln verdeckt und kein Unterschied zwischen den beiden mehr ausgemacht werden kann.

Abb. 1: exp((x+y)/(x-y)) auf einem Polygon

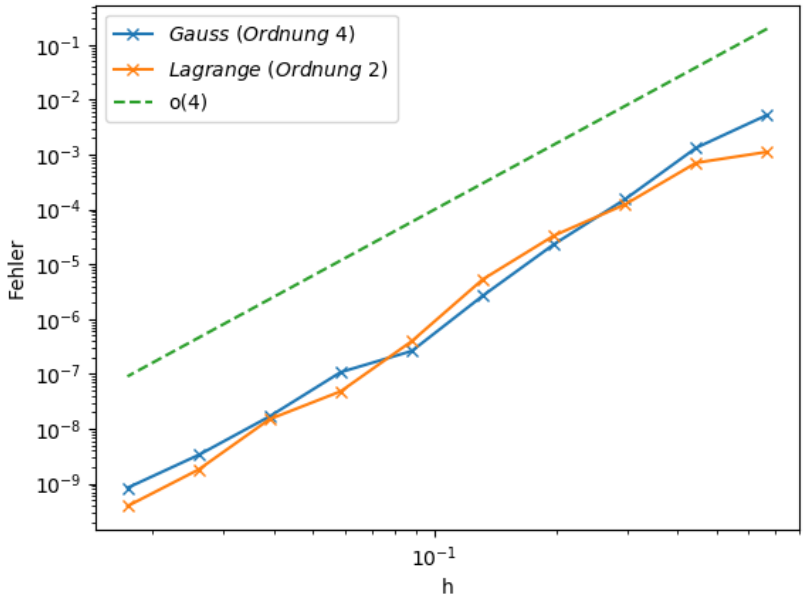


Abb. 2: sin(x²+y²) auf dem Einheitskreis