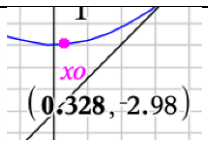
Hubert Langlotz
Sebastian Rauh, Veit Berger

Das Newtonverfahren mit Notes bzw. durch Programmierung

Wir wollen hier als Ergänzung zum Beitrag von Wilfried Zappe „Die letzte Antwort“ – Gleichungen lösen mit ANS“, in dem u.a. das Newtonverfahren beschrieben wird, einige weitere Möglichkeiten vorstellen, wie das Newtonverfahren im Unterricht auch genutzt werden kann. Die im Artikel von Wilfried Zappe erwähnten theoretischen Grundlagen des Verfahrens sowie dessen Grenzen werden wir hier nicht noch einmal thematisieren.

1. Graphische Veranschaulichung mittels Notes

Notes ist die dynamische Oberfläche des TI Nspire. Hier lässt sich mit dem mächtigen Befehl seqGen() das Newtonverfahren extrem kompakt realisieren. Die Visualisierung erfolgt in einem „Graphs“-Fenster, genau wie die dynamische Auswahl der Startstelle x_0 .

Zunächst wird die Ausgangsfunktion als $f(x)$ definiert. Die numerische Ableitung ist lediglich der Differenzenquotient mit einem relativ großen h.	Ausgangsfunktion $f(x):=x^2-4$ ▸ *Fertig* (numerische) Ableitung $fs(x):=\frac{f(x+h)-f(x)}{h}\|h=\frac{1}{100.}$ ▸ *Fertig*
Im „Graphs“-Fenster wird ein Punkt auf dem Graphen als Startstelle x_0 festgelegt. Die x-Koordinate wird als $x0$ gespeichert.	
Außerdem wird im „Graphs“-Fenster ein Schieberegler eingefügt, der die Anzahl der Iterationen festlegt. Damit lässt sich später der schrittweise Prozess der Näherung beobachten.	
Nun kann das Newtonverfahren entsprechend der Vorschrift $x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}$ durchgeführt werden.	$\text{seqGen}\left(x(n-1)-\frac{f(x(n-1))}{fs(x(n-1))},n,x,\{1.,nn\},\{x0\}\right)$ Bildungsvorschrift Laufparameter Variable Bereich von n Startwert
Das Ergebnis dieser Berechnung ist eine Liste, die zu einer Nullstelle konvergiert.	$\{0.328,6.17,3.41,2.29,2.02,2.\}$

Erweiterung:
Nun soll der Iterationsprozess graphisch mit Tangenten dargestellt werden.

Zunächst muss die Liste der Näherungen gespeichert werden. In diesem Fall als „l1“	$l1:=\text{seqGen}\left(x(n-1)-\frac{f(x(n-1))}{fs(x(n-1))}\ldots\right.$
Aus diesen Nullstellen können mit der Funktion tangentLine die Funktionsgraphen der Tangenten bestimmt werden.	Für die Tangenten $f2(x):=\text{tangentLine}(f(x),x,l1[nn-1])$

Um den Verlauf der Nullstellen beobachten zu können, sollen die Nullstellen ebenfalls dargestellt werden. Dazu muss eine Liste mit Nullen erzeugt werden, die die gleiche Dimension wie die Liste der Nullstellen hat.	$yy:=seqn(0,nn) \rightarrow \{0.,0.,0.,0.,0.\}$
Im letzten Schritt werden im „Graphs“-Fenster die Tangenten mit dem Namen $f2(x)$ und der Streuplot der Nullpunkte angezeigt.	s1 $\begin{cases} x \leftarrow l1 \\ y \leftarrow yy \end{cases}$

Abschließend ergibt sich dieses Bild:

Ausgangsfunktion

$f(x):=x^2-4 \rightarrow Fertig$

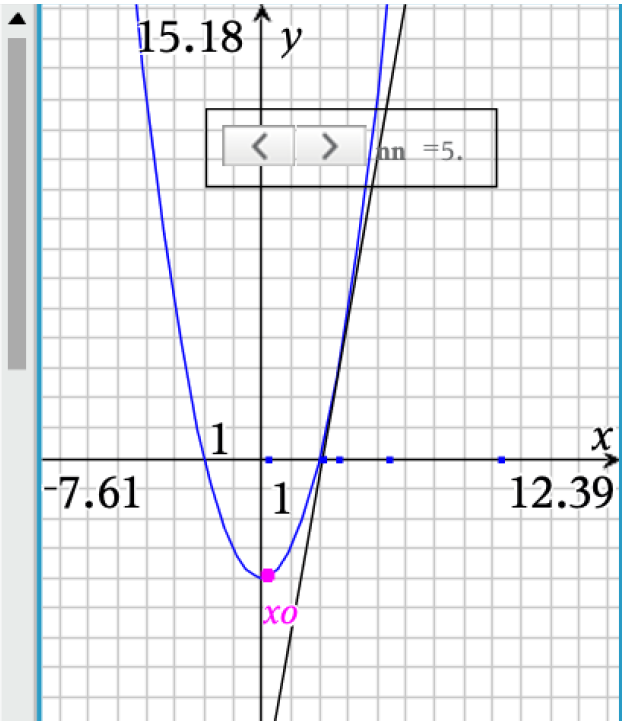
(numerische) Ableitung

$fs(x):=\frac{f(x+h)-f(x)}{h}|h=\frac{1}{100} \rightarrow Fertig$

Newton−Verfahren

$l1:=seqGen\left(x(n-1)-\frac{f(x(n-1))}{fs(x(n-1))},n,x,\{1,nn\},\{x0\}\right)$

$\rightarrow \{0.24,8.3,4.39,2.65,2.08\}$



Ergänzung: Verwendung der Ableitung anstelle des Differenzenquotienten.

Dazu wird die Funktion $f2(x)$ als Ableitung der Funktion $f(x)$ definiert	Ableitung $fs2(x):=\frac{d}{dx}(f(x)) \rightarrow Fertig$
Im Befehl seqGen() wird folgende Änderung vorgenommen	$\ldots \frac{f(x(n-1))}{fs2(k)\|k=x(n-1)} \ldots$

2. Zwei Basicprogramme

Mit der Programmiersprache Basic lässt sich das Verfahren relativ einfach implementieren.

Beispiel 1: Die zu untersuchende Funktion f muss dabei schon definiert sein. Als Parameter werden dann in die Funktion newton1(s,e) nur noch der Startwert s und die gewünschte Genauigkeit e eingegeben. Solange der absolute Betrag des Funktionswertes größer als die gewünschte Genauigkeit ist, wird die While-Schleife durchlaufen.	Define **newton1**(s,e)= Func Local a,fs,x $a:=s$ $fs(x):=\frac{d}{dx}(f(x))$ While $\lvert f(a)\rvert>e$ $a:=a-\frac{f(a)}{fs(a)}$ EndWhile Return a EndFunc
Zum Vergleich der gefundenen Lösung kann man z. B. den Befehl nSolve nutzen.	$f(x):=x^3-3$ *Fertig* $newton1(1,0.1)$ 1.44281 $newton1(1,0.01)$ 1.44281 $newton1(2,0.01)$ 1.44235 $newton1(20,0.01)$ 1.44231 nSolve$(f(x)=0,x=2)$ 1.44225
Wie schon erwähnt, versagt das Verfahren für „ungünstige Startwerte“. (Hinweis: Dies kann man mit einer entsprechenden Abfrage im Basicprogramm z. T. abfangen).	$f(x):=x^3-3$ *Fertig* $newton1(0,0.01)$ "Fehler: Eine bedingte Anweisung hat RICHTIG oder FALSCH nicht geklärt"

Beispiel 2 (Autor Josef Böhm) Im Unterschied zu Beispiel 1 wird der Funktionsterm der Funktion f als auch die unabhängige Variable x (hier als Parameter v) erst zur Laufzeit des Programms eingegeben. Ansonsten entspricht die Ausführung dem Beispiel 1.	Define **newton2**(f,v,s,e)= Func Local fs $fs:=\frac{d}{dv}(f)$ While $\lvert f\rvert>e\lvert v=s$ $s:=s-\frac{f}{fs}\lvert v=s$ EndWhile EndFunc
	$newton2(x^3-3,x,2,0.01)$ 1.44235

3. Varianten mit Python

Im Beispiel 1 soll ein Programm vorgestellt werden, mit dem das Newton-Verfahren ausschließlich in der Python-Umgebung des TI Nspire realisiert wird.

Beispiel 1:
Um alle bekannten Funktionen des Mathematikunterrichts zu nutzen, wird die Bibliothek math eingebunden.
Der Funktionsterm wird als Zeichenkette übergeben und zu einer Python-Funktion evaluiert. Dazu muss das Argument x als globale Variable ausgewiesen sein.
Die Ableitung der Funktion wird in Anlehnung zur Definition des Differentialquotienten als Differenzenquotient bei einer sehr kleinen Intervallbreite definiert.

```
from math import *

def fkt(string, val):
    global x
    x = val
    return eval(string)

def abl(string, x):
    h = 1e-10
    return (fkt(string, x + h) - fkt(string, x)) / h
```

Hier erfolgt die rekursive Implementierung des Newton-Verfahrens:
Falls der Rekursionszähler herabgezählt wurde, gib False zurück.
Anderenfalls prüfe, ob der Funktionswert an der Stelle x in einer hinreichend kleinen Epsilon-Umgebung liegt.
Gib in diesem Falle x zurück.
Anderenfalls führe das Newton-Verfahren für den nächsten Näherungswert von x bei vermindertem Rekursionszähler durch.

```
def newton(string, x, max):
    eps = 1e-08
    if max < 0:
        return False
    elif abs(fkt(string, x)) < eps:
        return x
    else:
        x = x - fkt(string, x) / abl(string, x)
        return newton(string, x, max - 1)
```

Nach der Eingabe des Funktionsterms und des Startwertes wird die Funktion newton() mit der maximalen Rekursionstiefe 25 aufgerufen.
Falls eine reelle Nullstelle ermittelt werden konnte, wird diese ausgegeben.
Anderenfalls erfolgt der Hinweis, dass keine Nullstelle ermittelt werden konnte.

```
term = input("f(x) = ")
x0 = float(input("x = "))
nst = newton(term, x0, 25)
if nst:
    print("Nullstelle: x0 =", nst)
else:
    print("Keine Nullstelle gefunden!")
```

Programmtest mit der Funktion $f(x) = x^2 - 2$. Der Startwert ist 5.
Das ermittelte Ergebnis stimmt sehr gut mit der Berechnung von $|x_0| = \sqrt{2}$ überein.

```
>>>#Running newton3.py
>>>from newton3 import *
f(x) = x**2 - 2
x = 5
Nullstelle: x0 = 1.414213562373089
>>>print(sqrt(2))
1.414213562373095
```

Das Programm terminiert, wenn keine reelle Nullstelle vorhanden ist.

```
>>>#Running newton3.py
>>>from newton3 import *
f(x) = x**2 + 2
x = 5
Keine Nullstelle gefunden!
```

In der Micropython-Umgebung sind leistungsstarke "Austauschfunktionen" verfügbar, mit denen ein Datenaustausch mit der TI-Basic-Umgebung des TI Nspires möglich ist.
Damit können, wie in den Beispielen 2 und 3, die TI-Basic-Funktionen in Python genutzt werden.

Beispiel 2: Die zu untersuchende Funktion sowie der Startwert werden in den Notes definiert.	Funktion: $\mathbf{f}(x):=x^2-2$ ▸ *Fertig* Startwert x0: $\mathbf{x0}:=5$ ▸ 5

Die Übernahme dieser Definitionen erfolgt mit den "Austauschfunktionen" recall_value() und eval_function().	`from ti_system import *`
	
	`x0 = recall_value("x0")`
	
	`def f(x):`
	`  return eval_function("f",x)`

Die Implementationen der ersten Ableitung sowie des Newton-Verfahrens vereinfachen sich entsprechend.
 	
Hier wird der Algorithmus iterativ implementiert. In der For-Schleife werden alle ermittelten Näherungswerte für die Nullstelle in der Liste ls gesammelt.
Die For-Schleife wird vorzeitig mit der Rückgabe der Nullstelle verlassen, falls diese existiert.
Wird die Schleife vollständig durchlaufen, existiert vermutlich keine reelle Nullstelle.
	`def fs(x):`
	`  h = 1e-10`
	`  return (f(x + h) - f(x)) / h`
	
	`def newton(x, ls):`
	`  eps = 1e-08`
	`  max = 100`
	`  for i  in range(max):`
	`  if abs(f(x)) < eps:`
	`  return x`
	`  else:`
	`  ls.append(x)`
	`  x = x - f(x) / fs(x)`
	`  return False`

Nach dem Funktionsaufruf wird die Liste mit den Näherungswerten der Nullstelle ausgegeben.
Falls die Nullstelle existiert, wird ihre Ausgabe auf 4 Nachkommastellen formatiert.
	`erg=[]`
	`nst = newton(x0, erg)`
	`print(erg)`
	`if nst:`
	`  print("Nullstelle: x0 = %1.4f" % nst)`
	`else:`
	`  print("Keine Nullstelle gefunden!")`

Alle Ergebnisse werden sofort nach dem Programmstart ausgegeben.	`>>>#Running newton4.py`
	`>>>from newton4 import *`
	`[5, 2.700000190302838, 1.720370451424432, 1.441395350033365, 1.414443768294422, 1.414213671592264]`
	`Nullstelle: x0 = 1.4142`

Beispiel 3: Mit der gleichen Idee wird nun auch die erste Ableitung als CAS-Funktion definiert. Damit kann das Newton-Verfahren mit einem "echten" Differentialquotienten implementiert werden.	Funktion: $\mathbf{f}(x):=x^2-2$ ▸ *Fertig* $\mathbf{fs}(x):=\frac{d}{dx}(\mathbf{f}(x))$ ▸ *Fertig* Startwert x0: $\mathbf{x0}:=5$ ▸ 5

Entsprechend dieser Definition vereinfacht sich die Python-Implementation der ersten Ableitung.
Alle weiteren Definitionen entsprechen dem Beispiel 2.
	`from ti_system import *`
	
	`x0 = recall_value("x0")`
	
	`def f(x):`
	`  return eval_function("f",x)`
	
	`def fs(x):`
	`  return eval_function("fs",x)`

In der Python-Umgebung des TI Nspires stehen leistungsfähige Grafikbibliotheken bereit. Damit kann das Newton-Verfahren mit etwas Aufwand auch visualisiert werden.

Beispiel 4: In der Applikation Notes werden zusätzlich Angaben zum Koordinatensystem ergänzt.	Funktion: $\mathbf{f}(x):=3\cdot\sin(x-1)$ ▸ *Fertig* $\mathbf{fs}(x):=\frac{d}{dx}(\mathbf{f}(x))$ ▸ *Fertig* Startwert x0: $\mathbf{x0}:=2$ ▸ 2 Koordinatensystem: $\mathbf{x\_min}:=-10$ ▸ -10 $\mathbf{x\_max}:=10$ ▸ 10 $\mathbf{y\_min}:=-5$ ▸ -5 $\mathbf{y\_max}:=5$ ▸ 5

Zur grafischen Darstellung müssen weitere Python-Bibliotheken geladen werden.
Neben der Übernahme der Daten zur Darstellung des Koordinatensystems erfolgt die Definition einer allgemeinen linearen Funktion, mit der die Tangenten zum Newton-Verfahren visualisiert werden.

```
from ti_system import *
from ti_draw import *
from math import pow, log10
import ti_plotlib as plt

def f(x):
    return eval_function("f",x)

def fs(x):
    return eval_function("fs",x)

def ft(x, m, n):
    return m * x + n

x0 = recall_value("x0")
x_min = recall_value("x_min")
x_max = recall_value("x_max")
y_min = recall_value("y_min")
y_max = recall_value("y_max")
```

Im iterativen Algorithmus des Newton-Verfahrens werden die Parameter der Tangentengleichungen ermittelt und in entsprechenden Listen gespeichert.
Für den Anstieg m sowie das Absolutglied n der Tangenten gelten:

$$m = f'(x)$$

Mit dem Näherungswert x für eine Nullstelle folgt:

$$0 = m \cdot x + n$$

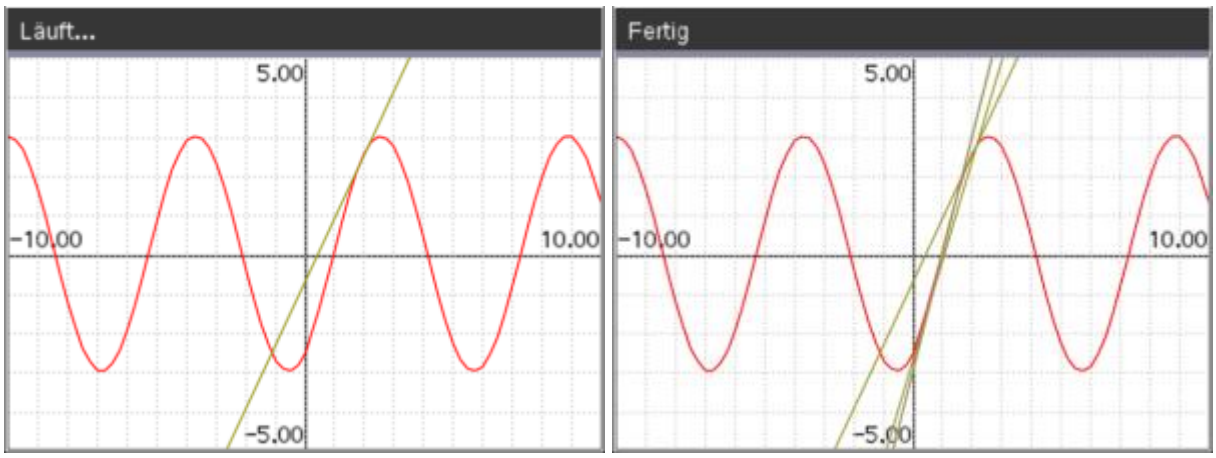
$$n = -m \cdot x$$

```
def newton(x, ls_x, ls_m, ls_n):
    eps = 1e-10
    max = 100
    for i  in range(max):
        ls_x.append(x)
        if abs(f(x)) < eps:
            return x
        else:
            m = fs(x)
            x = x - f(x) / m
            n = -m * x
            ls_m.append(m); ls_n.append(n)
    return False
```

Mit dieser Definition wird die Schrittweite des Rasters im Koordinatensystem an den darzustellenden Definitions- bzw. Wertebereich der Funktion angepasst.

```
def dist(min, max):
    k = log10(abs(max - min))
    d = pow(10, floor(k - 0.5))
    if d < 1: d = 1
    return floor(d)
```

Eine kurze Erklärungen zur Orientierung:
Festlegung der Weite des Rasters der x- und y-Achse. Festlegung des Koordinaten-systems mit Raster.
 	
Kleinschrittige Darstellung des Funktionsgraphen.
 	
Auslesen der Parameter m und n aus den entsprechen-den Listen und Darstellung der Tangenten.
Programmunterbrechung bis zum Betätigen einer beliebi-gen Taste
	def grafik(ls_m, ls_n):
	  use_buffer()
	  width = 80
	  xsc = dist(x_min, x_max)
	  ysc = dist(y_min, y_max)
	  plt.window(x_min, x_max, y_min, y_max)
	  plt.axes("on")
	  plt.grid(xsc,ysc,"dotted")
	  dx = (plt.xmax − plt.xmin) / width
	  #-- Funktion --
	  x0 = plt.xmin
	  x = x0 + dx
	  set_color(255, 0, 0)
	  while x <= plt.xmax + dx:
	  xp0, yp0 = plt.XY(x0, f(x0))
	  xp, yp = plt.XY(x, f(x))
	  draw_line(xp0, yp0, xp, yp)
	  x0 = x
	  x += dx
	  paint_buffer()
	  #-- Tangenten --
	  for i in range(len(ls_m)):
	  set_color(150, 150, (25 * (i + 1)) % 255)
	  xp0, yp0 = plt.XY(plt.xmin, ft(plt.xmin, ls_m[i], ls_n[i]))
	  xp, yp = plt.XY(plt.xmax, ft(plt.xmax, ls_m[i], ls_n[i]))
	  draw_line(xp0, yp0, xp, yp)
	  get_key(1)
	  paint_buffer()



Autoren:
Hubert Langlotz
Sebastian Rauh
Veit Berger