Das babylonisch-sumerische Verfahren zum Wurzelziehen

– auch als *Heron-Verfahren* bezeichnet –
und seine Deutung als Spezialfall des *Newton-Verfahrens*

Prof. Dr. J. Ziegenbalg
Institut für Mathematik und Informatik
Pädagogische Hochschule Karlsruhe

electronic mail: ziegenbalg@ph-karlsruhe.de
homepage: http://www.ph-karlsruhe.de/wp/ziegenbalg/

Herrn Franz Fritsche, Duisburg, bin ich für eine Reihe von Hinweisen zu diesem Text sehr zu Dank verpflichtet.

■ 1. Literaturhinweise

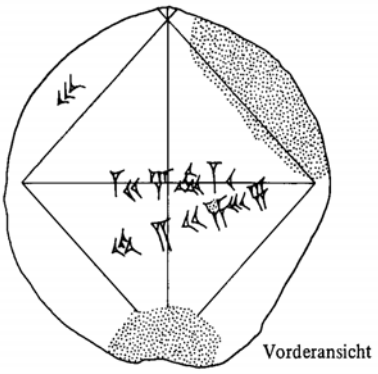
H. Heuser, Lehrbuch der Analysis, Teil 1, B.G. Teubner Verlag, Stuttgart 1990, Kapitel IX, Abschnitt 70, Seite 406 ff

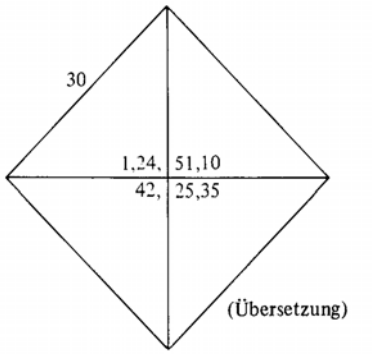
B. L. van der Waerden, Erwachende Wissenschaft, Basel 1966, 71-72

J. Ziegenbalg, O. Ziegenbalg, B. Ziegenbalg, Algorithmen von Hammurapi bis Gödel, Heidelberg 1996, 54-59
2., erweiterte Auflage: Harri Deutsch Verlag, Frankfurt am Main 2007

■ 2. Einführung

Im Zusammenhang mit Problemen der Flächenmessung, insbesondere der Landvermessung, entwickelten die Babylonier Verfahren zur Lösung quadratischer Gleichungen, die uns aus der Epoche von Hammurapi (etwa 1700 v. Chr.) auf Keilschriften überliefert sind (vgl. Abbildung unten). Eine Vorstufe zur Lösung allgemeiner quadratischer Gleichungen ist die Bestimmung von Quadratwurzeln. Das Verfahren wurde später etwa um 100 n.Chr. von den Griechen (Heron von Alexandria) aufgegriffen und beschrieben; es wird heute oft auch als *Heron-Verfahren* bezeichnet.





Heron von Alexandria lebte um 100 n.Chr. Seine Werke stellen eine Art Enzyklopädie in angewandter Geometrie, Optik und Mechanik dar. Sie haben oft den Charakter einer Formelsammlung. Viele der ihm zugeschriebenen Formeln und Verfahren waren schon vorher bekannt; so soll die *Heronsche Formel* für den Flächeninhalt von Dreiecken von Archimedes stammen; das *Heron-Verfahren* zum Wurzelziehen wurde schon viele Jahrhunderte vorher von den Babyloniern praktiziert.

Für die Technik ist Heron von Interesse durch seine Beschreibung früher Maschinen: Er beschrieb die Konstruktion von Vermessungsinstrumenten, Geschützen, einfachen Maschinen wie Hebel, schiefe Ebene, Keil, Flaschenzug, von Winden, Wasseruhren, Gewölben und von durch Dampfdruck angetriebenen Automaten. Durch Ausnutzung des Dampfdrucks oder von Luftdruckunterschieden gelangen ihm z.B. mechanische Konstruktionen zum automatischen Öffnen von Tempeltüren.

■ 3. Problemstellung, Interpretation und heuristische Grundidee

Das Problem: Gegeben sei eine Zahl a. Gesucht ist eine Zahl b mit der Eigenschaft $b * b = a$. In unserer heutigen Terminologie ist b die (Quadrat-) Wurzel von a; im Zeichen: $b = \sqrt{a}$.

Geometrische Interpretation: Man ermittle die Seitenlänge *b* eines Quadrats vom Flächeninhalt *a*. Oder: Man konstruiere ein Quadrat mit dem Flächeninhalt a.

Die Grundidee des babylonischen Verfahrens beruht auf dieser geometrischen Veranschaulichung und der folgenden Anwendung des *"Prinzips der Bescheidenheit"*: Wenn man das zum Flächeninhalt a gehörende Quadrat nicht sofort bekommen kann, begnüge man sich mit etwas weniger, etwa mit einem Rechteck des Flächeninhalts *a*. Ein solches ist leicht zu konstruieren: Man wähle z.B. eine Seitenlänge gleich a und die andere Seitenlänge gleich 1 (also eine Längeneinheit). Das Störende daran ist, dass die so gewählten Seitenlängen im allgemeinen verschieden sind, dass das Rechteck also kein Quadrat ist. Man versucht nun schrittweise, ausgehend von dem Ausgangsrechteck immer "quadrat-ähnlichere" Rechtecke zu konstruieren.

Dazu geht man folgendermaßen vor: Man wählt die eine Seite des neuen Rechtecks als das arithmetische Mittel der Seiten des Ausgangsrechtecks und passt die andere Seite so an, dass sich der Flächeninhalt a nicht verändert. Sind x_0 $(x_0 = a)$ und y_0 $(y_0 = 1)$ die Seiten des Ausgangsrechtecks, so lauten die Seitenlängen des neuen Rechtecks:

$$x_1 = \frac{x_0+y_0}{2} \text{ und } \quad y_1 = \frac{a}{x_1} \qquad \text{(Heron\_1)}$$

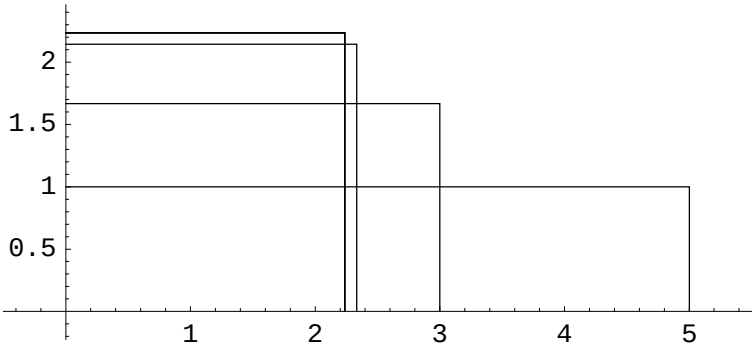
Entsprechend fährt man mit dem neuen Rechteck an Stelle des Ausgangsrechtecks fort und erhält so die allgemeine Iterationsvorschrift des Heron-Verfahrens:

$$x_{n+1} = \frac{x_n+y_n}{2} \text{ und } \quad y_{n+1} = \frac{a}{x_{n+1}} \qquad \text{(Heron\_2)}$$

Diese "gekoppelte" Iteration wird oft nach einer naheliegenden Umformung in der folgenden Form geschrieben, in der nur noch die (indizierte) Variable x vorkommt:

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right) \qquad \text{(Heron\_3)}$$

Die folgende Abbildung veranschaulicht diese Vorgehensweise (und die Effizienz des Verfahrens) am Beispiel $\sqrt{5}$. Bereits nach dem dritten Iterationsschritt sind (bei diesen Ausgangswerten) die iterativ produzierten Rechtecke optisch praktisch nicht mehr zu unterscheiden.



Die graphische Deutung des Verfahrens legt die Vermutung der Konvergenz nahe - sie ist natürlich noch kein definitiver Beweis.

■ 4. Der Algorithmus

Umgangssprachlich lässt sich das Verfahren wie folgt beschreiben; dabei seien in den Klammern des Typs (* und *), wie in vielen Programmiersprachen üblich, *Kommentare* eingeschlossen. (Dies sind Erläuterungen ohne Auswirkung auf den Ablauf des Algorithmus.) In der folgenden Fassung kommt es nicht primär auf die Knappheit der Formulierung an, sondern auf ihre Verständlichkeit. Wenn man unbedingt wollte, könnte man noch auf einige Hilfsvariablen verzichten.

```
Heron(a)
  Hilfsvariable:  x, y, xneu, yneu;
                 (* Vereinbarung von lokalen Hilfsvariablen *)
  x:=a;  y:=1;   (*  :=  Wertzuweisung; hier: Anfangswerte  *)
  Solange |x^2 - a| > 0.000001 (* solange nicht genau genug *)
    tue folgendes:
    [ xneu := (x+y)/2;   (* Die eckigen Klammern    *)
      yneu := a/xneu;    (* legen den Gueltigkeits- *)
      x := xneu;         (* bereich der Solange-    *)
      y := yneu ];       (* Kontrollstruktur  fest. *)
  Rueckgabe x            (* x wird als Funktions-   *)
                         (* wert zurueckgegeben.    *)
Ende.
```

■ Umsetzung des Algorithmus in *Mathematica*

In[1]:=

```
Heron[a_] :=
 Module[{x = a,  y = 1.0,  xneu,  yneu},
  While[Abs[x^2 - a] > 0.000001,
   xneu = (x + y)/2;  yneu = a/xneu;
   x = xneu;  y = yneu];
  Return[x]]
```

Einige Auswertungsbeispiele:

In[2]:=

```
Heron[2]
```

Out[2]=

```
1.41421
```

In[3]:=

```
Heron[12345]
```

Out[3]=

```
111.108
```

In[4]:=

```
Heron[169]
```

Out[4]=

```
13.
```

In[5]:=
```
a = 123456;
Heron[a]
```
Out[6]=
```
351.363
```
In[7]:=
```
Heron[a]^2
```
Out[7]=
```
123456.
```

Die interne Rechengenauigkeit von *Mathematica* ist größer als die Genauigkeit in der Standard-Darstellung am Bildschirm. Mit dem Kommando N (für "numerisch") kann man diese interne höhere Genauigkeit sichtbar machen:

In[8]:=
```
Sqrt[2]
Sqrt[2] // N
N[Sqrt[2], 100]
```
Out[8]=

$\sqrt{2}$

Out[9]=
```
1.41421
```
Out[10]=
```
1.41421356237309504880168872420969807856967187537694807317667973799
 07324784621070388503875343276415 73
```

Nähere Erläuterungen zum Thema numerische Genauigkeit finden sich weiter unten in diesem Notebook.

■ 5. Verallgemeinerung: Die *k*-te Wurzel

Das Verfahren (und seine Heuristik) lässt sich ohne weiteres auf höhere Dimensionen übertragen. Man erhält so ein Verfahren zur Ermittlung von $\sqrt[k]{a}$.

Wir führen zunächst eine Vorbetrachtung für den Fall der 5. Wurzel durch. Dabei wird (in Analogie zur Vorgehensweise bei der Quadratwurzel) $\sqrt[5]{a}$ als die Seitenlänge eines 5-dimensionalen Würfels angesehen, der durch eine Folge von 5-dimensionalen Quadern angenähert wird. Die Seiten der 5-dimensionalen Quader seien mit x_1, x_2, x_3, x_4, y bezeichnet. Entsprechend dem 2-dimensionalen Fall führen wir die folgende Mittelbildung durch: 4 der neuen Seiten (im folgenden Algorithmus die Seiten x_1, x_2, x_3, x_4) erhalten als neue Seitenlänge das

arithmetische Mittel der alten (fünf) Seiten; eine neue Seite (im folgenden Algorithmus die Seite *y*) wird dann so angepasst, dass das Volumen "stimmt". (Etwas genauer könnte man also formulieren: Die 5-dimensionalen Quader sind 5-dimensionale Säulen mit einer aus einem 4-dimensionalen Würfel bestehenden "Grundfläche" und der Höhe *y*.)

In[11]:=

```
Heron5[a_] :=
 Module[{x1 = x2 = x3 = x4 = a,  y,  xneu,  yneu},
  y = N[a / (x1 * x2 * x3 * x4)];
  While[Abs[x1^5 - a] > 0.000001,

   xneu = (x1 + x2 + x3 + x4 + y) / 5;

   x1 = x2 = x3 = x4 = xneu;

   yneu = a / (x1 * x2 * x3 * x4);

   y = yneu];
  Return[x1]]
```

Probe:

In[12]:=

```
Heron5[234]
```

Out[12]=

```
2.97744
```

Probe (das Prozentzeichen % bewirkt, dass die letzte Ausgabe als Eingabe verwendet wird):

In[13]:=

```
%^5
```

Out[13]=

```
234.
```

Zum Vergleich: Ermittlung des Ergebnisses mit der "eingebauten" Wurzel-Funktion $\sqrt{\square}$:

In[14]:=

```
N[Sqrt[234, 5]]
```

Out[14]=

```
2.97744
```

Grundsätzlich ist festzuhalten, dass die Algorithmen bzw. Programme in diesem Notebook nicht unbedingt so geschrieben sind, dass Variablennamen "mit aller Gewalt" eingespart werden. Vorrang hat die Transparenz des Algorithmus. So hätte man im ursprünglichen **Heron**-Algorithmus nicht unbedingt die beiden Variablen **xneu** und **yneu** gebraucht. Sie wurden dennoch eingeführt, da so das Programm besser verständlich ist.

Aber: Ein Blick auf den Algorithmus **Heron5** zeigt, dass die Einführung der Variablen **x2**, **x3** und **x4** etwas sehr verschwenderisch ist, denn sie erhalten stets nur dieselben Werte wie die Variable **x1**. Die folgende Version geht etwas sparsamer mit den Variablen um - und gibt zugleich einen Hinweis darauf, wie ein Programm für die *k*-te Wurzel gestaltet werden kann.

In[15]:=

```
Heron5neu[a_] :=
 Module[{x = a, y, xneu, yneu},
  y = N[a/x^4];
  While[Abs[x^5 - a] > 0.000001,
   xneu = (4 * x + y)/5;
   yneu = a/xneu^4;
   x = xneu; y = yneu];
  Return[x]]
```

In[16]:=

```
Heron5neu[123]
```

Out[16]=

```
2.61807
```

In[17]:=

```
Heron5[123]
```

Out[17]=

```
2.61807
```

In[18]:=
```
Sqrt[123, 5] // N
```
Out[18]=
```
2.61807
```

Zur Berechnung der *k*-ten Wurzel (mit variablem *k*) wird wird die Strategie der Mittelbildung im folgenden Algorithmus auf einen Würfel der Dimension *k* verallgemeinert. Man kann sich den jeweiligen *k*-dimensionalen Würfel dabei stets, etwas genauer gesprochen, als eine Säule über einem (*k*-1)-dimensionalen Würfel vorstellen.

In[19]:=
```
Heron[k_, a_] :=
 Module[{x = a, y, xneu, yneu},
  y = N[a/x^(k-1)];
  While[Abs[x^k - a] > 0.000001,
   xneu = ((k - 1) * x + y)/k;
   yneu = a/xneu^(k - 1);                (* Heron_k *)
   x = xneu;  y = yneu];
  Return[x]]
```
In[20]:=
```
Heron[13, 987654]
```
Out[20]=
```
2.8915
```

Probe:

In[21]:=
```
%^13
```
Out[21]=
```
987654.
```
In[22]:=
```
N[Sqrt[987654, 13]]
```
Out[22]=
```
2.8915
```

Bemerkung 1: Vor allem im Hinblick auf das weiter unten zu diskutierende *Newton-Verfahren* sei der in der Zeile

(* Heron_k *) beschriebene Iterationsschritt noch in der mathematischen Standard-Notation dargestellt:

Aus $x_n = \frac{1}{k}\left((k-1)\cdot x_{n-1} + y_{n-1}\right)$ und $y_n = \frac{a}{x_n^{k-1}}$

ergibt sich die folgende "autonome" Darstellung:

$$x_n = \frac{1}{k}\left((k-1)\cdot x_{n-1} + \frac{a}{x_{n-1}^{k-1}}\right) \qquad \text{(* Heron-autonom *)}$$

Bemerkung 2: Unabhängig von der obigen geometrischen Motivation sei an dieser Stelle noch gezeigt: Wenn die Folge (x_n) aus (* Heron-autonom *) einen Grenzwert g $(g > 0)$ hat, dann ist dies $g = \sqrt[k]{a}$.

Beweis: Nach Voraussetzung ist $g = \lim_{n\to\infty} x_n$

Aus der Gleichung (* Heron-autonom *) folgt durch Anwendung der Grenzwertoperation auf beiden Seiten der Gleichung

$$\lim_{n\to\infty} x_n = \lim_{n\to\infty} \frac{1}{k}\left((k-1)\cdot x_{n-1} + \frac{a}{x_{n-1}^{k-1}}\right)$$

Wegen der (in diesem Fall gegebenen) Vertauschbarkeit der Grenzwert-Operation mit den arithmetischen Verknüpfungen gilt:

$$\lim_{n\to\infty} \frac{1}{k}\left((k-1)\cdot x_{n-1} + \frac{a}{x_{n-1}^{k-1}}\right) = \frac{1}{k}\left((k-1)\cdot \lim_{n\to\infty} x_{n-1} + \frac{a}{(\lim_{n\to\infty} x_{n-1})^{k-1}}\right)$$

Da die Folge (x_{n-1}) offensichtlich denselben Grenzwert hat wie die Folge (x_n), ist $\lim_{n\to\infty} x_{n-1} = g$; und somit gilt:

$$g = \frac{1}{k}\left((k-1)\cdot g + \frac{a}{g^{k-1}}\right).$$

Daraus folgt

$$k\cdot g = (k-1)\cdot g + \frac{a}{g^{k-1}} \quad ,$$

d.h.

$$g = \frac{a}{g^{k-1}}$$

und daraus ergibt sich schließlich

$$g^k = a \quad \text{bzw.} \quad g = \sqrt[k]{a}.$$

■ 6. Simulation der babylonischen Berechnung (einschließlich der Darstellung im 60-er System)

Im folgenden soll das auf der Keilschrift dargestellte konkrete Verfahren (vgl. B. L. van der Waerden, Erwachende Wissenschaft, Basel 1966, S. 71-72) durch eine halbautomatische, interaktive Simulation "nachgespielt" werden.

Dazu benötigen wir zunächst noch das folgende Hilfsprogramm `Basis`, das dazu dient, eine im Dezimalsystem gegebenen Zahl ins 60-er System umzuwandeln. Als "Ziffern" des 60-er Systems verwenden wir die 60 Symbole 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ... , 58, 59. Dabei ist zu beachten, dass die Symbole 10, 11, 12, ... , 58, 59 jeweils **eine einzige** Ziffer im 60-er System darstellen. (Um in diesem Zusammenhang keine Mehrdeutigkeiten aufkommen zu lassen, wird das Ergebnis der Funktion `Basis` als Liste von Ziffern aufbereitet, wobei die Ziffern durch Kommata getrennt sind.) Weiterhin ist zu beachten, dass das folgende Programm `Basis` mit der Ziffer 0 arbeitet, über die die Babylonier noch nicht verfügten.

In[23]:=
```
Basis[b_, n_] :=
 If[n == 0, {}, Append[Basis[b, Quotient[n, b]], Mod[n, b] ] ]
```

In[24]:=
```
Basis[60, 234]
```

Out[24]=
```
{3, 54}
```

Probe:

In[25]:=
```
3 * 60 + 54
```

Out[25]=
```
234
```

Auf der Basis eines Ausgangs-Rechtecks mit den Seiten

In[26]:=
```
a0 = 2;
b0 = 1;
```

ergeben sich durch Mittelbildung die folgenden weiteren Werte:

In[28]:=

$$a1 = \frac{a0 + b0}{2}$$

$$b1 = \frac{2}{a1}$$

Out[28]=

$$\frac{3}{2}$$

Out[29]=

$$\frac{4}{3}$$

Im 60-er System:

In[30]:=

```
Basis[60, a1]
```

Out[30]=

$$\left\{\frac{3}{2}\right\}$$

In[31]:=

```
Basis[60, b1]
```

Out[31]=

$$\left\{\frac{4}{3}\right\}$$

Dies bedeutet für a_1: $\frac{3}{2}$ Einheiten an der "Einerstelle", bzw. für b_1: $\frac{4}{3}$ Einheiten an der "Einerstelle". Das ist zwar korrekt, aber eigentlich hätte man gern eine "bruchfreie" Darstellung. Wie kommt man zu einer bruchfreien Darstellung?

Eine Analogie im Zehnersystem: Wir betrachten den Bruch $\frac{3}{2}$. Mit 10 multipliziert ergibt dies 15. Diese Multiplikation lässt sich nun durch "Kommaverschiebung" rückgängig machen: $\frac{3}{2} = 1,5$.

> *Eine Neben-Bemerkung*: Die Babylonier verfügten noch nicht über ein voll entwickeltes Stellenwertsystem und schrieben einfach die "Ziffern" nebeneinander auf; im Analogiefall hätten sie (ohne Dezimalkomma) geschrieben 1 5 . Dass dies Probleme im Hinblick auf die richtige "Wertigkeit" von Ziffern mit sich bringt, liegt auf der Hand. Dieses Problem wurde erst durch die Einführung der Ziffer 0 durch die Inder (ca. 600 n.Chr.) befriedigend gelöst.

Da die Babylonier mit einem 60-er System arbeiteten, mussten sie den Bruch $\frac{3}{2}$ (statt mit der Basis 10) mit der Basis 60 multiplizieren und erhielten nach der entsprechenden "Kommaverschiebung" (dargestellt in unserer Notation):

In[32]:=
```
Basis[60, a1 * 60]
```
Out[32]=
```
{1, 30}
```

Dies ist zu lesen als: 1 Ganzes plus 30 Sechzigstel.

In[33]:=
```
Basis[60, b1 * 60]
```
Out[33]=
```
{1, 20}
```

Also: $\frac{4}{3}$ = 1 Ganzes plus 20 Sechzigstel.

Weitere Näherungswerte:

In[34]:=
```
a2 = (a1 + b1)/2
```
Out[34]=
$$\frac{17}{12}$$

In[35]:=
```
b2 = 2/a2
```
Out[35]=
$$\frac{24}{17}$$

In[36]:=
```
Basis[60, a2 * 60^2]
Basis[60, b2 * 60^2]
```
Out[36]=
```
{1, 25, 0}
```
Out[37]=
$$\left\{1, 24, \frac{720}{17}\right\}$$

In[38]:=

$$a3 = \frac{a2 + b2}{2}$$

$$b3 = \frac{2}{a3}$$

Out[38]=

$$\frac{577}{408}$$

Out[39]=

$$\frac{816}{577}$$

In[40]:=

```
Basis[60, a3 * 60^3]
Basis[60, b3 * 60^3]
```

Out[40]=

$$\left\{1, 24, 51, \frac{180}{17}\right\}$$

Out[41]=

$$\left\{1, 24, 51, \frac{5580}{577}\right\}$$

In[42]:=

$$a4 = \frac{a3 + b3}{2}$$

$$b4 = \frac{2}{a4}$$

Out[42]=

$$\frac{665857}{470832}$$

Out[43]=

$$\frac{941664}{665857}$$

In[44]:=

```
Basis[60, a4 * 60^4]
Basis[60, b4 * 60^4]
```

Out[44]=

$$\left\{1, 24, 51, 10, \frac{76200}{9809}\right\}$$

Out[45]=

$$\left\{1, 24, 51, 10, \frac{5172600}{665857}\right\}$$

An dieser Stelle wurde das auf der Keilschrift dargestellte Verfahren beendet mit dem Ergebnis:

```
In[46]:=
a4babylonisch = 1 + 24 / 60 + 51 / 60^2 + 10 / 60^3;
```

Der Unterschied der beiden Rechtecksseiten betrug damit:

```
In[47]:=
N[a4 - b4]
Print["a4 - b4 = ",
 PaddedForm[ N[a4 - b4], 20,  ExponentFunction -> (Null &) ]]
```

```
Out[47]=
3.18972 × 10^-12
```

```
a4 - b4 =  0.00000000000318972364921191 1
```

Da die betrachtete Seite jedoch eine Seitenlänge von 30 Einheiten haben sollte (vgl. das Keilschrift-Bild), ist das Ergebnis noch mit 30 zu multiplizieren:

```
In[49]:=
Basis[60, a4babylonisch * 60^3]
Basis[60, 30 * a4babylonisch * 60^3]
```

```
Out[49]=
{1, 24, 51, 10}
```

```
Out[50]=
{42, 25, 35, 0}
```

Lässt man die letzte Null weg (die Null wurde erst etwa um 600 n.Chr. von den Indern "erfunden", in früheren Zahlensystemen schrieb man die Zahl 0 einfach nicht auf), so sind dies genau die Zahlen, die auf der Keilschrift (in der zweiten Zeile) eingetragen sind.

■ 7. Das Newton-Verfahren

[*Isaac Newton*, 1643-1727, englischer Mathematiker, Physiker und Naturforscher]

■ Einführung

Das Heron-Verfahren lässt sich als Spezialfall des wesentlich allgemeineren *Newton-Verfahrens* deuten (vgl. Abschnitt: "Das Heron-Verfahren als Spezialfall des Newton-Verfahrens" weiter unten).

Problemstellung: Gegeben sei eine hinreichend "gutartige" Funktion $x \to f(x)$; gesucht ist eine Nullstelle von f.

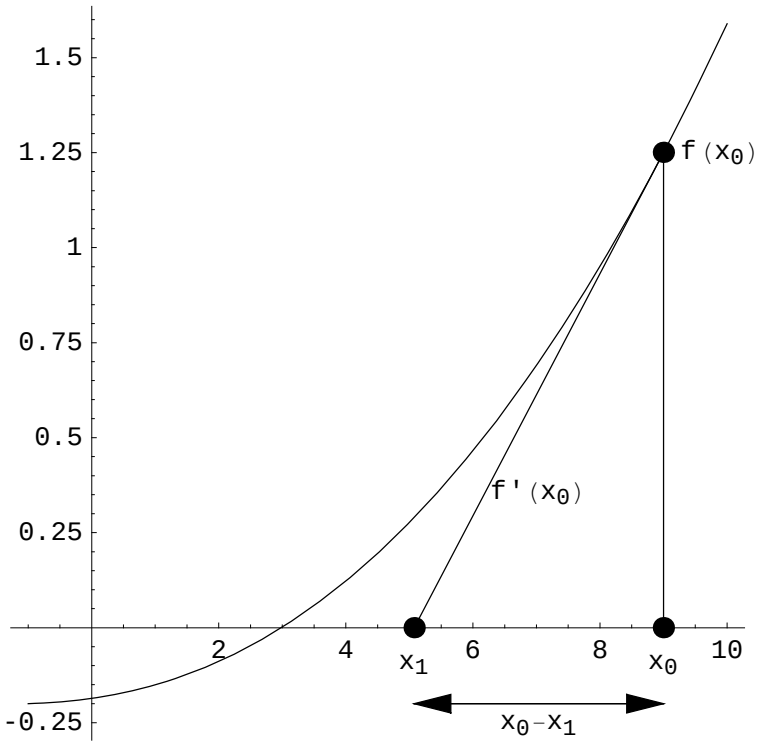
("Hinreichend gutartig" soll im Augenblick "differenzierbar" bedeuten; die genaueren Voraussetzungen werden weiter unten im Abschnitt "Analyse des Newton-Verfahrens" beschrieben.)

Nach dem Verfahren von Newton (bzw. Newton-Raphson; *Joseph Raphson*, engl. Mathematiker, 1648-1715) geht man folgendermaßen vor: Man such sich in dem Intervall, wo man an einer Nullstelle interessiert ist, einen ersten Näherungswert x_0 (z.B. durch eine grobe "nullte" Abschätzung). Basierend auf diesem Anfangswert setzt man nun ein Verfahren zur Bestimmung weiterer Näherungswerte $x_1, x_2, \ldots, x_n, x_{n+1}, \ldots$ durch iteratives Anwenden der folgenden Formel in Gang.

$$x_{n+1} := x_n - \frac{f(x_n)}{f'(x_n)}$$

Die Motivation zu dieser Iterationsgleichung kann man dem "Steigungsdreieck" in der folgenden Abbildung entnehmen:

$$f'(x_0) = \frac{f(x_0)}{x_0 - x_1}$$



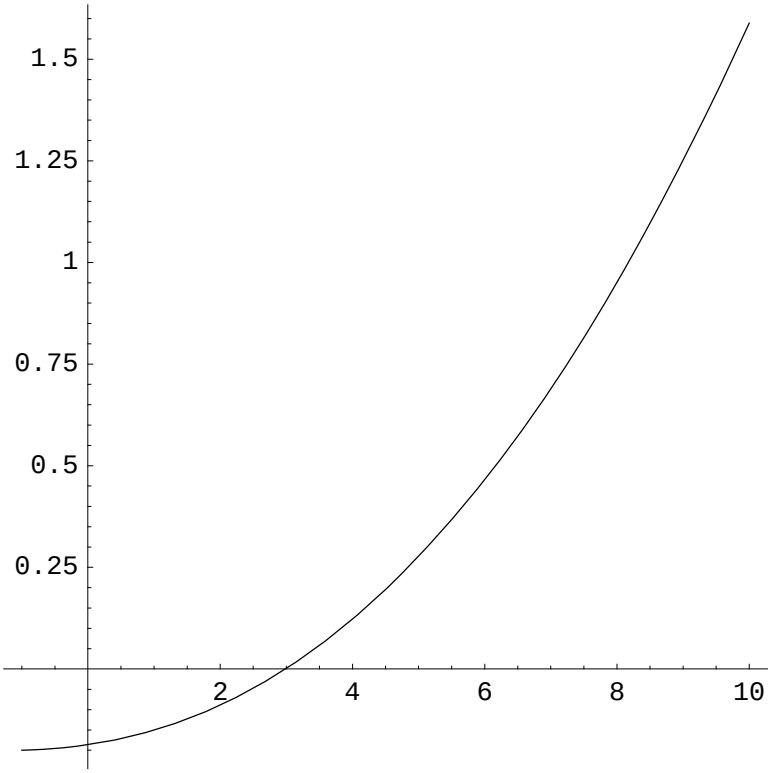
Das Newton-Verfahren sei nun in mehreren Schritten am Beispiel der Funktion $f(x) = 0.005 \cdot (x+2)^2 \cdot \mathrm{Log}(x+2) - 0.2$ veranschaulicht.

In[51]:=

```
f[x_] := 0.005 * (x + 2) ^ 2 * Log[(x + 2)] - 0.2;
```

In[52]:=

```
Plot[f[x], {x, -1, 10}, AspectRatio → 1]
```

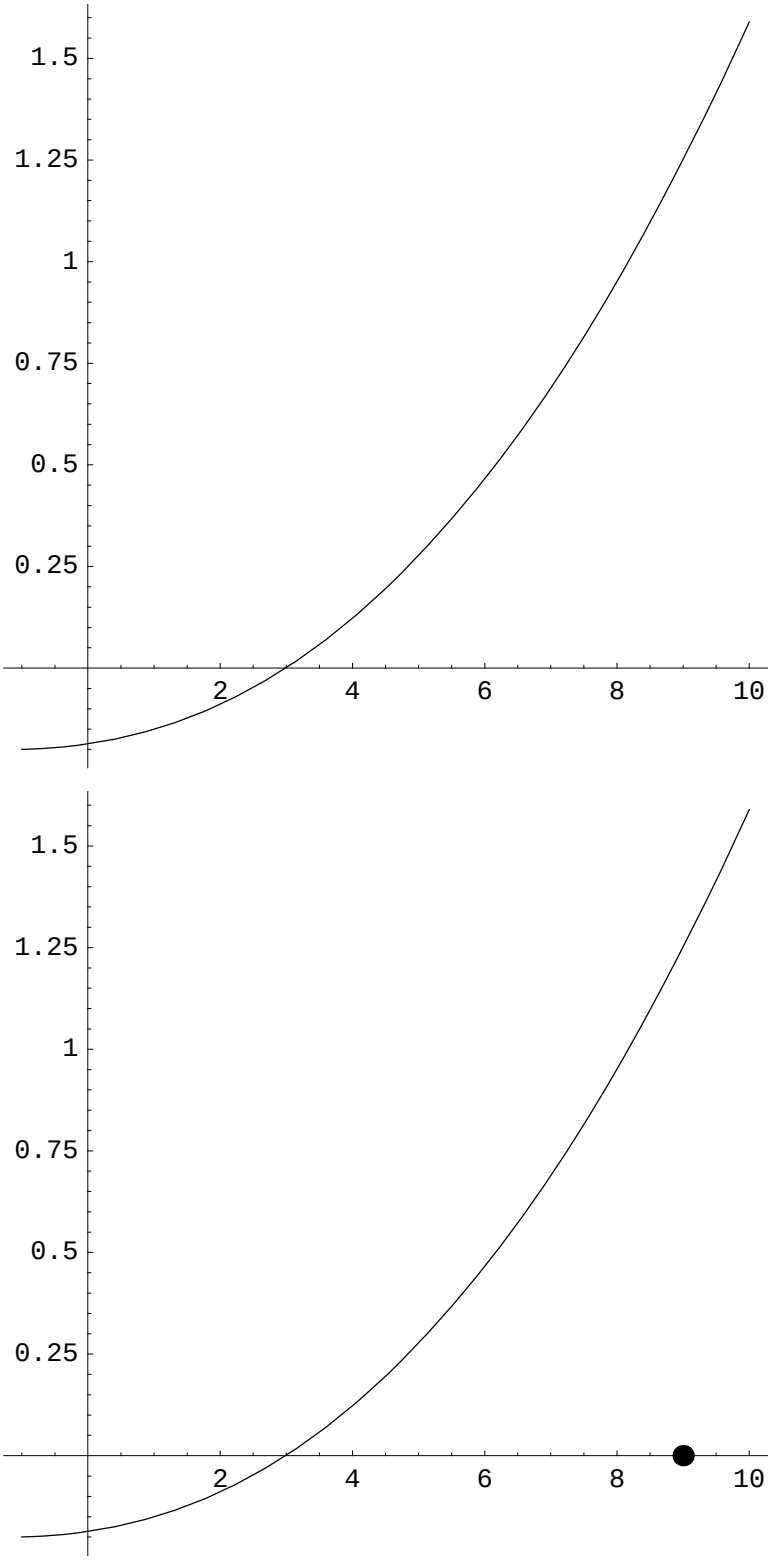


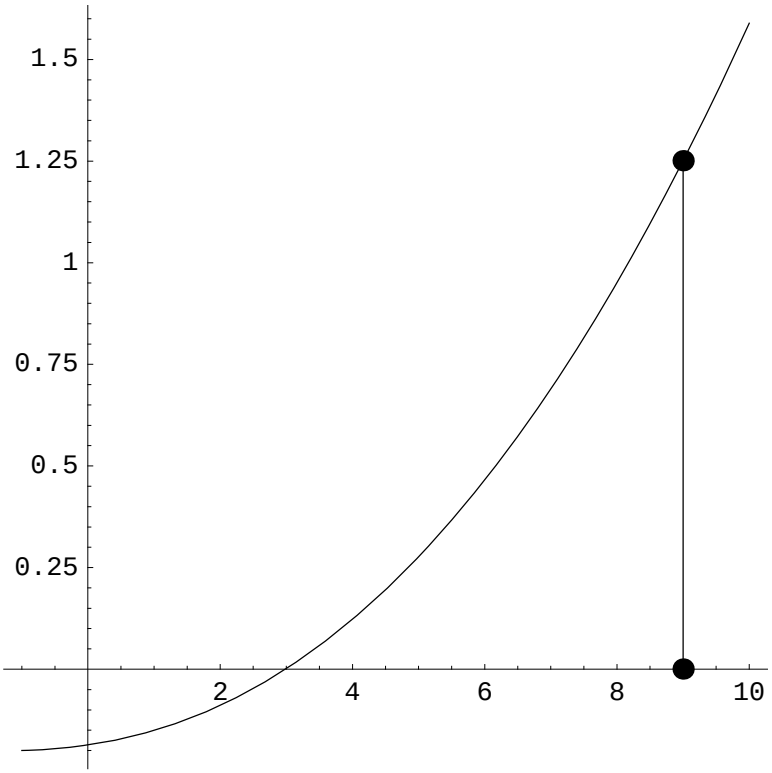
Out[52]=

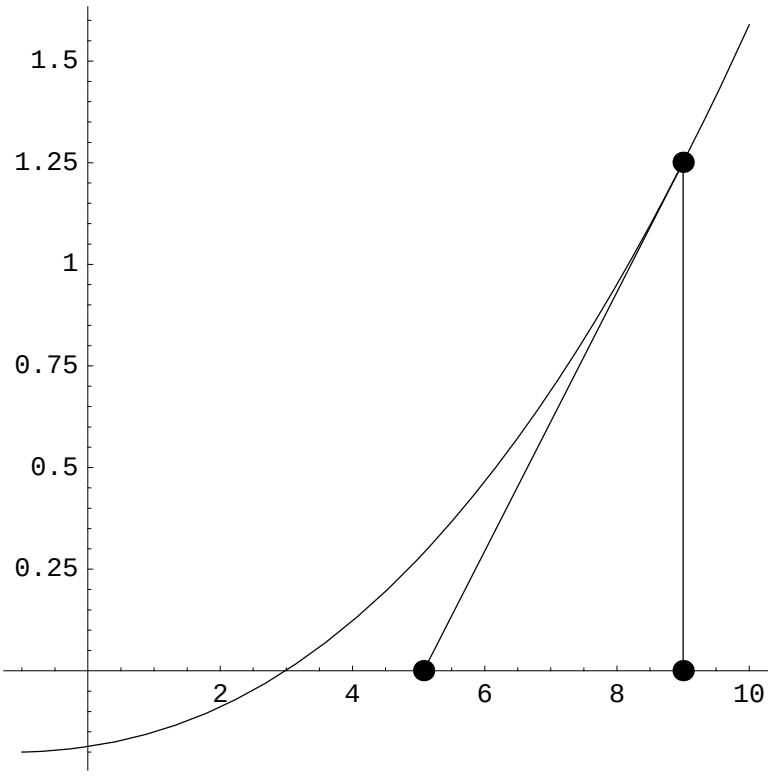
```
- Graphics -
```

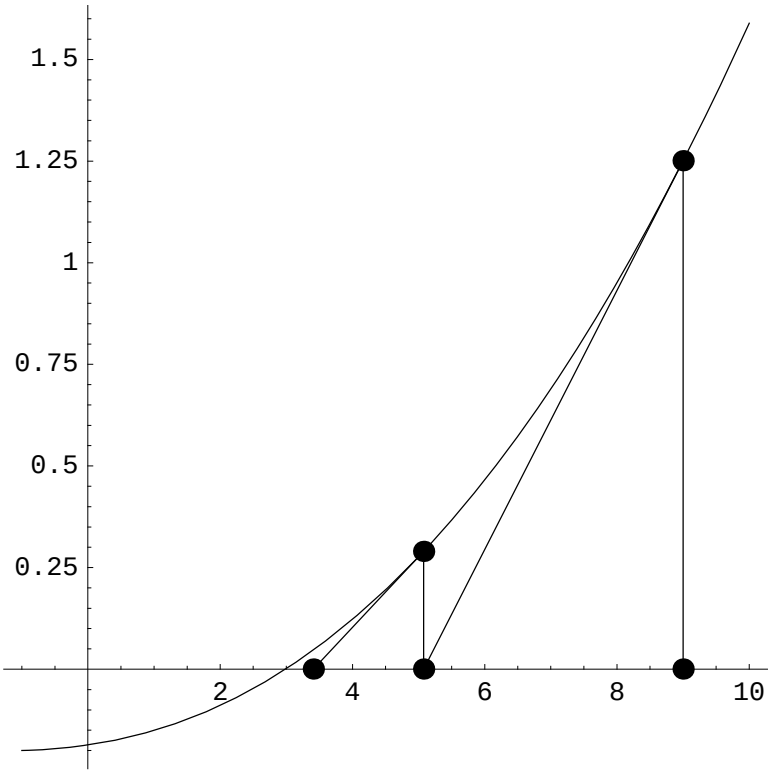
Gesucht sei die Nullstelle dieser Funktion im Intervall (0; 10). Nur der besseren Veranschaulichung halber wählen wir den relativ schlechten Anfangswert $x_0 = 9$. Danach zeichnen wir die Ordinate über x_0 und die Tangente im Punkt $(x_0, f(x_0))$. Sie schneidet die x-Achse im Punkt x_1, mit dem (an Stelle von x_0) das Verfahren entsprechend weiter läuft.

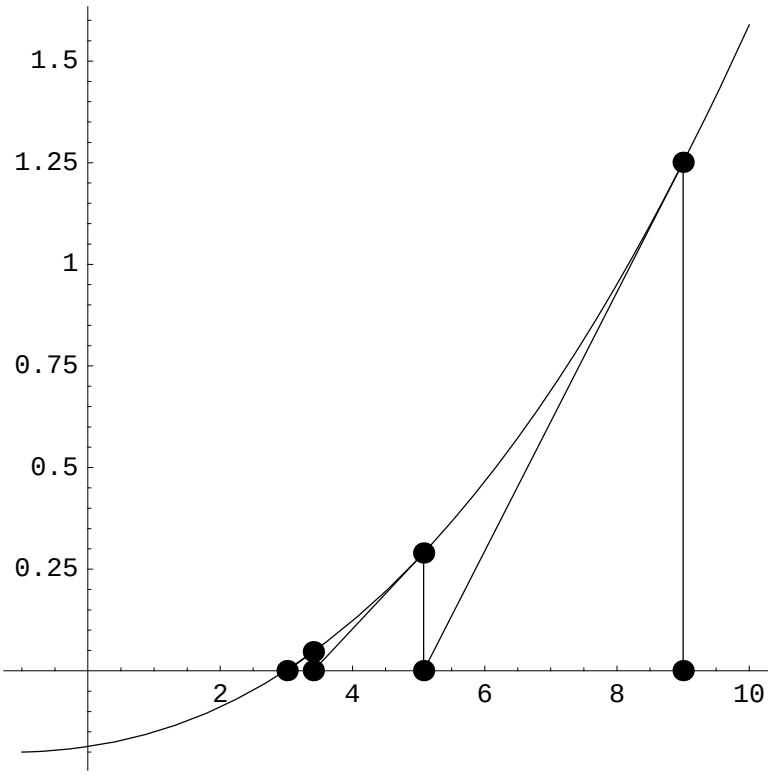
■ Visualisierung des Verfahrens, Animation




1.5
1.25
1
0.75
0.5
0.25
2
4
6
8
10


1.5
1.25
1
0.75
0.5
0.25
2
4
6
8
10





Mathematica verfügt über Möglichkeiten der Animation. Dazu markiere man im vorigen Zellenkomplex die zweite Zellenklammer - von links aus gesehen. Die so markierte Zelle enthält die zu animierenden Graphiken. Danach gebe man das Kommando Control-Y (oder man wähle den Menüpunkt: Cell / Animate Selected Graphics).
Probieren Sie es aus!

■ Analyse des Newton-Verfahrens

Nach *H. Heuser, Lehrbuch der Analysis,* Teil 1, B.G. Teubner Verlag, Stuttgart 1990
Kapitel IX, Abschnitt 70, Seite 406 ff

Zitat:

In diesem Buch, insbesondere in seinem Kapitel VII über Anwendungen der Differentialrechnung, sahen wir uns immer wieder vor die Aufgabe gestellt, Gleichungen der Form $f(x) = 0$ aufzulösen (man erinnere sich etwa an die Bestimmung der Extremalstellen einer Funktion; überhaupt ist das Gleichungsproblem eines der ältesten Probleme der Mathematik, dem jede höhere Zivilisation bereits auf der Stufe ihrer ersten Entfaltung begegnet und das wir denn auch ganz folgerichtig schon bei den Babyloniern um 3000 v. Chr. antreffen). Im Abschnitt 35 hatten wir schon einige Mittel zur Bewältigung von Gleichungen bereitgestellt; insbesondere ist hier der Kontraktionssatz und der Bolzanosche Nullstellensatz zu nennen. Auf dem nunmehr erreichten Entwicklungsstand sind wir in der Lage, ein Verfahren zur (näherungsweisen) Auflösung von Gleichungen vorzustellen und zu begründen, das wegen seiner raschen Konvergenz von eminenter Bedeutung für die Praxis ist und weittragende Verallgemeinerungen gestattet (s. Nr. 189).

Das Verfahren geht auf Newton zurück und ist anschaulich ganz naheliegend. Die Gleichung $f(x) = 0$ aufzulösen, bedeutet doch, den Schnittpunkt (oder die Schnittpunkte) des Schaubildes von f mit der x-Achse zu bestimmen (siehe Figur). Hat man nun bereits eine Näherungslösung x_0 gefunden, so ersetze man, kurz gesagt, die Funktion f durch ihre Tangente im Punkte $(x_0, f(x_0))$ und bringe diese zum Schnitt mit der x-Achse (siehe wieder Figur). Die Gleichung der Tangente ist

$$y = f(x_0) + f'(x_0) \cdot (x - x_0);$$

infolgedessen berechnet sich die fragliche Schnittabszisse x_1 aus der Bedingung $f(x_0) + f'(x_0) \cdot (x - x_0) = 0$ zu

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

x_1 wird in vielen Fällen eine "Verbesserung" von x_0 sein. Wendet man nun dieselbe Überlegung auf x_1 an, so findet man eine weitere "Verbesserung" $x_2 = x_1 - \frac{f(x_1)}{f'(x_1)}$. So fortfahrend erhält man sukzessiv die Zahlen

$$x_{n+1} := x_n - \frac{f(x_n)}{f'(x_n)}, \qquad (70.1)$$

wobei freilich stillschweigend vorausgesetzt wurde, daß die x_n unbeschränkt gebildet werden können. Ist nun f' stetig und *konvergiert* die Newtonfolge (x_n) gegen ein ξ mit $f'(\xi) \neq 0$, so folgt aus (70.1) sofort

$$\xi = \xi - \frac{f(\xi)}{f'(\xi)},$$

also $f(\xi) = 0$: $\lim_{n\to\infty} x_n$ löst die Gleichung $f(x) = 0$.

Mit der Konvergenz der Newtonfolge (x_n) ist es aber manchmal nichts. Diesen peinlichen Umstand offenbart uns etwa die Funktion

$$f(x) := -x^4 + 6x^2 + 11$$

mit den reellen Nullstellen $\pm$ 2,7335... . Ausgehend von $x_0 := 1$ ist hier ständig $x_{2n} = 1$ und $x_{2n+1} = -1$, von Konvergenz der *vollen* Folge (x_n) kann also gewiß nicht die Rede sein. (Kommentar *Ziegenbalg*: Man vergleiche dazu die Graphik, weiter unten im Abschnitt "Die Newton-Graphik: Programmgesteuert".)

Um so wohltuender wirkt der

Satz *Die Funktion* $f : [a, b] \to \mathbb{R}$ *erfülle die folgenden Voraussetzungen*:
a) f'' *ist vorhanden, stetig und* $\geqslant 0$ *bzw.* $\leqslant 0$ (f *ist also konvex bzw. konkav*).
b) f' *hat keine Nullstellen* (f *selbst ist also streng monoton*).
c) *Es* ist $f(a) \cdot f(b) < 0$.

Dann besitzt die Gleichung $f(x) = 0$ *in* $[a, b]$ *genau eine Lösung* ξ. *Die zugehörige Newtonfolge* x_n *konvergiert immer dann — und zwar sogar monoton — gegen* ξ, *wenn man ihren Startpunkt* x_0 *folgendermaßen wählt:*

In den beiden Fällen
(α) $f(a) < 0$, $f'' \leqslant 0$ *und* (β) $f(a) > 0$, $f'' \geqslant 0$
sei $x_0 \in [a, \xi]$, *z.B.* $x_0 := a$. *Es strebt dann* $x_n \nearrow \xi$.

In den zwei restlichen Fällen
(γ) $f(a) < 0$, $f'' \geqslant 0$ *und* (δ) $f(a) > 0$, $f'' \leqslant 0$
sei $x_0 \in [\xi, b]$, *z.B.* $x_0 := b$. *Es strebt dann* $x_n \searrow \xi$.

In allen diesen Fällen haben wir für jedes x_n *die Fehlerabschätzung*

$$|x_n - \xi| \leqslant \frac{|f(x_n)|}{\mu} \quad \text{mit} \quad \mu := \min_{a\leqslant x\leqslant b} |f'(x)|.$$

Besitzt f auf $[a, b]$ *überdies auch noch eine stetige Ableitung dritter Ordnung, so konvergiert* (x_n) *sogar "quadratisch" gegen* ξ, *d. h. so schnell, daß gilt:*

$$|x_{n+1} - \xi| \leqslant K \cdot (x_n - \xi)^2 \quad \text{für} \quad n = 0, 1, 2, \ldots \quad \text{mit konstantem } K.$$

Beweis: [Heuser, Seite 408 ff]

■ Interaktive Entwicklung des Newton-Verfahrens

Wir entwickeln das Verfahren zunächst interaktiv und geben nachher weiter unten (unter dem Programm-Namen "NewtonGraphik") auf der Basis dieser Entwicklung eine "automatisierte", programmgesteuerte Version.

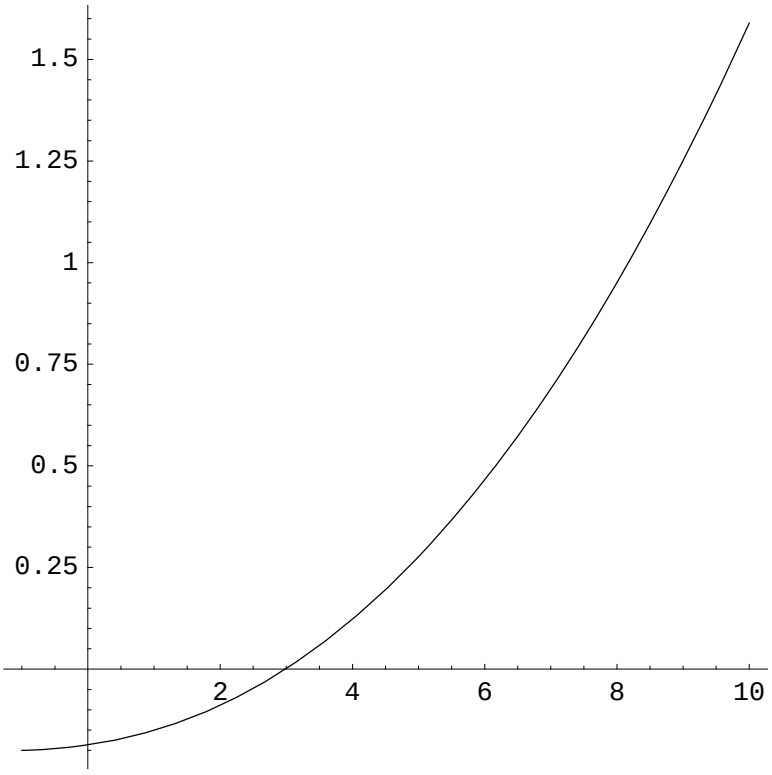
Zur Veranschaulichung des Verfahrens wählen wir die folgende Funktion.

In[53]:=

```
f[x_] := 0.005 * (x + 2)^2 * Log[(x + 2)] - 0.2;
```

In[54]:=

```
Plot[f[x], {x, -1, 10}, AspectRatio → 1]
```



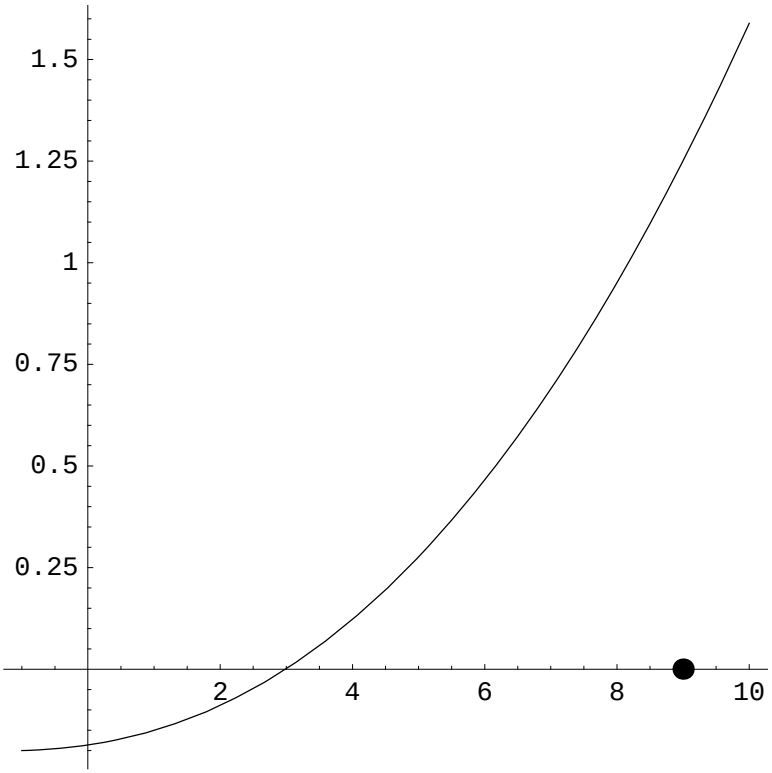
Out[54]=

```
- Graphics -
```

Wir wählen einen Startpunkt x_0.

In[55]:=

```
Remove[g1]; x0 = 9.0;
g1 =
  Graphics[Plot[f[x], {x, -1, 10}, DisplayFunction → Identity]];
g2 = Graphics[{AbsolutePointSize[8], Point[{x0, 0.0}]}];
Show[g1, g2, AspectRatio → 1, DisplayFunction → $DisplayFunction]
```



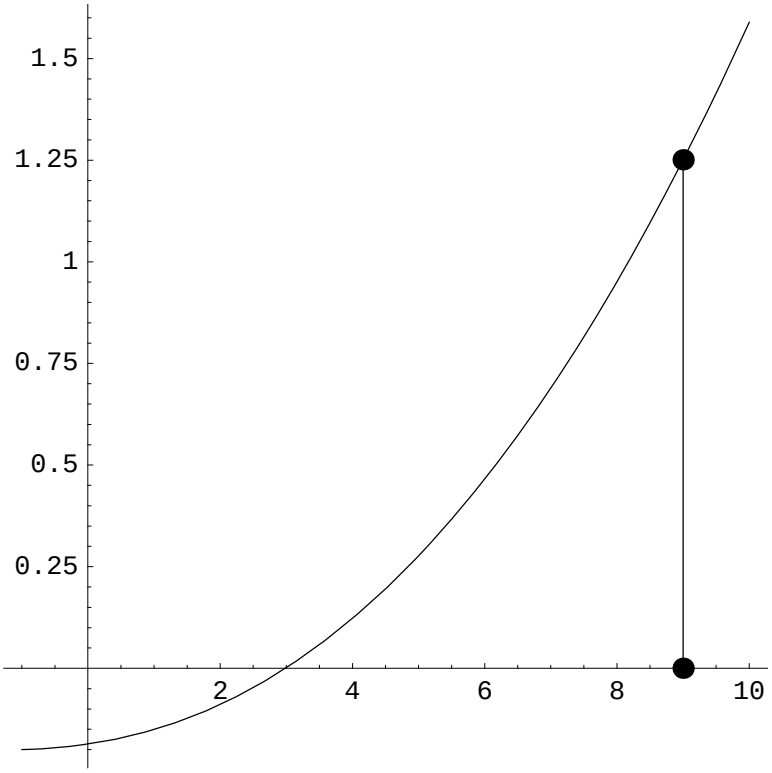
Out[58]=

```
- Graphics -
```

Nun zeichnen wir die Ordinate über dem Punkt x_0.

In[59]:=

```
g3=Graphics[{Line[{{x0,0},{x0,f[x0]}}],
   {AbsolutePointSize[8], Point[{x0, f[x0]}]}}];
Show[g1, g2, g3, AspectRatio->1, DisplayFunction -> $Display-
Function ]
```



Out[60]=

```
- Graphics -
```

Als nächstes zeichnen wir die Tangente im Punkt $(x_0,\ f(x_0))$ bis zum Schnittpunkt mit der x-Achse. Wir benutzen die Ableitungs-Funktion (die uns *Mathematica* aufgrund seiner Fähigkeiten zur symbolischen Differentiation liefert).

In[61]:=

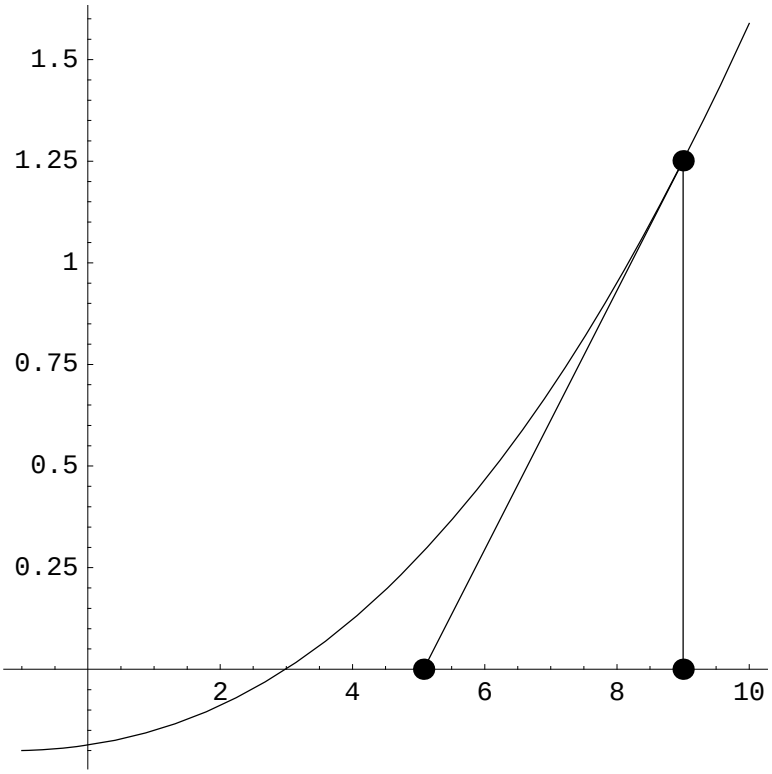
```
f'[x]
```

Out[61]=

```
0.005 (2 + x) + 0.01 (2 + x) Log[2 + x]
```

In[62]:=

```
x1 = x0 - f[x0] / f'[x0];
g4 = Graphics[{Line[{{x0, f[x0]}, {x1, 0}}],
    {AbsolutePointSize[8], Point[{x1, 0}]}},
   DisplayFunction → Identity];
Show[g1, g2, g3, g4, AspectRatio → 1,
 DisplayFunction → $DisplayFunction]
```



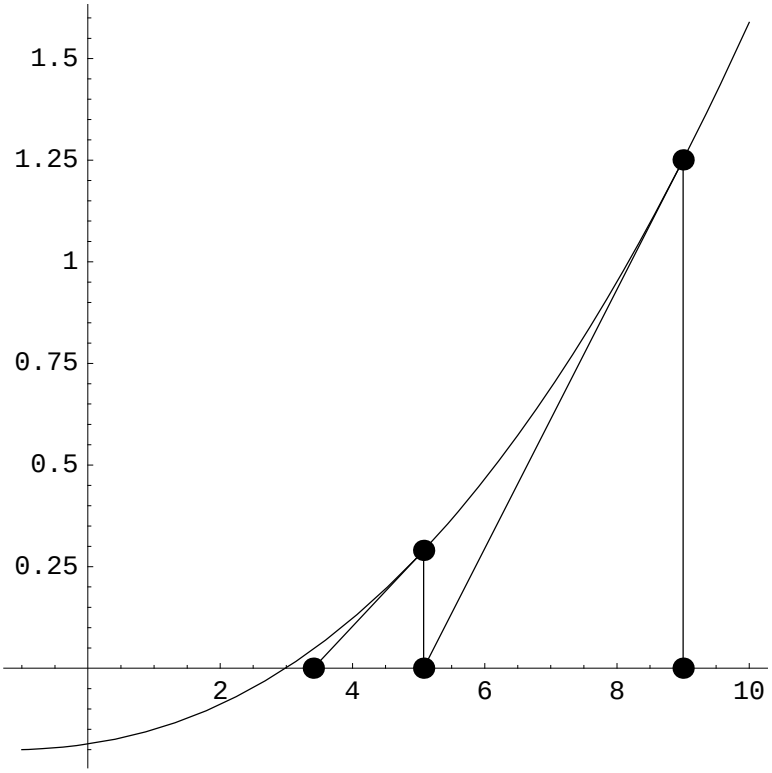
Out[64]=

```
- Graphics -
```

Der nächste Schritt:

In[65]:=

```
g5=Graphics[{Line[{{x1,0},{x1, f[x1]}}], {AbsolutePoint-
Size[8], Point[{x1, f[x1]}] } } , DisplayFunction -> Identity
];
x2=x1-f[x1]/f'[x1];
g6=Graphics[{Line[{{x1, f[x1]},{x2,0}}], {AbsolutePoint-
Size[8], Point[{x2, 0}] } } , DisplayFunction -> Identity ];
Show[g1, g2, g3, g4, g5, g6, AspectRatio->1, DisplayFunction
-> $DisplayFunction ]
```



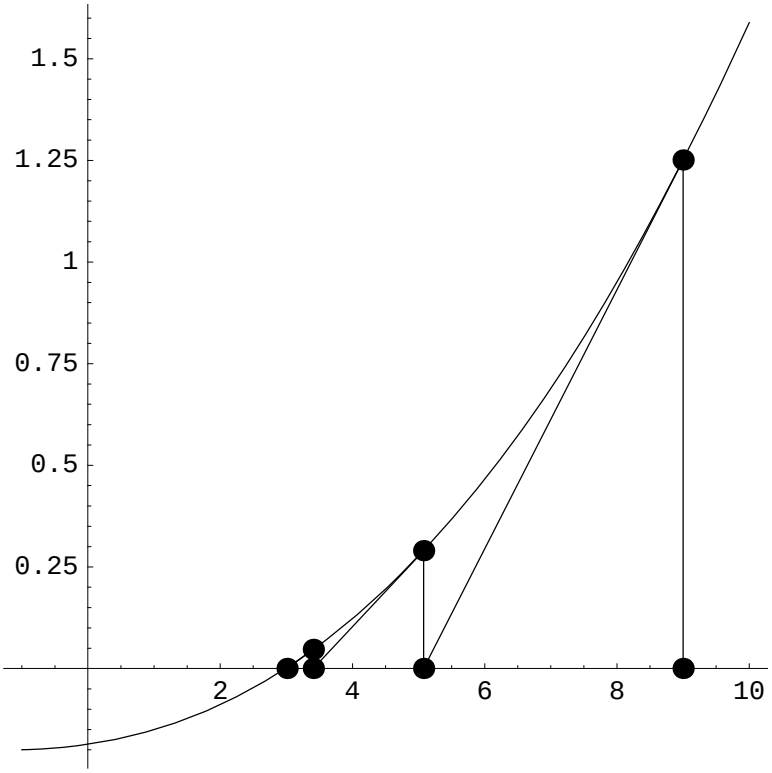
Out[68]=

- Graphics -

Nach dem nächsten Schritt lässt sich der Iterationspunkt x_3 optisch schon nicht mehr von der Nullstelle unterscheiden.

In[69]:=

```
g7=Graphics[{Line[{{x2,0},{x2, f[x2]}}], {AbsolutePoint-
Size[8], Point[{x2, f[x2]}] } } , DisplayFunction -> Identity
];
x3=x2-f[x2]/f'[x2];
g8=Graphics[{Line[{{x2, f[x2]},{x3,0}}], {AbsolutePoint-
Size[8], Point[{x3, 0}] } } , DisplayFunction -> Identity ];
Show[g1, g2, g3, g4, g5, g6, g7, g8, AspectRatio->1, Display-
Function -> $DisplayFunction ]
```



Out[72]=

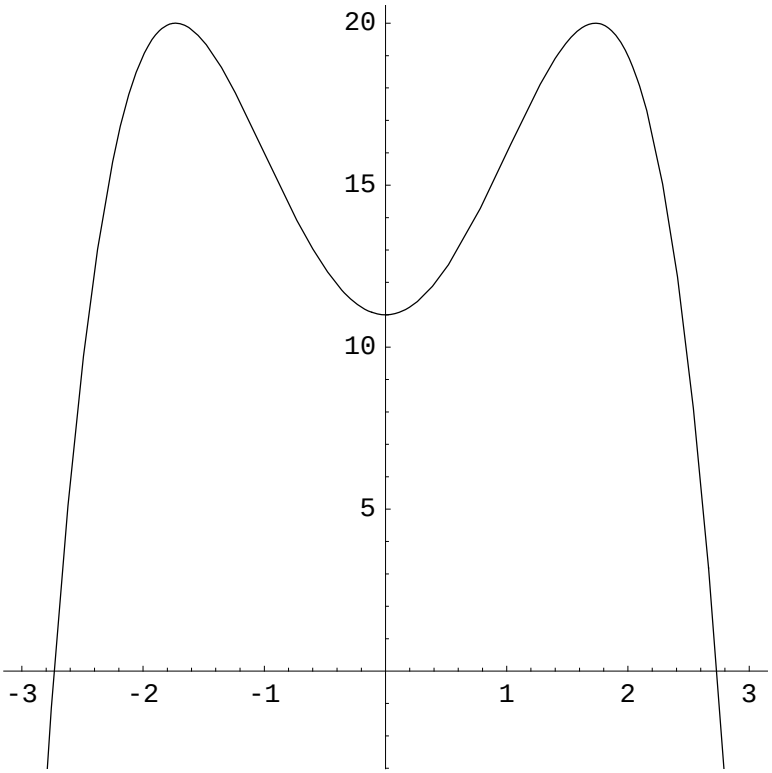
- Graphics -

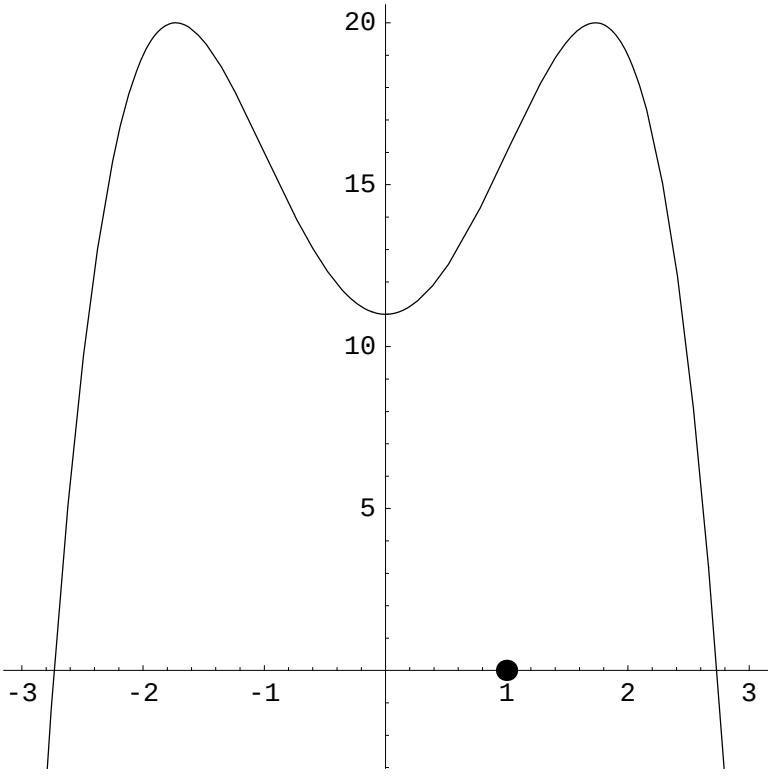
- **Die Newton-Graphik: Programmgesteuert**
- **Ein Fall, bei dem die Voraussetzungen des Satzes nicht erfüllt sind**

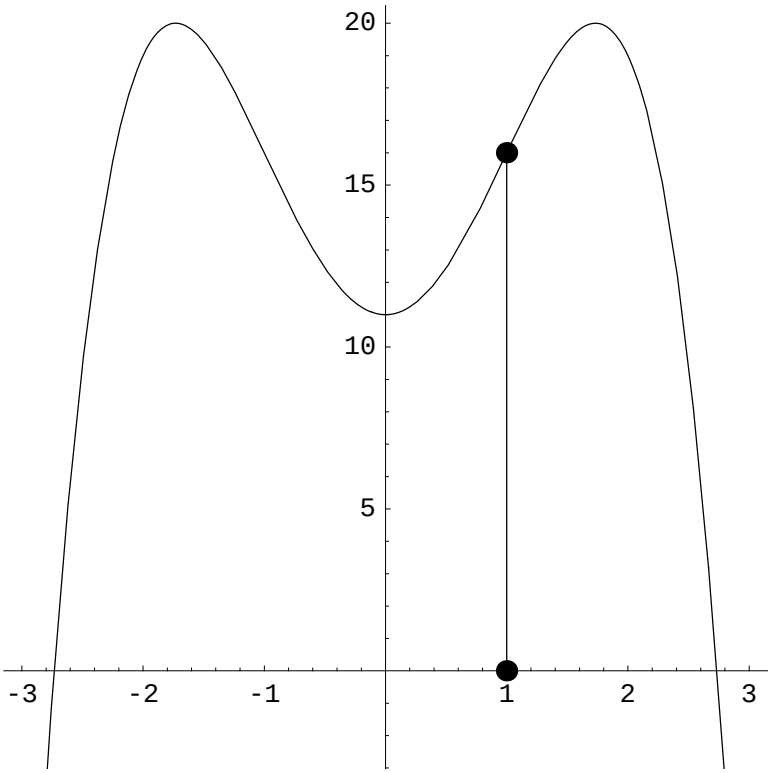
(Vgl. Heuser-Zitat oben)

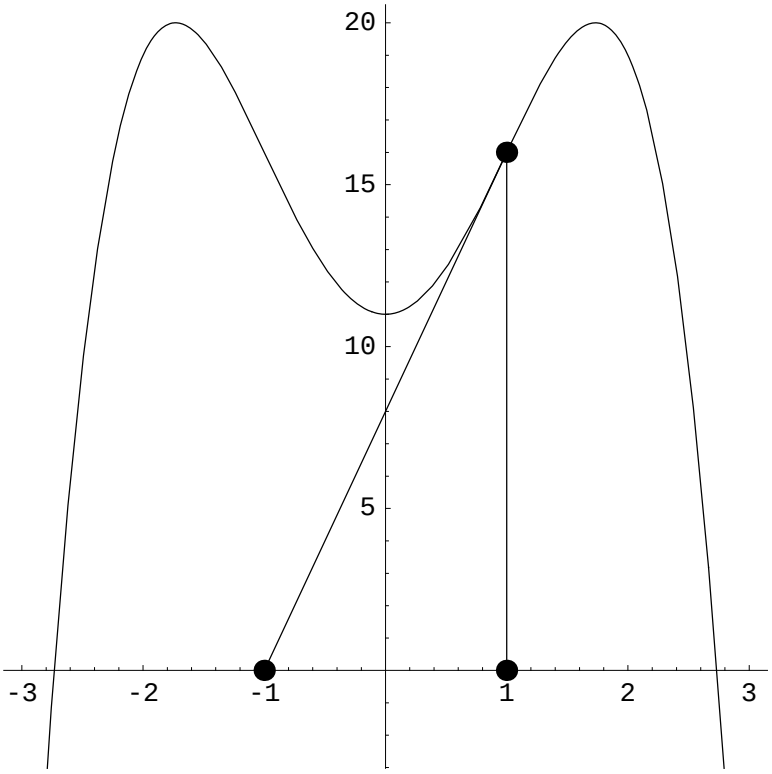
In[76]:=

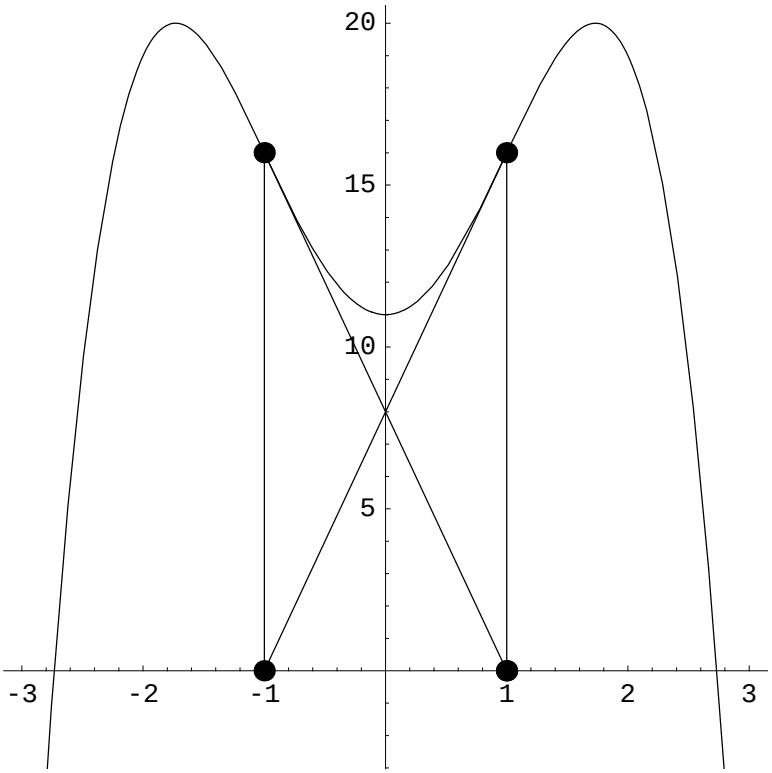
```
f[x_] := -x^4 + 6 x^2 + 11;
NewtonGraphik[f, -3, 3, 1]
```

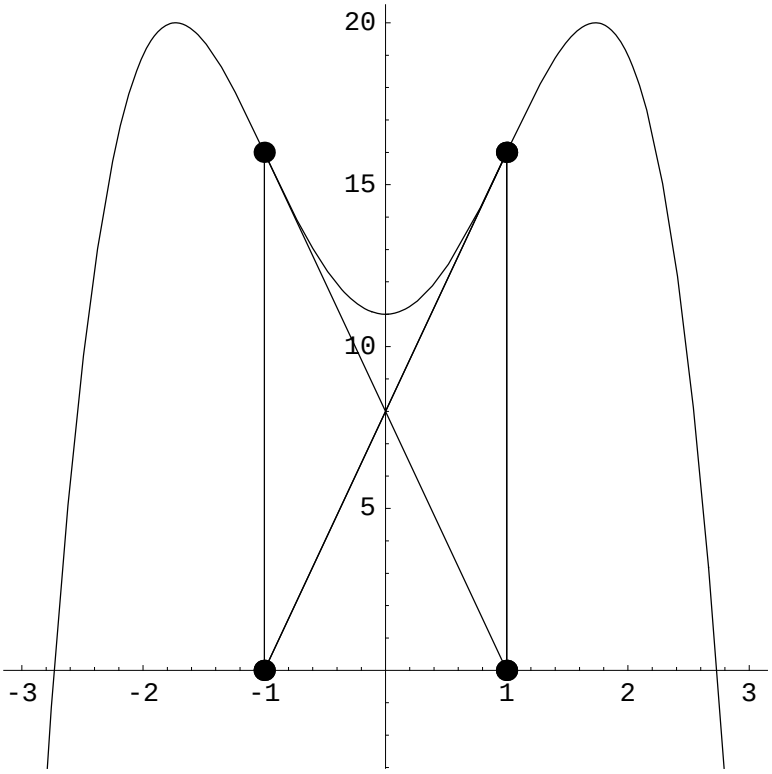

20
15
10
5
-3
-2
-1
1
2
3


20
15
10
5
-3
-2
-1
1
2
3


20
15
10
5
-3
-2
-1
1
2
3


20
15
10
5
-3
-2
-1
1
2
3





Out[77]=

- Graphics -

Aufgabe: Führen Sie, wie oben beschrieben, eine Animation durch.

■ 8. Das Heron-Verfahren als Spezialfall des Newton-Verfahrens

■ Die Quadratwurzel

Das Problem, eine Zahl x mit der Eigenschaft $x^2 = a$ zu finden, ist gleichbedeutend mit dem Problem, eine Nullstelle der Funktion $f(x) = x^2 - a$ zu finden.

```
In[78]:=
f[x_] := x^2 - a;
```

Die Ableitung dieser Funktion ist $f'(x) = 2\,x$.
Computeralgebra Systeme beherrschen die Differentialrechnung in der Regel.

```
In[79]:=
f'[x]
```

```
Out[79]=
2 x
```

Die allgemeine Iterationsvorschrift des Newton-Verfahrens lautet

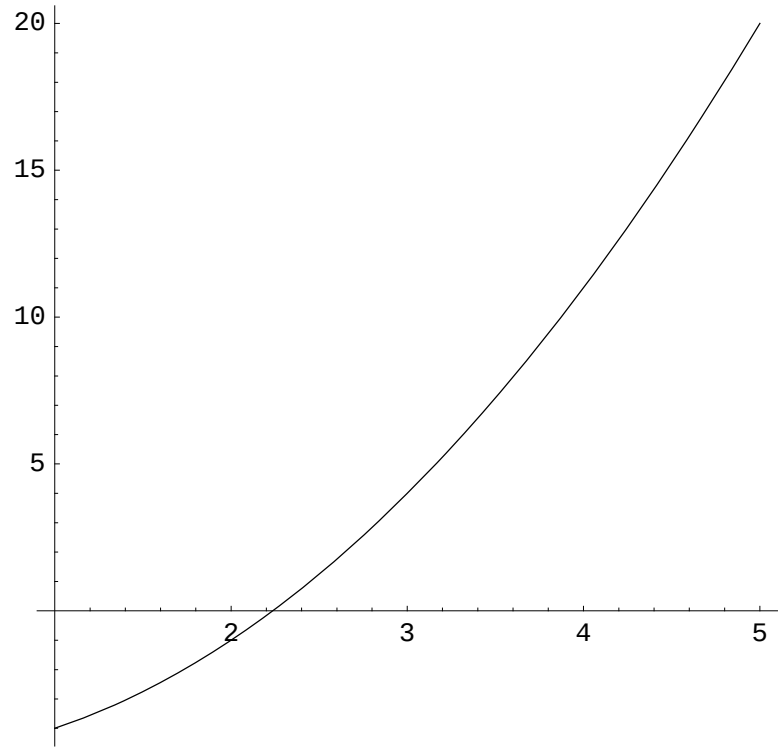
$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

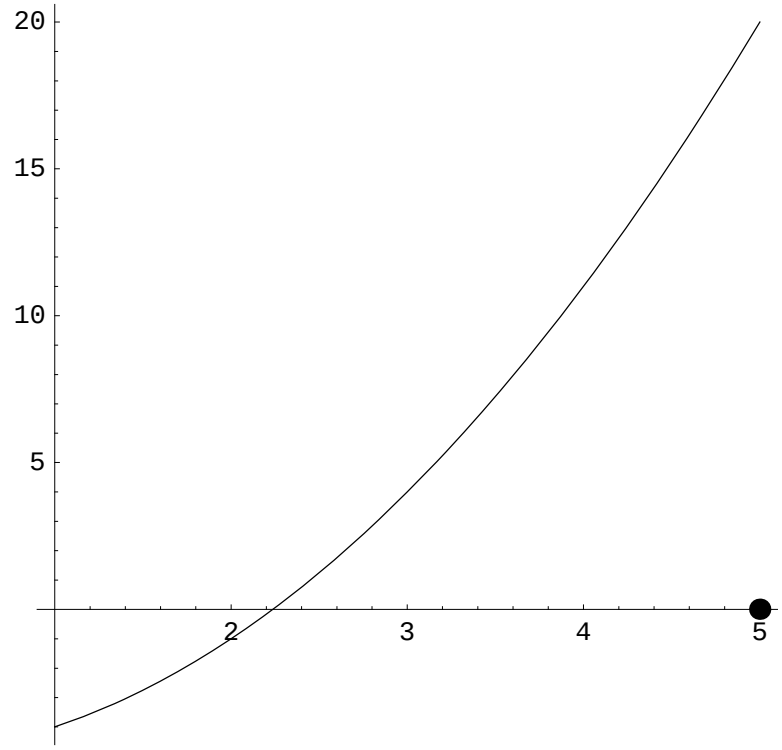
Angewandt auf die Funktion $f(x) = x^2 - a$ wird daraus

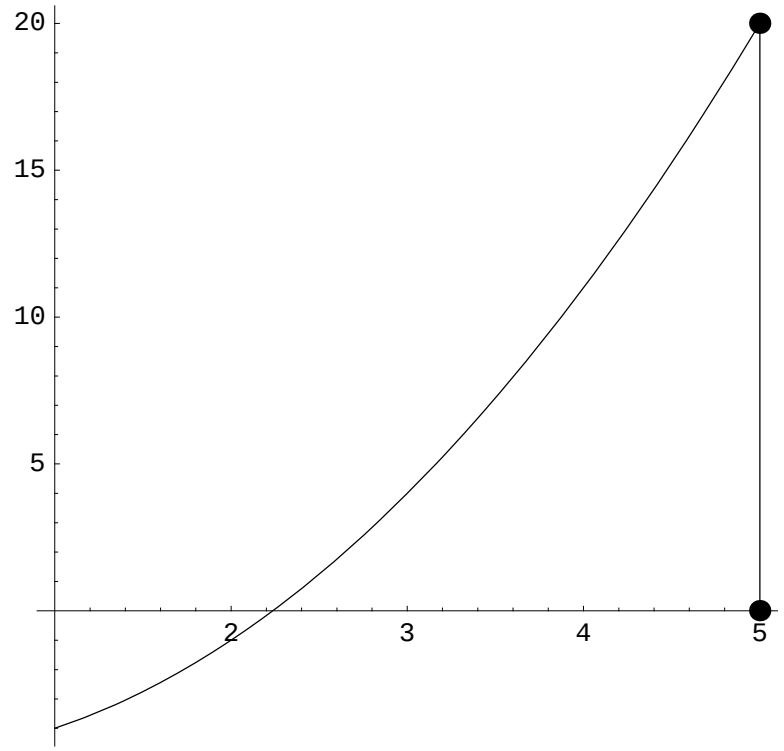
$$x_{n+1} = x_n - \frac{x_n^2 - a}{2\,x_n} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$ (vgl. Heron_3, weiter oben)

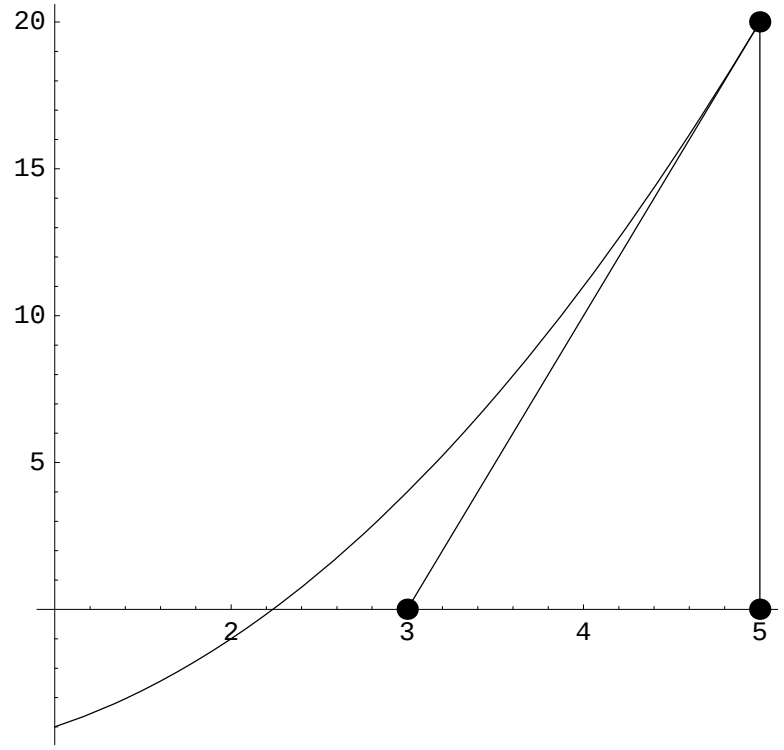
■ Graphik

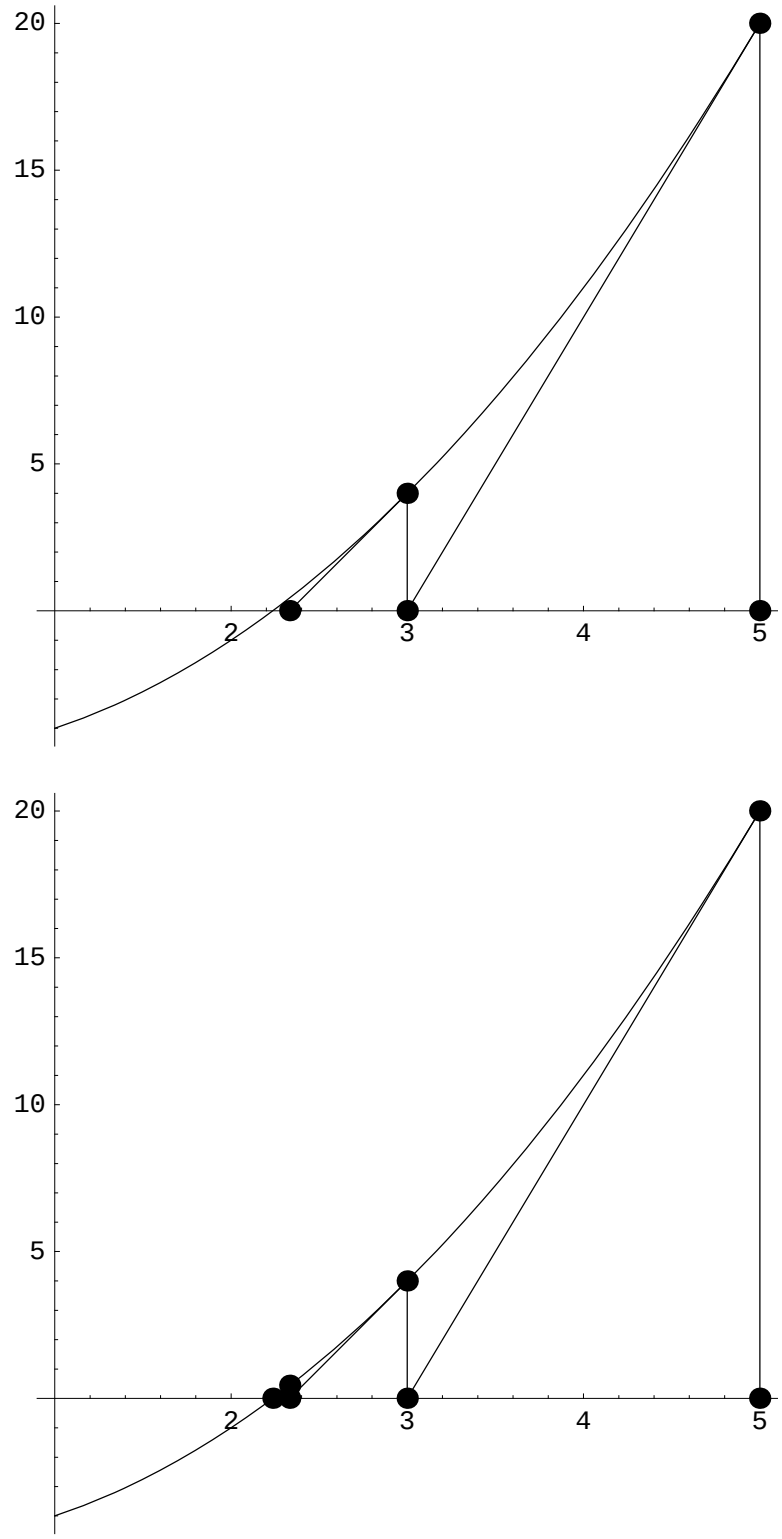
```
In[80]:=
f[x_] := x^2 - a;
a = 5;
g = Graphics[NewtonGraphik[f, 1, 5, a]]
```


20
15
10
5
2
3
4
5


20
15
10
5
2
3
4
5


20
15
10
5
2
3
4
5


20
15
10
5
2
3
4
5



Out[82]=

- Graphics -

■ Die k-te Wurzel

Im Falle der k-ten Wurzel ist eine Nullstelle der Funktion $f(x) = x^k - a$ gesucht. Die Ableitung dieser Funktion ist dann $f'(x) = k \cdot x^{k-1}$ und die allgemeine Iterationsvorschrift des Newton-Verfahrens lautet

$$x_{n+1} = x_n - \frac{x_n^k - a}{k \cdot x_n^{k-1}} = \frac{1}{k}\left((k-1)\cdot x_n + \frac{a}{x_n^{k-1}}\right)$$

In[83]:=
```
Remove[a];
f[x_] := x^k - a;
```

In[85]:=
```
f'[x]
```

Out[85]=
$$k\, x^{-1+k}$$

In[86]:=
```
x_{n+1} = x_n - f[x_n] / f'[x_n]
```

Out[86]=
$$x_n - \frac{x_n^{1-k}\,(-a + x_n^k)}{k}$$

In[87]:=
```
x_{n+1}
```

Out[87]=
$$x_n - \frac{x_n^{1-k}\,(-a + x_n^k)}{k}$$

Im folgenden überprüfen wir die Korrektheit dieses Ergebnisses mit Hilfe der Symbolverarbeitungsmöglichkeiten von Computeralgebra-Systemen.

In[88]:=
```
x_n - x_n^(1-k) (-a + x_n^k) / k == 1/k * ((k - 1) * x_n + a / x_n^(k-1))   // Simplify
```

Out[88]=
```
True
```

■ 9. Erste Bemerkungen zum Themenkomplex Korrektheit und numerische Genauigkeit von Ergebnissen in Computeralgebra Systemen, insbesondere in *Mathematica*

Computeralgebra Systeme (CAS) sind in der Regel bemüht, jeden Ausdruck nach den folgenden Prinzipien auszuwerten. (Diese Prinzipien sind in Softwaresystemen und Programmiersprachen weit weniger selbstverständlich als man annehmen sollte.)

1. Das Ergebnis sollte stets so korrekt wie möglich sein.
2. Das Ergebnis sollte nie falsch sein. Nur näherungsweise richtige (also streng genommen falsche) Ergebnisse werden aber dann zugelassen, wenn der Benutzer dies explizit so wünscht.
3. Die strukturelle Aufbereitung des Ergebnisses sollte eine möglichst große Flexibilität im Hinblick auf weitere Auswertungen zulassen.
4. Das Ergebnis sollte so einfach wie möglich dargestellt werden.

Hierzu seien im folgenden, angelehnt an das Heron-Verfahren, einige Konkretisierungen gegeben.

■ 9.1 Das Ergebnis sollte stets so korrekt wie möglich sein.

Dazu gehören z.B. die folgenden Eigenschaften.

■ 9.1.1 Volle Genauigkeit im Ganzzahl-Bereich; zum Beispiel:

```
In[89]:=
100!
```

```
Out[89]=
933262154439441526816992388562667004907159682643816214685929638952 ↪
 759999322991560894146397615651828625369792082722375825118521091686 ↪
 400000000000000000000000
```

■ 9.1.2 Die Fähigkeit, mit Brüchen zu rechnen; zum Beispiel:

```
In[90]:=
1/6 + 2/33
```

```
Out[90]=
5/22
```

In[91]:=

```
HeronBruch[a_] :=
 Module[{x = a, y = 1, xneu, yneu},
  While[Abs[x^2 - a] > 0.000000001,
   xneu = (x + y)/2; yneu = a/xneu;
   x = xneu; y = yneu];
  Return[x]]
```

In[92]:=

```
HeronBruch[2]
```

Out[92]=

$$\frac{665857}{470832}$$

Das Rechnen mit Brüchen ist aber erheblich aufwendiger als das Rechnen mit Dezimalzahlen. Deshalb verfügen Computeralgebra Systeme auch über die Fähigkeit, mit Dezimalzahlen zu rechnen. Da viele Brüche eine nichtabbrechende Dezimalzahl-Darstellung haben, müssen dann aber oft nur näherungsweise korrekte Ergebnisse in Kauf genommen werden. Man vergleiche dazu die Ausführungen im Abschnitt 9.2.

■ 9.1.3 Die Fähigkeit, symbolisch zu rechnen; zum Beispiel:

In[93]:=

```
Sqrt[2] * Sqrt[3]
```

Out[93]=

$$\sqrt{6}$$

In[94]:=

```
Sqrt[5, 3] * Sqrt[5, 4]
```

Out[94]=

$$5^{7/12}$$

In[95]:=

```
Sqrt[2] * Sqrt[2]
```

Out[95]=

$$2$$

In[96]:=
```
E^(2 Pi I)
```

Out[96]=
```
1
```

■ 9.1.4 Korrektheit und Flexibilität im interaktiven Dialog; zum Beispiel:

Nehmen wir an, der Benutzer benötigt eine selbst zu schreibende Funktion `Hugo[x]`. Wenn `Hugo` noch nicht definiert ist, liefert der Aufruf in *Mathematica*

In[97]:=
```
Hugo[17]
```

Out[97]=
```
Hugo[17]
```

Dies ist zweifellos korrekt, genauso wie `Sin[2]` stets gleich `Sin[2]` ist. Manche anderen Systeme würden in einer entsprechenden Situation vielleicht eine Fehlermeldung der Art "`Hugo`: unbekannte Funktion" ausdrucken und die Auswertung abbrechen. Die unveränderte Rückgabe des Aufrufs (wie im Beispiel `Hugo[17]`) erlaubt es aber, eine potentielle Auswertung (bzw. Auswertungskette) auch mit der zunächst noch undefinierten Funktion `Hugo` zu starten - mit der Möglichkeit, die Auswertung von `Hugo[17]` später nachzuliefern und in die noch nicht abgeschlossene Auswertungskette einzuspeisen.

■ 9.2 Das Ergebnis sollte nie falsch sein. Näherungsweise richtige Ergebnisse werden aber dann zugelassen, wenn der Benutzer dies explizit so wünscht.

Symbolische Ausdrücke, werden "ohne Not" nie in Kommazahlen umgewandelt,

In[98]:=
```
Sin[2]
```

Out[98]=
```
Sin[2]
```

es sei denn, der Benutzer wünscht dies explizit:

In[99]:=
```
N[Sin[2]]
```

Out[99]=
```
0.909297
```

Dieses Ergebnis ist nur näherungsweise richtig (also genau genommen falsch). Im Falle von Auswertungen, die zu Dezimalzahlen (bzw. allgemein zu Kommazahlen) führen, kann die gewünschte Genauigkeit beliebig vorgegeben werden.

In[100]:=
```
N[Sin[2], 100]
```
Out[100]=
```
0.90929742682568169539601986591174484270225497144789026837897301153
 09673015407835446201266889249593803
```

In[101]:=
```
N[HeronBruch[2], 100]
```

Out[101]=
```
1.41421356237468991062629557889013491011655962211574404458490501920
 00543718353892683589900431576443 40
```

Das Rechnen mit Brüchen ist in der Regel sehr viel aufwendiger als das Rechnen mit Kommazahlen. Deshalb verwendet man oft Kommazahlen, auch in Kenntnis der Tatsache, dass die damit erzielten Ergebnisse bestenfalls näherungsweise korrekt sind. So z.B. in der eingangs formulierten Version von Heron:

In[102]:=
```
Heron[a_] :=
 Module[{x = a, y = 1.0, xneu, yneu},
  While[Abs[x^2 - a] > 0.000001,
   xneu = (x + y)/2; yneu = a/xneu;
   x = xneu; y = yneu];
  Return[x]]
```

In[103]:=
```
Heron[2]
```
Out[103]=
```
1.41421
```

In diesem Beispiel wurde die Variable y durch den Zuweisungs-Befehl y = 1.0 mit einer Kommazahl belegt und dadurch quasi als Zahl "minderer Präzision" gekennzeichnet. Alle Rechnungen, in denen die Variable y vorkommt, werden damit in der Folge auch nur als Näherungsrechnungen durchgeführt. Alle Zwischen- und Endergebnisse von Rechenketten, in denen y vorkommt, haben dann höchstens die Präzision von y.

Computeralgebra Systeme besitzen allerdings im Gegensatz zu gewöhnlichen Softwaresystemen Möglichkeiten, die Genauigkeit von Rechnungen mit Kommazahlen beliebig zu kontrollieren. In der folgenden Version wird die Variable `y` durch den Befehl `y = SetAccuracy[1.0, 100]` als hochgenaue Kommazahl mit 100 exakten Nachkommastellen definiert und alle Rechnungen werden nun so durchgeführt, dass die Ergebnisse stets mit einer Genauigkeit von 100 Nachkommastellen ermittelt werden.

In[104]:=

```
HeronDezimalbruchHochgenau[a_] :=
 Module[{x = a, y = SetAccuracy[1.0, 100], xneu, yneu},
  While[Abs[x^2 - a] > 0.000000001,
   xneu = (x + y)/2; yneu = a/xneu;
   x = xneu; y = yneu];
  Return[x]]
```

In[105]:=

```
HeronDezimalbruchHochgenau[2]
```

Out[105]=

```
1.4142135623746899106262955788901349101165596221157440445849050192 ↓
 0054371835389268358990043157644 3402
```

Zu einer vertieften Behandlung dieses Themas konsultiere man das *Mathematica* Handbuch, Abschnitt 3.1.4 Numerical Precision. Man beachte insbesondere den Unterschied zwischen *arbitrary-precision numbers* und *machine-precision numbers*. Grob gesprochen, wird bei arbitrary-precision numbers die Präzision (auf Kosten der Rechengeschwindigkeit) und bei machine-precision numbers die Rechengeschwindigkeit (auf Kosten der Präzision) optimiert.

Rechnungen mit machine-precision numbers verwenden die gewöhnliche in den Computern eingebaute (allerdings oft fehlerhafte) Arithmetik (näheres zur Fehleranfälligkeit von "eingebauter" Software in: *Ziegenbalg*: Algorithmen von Hammurapi bis Gödel, Kapitel 6); für Rechnungen mit arbitrary-precision numbers wurden spezielle Softwarepakete entwickelt, die in der Regel nur in Computeralgebra Systemen vorzufinden sind.

■ 9.3 Die strukturelle Aufbereitung des Ergebnisses sollte eine möglichst große Flexibilität im Hinblick auf weitere Auswertungen zulassen.

Das obige *Mathematica*-Programm `Heron` liefert genau das, was es soll, nämlich die Quadratwurzel des Eingabewerts. Die im Verfahren errechneten Zwischenwerte x_i und y_i werden "weggeworfen".
Für bestimmte Zwecke, wie z.B. die graphische Darstellung erweist es sich jedoch als nützlich, wenn die Zwischenwerte auch nach Beendigung des Verfahrens noch zur Verfügung stehen. Aus diesem Grund wird im folgenden Progamm `HeronList` eine Liste mit allen Zwischenwerten angelegt.

In[106]:=
```
HeronList[a_] :=
  Module[{x = a, y = 1.0, xneu, yneu, L = {{a, 1}}},
    While[Abs[x^2 - a] > 0.000001,
      xneu = (x + y) / 2 ;   yneu = a / xneu;
      L = Append[L, {xneu, yneu}];
      x = xneu; y = yneu];
    Return[L]]
```

In[107]:=
```
HeronList[560]
```

Out[107]=
```
{{560, 1}, {280.5, 1.99643}, {141.248, 3.96465},
 {72.6064, 7.71282}, {40.1596, 13.9444}, {27.052, 20.7009},
 {23.8764, 23.4541}, {23.6653, 23.6634}, {23.6643, 23.6643}}
```

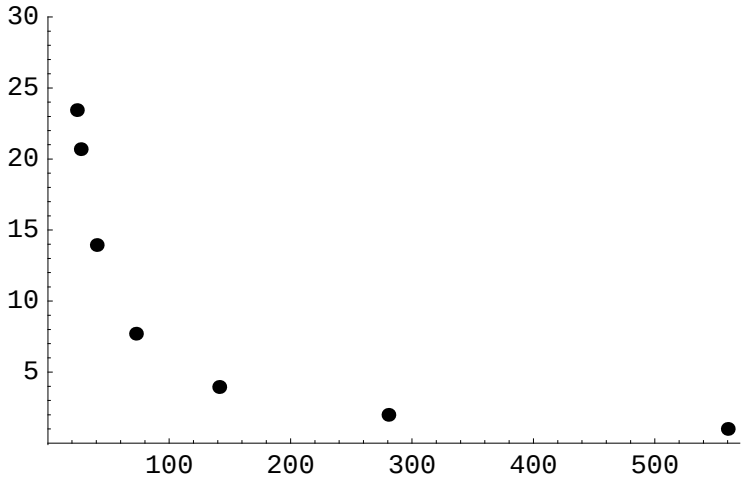
Dieses Ergebnis enthält alle relevanten Informationen in einer angemessen strukturierten Form. Es kann nun leicht in geeignete Routinen zur "medialen" Aufbereitung eingespeist werden.

In[108]:=
```
TableForm[HeronList[560]]
```

Out[108]//TableForm=

560	1
280.5	1.99643
141.248	3.96465
72.6064	7.71282
40.1596	13.9444
27.052	20.7009
23.8764	23.4541
23.6653	23.6634
23.6643	23.6643

In[109]:=
```
ListPlot[HeronList[560], AxesOrigin -> {0, 0},
 PlotRange -> {{-0.1, 570}, {-0.1, 30}},
 PlotStyle -> PointSize[0.02]]
```



Out[109]=
```
- Graphics -
```

Wollte man doch nur den (letzten) Näherungswert, so ginge dies z.B. folgendermaßen:

In[110]:=
```
Last[Last[HeronList[560]]]
```

Out[110]=
```
23.6643
```

■ 9.4 Das Ergebnis sollte so einfach wie möglich dargestellt werden.

In[111]:=
```
1/2 + 1/3 + 1/6
```

Out[111]=
```
1
```

In[112]:=
```
E^(Pi I)
```

Out[112]=
```
-1
```

Was "einfach" bedeutet, ist jedoch nicht immer von vornherein klar. Ist $(a+b)^2$ einfacher als $(a^2+2ab+b^2)$? Da die Antwort auf diese Frage nicht eindeutig ist, gibt es eine Reihe von Funktionen zur Manipulation von symbolischen Ausdrücken. `Simplify` und `Expand` sind zwei derartige Funktionen.

In[113]:=
```
Remove[a, b]
```

In[114]:=
```
Expand[(a + b)^2]
```

Out[114]=
$$a^2 + 2\,a\,b + b^2$$

In[115]:=
```
Simplify[a^2 + 2 a b + b^2]
```

Out[115]=
$$(a + b)^2$$

In[116]:=
```
(a + b)^2 == a^2 + 2 a b + b^2
```

Out[116]=
$$(a + b)^2 == a^2 + 2\,a\,b + b^2$$

Das letzte Ergebnis war zwar richtig, aber nicht das, was vermutlich erwartet wurde. Man vergleiche hiermit:

In[117]:=
```
Simplify[(a + b)^2 == a^2 + 2 a b + b^2]
```

Out[117]=
```
True
```

■ Hilfsprogramme