

---

# Hinweise zur Arbeit mit Octave

## im Rahmen der Vorlesung Mathematik IT 3 (Analysis)

für die Studiengänge Informatik, IMT und eBusiness

---

Bei der praktischen Lösung verschiedenster Aufgaben stellen wir immer wieder fest, dass nicht wenige Studierende oft keine Vorstellung davon haben, was sie eigentlich berechnen oder beweisen sollen. Nicht selten wird dann „irgendetwas“ gerechnet, ohne auch nur einen Gedanken an den Sinn (oder Unsinn) einer Rechnung zu verschwenden. Auch mangelt es oft an der notwendigen Selbstkritik bzw. Selbstkontrolle.

Um einen leichteren Zugang zu den in dieser Vorlesung diskutierten Fragestellungen der Analysis zu bekommen, kann es durchaus zweckmäßig sein, die gestellten Probleme erst einmal grafisch darzustellen (z.B. bei der Untersuchung von Funktionen in einer Variablen) oder einfache Testrechnungen durchzuführen (z.B. um eine Vermutung über die Konvergenz oder Divergenz von Folgen und Reihen zu erhalten oder berechnete Ergebnisse grob zu überprüfen).

Wenn das Zeichnen mit Papier und Bleistift zu umständlich/aufwändig ist oder bei komplizierten Rechnungen auch der Taschenrechner nicht weiterhilft, dann können und sollen die Teilnehmer auch nicht davor zurückscheuen, entsprechende Mathematik-Software auf einem Computer zu benutzen. Für den Hausgebrauch empfiehlt sich dabei die Verwendung der kostenfreien Software Octave, die zu einem großen Teil zu dem kostenpflichtigen Softwarepaket Matlab ähnlich ist, aber auch nicht alle Funktionalitäten von Matlab bereitstellt. Matlab und Octave sind Softwarepakete zur Berechnung, Visualisierung und auch zur Programmierung technisch-wissenschaftlicher, insbesondere mathematischer Ausdrücke. Die Octave-Grundausstattung kann z.B. über <http://www.gnu.org/software/octave/> abgerufen werden.

Der Umgang mit Octave ist leicht zu erlernen. Nach dem Start von Octave können im Kommandofenster Kommandos eingegeben werden, wobei auf weiter zurückliegende Eingaben mit den „Auf/Ab“-Pfeiltasten zugegriffen werden kann. Man kann die Ergebnisse der Berechnungen Variablen mit beliebigen Namen zuweisen. Die Zuweisung erfolgt mit Hilfe des Gleichheitszeichens. Ist kein Variablenname vorhanden, dann generiert Octave die Variable `ans` und weist dieser das Ergebnis zu. Grundsätzlich werden die Berechnungsergebnisse angezeigt, dies kann aber durch anhängen eines Semikolons am Ende des Befehls unterdrückt werden. Jede Eingabe ist mit ENTER abzuschließen. Wie in jedem ähnlichen Softwarepaket werden auch hier die normalen Operatoren (+, -, \*, /) unterstützt. Bei Dezimalzahlen ist ein Punkt als Trennzeichen zu verwenden.

Octave stellt eine Vielzahl einfacher und komplexerer Funktionen zur Verfügung. Zu den einfachen gehören zum Beispiel `abs`, `sin`, `cos`, `exp`, `log`, `sqrt`, `sum`, `mean`, `setdiff`, `union`, `and`, `or`, `not`. Durch einige Anweisungen wird „richtiges“ programmieren möglich, zum Beispiel können die auch von diversen Programmiersprachen bekannte `if-else`-Anweisung, `for`- und `while`-Schleife verwendet werden. Um lokale und globale Variablen muss sich der Anwender allgemein keine Sorgen machen. Weitere Details zur Programmierung mit Octave entnehme man einschlägigen Lehrbüchern oder diversen im Internet abgelegten Dokumentationen und Einführungen (insbesondere auch solchen, die sich mit Matlab befassen). Eine sehr kurze Übersicht findet man z.B. auch unter <http://vieta.math.tu-cottbus.de/~kunath/Allgemein/matlabEinfuehrung.pdf>.

Es folgen einige Beispiele:

**Beispiel 1:** Berechnung von Gliedern der explizit definierten Folge  $(a_n)_{n \in \mathbb{N}}$  mit  $a_n := \sqrt{n^2 - 1} - \sqrt{2n^2 + 1}$

- direkte Eingabe z.B. für  $n = 10$ : `a = sqrt(10^2-1) - sqrt(2*10^2+1) → Ergebnis: a = -4.2276`
- weitere Nachkommastellen anzeigen: `num2str(a,20) → Ergebnis: ans = -4.227572507691626`
- Definition von  $(a_n)_{n \in \mathbb{N}}$  als Funktion (man beachte den `.` vor `^!`):

`an = @(n) sqrt(10.^2-1) - sqrt(2*n.^2+1);`

- Berechnung und Anzeige der Folgenglieder  $a_1, a_{10}, a_{50}, a_{100}, a_{1000}, a_{10000}$  mittels **for**-Schleife

```
for (n=[1,10,50,100,1000,10000]), disp(num2str(an(n),20)), end;
```

bzw. Berechnung und Anzeige ohne **for**-Schleife

```
disp(num2str(an([1;10;50;100;1000;10000]),20))
```

Ergebnis (was die Divergenz der Folge vermuten läßt, der Beweis sei zu gegebener Zeit als Übung empfohlen!):

```
8.2178235634973227
-4.227572507691626
-60.767874461882371
-131.47501735595617
-1404.2640415553751
-14132.185784715222
```

**Beispiel 2:** Berechnung der  $k$ -ten Partialsumme der Reihe  $\sum_{n=1}^{\infty} \frac{(-3)^n + (-1)^{n+1} + \cos(n)}{4^n + 8}$

- Berechnung und Ausgabe mittels **for**-Schleife z.B. für  $k = 10$ :

```
sk = 0; for (n=1:10), sk = sk + ((-3)^n+(-1)^(n+1)+cos(n))/(4^n+8); end, disp(num2str(sk,20))
```

Ergebnis: 0.011447147682766765

- Definition des  $n$ -ten Summanden als Funktion (man beachte den `.` vor den Punktoperationen `^` und `/`)

```
an = @(n) ((-3).^n+(-1).^(n+1)+cos(n))./(4.^n+8);
```

Berechnung und Ausgabe mit **for**-Schleife:

```
sk = 0; for (n=1:10), sk = sk + an(n); end, disp(num2str(sk,20))
```

und ohne

```
sum(an(1:10)); disp(num2str(sk,20))
```

- Berechnung und Ausgabe der  $k$ -ten Partialsumme für  $k = 1, 2, 3, 4, 5, 10, 20, 50, 100, 195, 196, 197, 198, 200$ :

```
for (k=[1:5,10,20,50,100,195:200]), disp(num2str(sum(an(1:k)),20)), end
```

Ergebnis (was die Konvergenz der Reihe vermuten läßt, der Beweis sei zu gegebener Zeit als Übung empfohlen!):

```
-0.12164147451098835
0.19435240729954739
-0.18050859959768106
0.12004578062632038
-0.11417547694590519
0.011447147682766765
-0.011327801910266735
-0.012686650031958032
-0.012686892741101909
-0.012686892741239363
-0.012686892741239363
-0.012686892741239363
-0.012686892741239363
-0.012686892741239363
-0.012686892741239363
-0.012686892741239363
```

**Beispiel 3:** Plotten des Funktionsgraphen von  $f(x) := \exp\left(-\frac{x^2 - 2x + 3 + |x^3|}{x^4 + 1}\right)$  im Intervall  $[-5, 5]$

- Definition der Funktion (man beachte den `.` vor den Punktoperationen `*`, `/`, `^`):

```
f = @(x) exp(-(x.^2-2*x+3+abs(x.^3))./(x.^4+1));
```

- Der Graph einer Funktion lässt sich nie exakt darstellen, sondern nur als Approximation (d.h. Annäherung) mit Hilfe eines Polygonzuges. Wie gut diese Approximation ist, hängt von der Wahl des Approximationsgitters ab, das hier z.B. wie folgt definiert wird:

```
t = -5:0.01:5;
```

Die erste bzw. letzte Zahl sind Anfang bzw. Ende des Intervalls, in dem  $f$  dargestellt werden soll, die mittlere Zahl 0.01 zwischen den Doppelpunkten gibt die Gitterweite an, d.h. der durch Verbindung der Punkte  $(-5 + k \cdot 0.01, f(-5 + k \cdot 0.01))$ ,  $k = 0, 1, \dots, 10000$  entstehende Polygonzug wird zur grafischen Darstellung von  $f$  verwendet:

```
plot(t,f(t))
```

Wird das Approximationsgitter zu grob gewählt, dann ergibt dies eine „schlechte“ (und oft sogar falsche) Darstellung einer Funktion. Zum Beispiel mit

```
t2 = -5:2.5:5;
```

sieht der Funktionsgraph etwas kantig aus, wichtige Details wie Wendepunkte und lokale Extremwerte sind mit dem Auge kaum noch zu erkennen. Das macht man sich z.B. deutlich, wenn man die beiden Polygonzüge in einer Grafik gegenüberstellt (`'r'` steht dabei für rot, `'g'` für grün):

```
plot(t,f(t),'g',t2,f(t2),'r'), legend('gute Approx. von f','schlechte Approx. von f')
```

Ergebnis des `plot`-Befehls ist eine Grafik, die (mit Octave erzeugt) nicht weiter bearbeitet werden und auch nicht gespeichert oder exportiert werden kann. Zum exportieren gibt es aber den `print`-Befehl mit diversen Optionen. Einfachste Vorgehensweise:

```
h = figure;
plot(t,f(t),'r',t2,f(t2),'g')
legend('gute Approx. von f','schlechte Approx. von f')
print(h,'-dpdf','datei_als_pdf.pdf')
print(h,'-djpeg','datei_als_jpg.jpg')
print(h,'-dbmp','datei_als_bmp.bmp')
```

Das erste Argument ist dabei der Plot selbst, das zweite Argument gibt das Ausgabeformat an, schließlich folgt der Dateiname (wobei man die Dateiendung auch weglassen kann). Gespeichert wird dabei immer im aktuellen Arbeitsverzeichnis. Man kann aber auch in beliebige Verzeichnisse speichern, dazu ist dem Dateinamen einfach den Dateipfade voranstellen, z.B.

```
print(h,'-dpdf','D:/Unterverzeichnis/Unterunterverzeichnis/datei_als_pdf.pdf')
```

**Beispiel 4:** Plotten des Funktionsgraphen der stückweise definierten, stetigen Funktion

$$f(x) = \begin{cases} (x+1)^2 & , \quad x < -1 \\ -(x+1)^2 & , \quad -1 \leq x \leq 1 \\ x-5 & , \quad x > 1 \end{cases}$$

im Intervall  $[-2, 3]$ , zusätzlich Markierung der Funktionswerte  $f(-1)$  und  $f(1)$

- Definition der Funktion (man beachte den `.` vor den Punktoperationen `*`, `/`, `^`):

```
f = @(x) ((x+1).^2).*(x < -1);
f = @(x) f(x) + (-(x+1).^2).*(and(-1 <= x, x <= 1));
f = @(x) f(x) + (x-5).*(x > 1);
```

Man beachte, dass die Definitionsintervalle eines Funktionsabschnitts als logische Ausdrücke an die Definition des jeweiligen Funktionsabschnitts „multipliziert“ und die Funktionsabschnitte „addiert“ werden.

- weiter analog zu Beispiel 3:

```
t = -2:0.01:3;
plot(t,f(t),'r',[-1,1],f([-1,1]),['o','b'])
```

Durch die Anweisungen `[-1,1],f([-1,1]),['o','b']` werden die Punkte  $(-1, f(-1))$  und  $(1, f(1))$  als blaue Kreise auf dem Funktionsgraphen markiert.

**Beispiel 5:** Plotten des Funktionsgraphen der stückweise definierten, in  $x = 1$  unstetigen Funktion

$$f(x) = \begin{cases} (x+1)^2 & , \quad x < -1 \\ -(x+1)^2 & , \quad -1 \leq x \leq 1 \\ x-3 & , \quad x > 1 \end{cases} \quad \text{im Intervall } [-2, 3]$$

- Definition der Funktion (man beachte den `.` vor den Punktoperationen `*`, `/`, `^`):

```
f = @(x) ((x+1).^2).*(x < -1);
f = @(x) f(x) + (-(x+1).^2).*(and(-1 <= x, x <= 1));
f = @(x) f(x) + (x-3).*(x > 1);
```

- Ein weiteres Vorgehen analog zu Beispiel 3 führt zu einem geschlossenen Polygonzug, d.h. in der Grafik wirkt die unstetige Funktion so, als wäre sie doch stetig. Dies kann zu Fehlinterpretationen führen. Da Unstetigkeitsstellen nicht von selbst erkannt werden, muss der Anwender dies dem Programm selbst beibringen. Hier ein Vorschlag:

```
t1 = -2:0.01:1;
t2 = 1:0.01:3;
t2(1) = 1.0001;
plot(t1,f(t1),'r',t2,f(t2),'r')
```

**Beispiel 6:** Probleme bei der Berechnung des Funktionswertes  $f(0)$  der stückweise definierten Funktion

$$f(x) := \begin{cases} (x+1) \cdot e^{2x} & , \quad x \leq 0 \\ x \cdot \ln(2x) & , \quad x > 0 \end{cases}$$

- Definition der Funktion analog zu den Beispielen 4 und 5:

```
f = @(x) (exp(2*x)).*(x+1)).*(x <= 0);
f = @(x) f(x) + (x.*(log(2*x))).*(x > 0);
```

- Offenbar ist  $f(0) = 1$ . Die Eingabe `a = f(0)` liefert aber das Ergebnis `a = NaN`, obwohl  $f$  in  $x = 0$  explizit definiert ist. Hier kommt es vermutlich zu numerischen Problemen, die zwar beim Plotten des Funktionsgraphen keine Probleme verursachen, da `Not-a-Number` hier einfach ignoriert wird. Für weitere Rechnungen ist das natürlich nicht akzeptabel, und mahnt zur Vorsicht bei der Behandlung von stückweise definierten, unstetigen Funktionen nach dem o.g. Muster. In so einem Fall sollte man also die beiden Funktionsabschnitte besser einzelnen definieren, wobei man auch den Definitionsbereich einschränken kann, z.B.

```
f1 = @(x) (exp(2*x).*(x+1)).*(x <= 0);
f2 = @(x) x.*(log(2*x)).*(x > 0);
```

Dann ergibt die Eingabe `a = f1(0)` das Ergebnis `a = 1`. Würde man hier allerdings versehentlich `a = f2(0)` eingeben, so liefert dies `a = NaN`, obwohl `f2` für  $x = 0$  gar nicht definiert wurde. Für Werte, die nicht zum Definitionsbereich gehören, wird ansonsten 0 zurückgegeben.

- Die genannten Probleme kann man umgehen, wenn man die Funktion  $f$  nicht als „anonyme“ Funktion definiert, sondern als „richtige“ Funktion (*m*-File) definiert. Der folgende Beispielcode

```
01 function[funktionswerte] = f_stueckweise(x)
02     n = length(x); % Anzahl der Stellen x, für die f(x) berechnet werden soll
03     funktionswerte = zeros(1,n); % Nullvektor der Länge n, wird später mit den Funktionswerten überschrieben
04
05     index = 1:n; % Indizes für die logische Indizierung
06
07     ind = index(x <=0); % Indizes aller x, die kleiner gleich Null sind
08     funktionswerte(ind) = exp(2*x(ind)).*(x(ind)+1);
09
10     ind = index(x >0); % Indizes aller x, die größer Null sind
11     funktionswerte(ind) = x(ind).*(log(2*x(ind)));
```

ist dazu im gewünschten Arbeitsverzeichnis unter dem Namen `f_stueckweise.m` zu speichern. Ist das erledigt, dann kann man damit wie eben arbeiten, d.h. zur Berechnung von  $f(0)$  gibt man

```
a = f_stueckweise(0)
```

ein, und erhält das richtige Ergebnis. Man kann auch folgendes eingeben:

```
f = @f_stueckweise;
a = f(0)
```

Man beachte, dass auch mit der Funktion `f_stueckweise.m` beliebig viele Funktionswerte gleichzeitig berechnet werden können. Diese sind dann analog zu den obigen Beispielen als Vektor zur übergeben, d.h. `f_stueckweise([-1,0,1,2])` liefert einen (Zeilen)vektor mit den Funktionswerten  $f(-1), f(0), f(1), f(2)$ .

**Beispiel 7:** Numerische Berechnung des bestimmten Integrals  $\int_{-3}^2 x^2 + 2x - 1 \, dx$

Bei exakter Rechnung erhält man

$$\int_{-3}^2 x^2 + 2x - 1 \, dx = \frac{5}{3}$$

Zur Selbstkontrolle kann man das Endergebnis mit Octave zumindest näherungsweise mit Hilfe der Funktion `quad.m` berechnen. Dazu ist folgendes einzugeben:

```
f = @(x) x.^2+2*x-1;
erg = quad(f,-3,2,1.0e-012);
```

Die letzte Zahl nach der Funktion und den Integrationsgrenzen gibt dabei die Genauigkeit an, mit der die numerische Rechnung durchgeführt werden soll. Sie sollte immer möglichst klein gewählt werden, da das Ergebnis sonst sehr ungenau sein kann. Selbst mit der hier gewählten Genauigkeit von  $10^{-12}$  wird das exakte Ergebnis nicht erhalten, wie die Eingabe von `disp(num2str(erg,20))` zeigt:

```
1.66666666666666662966
```

**Beispiel 8:** Numerische Lösung des Anfangswertproblems  $y' = 2xy$ ,  $y(0) = \pi$ 

Die Lösung dieser homogenen Differentialgleichung erster Ordnung nach der Methode der Trennung der Variablen liefert die Lösungsfunktion

$$y(x) = \pi \cdot e^{x^2}.$$

Zur Selbstkontrolle ist es natürlich leicht möglich,  $y$  und  $y'$  in die Differentialgleichung einzusetzen. Bei komplizierteren Differentialgleichungen und/oder Lösungsfunktionen kann man die Lösung aber auch mit Hilfe von Octave überprüfen. Eine exakte Berechnung des Ergebnisses ist nicht möglich, aber immerhin können Anfangswertprobleme numerisch gelöst werden. Dazu kann die Funktion `lsode(y_strich,y_0,t)` verwendet werden. Der Parameter `y_strich` enthält die Ableitung  $y' = f(x, y)$ , der Parameter `y_0` den Anfangswert  $y_0 = y(x_0)$  und der Parameter `t` ist ein Vektor  $(x_0, x_1, x_2, \dots, x_m)$ ,  $m \in \mathbb{N}_0$ , wo  $x_0$  stets das zum Anfangswert  $y_0$  zugehörige Argument ist. Die Funktion gibt den Spaltenvektor  $(y(x_0), y(x_1), \dots, y(x_m))^T$  mit den Funktionswerten der Lösungsfunktion  $y$  zurück. Die Kontrolle einer mit Papier und Stift berechneten Lösung kann man damit z.B. entweder durch Berechnung einzelner Funktionswerte oder durch grafischen Vergleich der exakten mit der numerisch berechneten Lösungskurve vornehmen. Für das Beispiel ist dazu folgendes einzugeben:

```
y_strich = @(x,y) 2*x.*y;
y_per_hand_berechnet = @(x) pi*exp(x.^2);
t = 0:0.25:2;
disp(num2str([transpose(t),lsode(y_strich,pi,t),transpose(y_per_hand_berechnet(t))]))
```

Ergebnis ist folgende Ausgabe in der Kommandozeile (die die Rechnung per Hand bestätigen):

|        |          |          |
|--------|----------|----------|
| 0      | 3.1416   | 3.1416   |
| 0.2500 | 3.3442   | 3.3442   |
| 0.5000 | 4.0339   | 4.0339   |
| 0.7500 | 5.5137   | 5.5137   |
| 1.0000 | 8.5397   | 8.5397   |
| 1.2500 | 14.9877  | 14.9877  |
| 1.5000 | 29.8066  | 29.8066  |
| 1.7500 | 67.1703  | 67.1702  |
| 2.0000 | 171.5253 | 171.5251 |

Man beachte die kleinen Unterschiede zwischen der numerischen (2. Spalte) und der exakten Lösung (3. Spalte). Ein grafischer Vergleich kann z.B. wie folgt aussehen:

```
y_strich = @(x,y) 2*x.*y;
y_per_hand_berechnet = @(x) pi*exp(x.^2);
t = 0:0.01:2;
plot(t,lsode(y_strich,pi,t),'r',t,y_per_hand_berechnet(t),'g')
```

Zwischen beiden Kurven sind keine Unterschiede zu sehen, also wurde per Hand offenbar richtig gerechnet.

## W A R N U N G !

**Die Verwendung von Computersoftware wie zum Beispiel das hier kurz vorgestellte Octave kann und soll eine Unterstützung beim Lernen und Verstehen sein, kann aber die intensive Auseinandersetzung mit dem Vorlesungsstoff nicht ersetzen!** Letzteres sollte vor allem bei der Bearbeitung der Haus- und Übungsaufgaben ganz altmodisch mit Papier und (Blei)stift (und insbesondere nicht durch Abschreiben) erfolgen. Die Hausaufgaben werden in der Regel so gestellt, dass bei einfachen Rechnungen die Ergebnisse auch ohne (Taschen)rechner exakt berechnet werden können. Dies trainiert gleichzeitig für die Prüfungsklausur, bei der kein Taschenrechner verwendet werden darf. Daher werden wir bei den Hausaufgaben auch keine Näherungslösungen akzeptieren, wie Sie z.B. bei  $\exp$  –,  $\ln$  –,  $\sin$  –,  $\cos$  – oder Wurzelausdrücken berechnet werden können.