Programmieren mit *DERIVE*

Josef Böhm

bk teachware Schriftenreihe Nr. SR-32
ISBN 3-901769-50-1

Inhalt

Einleitung

Dieses Büchlein entstand aus der langjährigen Beschäftigung des Autors mit *DERIVE* als Begründer der *International DERIVE & TI-92 User Group* (*DUG*), als Lehrerfortbildner und vor allem als Lehrer, der lange und erfolgreich *DERIVE* im Mathematikunterricht an einer Handelsakademie eingesetzt hat.

Mit der Programmierfähigkeit hat *DERIVE 5* eine von vielen Benützern schon lange geforderte Eigenschaft erhalten. Damit läßt sich die ungeheure „power" eines sehr guten CAS mit der Vielseitigkeit einer Programmiersprache verbinden. Wenn auch die Programmierumgebung – Editieren, Debuggen, usw. – noch einiges an Wünschen offen läßt, kann man auf einfache Art und Weise bemerkenswerte Resultate erzielen, vor allem dann, wenn man schon Programmiererfahrung hat. Für das Editieren und Debuggen werden in diesem Buch sehr brauchbare Hilfen angeboten.

Wie ich aus Erfahrung zu wissen glaube, wurden nun einige Generationen von PC-Nutzern herangezogen, denen fast ausschließlich der Umgang mit Standardpaketen beigebracht wurde, und für die das Selbsterstellen von Programmen nicht mehr notwendig schien. Zur Zeit ist jedoch ein Umdenken im Gange, da die Herstellung von Makros, Java-Applikationen u.ä. wieder zum Programmieren hinführen sollen.

Dieses Buch wendet sich nicht unbedingt an den *DERIVE*-Einsteiger. Grundkenntnisse im Umgang mit dieser wunderbaren Software wird vorausgesetzt. Der Autor hofft, dass aber auch der bereits etwas kundige DERIVIANER neben dem Programmieren viel Wissenswertes über *DERIVE*-Funktionen und -Syntax mitnehmen kann.

Ich möchte ausdrücklich darauf hinweisen, dass sich sicherlich alle vorgestellten Programme auch noch anders – und vielleicht auch kürzer und/oder eleganter – schreiben lassen. Ich habe aber bewusst der Verständlichkeit den Verzug gegenüber dem Griff in die Trickkiste gegeben.

Jedem Kapitel sind Herausforderungen angeschlossen, die den Leser zu eigenen Programmierabenteuern ermuntern sollen. Manche Aufgabenstellungen lassen sich aus dem sehr komplexen Funktionsvorrat von *DERIVE* auch ohne Programm lösen. Aus didaktischen Gründen sollten sie aber mit Hilfe von Programmen bearbeitet werden, sonst müsste man gleich mit zu umfangreichen Programmen beginnen, die für ein doch mehr einführendes Werk ungeeignet wären. Für die meisten der Herausforderungen sind Lösungsmöglichkeiten auf der beigelegten Diskette zu finden.

Die Programme wurden mit der aktuellen - englischen - *DERIVE*-Version 5.05 gestestet, aber wie immer können sich doch trotz größter Sorgfalt Fehler eingeschlichen haben. Ich bin für Rückmeldungen sehr dankbar.

Ich möchte Sie überhaupt auffordern, sich bei mir zu melden, wenn Sie Probleme haben, wenn Sie interessante Aufgabenstellungen selbst gelöst haben, oder um Lösungen suchen. Auch für Verbesserungsvorschläge für *DERIVE* bin ich sehr dankbar. Über die *DERIVE User Group* habe ich einen sehr guten und direkten Kontakt zu den Softwareschmieden, die gerne Anregungen von Benützern entgegennehmen.

Beachten Sie bitte auch die ausführliche Referenzliste am Ende des Buchs. In der Referenzliste wird Ihnen auffallen, dass sehr viele Beiträge aus den *DERIVE Newsletters* (DNL) genannt sind. Diese sind wirklich eine fast unerschöpfliche Quelle für Programmierideen. Wo früher lange Funktionsketten notwendig waren, kann heute ein kompaktes Programm erzeugt werden.

DERIVE-Programme können Sie auch in einigen aktualisierten Utility-Dateien finden, die zum Lieferumfang von *DERIVE* gehören, z.B. NUMBER.MTH (entstanden unter entscheidender Mitwirkung von Johann Wiesenbauer). Außerdem publizieren wahre *DERIVE*-Spezialisten wie Johann Wiesenbauer, Richard Schorn, Stefan Welke, Josef Lechner, Rüdeger Baumann, Terence Etchells u.a. *DERIVE*-Programme für den DNL, die aber den Rahmen dieses Büchleins bei weitem sprengen würden. Albert Rich, einer der Gründerväter von *DERIVE*, hilft bei Problemen immer sehr rasch. Ich möchte mich hier bei allen, die immer wieder helfen, die Möglichkeiten von *DERIVE* auszuloten, ganz herzlich bedanken.

Ganz besonderer Dank gilt Richard Schorn und Rüdeger Baumann, die mir wertvolle Hinweise gegeben haben.

Rüdeger Baumann hat auch eine Variante des BRIANCHON-Programms geliefert, das auf der Diskette unter BRIBAUM.MTH zu finden ist. Herzlichen Dank auch dafür.

Und jetzt bleibt mir nur noch, Ihnen viel Spaß und Erfolg beim Programmieren mit *DERIVE* zu wünschen.

Josef Böhm (nojo.boehm@pgv.at)

1 Woraus ein *DERIVE* Programm besteht – das erste Programm

Eine der wichtigsten Neuerungen von *DERIVE* 5 ist die Möglichkeit, Programme zu schreiben. Es war auch in früheren Versionen schon möglich, sehr komplexe Abläufe und Algorithmen in *DERIVE* zu realisieren, aber doch nur in der Weise, dass man oft eine Vielzahl von Funktionen definieren musste, die sich gegenseitig aufriefen. Damit wurde das Nachvollziehen, bzw. Ändern eines „Programms alter *DERIVE*-Art“ in vielen Fällen recht mühsam.

Die wichtigste Neuerung ist nun, dass man alle diese Funktionen in ein „Paket“ schnüren kann. Mit der IF-Konstruktion steht schon lange eine mächtige Kontrollstruktur zur Verfügung, wogegen man bisher die Möglichkeit von zähler- und ablaufgesteuerten Schleifen vermissen musste.

Es gibt zwar auch hier noch keine FOR – NEXT-Schleife, und auch das WHILE und UNTIL werden Programmierer vergeblich suchen. Doch mit der `Loop`-Anweisung kann man das alles relativ einfach verwirklichen. Das ist die zweite wesentliche Erweiterung des bisher in *DERIVE* vorhandenen Befehlsvorrats.

Gemeinsam mit der hohen Funktionalität des Computer Algebra Systems sind beachtenswerte Ergebnisse zu erzielen.

Um den Leser ein wenig neugierig zu machen, stelle ich ein Resultat eines Programmierversuchs vor: Sie alle kennen den Schrecken der Schüler – Extremwertaufgaben. Nun, und da gibt es ein Werkzeug, das nach Eingabe von Haupt- und Nebenbedingung, den beiden Variablen und den Definitionsgrenzen einer dieser beiden variablen Größen, die komplette Lösung des Problems in einer ansprechenden Form präsentiert.

Mit einer herkömmlichen Programmiersprache ist das wohl nicht leicht möglich, da das exakte Differenzieren nicht verfügbar ist. Aber hier:

$$\text{extrem}\left(x^2 \cdot y \cdot \pi,\ \frac{9}{6} = \frac{y}{6 - x},\ x,\ y\right)$$

x	y	Optimum	Art
4	3	$48 \cdot \pi$	lokales Maximum
0	9	0	lokales Minimum

Die Berücksichtigung von allfälligen Randextrema finden Sie ebenso wie das komplette Programm in Kapitel 9.

Leider wird weder im Handbuch noch in der *DERIVE*-Online Hilfe ausführlich auf das Programmieren mit *DERIVE* eingegangen. Das vorliegende Büchlein soll helfen, diese Lücke zu schließen. Daneben sollen einige – vielleicht auch für Sie neue – Eigenschaften von *DERIVE* gezeigt werden, und dies alles an Hand von ausgesuchten Beispielen, die sich alle in der

Sekundarstufe II einsetzen lassen. Ich bin nämlich der Meinung, dass man durchaus auch das Programmieren – in kleinen Dosen - wieder in den Mathematikunterricht einfließen lassen könnte und dies auch tun sollte.

Wir beginnen mit einem einfachen Auftrag aus der Finanzmathematik:

> Ein Startkapital soll über eine bestimmte Anzahl von Perioden jeweils am Ende der Periode um einen festen Betrag vermehrt werden. Welcher Betrag steht nach Ablauf der Perioden zur Verfügung, wenn eine gegebene Verzinsung (bei Zinseszinsen) zu berücksichtigen ist?
> Die Ausgabe soll nicht nur den Endbetrag liefern, sondern in einer Tabelle den jeweiligen Kapitalstand am Ende jeder Periode ausweisen.

Dem Programm geben wir den Namen `FVPROG` erhalten (`FV` von Endwert = Future Value). Als Parameter sind das Startkapital, der konstante Zuwachs, die Verzinsung und die Anzahl der Perioden notwendig.

Ich würde mir beim Startkapital 4000€, das fünfmal um je 2000€ aufgestockt wird und bei 5% Zinsen eine Ausgabe etwa in der folgenden Form wünschen:

```
FVPROG(4000, 2000, 5, 5) =
```

Periode	Kapitalstand
0	4000
1	6200
2	8510
3	10935.5
4	13482.275
5	16156.38875

Den kleinen Schönheitsfehler mit den Dezimalstellen werden wir mit einer geeigneten Rundungsroutine noch ausmerzen können.

Mir ist klar, dass man für dieses Problem kein Programm benötigt, sondern mit einer entsprechenden Formel aus der Rentenrechnung sofort zum Ergebnis kommen kann.

$$\text{TABLE}\left(\left[4000\cdot 1.05^{n} + \frac{2000\cdot(1.05^{n}-1)}{1.05-1}\right], n, 0, 5\right)$$

0	4000
1	6200
2	8510
3	10935.5
4	13482.275
5	16156.38875

DERIVE-Kenner werden lächeln und die Aufgabe rekursiv lösen, indem sie mit einem eleganten Einzeiler die `ITERATES`-Funktion einsetzen:

$$FV\_ITER(pv, add\_pmt, perc, periods) := ITERATES\left(\left[v_1 + 1, v_2 \cdot \left(1 + \frac{perc}{100}\right) + add\_pmt\right], v_, [1, pv], periods\right)$$

$$FV\_ITER(4000, 2000, 5, 5)$$

Ich möchte aber dieses Beispiel als Einstieg nutzen, um gleich eine zählergesteuerte Schleife einzusetzen und außerdem einige ganz wesentliche Grundlagen zum Programmieren mit *DERIVE* hervorzuheben.

Da man eine Rundungsfunktion sicherlich öfters gebrauchen kann, werden wir diese extern definieren und dann vom Programm her aufrufen.

$$runde(x, m) := \frac{FLOOR(10^m \cdot x + 0.5)}{10^m}$$

$$runde(16156.38875, 4) = 16156.3888$$

$$runde(16156.38875, 1) = 16156.4$$

$$runde(16156.38875, 2) = 16156.39$$

$$runde(16156.38875, -3) = 16000$$

Nun soll das Geheimnis gelüftet werden. Wie sieht ein *DERIVE*-Programm aus?

```
1  FVPROG(pv, add_pmt, perc, periods, n := 0, cap, values) :=
2    Prog
3         cap := pv
4         values := [[n, cap]]
5         Loop
6         n :+ 1
7             cap := (1 + perc/100)·cap + add_pmt
8             values := APPEND(values, [[n, runde(cap, 2)]
9             If n = periods
10               RETURN APPEND([["Periode", "Kapitalstand"]], values)
```

Die links stehenden, kursiv geschriebenen Ziffern gehören nicht zum Programm, sondern sind Referenznummern für Programmzeilen, auf die ich mich im Text beziehen werde.

Das sieht ja ganz schön aus und erklärt sich weitgehend von selbst. Aber beginnen wir ganz von vorne – und versuchen Sie bitte noch nicht, dieses Programm abzutippen!

`FVPROG` ist der Programm- (Funktions-) name mit den Parametern `pv` (Anfangskapital = = *principal value*), `add_pmt` (Zusatzzahlung = *additional payment*), `perc` (Prozentsatz für die Verzinsung) und `periods` für die Laufzeit in Perioden. Aber dann folgen ja noch weitere Variable in der Parameterliste?

Im allgemeinen müssen alle Variablen, die im Programm auftreten, als lokale Variable in der Parameterliste angeführt werden. Am Beginn stehen jene Parameter, die vom Benutzer belegt werden müssen, alle anderen können ungeordnet aufgezählt werden. Laufvariable in `VECTOR-, SELECT-` Befehlen u.ä. braucht man dabei nicht zu berücksichtigen.

Ich habe diese „Vorschrift“ eingerahmt, da sie die vielleicht wichtigste überhaupt ist. Es können auch globale Variable verwendet werden, die über das Programm hinaus „am Leben bleiben“.

Dann fällt noch `n:=0` in der Parameterliste auf. Startbelegungen für Variable können in die Parameterliste aufgenommen werden. Man hätte auch nur `n` hineinnehmen können und dann als dritte Programmzeile `n:=0` geschrieben.

In Zeile ***2*** steht, dass es sich um ein Programm handelt. `cap` ist eine Hilfsvariable die mit dem jeweils aktuellen Kapitalstand belegt wird, der zum Zeitpunkt $n = 0$ mit `pv` übereinstimmt. Mit `values` definieren wir eine Matrix, die vorerst nur aus einem Zeilenvektor (für die Periode 0) besteht.

Und nun beginnt in ***5***, eingeleitet von `Loop` die Schleife, die jeweils die Perioden hochzählt (`n :+ 1`) und aus dem alten Kapitalstand durch Verzinsung und Hinzufügung der zusätzlichen Zahlung das neue Kapital erzeugt.

Hinweis: `n :+ 1` ist eine Kurzform für `n := n + 1`. Analoges gilt für die anderen Grundrechnungsoperationen, z.B., `n :- 1, n :* 2, oder n :/ 10.`

Die Berechnung des neuen Kapitals erfolgt in Zeile ***7***. Die Matrix `values` wird durch Anhängen (`APPEND` = anfügen) der neuen Zeile aktualisiert. Und hier wird auch die Rundungsfunktion für die Ausgabe aufgerufen. Wenn der Zähler `n` die Anzahl der Perioden erreicht hat, dann ist es Zeit für die Ausgabe der Liste, nicht ohne vorher noch eine Überschrift vorangestellt zu haben.

So weit, so gut. Aber wie schreiben wir nun das Programm, da uns nur die Eingabezeile zur Verfügung steht? Dass man nicht direkt in das Algebrafenster hineinschreiben kann, haben sicherlich schon viele von uns oft bedauert.

Das ist nun das große Problem, dass uns in *DERIVE* – vorerst – kein Fullscreen Editor zur Verfügung steht. Das Programm ist in linearer Form, d.h., Zeichen für Zeichen, Klammer für Klammer zu schreiben und man beginnt:

```
FVPROG(pv,add_pmt,perc,periods,n:=0,cap,values):=PROG(cap:=pv,values:
=[[n,cap]],LOOP(n:+1,cap:=(1+perc/100)*cap+add_pmt,values:=APPEND(val
ues,[[n,runde(cap,2)]]),IF(n=periods,RETURN(APPEND([["Periode","Kapit
alstand"]],values)))))
```

Das ist nicht sehr übersichtlich und außerdem fehleranfällig. Ein besonderes Problem stellen die vielen notwendigen Klammern dar, und dabei ist das ja noch gar kein „richtiges" Programm. Bei geschachtelten Schleifen und verschachtelten IF-Konstruktionen kann es ohne hinreichende Übung schon recht unangenehm werden.

Ich empfehle aber trotzdem, dieses kleine Programm auf die konventionelle Art auf den *DERIVE*-Schirm zu zaubern. Achten Sie auf die Einrückungen, die dann auf dem Schirm zu sehen sind, und welche die Struktur des Programms erkennen lassen. Dies ist vorerst Ihre einzige Kontrolle, ob alles stimmt.

Ein Startkapital von 5000 € wird 10 Jahre hindurch um 200 € am Jahresende vermehrt. Wie sieht die Kapitalstandsübersicht aus, wenn 3 1/4 % Zinsen verrechnet werden?

FVPROG(5000, 200, 3.25, 10) =

Periode	Kapitalstand
0	5000
1	5362.5
2	5736.78
3	6123.23
4	6522.23
5	6934.2
6	7359.57
7	7798.75
8	8252.21
9	8720.41
10	9203.82

Ich habe vorhin erwähnt, dass alle in der Parameterliste aufgeführten Variablen „lokale" Variable sind. Sie existieren nicht außerhalb des Programms. Das können Sie leicht sehen, da unter `values` keine „Werte" gespeichert sind:

```
values = values
```

Es gibt aber doch Fälle, in denen „globale" Variable gewünscht sind (z.B. bei der Fehlersuche oder aus problembezogenen Gründen). Dann muss die Variable vor dem Programmaufruf als existent definiert werden und wird nicht in die Parameterliste aufgenommen (später werden wir darauf näher eingehen können):

```
values :=
FVPROG(pv, add_pmt, perc, periods, n := 0, cap) :=
```

Wenn wir nun das Programm ablaufen lassen, dann erkennt man vorerst keinen Unterschied. Aber nachher hat `values` den letzten im Programm zugewiesenen Wert behalten.

```
FVPROG(5000, 2000, 5, 3) =
```

Periode	Kapitalstand
0	5000
1	7250
2	9612.5
3	12093.13

```
values =
```

0	5000
1	7250
2	9612.5
3	12093.13

Was geschieht, wenn wir nach den Perioden noch einen weiteren Wert in der Parameterliste angeben? (An dieser Stelle wird sonst in der Liste die Variable `n` initialisiert.)

```
FVPROG(5000, 2000, 5, 10, 7)
```

Periode	Kapitalstand
7	5000
8	7250
9	9612.5
10	12093.13

Wir erkennen, dass wir damit den default-Wert für `n`, nämlich `n = 0`, überschreiben und ihm von außen den Wert $n = 7$ zuteilen können. Auch das werden wir später zu nutzen wissen.

Herausforderung 1

Schreiben Sie ein Programm für eine Kreditrückzahlung. Gegeben sind die Kreditsumme, die jährliche Rückzahlung (am Jahresende) und der Kreditzins.

Ausgabe in Form einer Tabelle, die für jedes Jahr die offene Restschuld ausweist.

```
kredit(100000, 20000, 10)
```

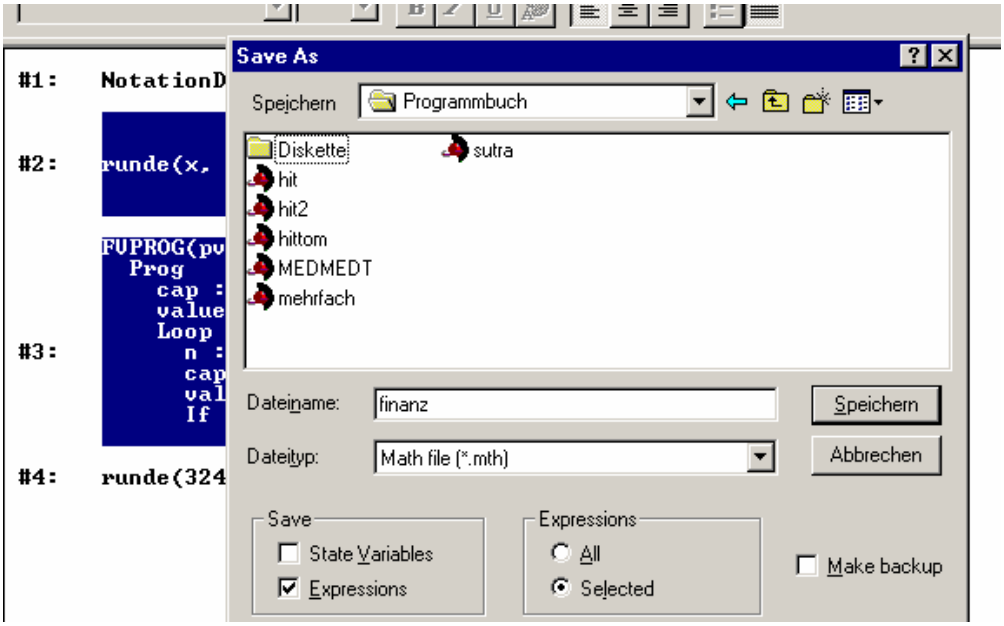
Periode	Kapitalstand
0	100000
1	90000
2	79000
3	66900
4	53590
5	38949
6	22843.9
7	5128.29

Bevor wir nun

weitere Aufgaben lösen, wollen wir überlegen, wie wir diese mühsame Eingabe des Programmcodes umgehen können? Ein alter Trick von mir ist es immer, sich das Format einer Datei genauer anzusehen, wenn dies möglich ist.

Zu diesem Zweck müssen wir eine bereits bestehende *DERIVE*-Datei heranziehen.

Wir markieren die Funktion `runde` und das Programm `FVPROG` und speichern diese beiden Funktionen als MTH-Datei. Dabei werden wir darauf aufmerksam gemacht, dass wir alle Formatierungen verlieren (und nur den reinen Code speichern).



Anschließend inspizieren wir diese Datei im Editor (findet sich im Zubehör der Windows-Programme). Man kann aber auch den Editor auf der DOS-Ebene einsetzen.

Wir sehen eine reine Textdatei, in der die einzelnen Funktionen und Programme durch eine Leerzeile getrennt sind, und falls der Code über eine Zeile hinausgeht, wird der Zeilenumbruch mit einer Tilde (~) gekennzeichnet.

So zeigt sich `finanz.mth` im Editor, wobei die Zeile immer mit dem ~ - Zeichen (der „Tilde“) endet. Wegen der Schriftgröße kann das Zeichen bei Ihnen etwas anders aussehen.

```
runde(x,m):=FLOOR(10^m*x+0.5)/10^m

FVPROG(pv,add_pmt,perc,periods,n:=0,cap,values):=PROG(cap:=pv,values~
:=[[n,cap]],LOOP(n:+1,cap:=(1+perc/100)*cap+add_pmt,values:=APPEND(v~
alues,[[n,runde(cap,2)]]),IF(n=periods,RETURN(APPEND([["Periode","Ka~
pitalstand"]],values)))))
```

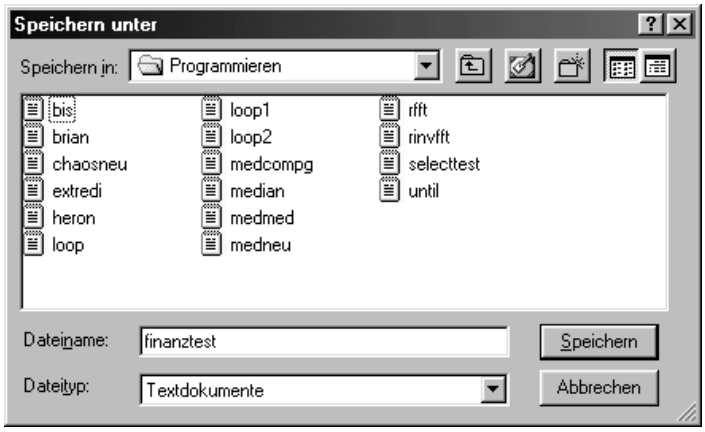
Jetzt werden wir einmal frech und verändern diesen Code nach unseren Vorstellungen – – nicht inhaltlich, aber formal – um dem Programm eine lesbare Struktur zu verleihen. Wenn *DERIVE* anschließend diese „geschönte“ Version versteht, könnten wir ja überhaupt im Editor das Programm schreiben und von dort nach *DERIVE* übernehmen.

Meine Vorstellung von ordentlich lesbar sieht nun so aus:

```
runde(x,m):=FLOOR(10^m*x+0.5)/10^m

FVPROG(pv,add_pmt,perc,periods,n:=0,cap,values):=~
            PROG(~
                cap:=pv,~
                values:=[[n,cap]],~
                LOOP(~
                  n:+1,~
                  cap:=(1+perc/100)*cap+add_pmt,~
                  values:=APPEND(values,[[n,runde(cap,2)]]),~
                  IF(n=periods,RETURN(APPEND([["Periode",~
                                              "Kapitalstand"]],values)))~
                     )~
                 )
```

Diese Textdatei wird unter dem Namen `finanztest` gespeichert.



Wechseln Sie zurück nach *DERIVE* und öffnen Sie die Datei `finanztest`. Sie werden sie weder unter den `.dfw`-, noch unter den `.mth`-Dateien finden. Suchen Sie sie bei dem Dateityp unter “`All files`”.

Nun sollten die bereits bekannten beiden Ausdrücke wieder im Auswahlfenster erscheinen. Speichern Sie die Datei als *DERIVE*-file und das Verfahren ist geglückt. Laden Sie bitte probeweise diese Datei in *DERIVE*. Sie werden bald noch bequemere Editiermöglichkeiten kennen lernen.

Ein wichtiger Hinweis:
Alle *DERIVE*-Befehle und Funktionen müssen, wenn Sie so mit dem Editor arbeiten, in Großbuchstaben geschrieben werden. Das Multiplikationszeichen ist zu schreiben!

Das nächste Programm werden wir vorerst auf diese Weise entwickeln und editieren.

Bereits erfahrene Programmierer werden noch eine Notwendigkeit beim Programmieren vermissen – eine geeignete Dokumentation. Aber auch das lässt sich leichter verwirklichen als angenommen. Hilfreiche Tipps für die Fehlersuche finden Sie auf Seite 25.

2 Die Mathematik der Veden in einem *DERIVE*-Programm

Schon die alten Inder wussten, dass sich jede positive ganze Zahl $x > 2$ als Differenz zweier Quadratzahlen m^2 und n^2 darstellen lässt. „Durch Addition – durch Subtraktion“ nennt sich die entsprechende Sutra, denn ....

..... wenn $x = a * b$, dann ergeben sich m und n als $(a + b)/2$ und $(a – b)/2$.

Unser Programm soll für jede Zahl x alle möglichen ganzzahligen Paare (m,n) ausgeben.

Zusatzaufgaben für Schüler:
- Beweise die Behauptung.
- Für welche Zahlen gibt es kein derartiges Paar?
- Warum muss man nur bis [DIM(v)/2] zählen?

Der Programmcode, wie er sich im Editor präsentieren könnte:

```
"@ NUMBER.MTH aus dem MATH-Verzeichnis muss vorhanden sein"

sutra(x,v,p,i,u:=[],a,b,m,n):=PROG(~
     "@ Plausibilitätskontrolle ",~
     IF(x<2 OR NOT(INTEGER?(x)),RETURN "unmöglich !"),~
     "@ alle Teiler von x werden in einer Liste gesammelt ",~
     v:=DIVISORS(x),~
     "@ die abgerundete 1.Hälfte der Liste wird verwendet ",~
     p:=FLOOR(DIM(v)/2),~
     i:=1,~
     LOOP(~
       IF(i>p,RETURN u),~
       " der i-te Divisor aus der Liste ",~
       a:=v SUB i,~
       b:=x/a,~
       m:=(a+b)/2,~
       n:=ABS((a-b)/2),~
       "@ nur wenn n und m ganz, wird das Paar gelistet ",~
       IF(INTEGER?(n) AND INTEGER?(m),~
                u:=APPEND(u,[['(m^2-n^2)]])~
         ),~
       i:+1~
         )~
     )
```

Hier finden Sie im Code dokumentierende Kommentare eingefügt. Sie stehen unter “ und werden zusätzlich mit dem @ eingeleitet, um sie als Dokumentation kenntlich zu machen.

`DIVISORS` ist eine Funktion, die das Utility-file `NUMBER.MTH` bereit stellt. Falls *DERIVE* ordungsgemäß installiert wurde, wird diese Datei beim ersten Aufruf von `DIVISORS` automatisch in den Hintergrund geladen. `DIVISORS(x)` liefert alle Teiler der Zahl *x* in einer geordneten Liste.

Beachten Sie die Funktion `INTEGER?`, die im Falle eines ganzzahligen Arguments den Wert `true` zurückgibt.

Auf ein kleines Zeichen möchte ich Sie aber besonders aufmerksam machen. Wenn Sie in *DERIVE* die Addition `1+1=` eingeben, dann erhalten Sie natürlich 2. Manchmal wäre es aber wünschenswert, dass *DERIVE* nicht in vorauseilendem Gehorsam gleich alles ausrechnet. Dafür ist der „Quote-Operator“ `'` vorgesehen (Sehen Sie den Unterschied?):

$$1 + 1 = 2$$

$$'(1 + 1) = 1 + 1$$

```
sutra(x, v, p, i, u := [], a, b, m, n) :=
  Prog
    "@ Plausibilitaetskontrolle "
    If x < 2 ∨ ¬ INTEGER?(x)
       RETURN "impossible !"
    "@ alle Teiler von x werden in einer Liste gesammelt "
    v := DIVISORS(x)
    "@ nur die erste Haelfte der Liste wird verwendet "
    p := FLOOR(DIM(v)/2)
    i := 1
    Loop
      If i > p
         RETURN u
      "@ der i-te Divisor aus der Liste "
      a := v↓i
      b := x/a
      m := (a + b)/2
      n := ABS((a - b)/2)
      "@ nur wenn n und m ganzzahlig sind, wird das Paar gelistet "
      If INTEGER?(n) ∧ INTEGER?(m)
         u := APPEND(u, [['(m^2 - n^2)]])
      i :+ 1
```

$$\text{sutra}(12) = \left[\left[4^2 - 2^2\right]\right]$$

Diesen Quote-Operator setzen wir dazu ein, dass die Differenz der Quadrate von *m* und *n* nicht ausgerechnet wird - wir wissen ja, dass sich die gegebene Zahl *x* ergibt -, sondern dass die Differenz der Quadrate wirklich ersichtlich wird. (Einfacher wäre die Ausgabe der Paare [m,n], aber die wäre doch nur halb so schön?)

Die doppelte eckige Klammer bei der Ausgabe im Falle von nur einer Lösung wird Puristen natürlich stören.

Da ich der Meinung bin, dass sich ein Programm durch eine geeignete Form der Ausgabe „verkaufen“ muss, wollen wir diesen kleinen Schönheitsfehler durch eine Abfrage unmittelbar vor der Ausgabe entfernen.

Die IF-Abfrage am Beginn der Schleife wird ersetzt durch die folgende geschachtelte IF-Abfrage:

```
IF(i>p,IF(DIM(u)=1,RETURN 'u SUB 1, RETURN u)),~
```

$$\text{sutra}(12) = \begin{bmatrix} 4^2 & - & 2^2 \end{bmatrix}$$

$$\text{sutra}(123456) = \begin{bmatrix} 30865^2 - 30863^2 \\ 15434^2 - 15430^2 \\ 10291^2 - 10285^2 \\ 7720^2 - 7712^2 \\ 5150^2 - 5138^2 \\ 3866^2 - 3850^2 \\ 2584^2 - 2560^2 \\ 1945^2 - 1913^2 \\ 1310^2 - 1262^2 \\ 691^2 - 595^2 \end{bmatrix}$$

Daneben können Sie die Ausgabe für eine etwas größere Zahl sehen.

Literaturhinweis: [1], [2]

Da in diesem Programm mit einer Liste gearbeitet wird, möchte ich nicht die Gelegenheit versäumen, darauf hinzuweisen, dass *DERIVE* auf der Programmiersprache LISP basiert. LISP ist listenorientiert, und man programmiert besonders *DERIVE*-gerrecht, wenn man die Listenbearbeitungsbefehle von *DERIVE* sinnvoll einsetzt. Ich verdanke diesen wichtigen Hinweis meinem langjährigen Mr. Titbits aus den *DERIVE* Newslettern, Johann Wiesenbauer. Sehen Sie nun auch diese Variante im Editor (nächste Seite).

Ich höre Ihre Fragen: Und wo sind nun die Tilden? Warum schreiben Sie hier die *DERIVE*-Funktionen nicht groß?

Nun, die Antwort ist ganz einfach:

Wenn Sie nur einen einzigen Ausdruck – sei es eine einfache Zuweisung oder eine lange Funktion oder ein umfangreiches Programm – aus dem Editor nach *DERIVE* transferieren wollen, dann können Sie das wie folgt tun:

Markieren Sie den ganzen Ausdruck und fügen sie ihn über die Zwischenablage mit erst `Strg+C` und dann mit `Strg+V` in die *DERIVE*-Eingabezeile ein. Wenn der Ausdruck syntaktisch richtig ist, dann sehen Sie ihn nach einem abschließenden `Enter` im Algebrafenster!

Vergleichen Sie bitte die Behandlung der Liste in der `Loop`-Konstruktion!

```
sutra_2(x,v,p,u:=[],a,b,m,n):=prog(
     "@ Plausibilitätskontrolle ",
     if(x<2 or not(integer?(x)),return "unmöglich !"),
     "@ alle Teiler von x werden in einer Liste gesammelt ",
     v:=divisors(x),
     "@ die abgerundete 1.Hälfte der Liste wird verwendet ",
     p:=floor(dim(v)/2),
          loop(
            if(dim(v)>p,return u),
            "das erste Element von v ",
            a:=first(v),
            b:=x/a,
            m:=(a+b)/2,
            n:=ABS((a-b)/2),
            "@ nur wenn n, m ganz, wird das Paar gelistet ",
            if(integer?(n) and integer?(m),
                        u:=append(u,[['(m^2-n^2)]])
              ),
            "@ das 1.El. wird aus der Liste genommen, mit",
            "@ dem Rest von v wird der Vorgang wiederholt ",
            v:=rest(v)
          )
     )
```

Herausforderung 2

Ein ganz einfaches Beispiel aus der Sekundarstufe 1: Von einer Pyramide mit rechteckiger Basis kennt man die Grundkanten a und b, sowie die Körperhöhe H. Schreiben Sie ein Programm, das neben den Ausgangsdaten auch das Volumen, die Oberfläche und die Gesamtkantenlänge der Pyramide ausgibt.

```
pyramide(3, 5, 10) = [ Pyramide mit a,b,H =  3  5    10 ]
                     [       Volumen =              50 ]
                     [    Oberflaeche =         96.4826 ]
                     [   Kantenlaenge =         57.6653 ]

pyramide(12.35, 21.05, 48.33) = [ Pyramide mit a,b,H =  12.35  21.05   48.33 ]
                                [       Volumen =                    4188.07 ]
                                [    Oberflaeche =                   1896.44 ]
                                [   Kantenlaenge =                   266.186 ]
```

Auch Ausgabe-, bzw. Berechnungsvereinbarungen lassen sich ins Programm integrieren.

```
Precision := Approximate,
NotationDigits := 6,
```

Diese werden aber erst nach erstmaligem Aufruf des Programms wirksam!

3 Milton´s Folge – ein Souvenir aus Kolumbien

Milton Lesmes Acosta, ein lieber Kollege aus Kolumbien, stellte mir die folgende *DERIVE*-Aufgabe [3]:

> Man startet mit einer beliebigen ganzen positiven Zahl und bildet die Summe der Quadrate der Ziffern. Das ergibt die nächste Zahl, von der wieder die Quadratsumme gebildet wird, usw. Dabei mündet man entweder bei der Zahl 1 oder bei 16, die dann den Anfang einer Periode bildet.
>
> Erstelle das zugehörige Programm und untersuche die Längen der Folgen für die ersten 10000 ganzen Zahlen.

Wenn man z.B. mit 128 beginnt, dann erzeugt man die Folge [128, 69, 117, 51, 26, 40, 16]

$$
\begin{array}{ll}
128 & 1^2 + 2^2 + 8^2 = 69 \\
69 & 6^2 + 9^2 = 117 \\
117 & 1^2 + 1^2 + 7^2 = 51 \\
51 & 5^2 + 1^2 = 26 \\
26 & 2^2 + 6^2 = 40 \\
40 & 4^2 + 0^2 = 16 \\
16 & 1^2 + 6^2 = 37 \\
 & \vdots
\end{array}
$$

```
sixteen(n, seq := []) :=
  Prog
    Loop
      seq := APPEND(seq, [n])
      "@ Endabfrage "
      If n = 16 ∨ n = 1
        RETURN seq
      " @"
      n := Σ(VECTOR(CODES_TO_NAME(k_)^2, k_, NAME_TO_CODES(n)))
      "@ Ende der Schleife "
    "@ Ende des Programms "

                          sixteen(128) = [128, 69, 117, 51, 26, 40, 16]
```

So kann man auch sofort die Periode erkennen, die mit 16 begonnen wird.

```
sixteen(37) = [37, 58, 89, 145, 42, 20, 4, 16]
```

Der Kern des Problems liegt darin, wie man die Zahl in ihre Dezimalstellen zerlegt, um deren Quadrate zu bilden, die dann summiert werden. Wie aus dem Programm ersichtlich, geschieht dies alles in der drittvorletzten Zeile, die ich aber näher erklären will.

Mit `NAME_TO_CODES(x)` wird eine Zeichenfolge *x* (String oder Zahl, aber keine Variable) in eine Liste der ASCII-Codes der einzelnen Zeichen umgewandelt.

```
NAME_TO_CODES(Josef) = [74, 111, 115, 101, 102]
NAME_TO_CODES(j·o·s·e·f) = NAME_TO_CODES(e·f·j·o·s)
NAME_TO_CODES(128) = [49, 50, 56]
```

Der erste ″`Josef`″ wurde unter Doppelhochkommata ″ eingegeben, die aber am *DERIVE*-Schirm nicht mehr gezeigt werden. Beim zweiten `Josef` ohne die beiden ″ arbeitet die Funktion nicht nach Wunsch, wogegen für die numerische Größe 128 wieder die Liste der Zahlencodes für 1, 2 und 8 ausgegeben wird.

```
VECTOR(CODES_TO_NAME(k_)^2, k_, [49, 50, 56]) = [1, 4, 64]
```

Diese Liste steht also am Ende des `VECTOR`-Befehls, in dem die „Umkehrfunktion“ `CODES_TO_NAME(k_)` jedes Element `k_` dieser Liste wieder in die Zahl rückverwandelt, die dann quadriert wird.

Der Summationsbefehl addiert alle Komponenten der auf diese Weise entstandenen Liste, womit die nächste Zahl erhalten wird und die Schleife ein nächstes Mal durchlaufen werden kann.
Die Eingabe des Programmcodes ist nun einfach, da das Programm sehr kurz ist:

```
sixteen(n,seq:=[]):=PROG(
                      LOOP(
                        seq:=APPEND(seq,[n]),
                        "@ Endabfrage ",
                        IF(n=16 OR n=1,RETURN(seq)),
                        " @",
                        n:=SUM(VECTOR(CODES_TO_NAME(k_)^2,k_,
                                      NAME_TO_CODES(n))),
                      "@ Ende der Schleife "
                      ),
                    "@ Ende des Programms "
                    )
```

Die alleinstehenden Klammern machen das Ende des Loops und des Programms deutlich.

Beachten Sie bitte, dass `k_` als Schleifenzähler nicht in die Parameterliste aufgenommen werden muss!

Anschließend wollen wir die Längen der entstehenden Folgen untersuchen.

```
TABLE(sixteen(x), x, 100, 110)
```

100	[100, 1]
101	[101, 2, 4, 16]
102	[102, 5, 25, 29, 85, 89, 145, 42, 20, 4, 16]
103	[103, 10, 1]
104	[104, 17, 50, 25, 29, 85, 89, 145, 42, 20, 4, 16]
105	[105, 26, 40, 16]
106	[106, 37, 58, 89, 145, 42, 20, 4, 16]
107	[107, 50, 25, 29, 85, 89, 145, 42, 20, 4, 16]
108	[108, 65, 61, 37, 58, 89, 145, 42, 20, 4, 16]
109	[109, 82, 68, 100, 1]
110	[110, 2, 4, 16]

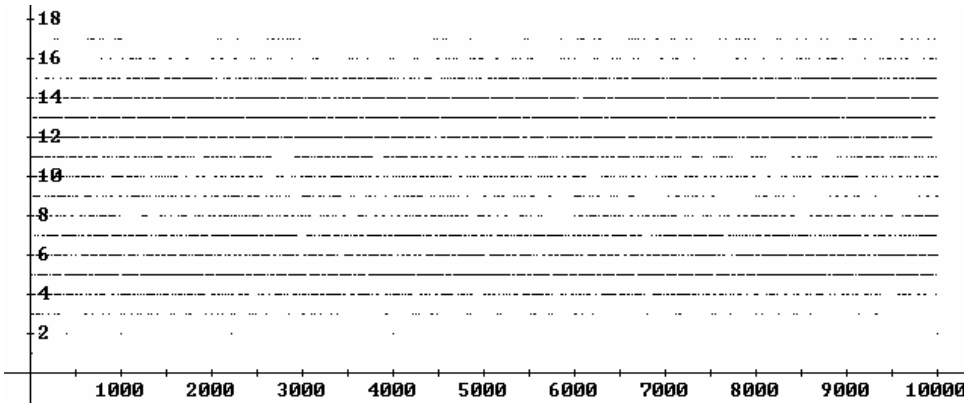
```
TABLE(DIM(sixteen(x)), x, 1000, 1010)
```

1000	2
1001	4
1002	11
1003	3
1004	12
1005	4
1006	9
1007	11
1008	11
1009	5
1010	4

Mit `TABLE(sixteen(...)` erhalten wir die Folgen, wogegen uns der zusätzliche `DIM` -Befehl nur die Anzahlen der Elemente ausgibt. Und diese Tabelle der Folgenlängen wollen wir graphisch für die ersten 10000 ganzen Zahlen darstellen lassen.

Lassen Sie aber bitte die Tabelle nicht im Algebrafenster ausgeben, sondern wechseln Sie unmittelbar ins 2D-Zeichenfenster, aktivieren Sie die *Option Approximate Before Plotting*, passen Sie die Skalierung an und lassen Sie die Punkte *Small* und *Connect No* erscheinen. Dann sollte die nach einiger Rechenzeit auftauchende Grafik ein recht chaotisches Bild darstellen. Jedenfalls ist unter den ersten 10000 Folgen keine länger als 17 Elemente!

```
TABLE(DIM(sixteen(x)), x, 0, 10000)
```



Das entstehende „Muster“ weist keinerlei Regelmäßigkeit auf, es scheint chaotisch zu sein. Eine geeignete Ausschnittvergrößerung macht das noch deutlicher. Auch die Untersuchung der Häufigkeit der auftretenden Folgenlängen - unter Verwendung der SELECT-Funktion - gibt keinen Hinweis auf eine besondere Verteilung:

```
TABLE(DIM(SELECT(k_ = x_, k_, VECTOR(DIM(sixteen(x)), x, 1, 1000))), x_, 2, 20)`
```

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
6	21	69	94	90	81	51	38	67	87	110	135	97	34	6	12	0	0	0

Ich habe in diesem Beispiel die beiden nützlichen Stringoperationen CODES_TO_NAME und NAME_TO_CODES vorgestellt. Hier sind sie aber nicht unbedingt erforderlich, da auf alle Zeichenketten (und auch auf Zahlen) die Vektoroperationen angewendet werden können.

```
128 sub 2 = 2
```

Daher lässt sich die entscheidende Programmzeile, die aus einem Folgenelement das nächste macht, umschreiben in:

```
n:=vector(n sub k,k,dim(n))^2.
```
[*]

Herausforderung 3

Das 3n+1 Problem oder die Collatzfolge [4]

Eine Zahlenfolge wird auf folgende Weise erzeugt: Der Benutzer gibt eine beliebige positive ganze Zahl ein; falls die Zahl gerade ist, wird sie halbiert, anderenfalls wird sie verdreifacht und um eins vermehrt. Diese so gewonnene neue Zahl ist das nächste Folgenelement; falls sie gerade ist, wird sie halbiert, anderenfalls .....
Die Zahlenfolge endet, sobald der Wert 1 erreicht ist.
Versuchen Sie 27, oder 26623, oder 60965, oder .....

Erzeugen Sie ein Übersicht der Folgenlängen für einen bestimmtes Intervall der ganzen Zahlen zusammen mit einer geeigneten graphischen Darstellung.

Herausforderung 4

Eine andere Zahlenfolge kann so definiert werden: Ein Startwert (ganz, positiv und durch drei teilbar) wird vorgegeben. Die nächste Zahl ist die Summe der dritten Potenzen ihrer Ziffern, usw. Erzeugen und untersuchen Sie derartige Folgen! [5]

[*] Rüdeger Baumann hat im *DERIVE Newsletter* #47 dieses Problem und eine interessante Verallgemeinerung auf rekursive Weise behandelt [32].

4 „Heronsche Dreiecke“ – Geschachtelte Schleifen

René Hugelshofer schrieb einen schönen Beitrag für den International *DERIVE* & *TI*-92 *Newsletter* mit dem Titel „*Dynamic Algebra*“ [6], in dem er *Heronsche Dreiecke* als solche definierte, in denen sowohl die Seitenlängen, als auch der Flächeninhalt ganzzahlige Werte annehmen (z.B., [3,4,5]). In einem Nebensatz machte er die Bemerkung, dass man auf dem *TI*-92 ein Programm zur Auflistung aller derartigen Dreiecke bis zu einer vorgegebenen maximalen Seitenlänge *s* schreiben könnte.

Dies war Anlass genug für mich, das Problem mit *DERIVE* zu lösen. Da man für alle drei Seiten alle möglichen ganzzahligen Werte $\leq s$ untersuchen muss, ergibt sich zwangsläufig die Notwendigkeit mit geschachtelten Schleifen zu arbeiten.

Nun möchte ich Sie mit einem weiteren, sehr mächtigen Werkzeug bekannt machen, das das Programmieren mit *DERIVE* ganz entscheidend vereinfachen kann.

Vor einiger Zeit erhielt ich im Rahmen meiner Arbeit für die *DUG* (*DERIVE User Group*) einen Brief von Herrn Walter Schiller aus Paderborn. Ich zitiere den Anfang des Schreibens:

> *..... ich habe einen einfachen Compiler geschrieben, der für die Programmierung von DERIVE-Prozeduren hervorragend eingesetzt werden kann. Der beiliegende Artikel beschreibt die einfache Syntax, begleitet mit einfachen Beispielen ....*

Ich war neugierig und begeistert zugleich und arbeite seither auch mit diesem Werkzeug, das Herr Schiller den Mitgliedern der *DUG* zur Verfügung gestellt hat. Er nannte den Compiler ganz einfach „TOM.EXE“ (TOM = TO Math) [7].

Inzwischen steht nach einem regen Briefwechsel die bereits sehr ausgereifte Version 3 von TOM für alle Mitglieder der *DUG* bereit.

Sehen Sie nun, wie die Heronschen Dreiecke in einem Quelltext für TOM.EXE, der wieder mit jedem Editor verfasst werden kann, erzeugt werden können:

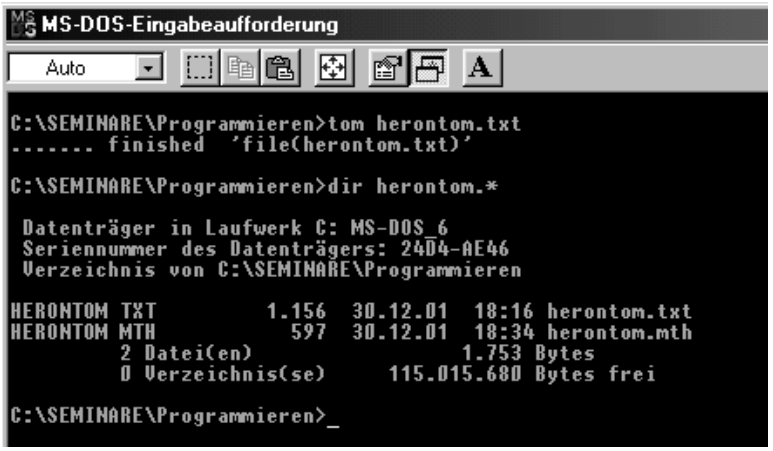
```
heron_triangle(x):=prog
             /* das ist die Überschrift */
                res:=[["a","b","c","FLAECHE"]]
                /* die Seiten werden drei Schleifen durchlaufen */
                loop c:= 1 to x by 1
                  loop b := 1 to x by 1
                    loop a := 1 to x by 1
                      /* nun folgt die Heronsche Flächenformel */
                      s:= (a+b+c)/2
                      ar:=sqrt(s*(s-a)*(s-b)*(s-c))
                      /*
                         nur im Falle von ganzzahligen und
                         nichtverschwindenden Flächeninhalten liegt
                         ein "Heronsches Dreieck vor
                                                                */
```

```
            if integer?(ar) and ar > 0
                 then
               res:= append(res,[[a,b,c,ar]])
          end
        end
      end
      /* die drei Schleifen werden geschlossen */
      /* Ausgabe der möglichen Dreiecke */
      return res
      end
```

Aller Text zwischen /* und */ sind Kommentare. Zeilenumbrüche können beliebig eingefügt werden, um den Code lesbarer zu machen. Der Compiler erzeugt die Liste der lokalen Variablen aus dem Quelltext heraus. Damit kann auch keine mehr vergessen werden. Globale Variable können zusätzlich definiert werden. Beachten Sie weiters die handliche Formulierung der Loops und der IF-Konstruktion. TOM bietet noch einige andere sehr praktische Möglichkeiten. Einen weiteren – gelungenen – Versuch finden Sie im Kapitel 11.

Nachdem der Text editiert und gespeichert wurde – hier unter dem Namen `herontom.txt` – wird der Quelltext in der DOS-Umgebung oder aus WINDOWS heraus compiliert.

Anschließend sollte die entsprechende MTH-Datei von *DERIVE* erkannt und geladen werden können.



Ich zeige im Folgenden nur einen kleinen Ausschnitt des compilierten *DERIVE*-Programms. Es sieht merkwürdig genug aus.

```
heron_triangle(x, c, s, "$start1", "$by3", "$to3",
  Prog
    res := [["a", "b", "c", "FLAECHE"]]
    "$start1" := TRUE
    c := 1
    "$to1" := x
    "$by1" := 1
    Loop
      If "$start1"
         "$start1" := FALSE
```

Wir erkennen eine Menge von neuen Variablen, die zur Übersetzung der Schleifen verwendet werden. Diese sind aber Strings. Auch Strings können als Variablenbezeichnungen verwendet werden!! Herr Schiller brauchte neue Variable, die aber auf keinen Fall mit anderen, vom Programmierer verwendeten Variablenbezeichnungen in Konflikt geraten dürften. Daher schaffte er künstliche Variable, die mit einem $-Zeichen beginnen, dies ist aber nur als String möglich.

Nun wollen wir noch schauen, ob das Programm auch funktioniert!

heron_triangle(6) =

a	b	c	FLAECHE
5	4	3	6
4	5	3	6
5	3	4	6
3	5	4	6
4	3	5	6
3	4	5	6
6	5	5	12
5	6	5	12
5	5	6	12

Im Prinzip erfüllt es unseren Zweck, allerdings brauchen wird die Kombination [3,4,5] nicht in allen Permutationen aufgelistet, einmal wird ja genügen.

Es fehlt also noch eine Routine, die diese Matrix, beginnend mit der zweiten Zeile auf das Mehrfachvorkommen von Zeilenvektoren überprüft.

Dies ist im nächsten Quelltext bereits berücksichtigt, den ich wieder so verfasst habe, dass er auch ohne Compiler von *DERIVE* verstanden werden kann:

```
heron(x,res:=[["a","b","c","AREA"]],a:=1,b:=1,c:=1,
              s,ar,i_:=2,j_:=3):=
              PROG(
              "@ Schleife fuer die Seite a ",
              LOOP(
                IF(a>x,exit),
                "@ Schleife fuer die Seite b ",
                LOOP(
                  IF(b>x,exit),
                  "@ Schleife fuer die Seite c ",
                    LOOP(
                      IF(c>x,exit),
                      "@ die Heron'sche Flaechenformel ",
                      s:=(a+b+c)/2,ar:=SQRT(s*(s-a)*(s-b)*(s-c)),
                      IF(ar>0 AND INTEGER?(ar),
                         res:=APPEND(res,[APPEND(SORT([a,b,c]),
                         [ar])])),
                         c:+1
                    ),
                  "@ c wird auf 1 zurückgesetzt und b um 1 erhöht ",
                  c:=1,
                  b:+1
                ),
```

```
      "@ b und c werden zurückgesetzt und a erhöht ",
      c:=1,
      b:=1,
      a:+1
    ),
    "@ Ende der äußersten Schleife ",
    "@ mehrfach vorkommende Tripel werden entfernt",
    LOOP(
      IF(i_=DIM(res),exit),
      LOOP(IF(j_>DIM(res),exit),
        IF(res SUB i_=res SUB j_ AND res SUB j_/=[[0]],
          res:=REPLACE(0,res,j_)),
          j_:+1~
      ),
      i_:+1,
      j_:=i_+1
    ),
    RETURN(SELECT(k/=0,k,res))
    )
```

Hier sehen Sie deutlich die Unterschiede zur Arbeit mit dem Compiler im Umgang mit lokalen Variablen in der Parameterliste, mit den Loops (Schleifen), mit der IF-Konstruktion und den Kommentaren. Auch die Klammer(n)setzung wird vom Compiler übernommen.

Auch hier wird nur ein Teil des *DERIVE*-Schirms gezeigt, zusammen mit einem neuerlichen Testlauf des Programms:

```
      "@ c wird auf 1 zurueckgesetzt und b um 1 erhoeht "
      c := 1
      b :+ 1
    "@ b und c werden zurückgesetzt und a erhöht "
    c := 1
    b := 1
    a :+ 1
  "@ Ende der äußersten Schleife "
  "@ zuletzt werden mehrfach vorkommende Tripel entfernt"
  Loop
    If i_ = DIM(res) exit
    Loop
      If j_ > DIM(res) exit
      If res↓i_ = res↓j_ ∧ res↓j_ ≠ [[0]]
        res := REPLACE(0, res, j_)
      j_ :+ 1
    i_ :+ 1
    j_ := i_ + 1
  RETURN SELECT(k ≠ 0, k, res)
```

$$\text{heron}(6) = \begin{bmatrix} a & b & c & AREA \\ 3 & 4 & 5 & 6 \\ 5 & 5 & 6 & 12 \end{bmatrix}$$

$$\text{heron}(10) = \begin{bmatrix} a & b & c & AREA \\ 3 & 4 & 5 & 6 \\ 5 & 5 & 6 & 12 \\ 5 & 5 & 8 & 12 \\ 6 & 8 & 10 & 24 \end{bmatrix}$$

Ich ersetze die wiederholt auftretenden Tripel zuerst durch eine 0 und selektiere dann alle von 0 verschiedenen Elemente der Ergebnisliste. Eine andere Möglichkeit, mehrfach auftretende Elemente aus einer Liste zu entfernen, finden Sie in Herausforderung 17, bzw. in der angebotenen Lösung angesprochen.

Eine weitere Lösung wurde von Rüdeger Baumann zur Verfügung gestellt, wofür ich ihm herzlich danke. (heron_baumann.dfw)

Ich möchte Ihnen aber auf keinen Fall die Lösung vorenthalten, die mir Johann Wiesenbauer angeboten hat. Beachten Sie vor allem den souveränen Umgang mit den Listen!

```
heron2(s, a_, s_ := []) :=
  Prog
    a_ := VECTOR(SORT(u_), u_, POWER_SET({1, ..., s}, 3))
    s := APPEND(a_, VECTOR(SORT(ADJOIN(FIRST(u_), u_)), u_, {1, ..., s}^2))
    Loop
      If s = []
        RETURN ADJOIN(["a", "b", "c", "AREA"], SORT(s_))
      a_ := √(Π([1, 1, 1; -1, 1, 1; 1, -1, 1; 1, 1, -1]·FIRST(s)))/4
      If a_ > 0 ∧ INTEGER?(a_)
        s_ := ADJOIN(APPEND(FIRST(s), [a_]), s_)
      s := REST(s)
```

Diese ist etwa um den Faktor 10 schneller als meine erste und noch immer 5 mal schneller als meine Lösung, die zur Herausforderung 17 gehört.

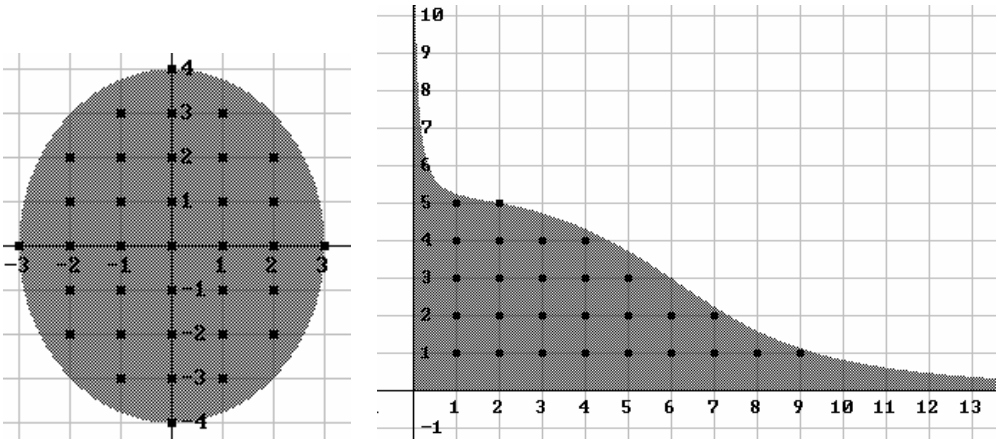
Herausforderung 5

Schreiben Sie ein Programm zur Lösung von Ungleichungen mit zwei (oder drei) Variablen über einer Teilmenge von Z x Z (bzw. Z x Z x Z).

Stellen Sie die Lösungsmengen auch graphisch dar und überzeugen Sie sich von der Richtigkeit Ihres Programms.

$$\text{ungl}(16\cdot x^2 + 9\cdot y^2 \le 144,\ x,\ -5,\ 5,\ y,\ -5,\ 5)$$

$$\text{ungl}\left(\frac{4\cdot x + y}{|x\cdot y|} > \frac{x^2 + y^2}{\sqrt{(50\cdot x\cdot y)}},\ x,\ 1,\ 10,\ y,\ 1,\ 10\right)$$



Tipps für das Entwanzen (Debuggen)!

- Die Funktionsweise eines Programms läßt sich schrittweise überprüfen, indem man an der entsprechenden Stelle mit `RETURN` eine Programmausgabe erzwingt. Das ist besonders interessant bei Schleifen, da man damit auch der Ursache von Endlosschleifen auf den Grund gehen kann. Auch der jeweilige Stand von lokalen Variablen kann angezeigt werden.
- Variable können auch abgefragt werden, indem man sie „globalisiert“ (siehe Hinweis auf Seite 9).
- Kritische Programmteile können mit Doppelhochkommata zu Kommentaren „degradiert“ und damit aus dem Programmfluss ausgenommen werden.

```
test(zahl, grenze, i := 0) :=
  Prog
    Loop
      i :+ 2
      If i > grenze exit
      zahl := zahl + i
      RETURN [zahl, i]
    RETURN zahl
```

```
test(10, 8) = [12, 2]
```

```
test(zahl, grenze, i := 0) :=
  Prog
    Loop
      i :+ 2
      If i > grenze exit
      zahl := zahl + i
      "RETURN [zahl, i]"
    RETURN zahl
```

```
test(10, 8) = 30
```

5 Kennen Sie die MedMed-Regression ?

Viele von Ihnen, die sich schon mit Regressionslinien auf den diversen Taschencomputern beschäftigt haben, mögen sich schon gefragt haben, was es mit der „Median-Median-Regression“ auf sich hat. Sie wird auch in den Statistik-Lehrbüchern erwähnt, wobei jeweils aber meist nur die Computerergebnisse angeführt werden. Ich verdanke meinem Kollegen Fritz Tinhof aus Eisenstadt den Hinweis auf eine Homepage [8] – die es mittlerweile nicht mehr gibt – auf der die Berechnung dieser Regressionslinie sehr schön erklärt war. So wie der Median einer Stichprobe allfällige Ausreißer weniger berücksichtigt als der Mittelwert, tut dies diese Regressionslinie auch im Vergleich zur linearen Regression.

Die Durchführung der Median-Median-Regression erfordert nur die Kenntnis des Medians und der Zweipunkt-, bzw. Punkt-Richtungsform der Geraden. Daher ist sie auch eine schöne Anwendung der linearen Funktion im Rahmen des Mathematikunterrichts [9].

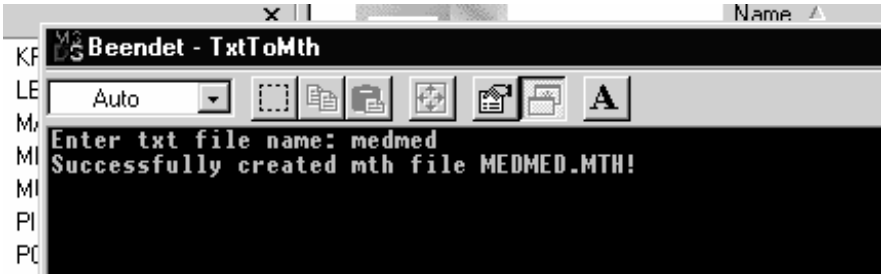
Der folgende Code ist hoffentlich weitgehend selbsterklärend, doch vorher gibt es noch ...

... eine Editierhilfe von Albert Rich

Wenn Sie das folgende Listing betrachten, werden Ihnen einige mit einem Strichpunkt eingeleitete Zeilen auffallen. Albert Rich schrieb für die Programmiergemeinde ein Hilfsprogramm TxtToMth.EXE, das txt-Dateien in MTH-Dateien konvertiert und Kommentare an beliebigen Stellen enthalten darf. Diese sind mit einem führenden ; zu versehen.

Ist das Programm editiert, ruft man im Kommandomodus dieses Dienstprogramm auf – durch einfachen Doppelklick im Explorer, gibt den Namen der txt-Datei ein und erhält eine gleichnamige MTH-Datei zurück. Großschreibung der *DERIVE*-Funktionen ist nicht notwendig.

Die einzelnen *DERIVE*-Ausdrücke sind durch Leerzeilen zu trennen.



Der Bildschirmausschnitt illustriert die Vorgangsweise.

```
;zuerst wird eine Funktion median(liste) erzeugt
median(liste,n):=PROG(
                      list:=SORT(liste),
                      n:=DIM(liste),
                      IF(MOD(n,2)=1,list SUB ((n+1)/2),
                        (list SUB (n/2)+list SUB (n/2+1))/2)
                          )
;diesen Kommentar sieht man im DERIVE-file nicht mehr

;aber die nächste Zeile schon
"@ mdreg(liste) erzeugt die MedMed-Regressionsgerade"

mdreg(liste,n,fl,r,grg,gr1,gr2,gr3,d2,p1,p2,p3,m_):=PROG(
    "@ die Liste wird nach steigenden x-Werten sortiert ...",
    liste:=SORT(liste),n:=DIM(liste),
    "@ ... und in moeglichst gleiche Gruppen geteilt ",
    fl:=FLOOR(n/3),r:=MOD(n,3),
    IF(r=0,grg:=[fl,2*fl],
       IF(r=1,grg:=[fl,2*fl+1],
              grg:=[fl+1,2*fl+1])),
    "@ die Gruppengrenzen richten sich nach Divisionsresten ",
    ;wird nicht nach DERIVE übertragen
    gr1:=liste ROW [1,...,grg SUB 1],
    gr2:=liste ROW [grg SUB 1+1,...,grg SUB 2],
    gr3:=liste ROW [grg SUB 2+1,...,n],
    "@ die Mediane der x- und y-Werte bestimmen 3 Punkte p1 bis p3",
    p1:=[median(gr1 COL 1),median(gr1 COL 2)],
    p2:=[median(gr2 COL 1),median(gr2 COL 2)],
    p3:=[median(gr3 COL 1),median(gr3 COL 2)],
    "@ p1 und p3 legen den Anstieg m_ der Regressionsgeraden fest ",
    m_:=(p3 SUB 2-p1 SUB 2)/(p3 SUB 1-p1 SUB 1),
    "@ d1 ist der Abschnitt auf der y-Achse ",
    d1:=p3 SUB 2-m_*p3 SUB 1,
    "@ durch p2 geht eine Parallele und bildet den Abschnitt d2 ",
    d2:=p2 SUB 2-m_*p2 SUB 1,
    "@ die endgueltige Gerade hat den Anstieg m_ und einen
           gewichteten Abschnitt",
    "@ und es ergibt sich die Regressionslinie: ",
    EXPAND(m_*x+(2*d1+d2)/3))

; medreg erzeugt die Ausgabe der Fehlerquadrate
medreg(liste,sse):=PROG(
       sse:=SUM((liste SUB i SUB 2-LIM(mdreg(liste),x,
                 liste SUB i SUB 1))^2,i,1,DIM(liste)),
       [["MedMed-Gerade:",mdreg(liste)],["SummeRes^2:",sse]])

; auch die Daten können gleich über den Editor eingegeben werden
hp:=[75,27;175,12;135,16;100,22;125,19;85,15;160,10;80,25]
```

Dieser Text erzeugt zuerst die Funktion `median(liste)`, anschließend im Herzstück die Median-Median-Regressionsgerade mit `mdreg(tabelle)` und schließlich mit `medreg(tabelle)` die Ausgabe mit der Summe der Fehlerquadrate.

Ein Hinweis zur Erzeugung der Gruppen 1 - 3 sei gegeben:

Bei 15 Datenpaaren ist *fl* = `FLOOR(15/3) = 5` und der Divisionsrest von 15 / 3 = 0 → *r* = `MOD(15,3)` = 0. Alle Gruppen haben gleich viele Elemente, nämlich 5, 5 und 5.

Bei 16 Datenpaaren ist *fl* = 5, aber r = *1*, die Gruppengrößen sind 5, 6 und 5.

Bei 17 Datenpaaren ist *fl* = 5, *r* = 2 und die Gruppengrößen sind 6, 5, 6.

Beachten Sie die Matrizenoperationen `ROW` und `COL`, mit denen man Zeilen und Spalten einer Matrix ansprechen kann.

Ich strebe eine Ausgabe in der folgenden Form an:

```
mdreg(hp) = 38.26041666 - 0.1625·x

                [ MedMed-Gerade:  38.26041666 - 0.1625·x ]
medreg(hp) =    [                                        ]
                [   SummeRes^2:          101.2508680     ]

FIT([x, a·x + b], hp) = 33.70730948 - 0.1322550544·x
```

Hier wurde zusätzlich auch die *lineare Regression* für die Datentabelle `hp` durchgeführt.

Der `median`-Baustein sollte am *DERIVE*-Schirm die folgende Gestalt haben:

```
median(liste, n) :=
  Prog
    list := SORT(liste)
    n := DIM(liste)
    If MOD(n, 2) = 1
       list↓((n + 1)/2)
       (list↓(n/2) + list↓(n/2 + 1))/2
```

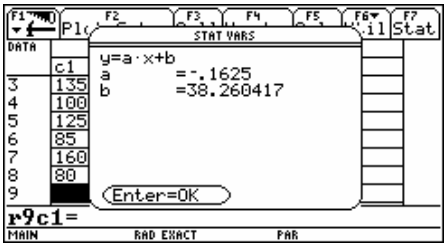
Testen Sie bitte diese Funktion mit einigen Listen!

Nun folgt das Kernprogramm inklusive der nötigen Erklärungen über die Median-Median-Prozedur. Das ist der Beginn des *DERIVE*-Erscheinungsbildes:

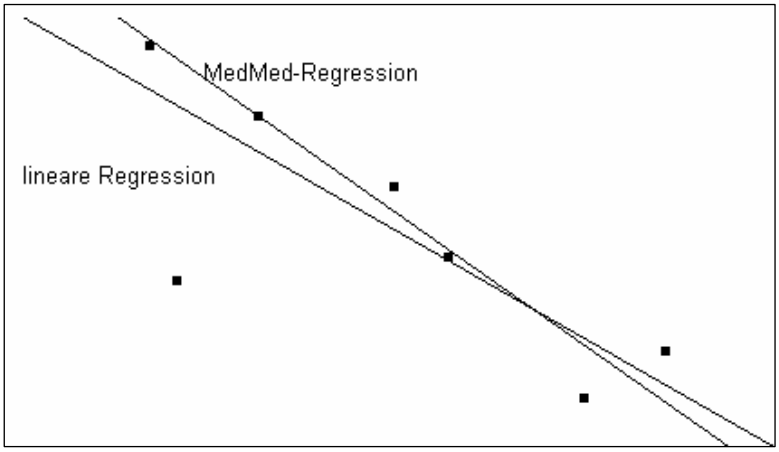
```
mdreg(liste, n, fl, r, grg, gr1, gr2, gr3, d2, p1, p2, p3, m_) :=
  Prog
    "@ die Liste wird nach steigenden x-Werten sortiert ..."
    liste := SORT(liste)
    n := DIM(liste)
    "@ ... und in moeglichst gleiche Gruppen geteilt "
    fl := FLOOR(n/3)
    r := MOD(n, 3)
    If r = 0
       grg := [fl, 2·fl]
       If r = 1
          grg := [fl, 2·fl + 1]
          grg := [fl + 1, 2·fl + 1]
```

$$hp := \begin{bmatrix} 75 & 27 \\ 175 & 12 \\ 135 & 16 \\ 100 & 22 \\ 125 & 19 \\ 85 & 15 \\ 160 & 10 \\ 80 & 25 \end{bmatrix}$$

Wenn Sie dann eine Datenmatrix wie die nebenstehende eingegeben haben und `medreg(daten)` aufrufen, sollten Sie ein Ergebnis wie auf der vorigen Seite gezeigt erhalten.



Zum Vergleich habe ich die Ausgabe am *TI*-92 Schirm abgebildet.
Zu guter letzt werden alle Daten und Ergebnisse geplottet und man kann deutlich den Unterschied zwischen linearer und Median-Median-Regression erkennen.



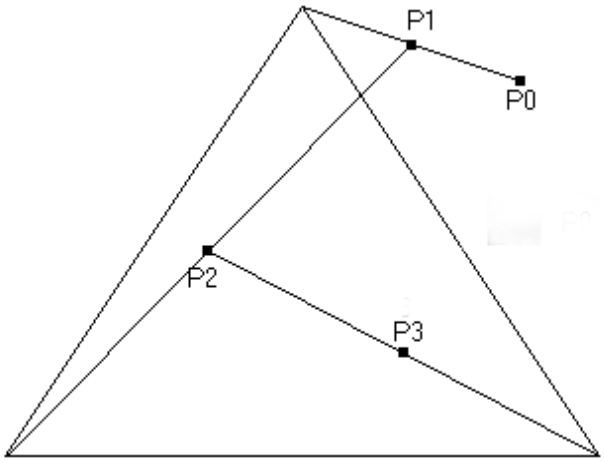
Herausforderung 6

Stellen Sie auch für andere Arten von Regressionslinien eine Ausgabe wie auf Seite 29 her, wie z.B.

```
cubicreg(liste)
[ Regressionslinie:  0.00605019·x^3 + 0.48887·x^2 + 1.53201·x + 4.05241 ]
[    SummeRes^2:                                123.725                  ]
[     BestMaß:                                  0.998516                 ]
linreg(liste2)
[ Regressionslinie:  33.7073 - 0.132255·x ]
[    SummeRes^2:            83.766       ]
[    KorrKoeff:           0.822922       ]
[     BestMaß:            0.677201       ]
```

6 Das Chaos – Spiel

In der Ebene wird ein gleichseitiges Dreieck ΔABC mit $A = (0|0)$ und $B = (1|0)$ gezeichnet. Ein beliebiger Startpunkt P_0 in der Ebene wird festgelegt.



Jetzt kann das Spiel beginnen:

Eine der drei Ecken des Dreiecks wird zufällig gewählt und der Mittelpunkt der Strecke zu dieser Ecke ist der nächste Punkt P_1; nun wird wieder eine Ecke zufällig gewählt und die Strecke von P_1 zu dieser Ecke wieder halbiert $\rightarrow P_2$, usw. [10], [11]

Probieren Sie, 20 Punkte auf diese Weise händisch zu konstruieren und verwenden Sie einen gewöhnlichen Spielwürfel für die Zufallsauswahl. Die Punktmenge scheint sich chaotisch innerhalb der Dreiecksgrenzen auszubreiten!

Wenn wir sehen wollen, wie dieses Chaos nach 100, 1000 oder gar 10000 Punkten aussieht, müssen wir wohl den Computer bemühen.

```
chaos(n,v_,xn,yn,ze,i:=1,zz,pts):= PROG(
               "@ das Startdreieck ",
               v_:=[[0,0],[1,0],[1/2,3^(1/2)/2]],
               "@ ein Zufallspunkt mit -1 <= x <= 2, -1 <= y <= 2 ",
               "@ RANDOM(1) ist eine Zufallszahl > 0 und < 1 ",
               xn:=3*RANDOM(1)-1,yn:=3*RANDOM(1)-1,
               "@ das ist der Startpunkt: ",
               pts:=[[xn,yn]],
               LOOP(
                 "@ RANDOM(3) gibt eine Zufallszahl aus {0,1,2} ",
                 "@ zz ist die Nr der Zufallsecke und ze ist diese ",
                 zz:=RANDOM(3)+1,
                 ze:=v_ SUB zz,
                 xn:=(xn+ze SUB 1)/2,yn:=(yn+ze SUB 2)/2,
                 pts:=APPEND(pts,[[xn,yn]]),
                 "@ Endabfrage fuer die Schleife ",
                 IF(i=n,RETURN(pts),i:+1)
                   )
                 )
```

Die Erzeugung des jeweils neuen Punktes kann auch über die Vektorrechung erfolgen. Das entsprechende Programm findet sich in der Datei `chaos.dfw`.

Auch dieses Programm hat noch einen Schönheitsfehler: Nehmen wir einmal an, dass Sie in einer Schulklasse diese Prozedur gemeinsam entwickeln und dann testweise die ersten 5 Punkte darstellen lassen. Von Zufall keine Spur. Alle Schüler – und auch Sie selbst – werden dieselbe Konfiguration auf Ihrem Schirm sehen. Und wenn Sie es das nächste Mal nach einem Neustart von *DERIVE* wieder probieren, dann sehen Sie wieder dieselben fünf Punkte.

Daran muss man immer dann denken, wenn man mit dem Zufallsgenerator arbeitet. Für Testzwecke ist es oft gar nicht schlecht, immer dieselben Daten zu erhalten, aber später sollte wirklich der Zufall walten – z.B., wenn man Simulationsprogramme erstellt.

Entweder, man simplifiziert am Beginn der Sitzung

```
RANDOM(0) = 3151648782
```

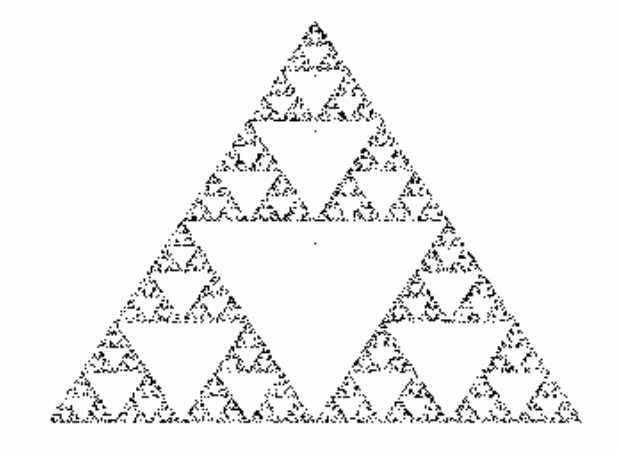
und intialisiert damit den Zufallsgenerator mit der Systemzeit des PC oder man integriert diesen Aufruf gleich ins Programm. Fügen Sie daher gleich nach `PROG(` eine „Dummy"-Variable ein:

```
dummy := RANDOM(0)
```

wobei anstelle von `dummy` jede andere Bezeichnung stehen kann. Vergessen Sie aber nicht, diese Variable in die Parameterliste aufzunehmen, damit sie lokal bleibt.

Jetzt wollen Sie doch sicher auch wissen, wie unser „Chaos" aussieht:

Während `chaos(20)` noch nichts enthüllt, zeigt sich bei `chaos(100)` doch schon die Andeutung eines Musters.



Und wenn man gar 5000 Punkte zeichnen läßt, dann entsteht ganz plötzlich das *Sierpinski* Dreieck.

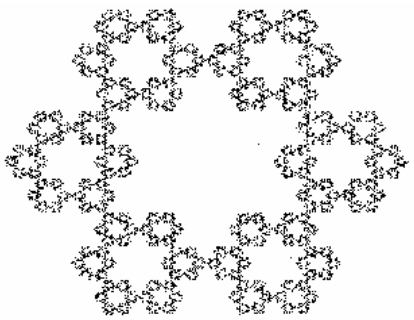
Dieses schöne Dreieck läßt sich auch mit einem „Einzeiler“ erzeugen – und wiederum mit dem sehr mächtigen `ITERATES`-Befehl.

```
sierp3(n) := ITERATES([1/2]·[z + [[0, 0], [1, 0], [1/2, √3/2]][1 + RANDOM(3)]], z, [1.5·RANDOM(1) - 3, 1.5·RANDOM(1) - 3], n)
```

Herausforderung 7

Chaospolygone [11]

Ausgehend von einem regelmäßigen p-Eck mit p > 4 ergibt sich der jeweils nächste Punkt nach der folgenden Vorschrift:



$$\overrightarrow{p_{n+1}} = f\ \overrightarrow{p_n} + (1-f)\ \overrightarrow{Zufallsecke}$$

$$\text{mit}\quad f = \frac{1}{2\left(1+\cos\frac{2\pi}{p}\right)}$$

In der Abbildung sehen Sie ein Resultat von `chaospol(6,10000)`

7 Der Satz von BRIANCHON – ein Satz über Kegelschnitte

Blaise Pascal entdeckte 1640 im Alter von 17 Jahren einen bemerkenswerten, nach ihm benannten Satz (siehe Herausforderung 9). Fast 200 Jahre später wendete Charles Brianchon das damals gefundene „Dualitätsprinzip“ der Projektiven Geometrie an und formulierte das duale Gegenstück, den *Satz von BRIANCHON* (1812):

> Wenn man einem beliebigen Kegelschnitt ein geschlossenes Sechsseit umschreibt und die entstehenden Ecken von 1 bis 6 durchnumeriert, dann gehen die Verbindungsgeraden von gegenüberliegenden Ecken ([1,4], [2,5] und [3,6]) immer durch einen gemeinsamen Punkt, den *BRIANCHONschen Punkt* (siehe auch die Herausforderung 7) [12].

Wir wollen ein Programm schreiben, das zu einer Ellipse mit frei gewählten Halbachsen *a* und *b* ein zufälliges umschriebenes Sechsseit erzeugt (die Tangenten in 6 zufälligen Punkten auf der Kurve). Die Tangentenschnittpunkte werden zu einem geschlossenen Sechsseit verbunden und die drei „Diagonalen“ (= Verbindungen der „Gegenecken“) gezeichnet. Sie müssen sich immer in einem Punkt treffen, der dann noch besonders herausgehoben wird.

Die Ausgabe soll so erfolgen, dass wir schrittweise die Konstruktion im 2D-Grafikfenster nachvollziehen können.

Ich zeige das Programm wieder, wie es im Editor aussieht, da die Übertragung aus dem *DERIVE*-Schirm nur nach vielen Anstrengungen gelingen würde.

```
brianchon(a,b,dummy,teil2,vals,markierung,brpt,teil1,ips,diags,
          pts,tgs,teil3):=PROG(
dummy:=RANDOM(0),
"",
"@ 6 Parameter zur Erzeugung von 6 Zufallspunkten auf der Ellipse",
"@ Dabei wird der erste Punkt am Ende nochmals gebraucht ",
"",
vals:=VECTOR(2*pi*RANDOM(1),k,6),vals:=APPEND(vals,[vals SUB 1]),
"@ das sind die 6 Punkte auf der Ellipse .... ",
pts:=VECTOR([a*COS(t),b*SIN(t)],t,vals),
"@ ... mit den zugehoerigen Tangenten (Pkt-Richtungsform) ",
tgs:=VECTOR([SUBST(y=-b*COS(t)/(a*SIN(t))*(x-a*COS(t))+b*SIN(t),
     t,k)],k,vals),
"@ ips wird die Liste der Tangentenschnittpunkte ",
ips := VECTOR(SOLUTIONS(tgs SUB i SUB 1 AND tgs SUB (i+1) SUB 1,
                        [x,y]) SUB 1,i,6),
"",
"@ damit entsteht dann ein geschlossenes Sechseck ... ",
"",
ips:=APPEND(ips,[ips SUB 1]),
"",
"@ die Diagonalen (1,4), (2,5) und (3,6) werden gebildet ",
"",
```

```
diags:=VECTOR(EXPAND([y=(ips SUB k SUB 2-ips SUB (k+3) SUB 2)/
           (ips SUB k SUB 1-ips SUB (k+3) SUB 1)*(x-ips SUB k SUB 1)+
            ips SUB k SUB 2]),k,1,3),
"",
"@ Der BRPkt entsteht im Schnitt von zwei dieser Diagonalen ",
"",
brpt:=SOLUTIONS(diags SUB 1 SUB 1 AND diags SUB 2 SUB 1,[x,y]) SUB 1,
"",
"@ der BrPkt soll durch einen dreifachen Kreis markiert werden ",
"@ dann Zusammenfassung der Ergebnisse und Ausgabe in einer Matrix ",
"",
marks:=EXPAND(VECTOR([(x-brpt SUB 1)^2+(y-brpt SUB 2)^2=k*(0.1)^2],
              k,3)),
teil1:=[["Ellipse:",b^2*x^2+a^2*y^2=a^2*b^2],["Punkte:",pts]],
teil2:=[["Tangenten:",tgs],["Sechsseit:",ips],["Diagonalen",diags]],
teil3:=[["Brianchon'scher Punkt:",brpt],["Markierung:",marks]],
RETURN(APPEND(teil1,teil2,teil3)))

NotationDigits:=4
```

Der Programmcode sollte selbsterklärend sein – wenn man davon absieht, dass sich die SUBs schwerer lesen lassen als die Indizes. Ein zusätzlicher Hinweise zur *DERIVE*-Syntax scheint aber angebracht:

`2*pi*RANDOM(1)` erzeugt eine Zufallszahl zwischen 0 und 2π.

Die Ausgabe mit 4 werthabenden Stellen können Sie natürlich auch weglassen und dann extern über *Declare Output Settings* individuell einstellen.

`brianchon(4,2)` brachte bei mir z.B. das folgende Ergebnis:

Ellipse:	$4 \cdot x^2 + 16 \cdot y^2 = 64$
Punkte:	$\begin{bmatrix} 3.999 & 0.03774 \\ 3.989 & 0.1422 \\ -0.6042 & -1.977 \\ 0.8390 & -1.955 \\ -3.787 & 0.6431 \\ 0.1874 & 1.997 \\ 3.999 & 0.03774 \end{bmatrix}$
Tangenten:	$\begin{bmatrix} y = 105.9 - 26.48 \cdot x \\ y = 28.11 - 7.011 \cdot x \\ y = -0.07640 \cdot x - 2.023 \\ y = 0.1072 \cdot x - 2.045 \\ y = 1.472 \cdot x + 6.219 \\ y = 2.002 - 0.02345 \cdot x \\ y = 105.9 - 26.48 \cdot x \end{bmatrix}$

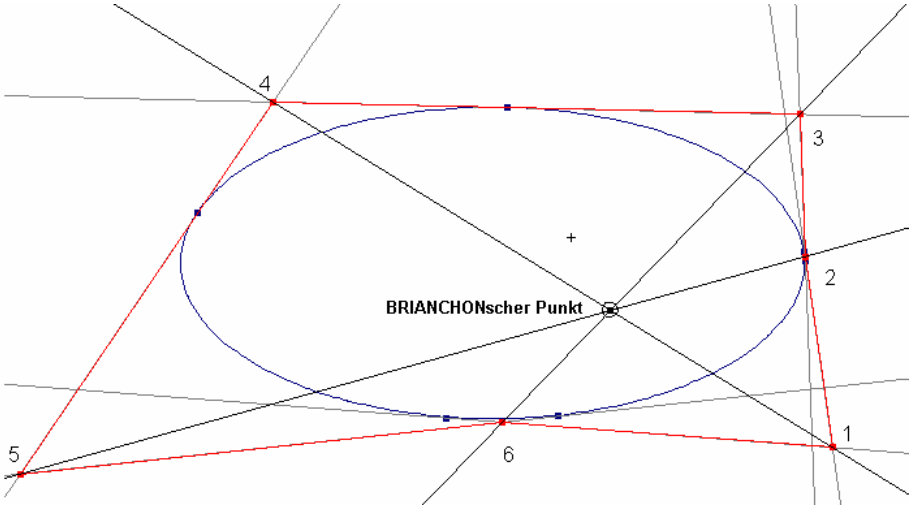
```
                                  ⎡  3.997   0.09006 ⎤
                                  ⎢  4.346   -2.355  ⎥
                                  ⎢ 0.1213   -2.032  ⎥
      Sechsseit:                  ⎢ -6.054   -2.695  ⎥
                                  ⎢ -2.819    2.068  ⎥
                                  ⎢  3.928    1.910  ⎥
                                  ⎣  3.997   0.09006 ⎦

                              ⎡ y = 0.2770·x - 1.017  ⎤
      Diagonalen              ⎢ y = 0.3277 - 0.6173·x ⎥
                              ⎣  y = 1.035·x - 2.158  ⎦

Brianchon'scher Punkt:              [1.503, -0.6007]

                              ⎡  2              2                  ⎤
                              ⎢ x  - 3.007·x + y  + 1.201·y + 2.622 = 0.01 ⎥
      Markierung:             ⎢  2              2                  ⎥
                              ⎢ x  - 3.007·x + y  + 1.201·y + 2.622 = 0.02 ⎥
                              ⎢  2              2                  ⎥
                              ⎣ x  - 3.007·x + y  + 1.201·y + 2.622 = 0.03 ⎦
```

Daraus entwickelte sich dann im Zeichenfenster eine Veranschaulichung des *Satzes von BRIANCHON*, die in Farben noch viel deutlicher wird.



Die Ausgabe der kompletten Matrix ist natürlich sehr aufwändig und die Zahlenwerte mögen uns meist nicht interessieren. Daher will ich Ihnen zeigen, wie man hier sehr sinnvoll *globale Variable* einsetzen kann.

Man definiert die Variablen vor dem Programm am besten in einer Liste:

```
[ellipse:=, pts:=, diags:=,brpt:=,marks:=,ips:=,tgs:=]
```

Natürlich wird man da im „Ernstfall“ sprechendere Variablennamen wählen. Dann sind diese Variablen aus der Parameterliste des Programms zu entfernen – und die **`ellipse`** ins Programm aufzunehmen.

```
brianchon2(a,b,dummy,teil1,teil2,teil3,vals):=PROG(
```

Außerdem ist eine kleine Änderung am Ende des Programms notwendig, wobei die ursprünglichen Ausgabesequenzen durchaus erhalten bleiben können, da sowieso nur der erste auftretende `RETURN`-Auftrag erfüllt werden kann:

```
marks:=EXPAND(VECTOR([(x-brpt SUB 1)^2+(y-brpt SUB 2)^2=k*(0.1)^2],
                     k,3)),
ellipse := b^2*x^2+a^2*y^2=a^2*b^2,
RETURN "Zeichne ellipse, pts, tgs,ips,diags,brpt,marks",
teil1:=[["Ellipse:",b^2*x^2+a^2*y^2=a^2*b^2],["Punkte:",pts]],
```

Im Algebrafenster sieht es jetzt viel einfacher und weniger dicht aus. Die graphische Darstellung kann nun komplett im 2D-Grafikfenster durchgeführt werden, indem man die Variablen der Reihe nach in der Eingabezeile aufruft und darstellt.

```
#4:   brianchon2(4, 2)
#5:                    Zeichne ellipse, pts, tgs,ips,diags,brpt,marks
#6:   ellipse
#7:   pts
#8:   tgs
#9:   ips                    Das sehen Sie dann im Algebrafenster
#10:  diags
#11:  brpt
#12:  marks
```

Herausforderung 7

Ändern Sie `brianchon()` in dem Sinn ab, dass Sie die Gültigkeit des Satzes auch für Hyperbeln, bzw. Parabeln sichtbar machen können.

Herausforderung 8

Verallgemeinern Sie `brianchon()` so, dass zufällig ein beliebiger Kegelschnitt entsteht. Ein Tipp dazu: Denken Sie an die Polarform von Kegelschnitten:

$$r(\varphi) = \frac{a}{1-\varepsilon\cos(\varphi)} \begin{cases} 0<\varepsilon<1 \ldots \text{Ellipse} \\ \varepsilon=1 \ldots \text{Parabel} \\ \varepsilon>1 \ldots \text{Hyperbel} \end{cases}$$

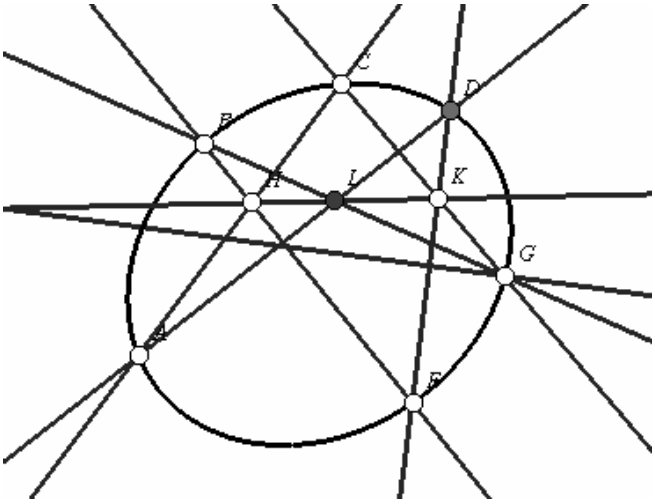
Herausforderung 9

Der *Satz von Pascal* lautet in moderner Sprechweise: Sind 1, ..., 6 beliebige Punkte eines Kegelschnitts und bezeichnet (1,2), (2,3), usw. die entsprechende Verbindungsgerade, (1,2)/(4,5) usw. den Schnittpunkt dieser Geraden, so liegen die Schnittpunkte (1,2)/(4.5), (2,3)/(5,6) und (3,4)/(6,1) auf einer Geraden, der Pascalschen Geraden [12],[13].

Überlegen Sie, dass die beiden Sätze (Brianchon und Pascal) durch Dualisierung auseinander hervorgehen. (Vertauschen Sie jeweils Punkt ↔ Gerade, Verbinden ↔ Schneiden).

Schreiben Sie nach der Vorlage `brianchon()` ein entsprechendes Programm `pascal()`.

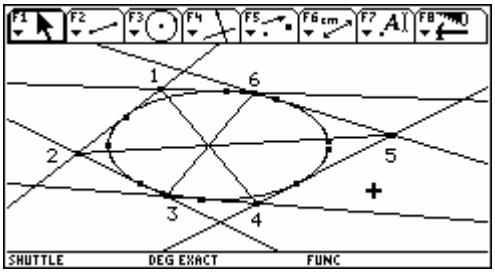
Das ist etwas einfacher als `brianchon()`.

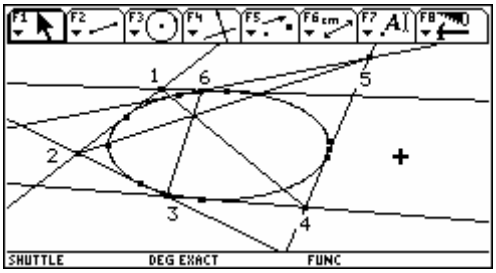


In der Abbildung können Sie eine Realisierung des Satzes von Pascal sehen, die mit dem DGS (Dynamischen Geometrie System) Cinderella erzeugt wurde [14].

Herausforderung 10

Falls Ihnen ein Dynamisches Geometriesystem zur Verfügung steht, dann erzeugen Sie die Darstellung von einem der beiden Sätze. (Hier wurde der *Satz von Brianchon* mit der Cabri-Applikation des TI-92 dargestellt. Das rechte Bild entstand aus dem linken durch Veränderung der Lage von zwei Ellipsenpunkten).





8 Und damit verlieren die Extremwertaufgaben ihren Schrecken

Wer kennt sie nicht, die Probleme der Schüler mit den Extremwertaufgaben? Wenn man sich als Lehrer einmal entschlossen hat, ein CAS im Unterricht einzusetzen, reduziert sich ab einem gewissen Moment – sicherlich nach der händischen Durchführung von einigen wenigen einführenden Aufgaben – der mathematische Gehalt auf das Auffinden und Formulieren von Haupt- und Nebenbedingung(en) (= hb, bzw. nb) und auf das Interpretieren der Ergebnisse. Alles andere (Substituieren für die Variablen, Differenzieren und Lösen der auftretenden Gleichungen) wird nach einem „Rezept" vom CAS übernommen.

Dann könnte man aber auch im Unterricht versuchen, dieses „Rezept" in Form eines ablauffähigen Algorithmus einem programmierbaren System zu „lernen". Damit wird man erst richtig gezwungen, ganz klar und sauber Schritt für Schritt zu überlegen – und auch allfällig auftretende Sonderfälle in seine Überlegungen mit einzubeziehen [15].

Im vorgestellten Programm werden auch mögliche Randextrema berücksichtigt. Kommentare sind eingefügt.

```
extrem(hb,nb,v1,v2,v3:="$",lr,rr,solhb,hb1,solsv1,auxv1,d2hb,i_,
       art,nsols1,vald2,j_,res,solsv2,sols1,va_,v4,randex):=
       PROG(
       "@ Menge der echten Variablen ",
3      va_:={v1,v2},
       "@ die NB wird nach v1 aufgeloest ",
       sols1:=SOLUTIONS(nb,v1),nsols1:=DIM(sols1),
       i_:=1,art:=[],
       "@ Schleife fuer Loesungen der NB ",
       LOOP(
         IF(i_>nsols1,exit),
         "@ die Loesungen aus der NB werden in die HB eingesetzt ",
         auxv1:=sols1 SUB i_,hb1:=SUBST(hb,v1,auxv1),
         "@ die HB haengt nur mehr von v2 ab und EW wird gesucht ",
         solsv2:=SOLUTIONS(DIF(hb1,v2)=0,v2),solsv1:=
         VECTOR(auxv1,v2,solsv2),
         solhb:=VECTOR(hb1,v2,solsv2),
         "@ Art der Extremwerte wird bestimmt ",
         d2hb:=DIF(hb1,v2,2),
         vald2:=VECTOR(d2hb,v2,solsv2),
                art:=VECTOR(IF(k_>0,"lokales Minimum",
                            IF(k_<0,"lokales Maximum",
                                    "kein EW")),k_,vald2),
         i_:+1
       ),
       "@ Ende der Schleife ",
       "@ Variable mit den gegebenen Grenzen wird berücksichtigt ",
1      IF(NOT STRING?(v3),
         PROG(
4          va_:=va_\{v3},v4:=(VARIABLES(va_)) SUB 1,
           sols1:=SOLUTIONS(nb,v4),hb1:=SUBST(hb,v4,sols1 SUB 1),
```

```
          "@ Bestimmung der Werte für HB am Rand des Def.Ber. ",
  randex:=IF(NAME_TO_CODES(STRING(v3))=NAME_TO_CODES(STRING(v1)),
  [[lr,SUBST(sols1 SUB 1,v3,lr),SUBST(hb1,v3,lr),"am unteren Rand"],
   [rr,SUBST(sols1 SUB 1,v3,rr),SUBST(hb1,v3,rr),"am oberen Rand"]],
  [[SUBST(sols1 SUB 1,v3,lr),lr,SUBST(hb1,v3,lr),"am unteren Rand"],
  [SUBST(sols1 SUB 1,v3,rr),rr,SUBST(hb1,v3,rr),"am oberen Rand"]]))
            ),
       "@ Zusammenfassung und Ausgabe des Endergebnisses ",
       res:=APPEND([APPEND([v1],solsv1)],[APPEND([v2],solsv2)],
                 [APPEND(["Optimum"],solhb)],[APPEND(["Art"],art)]),
       res:=res`,
       "@ randex ist bereits eine zweizeilige Matrix ",
2      IF(NOT STRING?(v3),RETURN(APPEND(res,randex)),RETURN res)
           )
```

Aus drucktechnischen Gründen mussten einige Einrückungen verändert werden, wenn die Schrift eine lesbare Größe behalten sollte.

Für `v3` wird die „unmögliche“ Variable `"$"` voreingestellt, und falls diese nicht überschrieben wird, dann wird die Frage nach Randextrema nicht gestellt. In ***1*** und ***2*** wird nach der Existenz dieses Strings gefragt und bei Nichtexistenz (`NOT STRING?(v3)`) wird die Routine zur Behandlung der Randextrema durchgeführt.

Die Einführung einer Menge der Variablen in ***3*** war deshalb notwendig, weil ja bei den Extremwertaufgaben auch Formvariable auftreten können, nach denen weder differenziert wird, noch werden für sie Randwerte gesucht. Wenn die Werte der Hauptbedingung an den Rändern berechnet werden sollen, müssen wir dem System mitteilen, welche der allfälligen noch vorhandenen Variablen die zweite bestimmende Variable des Problems ist. Dazu wird auch die Prozedur in ***4*** benötigt.

Eine Neuheit in diesem Programm ist, dass man in ein IF-Konstrukt auch Blöcke von Anweisungen einbauen kann. Diese müssen aber, wie aus ***1*** ersichtlich, in einer `PROG(....)`-Anweisung zusammengefasst werden. Sie sehen hier, wie das dann auf dem *DERIVE*-Schirm aussieht:

```
If ¬ STRING?(v3)
   Prog
     va_ := va_ \ {v3}
     v4 := (VARIABLES(va_))↓1
     sols1 := SOLUTIONS(nb, v4)
     hb1 := SUBST(hb, v4, sols1↓1)
     "@ Bestimmung der Werte fuer HB am Rand des Def.Ber. "
     randex := IF(NAME_TO_CODES(STRING(v3)) = NAME_TO_CODES(
```

Leider können in einem *DERIVE*-Programm eigene Funktionen nur mühsam definiert werden (ein Beispiel können Sie in BRIBAUM.MTH finden), daher erfolgt hier die Berechnung von Funktionswerten über das Substituieren oder über einen Grenzwert:

`SUBST(sols1 SUB 1,v3,rr)` oder `LIM(sols1 SUB 1,v3,rr)` setzen den rechten Rand des Definitionsbereichs für die Variable `v3` in den entsprechenden Term ein, der nach Auflösung der Nebenbedingung nach der anderen Variablen entstanden ist.

Besondere Sorgfalt verdient die Zusammenfassung der Ergebnisse in Form einer Matrix. Der Einsatz der eckigen Klammern ist ein wenig ausgeklügelt, da man sehr klar zwischen einem Vektor und einer einzeiligen Matrix, die mit [[...]] begrenzt wird, zu unterscheiden hat. In umgekehrter Reihenfolge braucht man ein `SUB` oder zwei `SUB`s, um ein Element aus dem Vektor (eindimensionale Liste) oder der Matrix anzusprechen. Beachten Sie bitte die entsprechenden Hinweise in der *DERIVE*-Hilfe-Datei.

Nun wollen wir aber unser selbst-geschaffenes Werkzeug nützen, um einige Extremwertaufgaben zu lösen, und damit gleich das Programm auf seine Tauglichkeit zu testen.

Auf Seite 5 habe ich die Lösung für die wohlbekannte Extremwertaufgabe gegeben, in der einem geraden Kreiskegel mit $r = 6$ und $h = 9$ der volumsgrößte Kreiszylinder einzuschreiben ist. Ich versuche nochmals, meinem Programm die Lösung zu entlocken, verwende aber andere Variable, vertausche die Reihenfolge der Variablen und gebe für r die Definitionsgrenzen mit 0 und 6 an:

$$\text{extrem}\left(r^2 \cdot h \cdot \pi,\ \frac{9}{6} = \frac{h}{6 - r},\ h,\ r,\ r,\ 0,\ 6\right)$$

h	r	Optimum	Art
9	0	0	lokales Minimum
3	4	$48 \cdot \pi$	lokales Maximum
9	0	0	am unteren Rand
0	6	0	am oberen Rand

Wenn Sie anstelle der Werte 9 und 6 mit allgemeinen Größen a und b arbeiten, funktioniert zwar die Extremwertsuche auch, aber die Art der Extremwerte kann nicht bestimmt werden, da *DERIVE* bei den allgemeinen zweiten Ableitungen nicht das Vorzeichen bestimmen kann.

Dass auch Wurzelfunktionen richtig bearbeitet werden, zeigt das folgende Beispiel aus [12]:

$$\text{extrem}\left(150 \cdot \sqrt{(15^2 + x^2)} + 250 \cdot \sqrt{(25 + y^2)},\ \frac{y}{5} = \frac{15}{x},\ x,\ y\right)$$

x	y	Optimum	Art
12.33106037	6.082201995	4881.079524	lokales Minimum
-12.33106037	-6.082201995	4881.079524	lokales Minimum

Beachten Sie aber, dass sie immer im `exact`-Modus rechnen (lassen) und erst das Ergebnis approximieren.

Besonders einfach, aber auch interessant sind die Ergebnisse für die nächsten beiden Aufgaben:

Einem gleichschenkligen Trapez ($a = 4$, $h = 1.5$ und $c = 3$) ist das
a) flächengrößte
b) umfangsgrößte Rechteck einzuschreiben.

Wenn man das erste Ergebnis richtig interpretiert - und nicht nur abschreibt - erkennt man, dass es ein Rechteck mit der Höhe 3 in einem Trapez, dessen Höhe nur 1,5 beträgt, nicht geben kann.

$$\text{extrem}\left(b \cdot h,\ \frac{1}{3} = \frac{\frac{4-b}{2}}{h},\ b,\ h\right)$$

b	h	Optimum	Art
2	3	6	lokales Maximum

$$\text{extrem}\left(b \cdot h,\ \frac{1}{3} = \frac{\frac{4-b}{2}}{h},\ h,\ b,\ h,\ 0,\ \frac{3}{2}\right)$$

Wenn wir für die Höhe h die Grenzen mit 0 und 3/2 festlegen, dann sehen wir, dass hier ein Randextremum vorliegt ($h = 3/2$, $b = 3$ und $Fläche = 9/2$)

h	b	Optimum	Art
3	2	6	lokales Maximum
0	4	0	am unteren Rand
$\frac{3}{2}$	3	$\frac{9}{2}$	am oberen Rand

Das Problem b) liefert gar keinen lokalen Extremwert!

$$\text{extrem}\left(2 \cdot b + 2 \cdot h,\ \frac{1}{3} = \frac{\frac{4-b}{2}}{h},\ h,\ b,\ h,\ 0,\ \frac{3}{2}\right)$$

h	b	Optimum	Art
0	4	8	am unteren Rand
$\frac{3}{2}$	3	9	am oberen Rand

Da die Hauptbedingung eine lineare Funktion bleibt, gibt es keine lokalen Extremwerte, nur Randextrema existieren.

Dem Programm kann man offensichtlich weitgehend vertrauen. Ich habe es mit vielen gängigen Aufgaben aus den Schulbüchern getestet. Es gibt aber wahrscheinlich Sonderfälle, die nicht richtig behandelt werden.

Aber auch – oder gerade – bei Computerprogrammen gilt die Lebensweisheit „Vertrauen ist gut – Kontrolle ist besser!“.

Zum Abschluss soll noch eine rechentechnisch schwierigere Aufgabe versucht werden:

> Aus 40cm Draht soll das Kantenmodell einer geraden Pyramide mit quadratischer Grundfläche gefertigt werden. Bei welchen Abmessungen hat die Pyramide das größte Fassungsvermögen?

Ich substituiere für *k* (aus `nb1_`) in `nb2_` und probiere mein Glück mit `extrem()`:

$$hb\_ := v = \frac{a^2 \cdot h}{3}$$

$$nb1\_ := 4 \cdot a + 4 \cdot k = 40$$

$$nb2\_ := k^2 = h^2 + \left(\frac{a \cdot \sqrt{2}}{2}\right)^2$$

Ich hatte Glück. So wie bei der traditionellen Durchführung muss man hier manchmal die Reihenfolge der Berechnung ändern. (Schon die Vertauschung der Reihenfolge von *a* und *h* zeigt kein so schönes Ergebnis mehr).

```
extrem(RHS(hb_), SUBST(nb2_, k, 10 - a), a, h)
```

a	h	Optimum	Art
4.64810	-4.22367	-30.4174	lokales Minimum
4.64810	4.22367	30.4174	lokales Maximum
0	-10	0	lokales Maximum
0	10	0	lokales Minimum

Die Länge der Seitenkante *k* muss man dann separat aus `nb1_` berechnen:

```
SOLVE(4·4.648162415 + 4·k = 40, k) = (k = 5.351837585)
```

Mit diesem Werkzeug lassen sich sicher viele der gängigen Extremwertaufgaben lösen. Wenn Ihnen aber einmal eine unterkommt, bei der `extrem()` versagt, dann bleiben Ihnen zwei Möglichkeiten:

- Das Programm analysieren und so verbessern, dass die neue Aufgabe auch gelöst werden kann, oder
- die Möglichkeiten eines CAS nützen und traditionell vorgehen.

Eine kritische Kontrolle des Ergebnisses bleibt aber immer angeraten.

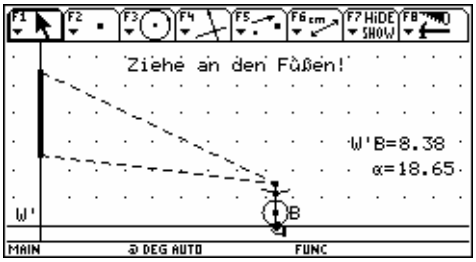
Herausforderung 11 (leicht)

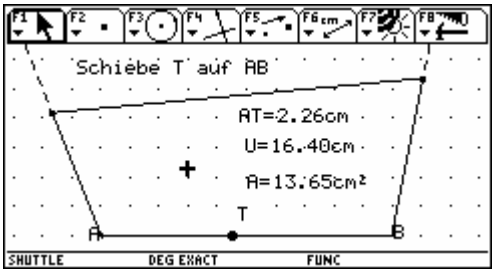
Wie lassen sich mit `extrem()` auch Aufgaben ohne Nebenbedingung lösen? Finden Sie dazu ein geeignetes Problem!

Herausforderung 12 (noch leicht?)

Lösen Sie die folgenden traditionellen Extremwertaufgaben [16] mit `extrem()`

Ein 3m hohes Bild hängt an der Wand eines Saales; sein unterer Rand ist 2,5m über dem Fußboden. Wie weit muss sich ein Betrachter, dessen Auge sich 150cm über dem Boden befindet, von der Wand entfernen, um das Bild unter einem möglichst großen Sehwinkel bewundern zu können?

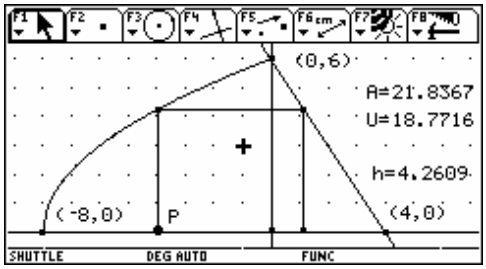




Gegeben ist das Viereck ABCD mit A(0/0), B(5/0), C(5.5/2.5), D(-1,2.5). T ist ein beliebiger Punkt auf AB. Auf AD ergibt sich der Punkt P mit AP = AT und auf BD der Punkt Q mit BQ = BT.

Für welche Lage von T hat das entstehende Viereck ABQP
a) größten Umfang und
b) größten Inhalt?

Die nach rechts offene Parabel mit dem Scheitel in (-8/0) geht durch den Punkt (0/6). Die Parabel und die Gerade g: 3x + 2y = 12 begrenzen mit y = 0 in der oberen Halbebene einen Bereich, dem ein achsenparalleles Rechteck eingeschrieben werden kann.
a) Bestimmen Sie das Rechteck mit dem größten Flächeninhalt.
b) Bestimmen Sie das Rechteck mit dem größten Umfang.



(Alle Beispiele sind dem bkt-Buch SR-06 entnommen: *Optimierungsaufgaben grafisch, numerisch und analytisch mit dem TI-92 lösen*. Die *TI*-92 Bilder zeigen CABRI-Modelle der gestellten Aufgaben).

Herausforderung 13 (aufwändig, aber leichter als `extrem()`)

Ein anderer „Dauerbrenner“ im Rahmen der Differentialrechung sind die Kurvenuntersuchungen [17], [18]. Schreiben Sie ein Programm, das nach Eingabe der Funktion als einzigen Parameter die Nullstellen, Extremwerte, Wendepunkte, Anstiege der Wendetangenten, Polstellen, Verhalten im Unendlichen, Symmetrie, ...) ausgibt.

Hier können Sie einen Ausschnitt einer Ausgabe sehen (für $f(x)=\frac{2x-3}{3x^2-5}$).

$$\left[\begin{array}{l} \left[\begin{array}{ll} (3\cdot x^2-5)^2 & \\ 0.7362 & 0.4527 \\ 2.263 & 0.1472 \end{array}\right] \\ \left[\begin{array}{ll} \text{Infl.points:} & \text{k(I)} \\ \dfrac{18\cdot(2\cdot x^3-9\cdot x^2+10\cdot x-5)}{(3\cdot x^2-5)^3}=0 & \\ 3.172 & 0.1327 \quad -0.02092 \\ \text{complex} & \text{!} \quad \text{!} \\ \text{complex} & \text{!} \quad \text{!} \end{array}\right] \\ \left[\begin{array}{l} \text{Poles:} \\ x=-1.29 \\ x=1.29 \end{array}\right] \end{array}\right]$$

Herausforderung 14 (nicht so aufwändig, leichter als `extrem()`).

Schreiben Sie ein Programm für eine „Umgekehrte Kurvendiskussion“ [18], [19].

Beispiel: Welche Polynomfunktion hat die Punkte A(0|2), B(1|-3) und C(4|4) als lokale Extremwerte? (Mein Programmvorschlag besteht aus 10 Zeilen!)

$$\text{funk}\left[\begin{array}{lll} 0 & 0 & 2 \\ 1 & 0 & 0 \\ 0 & 1 & -3 \\ 1 & 1 & 0 \\ 0 & 4 & 4 \\ 1 & 4 & 0 \end{array}\right] = \frac{313\cdot x^5}{432}-\frac{64\cdot x^4}{9}+\frac{3175\cdot x^3}{144}-\frac{4463\cdot x^2}{216}+2$$

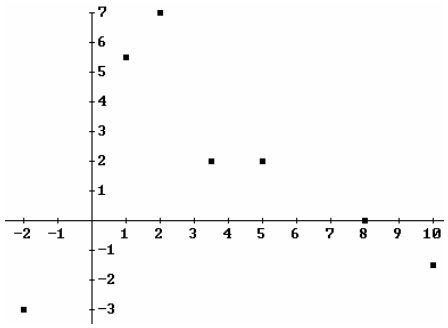
(Lesen Sie die beiden ersten Zeilen der Matrix als : die „nullte“ Ableitung an der Stelle *x* = 0 ist 2; die erste Ableitung an dieser Stelle ist 0.)

Und bei welcher Funktion treten diese Punkte A, B und C als Wendepunkte auf?

9 Polynominterpolation gegen kubische Splines

Stellen Sie sich vor, Sie hätten 7 Punkte in der Ebene gegeben und sollen die Polynomfunktion niedrigsten Grades finden, deren Graph diese Punkte (Stützpunkte) enthält.

$$\text{pkte} := \begin{bmatrix} -2 & -3 \\ 1 & 5.5 \\ 2 & 7 \\ 3.5 & 2 \\ 5 & 2 \\ 8 & 0 \\ 10 & -1.5 \end{bmatrix}$$



Üblicherweise werden Sie wahrscheinlich ein Polynom 6. Grades finden wollen. Das Aufstellen der sieben Gleichungen und auch das Lösen des Gleichungssystems stellen für ein CAS kein Problem dar. Wir wollen für diese Standardaufgabe ein Programm schreiben, das zu einer gegebenen Liste von Punkten das Interpolationspolynom ermittelt. Dabei wird kein Wert auf einen ausgetüftelten Algorithmus gelegt, sondern es soll nur die traditionelle Vorgangsweise in einem Programm – und in der *DERIVE*-Syntax verpackt werden.

```
polynom(m,m_,mat,koeff):=PROG(
          m_:=DIM(m),
          "@ ein Polynom vom Grad m_-1 wird gebildet, ein GLS mit  ",
          "@ m_ Gleichungen und m_ Variablen ist zu loesen ",
1          mat:=VECTOR(VECTOR(k^i,i,m_-1,0,-1),k,m COL 1),
          mat:=APPEND(mat`,[m COL 2`])`,
          koeff:=(ROW_REDUCE(mat) COL (m_+1))`,
          "@ oder VECTOR(x^i,i,m_-1,0,-1)*koeff ",
          SUM(x^(m_ - i) koeff SUB i, i, m_, 1, -1)
               )
```

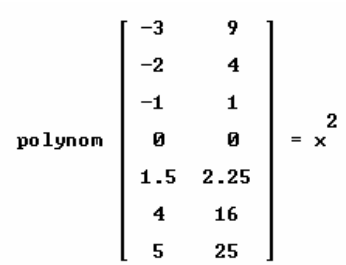
In Zeile **1** wird die Koeffizientenmatrix der Terme gebildet, die bei Belegung von $a*x^{n-1} + b*x^{n-2} + ...$ mit allen *x*-Werten (erste Spalte der Punktmatrix `m`, daher `m COL 1`) entsteht. In der nächsten Zeile wird diese Matrix um die rechten Seiten der Gleichungen (die *y*-Werte = zweite Spalte von `m`) erweitert. `koeff` ist dann die letzte Spalte der in die „reduzierte Zeilen-Stufenform" gebrachten, erweiterten Systemmatrix des Gleichungssystems.
Diese Koeffizienten verhelfen sofort – auf zwei mögliche Arten – zum gesuchten Polynom:

```
polynom(pkte)
```

$$\frac{102721 \cdot x^6}{25945920} - \frac{48523 \cdot x^5}{494208} + \frac{3514919 \cdot x^4}{4324320} - \frac{38376461 \cdot x^3}{17297280} - \frac{1276427 \cdot x^2}{720720} + \frac{55980481 \cdot x}{4324320} - \frac{193409}{46332}$$

Dieses Polynom soll vorerst nicht gezeichnet werden.

Man kann sich aber fragen, was passiert, wenn das Programm überbestimmt ist, dh.,wenn die vorliegenden *n* Punkte gar kein Polynom vom Grad *n-1* benötigen?



Gegen Polynome höheren Grades als Interpolationspolynome sprechen einige Gründe: das Arbeiten mit derartigen Polynomen ist recht umständlich. Da kann man aber dagegenhalten, dass dies im Zeitalter des Computers kein Problem mehr darstellt. Ok, dann warten Sie bitte, bis Sie den Graphen der Funktion gesehen haben, warum einiges für oder gegen das folgende Verfahren spricht:

Wir verbinden jeweils zwei benachbarte Stützpunkte durch eine Kurve 3.Grades, so dass dieser Übergang stetig - das ist klar - und differenzierbar - Ecke wollen wir keine drin finden - ist. Darüber hinaus sollen die Kurven in den Anschlusspunkten auch in ihren zweiten Ableitungen übereinstimmen, um einen besonders glatten - und ruckfreien - Übergang zu ermöglichen.

Diese Aufgabenstellung erinnert an die „umgekehrten Kurvendiskussionen“, wie sie in Österreich genannt werden, während ich in Deutschland schon den schönen Namen „Steckbriefaufgaben“ dafür gehört habe (siehe auch Herausforderung 14).

Wir bilden das mathematische Modell für *n* Stützstellen, die mit *n*-1 Polynomen 3. Grades p_k zusammengehalten werden. Da jedes dieser Polynome durch 4 Parameter bestimmt ist, werden $4n - 4$ Gleichungen für diese $4n - 4$ Variablen benötigt.

Ich empfehle Ihnen, eine kleine Skizze zu machen und mit mir mitzuzählen:

Das erste und letzte Polynom müssen durch den ersten, bzw letzten Punkt gehen:

$$p_1(x_1) = y_1 \qquad (1)$$
$$p_{n-1}(x_n) = y_n \qquad (1)$$

In den *n*-2 übrigen Punkten hängen die Polynome zusammen:

$$p_{k-1}(x_k) = y_k \text{ und } p_k(x_k) = y_k \qquad \text{mit } k = 2, ..., n - 1 \qquad (2n - 4)$$

In diesen Punkten stimmen 1. und 2. Ableitung benachbarter Polynome überein:

$$p_{k-1}'(x_k) = p_k'(x_k) \text{ oder } p_{k-1}'(x_k) - p_k'(x_k) = 0 \qquad (n - 2)$$
$$p_{k-1}''(x_k) = p_k''(x_k) \text{ oder } p_{k-1}''(x_k) - p_k''(x_k) = 0 \qquad (n - 2)$$

Und, um das System zu komplettieren, werden noch zwei Bedingungen benötigt. Zumeist lässt man die zweite Ableitung an den Randpunkten verschwinden:

$$p_1''(x_1) = 0 \text{ und } p_{n-1}''(x_n) = 0 \qquad (2)$$

Nun zählen wir zusammen: $1 + 1 + 2n - 4 + n - 2 + n - 2 + 2 = 4n - 4$.

Ist schon das Aufstellen des Gleichungssystems lästig genug (für unsere 7 Punkte wären 24 Gleichungen mit 24 Variablen notwendig), kapitulieren wir sofort bei der Auflösung des Systems.

Nun, es gibt verschiedene Algorithmen für dieses Standardproblem der numerischen Mathematik, aber wir wollen die händische Lösung simulieren. Das heißt, dass wir ehrlich die Matrix des Gleichungssystems aufbauen (lassen), auf die Matrix wieder `ROW_REDUCE` anwenden und schließlich aus dem resultierenden Lösungsvektor die „kubischen Splines" zusammenstellen.

Das Programm ist natürlich etwas länger und viel Platz wird dafür benötigt, die erweiterte $(4n - 4 \times 4n - 3)$-Matrix des Gleichungssystems zusammenzustellen. Ich werde nicht das ganze Listing hier anbieten, sondern verweise auf die mitgelieferte Diskette. Sie werden ja sicher nicht den Ehrgeiz haben, das Programm abzutippen!

Aber einige wesentliche Erläuterungen, die über die Kommentarzeilen hinausgehen, will ich schon noch anbringen.

```
splines(m,m_,z,z_,mat:=[],dmat,i,j,koeff,spls,x,splf,anf,ende):=
     PROG(
          m_:=DIM(m),
1         z:=VECTOR(0,k,4*m_-3),
          i:=1,z_:=z,
```

In ***1*** wird ein Nullvektor `z` bereitgestellt, in dem an den richtigen Stellen die Nullen durch die Koeffizienten der Gleichungen ersetzt werden. Ein besseres Verständnis für den Aufbau der Matrix erhält man, wenn man versucht, einige Zeilen selbst zu erstellen!

Im ersten `LOOP` wird das Argument der ersten Stützstelle in den ersten Spline eingesetzt und damit die ersten 4 Elemente von `z_` überschrieben (ersetzt) und anschließend die noch leere Matrix `mat` mit der ersten Zeile gefüllt. Anschließend folgt eine Schleife für die inneren Punkte, die jeweils zweimal Verwendung finden, gefolgt von der Berücksichtigung des Punktes am rechten Rand.

```
          LOOP(
            IF(i=5,EXIT),
            z_:=REPLACE(m SUB 1 SUB 1^(4-i),z_,i),
            i:+1),

          "@ der Funktionswert an die letzte Stelle gesetzt ",
          z_:=REPLACE(m SUB 1 SUB 2,z_,4*m_-3),
          mat:= APPEND(mat,[z_]),
          i:=1,z_:=z,
          j:=1,
```

```
        LOOP(
          i:=1,z_:=z,
          LOOP(
```

Nun folgt wieder eine geschachtelte Schleife, die die Zeilen der Matrix erzeugt, die aus der Übereinstimmung der ersten Ableitungen folgen. Diese wird in der Wiedergabe übersprungen und wir schauen uns die nächste Schleife näher an, die die geforderten Übereinstimmungen der zweiten Ableitungen beschreibt.

```
       "@ die zweiten Ableitungen müssen übereinstimmen ",
        j:=1,
        LOOP(
          i:=1,z_:=z,
          LOOP(
            IF(i=3,EXIT),
            z_:= REPLACE((4-i)!*m SUB (j+1) SUB 1^(2-i),z_,4j-4+i),
            i:+1),
          i:=5,
          LOOP(
            IF(i=7,EXIT),
            z_:= REPLACE(-(8-i)!*m SUB (j+1) SUB 1^(6-i),z_,4j-4+i),
            i:+1),
        mat:= APPEND(mat,[z_]),
        j:+1,
        IF (j=m_-1,EXIT)),
        "@ die zweiten Ableitungen an den Raendern verschwinden",
```

Die Kommentarzeile sagt schon wie es weitergeht. Ähnlich dem Beginn werden die zweiten Ableitungen an den Rändern berücksichtigt.

Ein letztes Mal wird die Matrix um eine Zeile erweitert und dann extrahieren wir die letzte Spalte der mit `ROW_REDUCE` behandelten Matrix. Ihre Elemente sind die Koeffizienten der gesuchten Splines - immer schön in Viererpäckchen, wenn die Matrix regulär ist.

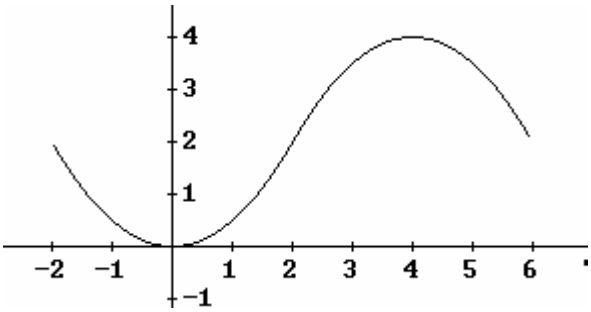
Das aber wird in Zeile ***2*** abgefragt.

```
        mat:= APPEND(mat,[z_]),
        koeff:=(ROW_REDUCE(mat) COL (4m_-3))`,
        "@ Bestimmung der Splinefunktionen ",
3       spls := VECTOR([x^3,x^2,x,1]* koeff ROW [4i_-3,...4i_],
                       i_,m_-1),
2       IF(NOT NUMBER?(SUBST(spls SUB 1,x,0)),
                            RETURN "singulaere Matrix !"),
```

```
4       splf := EXPAND(SUM(spls SUB i_ * CHI(m SUB i_ SUB 1,x,
                           m SUB (i_+1) SUB 1),i_,1,m_-1)),
        anf := m SUB 1 SUB 1, ende := m SUB m_ SUB 1,
        splf := IF(x<anf,?,IF(x<= ende,splf,?)),
        RETURN [APPEND(spls,[splf])]`
    )
```

In ***3*** werden die Splines zusammengestellt (man könnte dies auch mit dem Inneren Produkt erreichen, der Aufwand ist ziemlich der gleiche). Interessantes passiert in ***4*:** die einzelnen Teile werden von Punkt zu Punkt zusammengefügt und es entsteht eine abschnittweise definierte Funktion. Möglicherweise kennen Sie die Indikatorfunktion `CHI(a,x,b)` noch nicht, die dabei eingesetzt wird. Diese hat für $a < x < b$ den Wert 1 sonst überall den Wert 0. Wenn nun eine beliebige Funktion $f(x)$ mit CHI(a,x,b) multipliziert wird, dann erscheint sie nur innerhalb dieser Grenzen. Sehen Sie hier ein Beispiel mit zwei zusammengesetzten Parabeln.

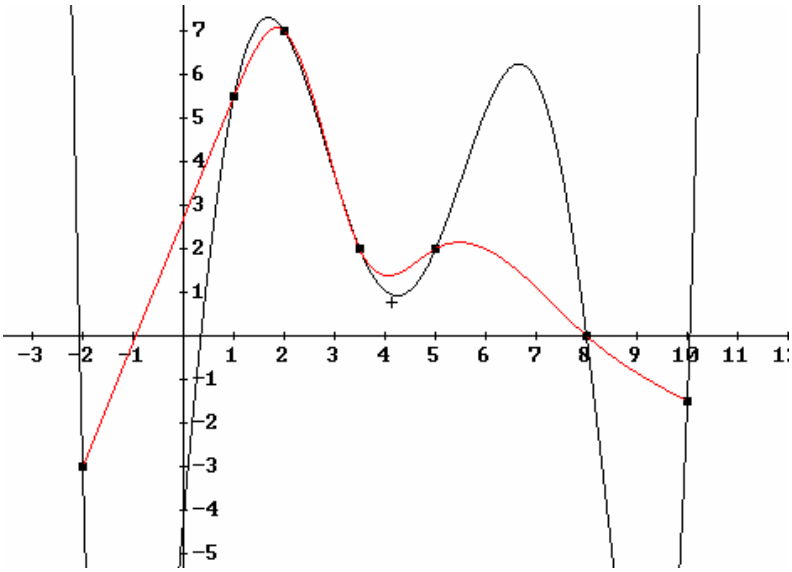
$$\frac{\mathrm{CHI}(-2,\ x,\ 2)\cdot x^2}{2} + \mathrm{CHI}(2,\ x,\ 6)\cdot\left(4 - \frac{(x-4)^2}{2}\right)$$



In der Ausgabe sehen wir dann den kompletten Spline als sehr langen und großen Ausdruck und eine Liste der einzelnen – hier sechs – verschiedenen kubischen Funktionen. Wenn Sie diese nicht brauchen, dann soll Ihre letzte Programzeile nur heißen: `RETURN spls`

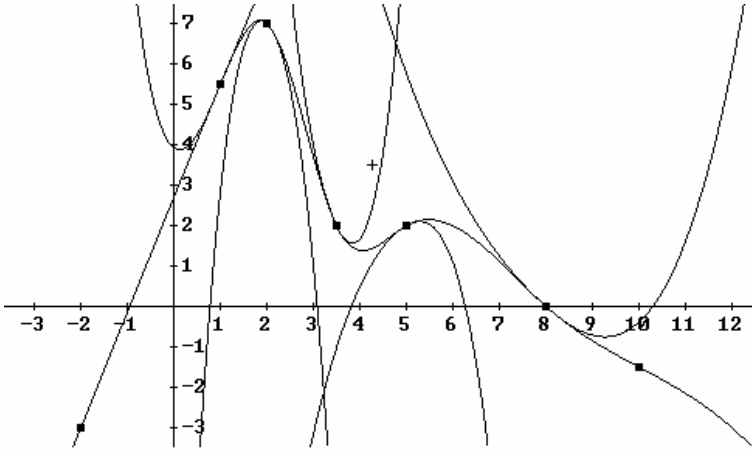
Wollen Sie trotzdem nicht auf die einzelnen Kurven verzichten, dann führen Sie `splf` als globale Variable ein, indem sie diese aus der Parameterliste entfernen und als allererste Kommandozeile `splf:=` in der Datei an die Spitze stellen. Unter `splf` sind alle kubischen Parabeln verfügbar, die im unteren Bild auf der nächsten Seite dargestellt sind.

Plotten Sie zuerst die gegebenen Punkte und legen Sie dann den Spline darüber. (Aktivieren Sie nur den Spline.) Anschließend zeichnen Sie bitte auch die Polynomfunktion dazu – wenn Sie nicht schon vorher neugierig gewesen sind.



Spätestens jetzt erkennen Sie den Vorteil des Splines: er osziliert wesentlich weniger als das Polynom 6. Grades. Der Linienfluss ist „natürlicher“. So oder so ähnlich würde man auch freihändig einen Versuch unternehmen, die Punkte zu verbinden.

Im nächsten Bild sieht man schön die glatten Übergänge in den Stützstellen. Natürlich lassen sich auch andere Randbedingungen vorgeben. So sind manchmal die Anstiege im linken und rechten Endpunkt vorgeschrieben.

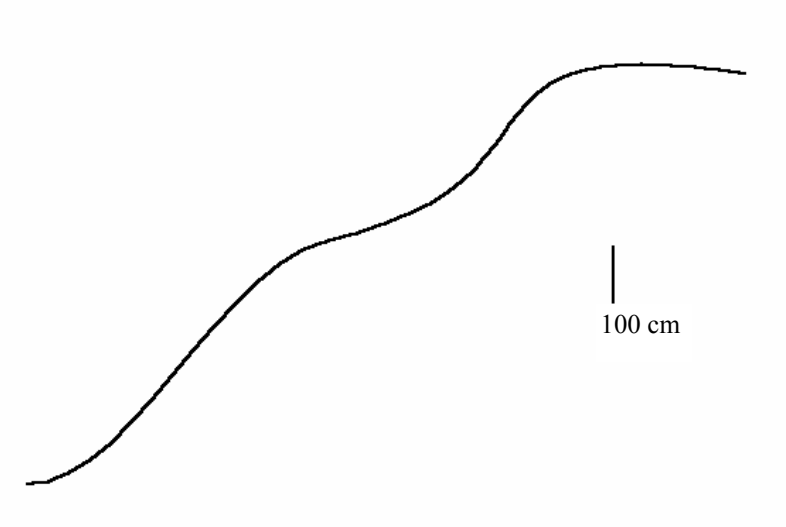


Zu den Splines gibt es sehr viel Literatur: [18], [20] - [24], [31], [33]

Herausforderung 14

Die vorliegende Kurve ist der halbe Querschnitt eines Drehkörpers - das könnte z.B. eine Turmhaube sein. Erzeugen Sie eine Spline - Anpassung an diese Kurve unter Verwendung von mindestens sechs geeigneten Stützstellen.

Verwenden Sie die gefundene Funktion zur Berechung des von diesem Drehkörper eingeschlossenen Volumens.

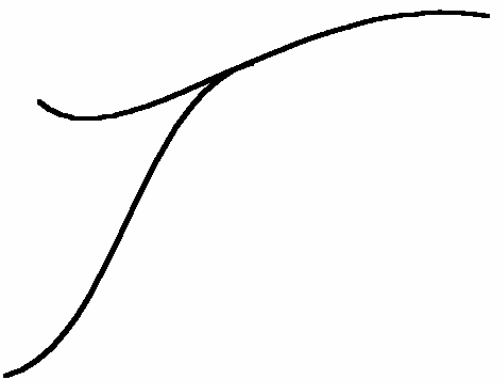


Herausforderung 15

Übertragen Sie aus einer Landkarte fünf oder mehr Landschaftspunkte, die durch eine Straße verbunden werden sollen. Konzipieren Sie die Trasse mit Hilfe von kubischen Splines.

Herausforderung 16

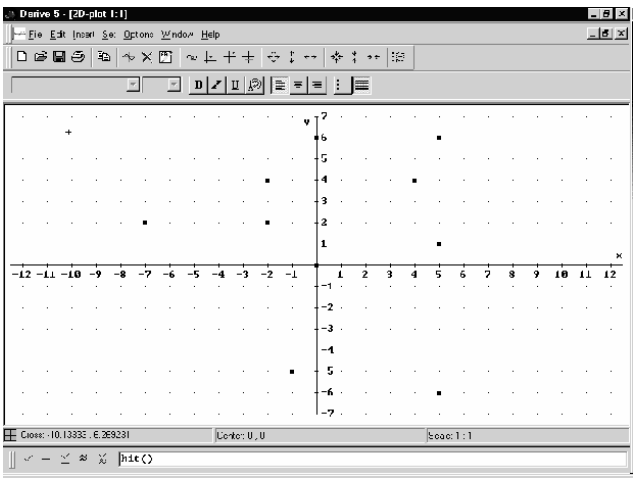
Das vorliegende kalligraphische „T“ soll durch zwei kubische Splines erzeugt werden. Wählen Sie geeignete Stützpunkte und erzeugen Sie die Splines in einem geeigneten Maßstab.



10 Programmieren mit *DERIVE* ist ein `hit()`

Das nächste Programm ist aus meiner schulischen Praxis entstanden, in der ich mehrere Übungs- und Trainingsprogramme für meine Schüler für den *TI*-92 geschrieben habe, darunter für das Arbeiten mit Termen, das Faktorisieren von Termen, für die Ableitungs- und Integrationsregeln und auch für einige Grundaufgaben mit den linearen Funktionen [25] Unter diesen erwies sich ein „Ballerspiel“ als besonders erfolgreich, bei dem es galt, zufällig im Koordinatensystem vorgegebene Punkte durch möglichst wenige Gerade „abzuschießen“. Es war nun für mich eine wahre Herausforderung, dieses Spiel nach *DERIVE* zu übertragen – wie auch alle anderen Trainingsprogramme.

Sehen Sie zuerst, wie so etwas aussehen kann:



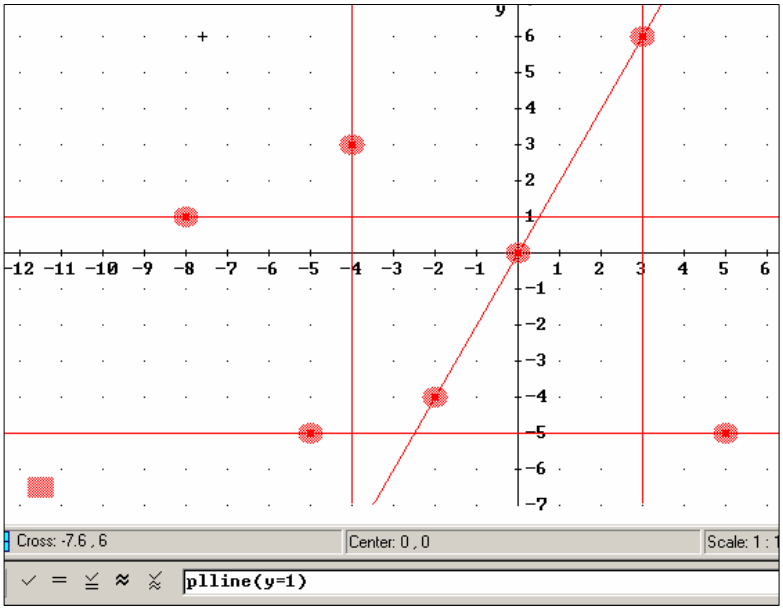
Der Übende ruft mit `hit()` zwischen 7 und 10 zufällige Punkte im Koordinatensystem auf, die alle ganzzahlige Koordinaten aufweisen. Das kann sofort im 2D-Plotfenster geschehen. Damit das Bild vernünftig wird, hat man auf einige Einstellungen im 2D-Fenster zu achten.

Um den Platz optimal zu nützen, habe ich die beiden Toolbars mit den mathematischen und griechischen Zeichen weggeschaltet. Unter *Options Display* empfiehlt sich bei *Grids* die *Intervals Horizontal* auf 28 und *Vertical* auf 15 zu setzen. Wenn man die *Lines* auf *On* stellt, soll man darauf achten, dass man eine sehr dezente Farbe wählt. Und zum Schluss noch eine ganz wichtige Einstellung: die Punkte sollen *Size Large* und *Connect No* aufweisen.

Jetzt kann es losgehen: Mit möglichst wenigen Geraden sollen alle Punkte „abgeschossen“ werden, wobei Sie den zusätzlichen Auftrag geben können, dass achsenparallele Gerade nicht gelten sollen – außer dies ist das erklärte Übungsziel. Genauso können Sie die Devise „Nur Gerade durch den Ursprung sind erlaubt!“ ausgeben. Aber unter diesen Voraussetzungen gilt dann die Resultatanzeige nicht.

Der Übende hat eine Gerade gefunden (im Kopf, mit einer Nebenrechnung, ...) - sie möge z.B. $y = 2x$ lauten - dann gibt er diese Gerade unter `plline(y = 2x)` ein und plottet sie.

Unmittelbar sieht er seine Treffer. Es ist nicht notwendig, die Gerade in expliziter Form anzugeben. Jede Darstellung ist erlaubt.



Nachdem der letzte Punkt getroffen wurde, wird das Ergebnis rechts unten angezeigt:

Werden die Punkte mit der theoretisch richtigen Mindestzahl von Versuchen getroffen, dann ist das Ergebnis – wie beim Golf – *PAR* und es erscheint ein Kreis. Für jeden zusätzlichen Versuch – *ÜBER PAR* – wird ein gefülltes Quadrat angezeigt und bei günstigen Punktkonstellationen und Geschick des Übenden soll es fallweise möglich sein, mehrere Punkte mit einer Geraden zu treffen, dann schafft man auch *UNTER PAR*, was mit einem oder mehr gefüllten Kreisen belohnt wird. In meinem Bild lautet das Ergebnis „*Eins über Par*“.

Hier finden wir eine sinnvolle Anwendung von globalen Variablen, da man sich ja die Punkte (`task`), das vorgegebene Par (`par`) und die Treffer (`score`) jeweils „merken“ muss. Daher sind globale Variable unbedingt notwendig.

Wie Sie sehen, ist das Programm gar nicht so umfangreich.
Das ist mein Programmvorschlag:

```
[score:=,task:=,par:=]

hit(pts_ := 0, dummy_, d_, k_, pt_, ttask_) := PROG(
   dummy_ := random(0),
   score := 0,
   "@ Man kann als ersten Parameter eine Anzahl von Pkten angeben",
   "@ oder mit hit() eine Zufallszahl akzeptieren",
```

```
   "@ hit(12) gibt 12 Punkte, hit() liefert eine zufällige Anzahl",
   IF(pts_= 0,
      d_ := 7 + RANDOM(4),
      d_ := pts_),
   "@ par = erlaubte Höchstzahl von Versuchen",
   par := FLOOR(d_/2 + 0.5),
   "@ erster Punkt in der Liste",
   task := [[RANDOM(23) - 11, RANDOM(13) - 6]],
   k_ := 1,
   LOOP(
        IF(k_ = d_,exit),
        "@ ein neuer Punkt wird erzeugt",
        pt_ := [RANDOM(23) - 11, RANDOM(13) - 6],
        "@ Testroutine um Dopplepunkte zu vermeiden",
        ttask_ := SELECT(pt_ = l_, l_, task),
        IF(ttask_ = [],
           task := APPEND([pt_], task),
           k_ := k_ - 1),
        k_ :+ 1
       ),
   RETURN task
   )

"@ nun folgt die Zeichen- und Überprüfungsroutine "

plline(equ_,yes_,resplot_,plot1_):=PROG(
   "@ der nächste Versuch",
   score :+1,
   "@ Liste der getroffenen Punkte",
① yes_:=SELECT(SUBST(equ_,[x, y],v_)=true,v_,task),
   "@ revidierte Punktliste",
② task:=SELECT(SUBST(equ_,[x,y],v_)=false,v_,task),
   "@ PAR: ein Kreis",
   IF(par-score=0,
      resplot_:=[(X+11.5)^2+(y+6.5)^2=0.6^2]),
   "@ ÜBER PAR: gefüllte Quadrate",
   IF(par-score<0,
      resplot_:=VECTOR(-12.8+i<=x<=-12.2+i and
                -6.8<=y<=-6.2,i,1,
      score-par)),
   "@ UNTER PAR: gefüllte Kreise",
   IF(par-score>0,
      resplot_:=VECTOR((x+12.5-i)^2+(y+6.5)^2<=0.4^2,i,1,
      par - score)),
      "@ Gerade zeichnen und getroffene Punkte markieren",
    ③ plot1_:=APPEND([equ_],
       VECTOR((x-yes_ sub i sub 1)^2+(y-yes_ sub i sub2)^2<= 0.3^2,
              i,1,DIM(yes_))),
    ④ IF(
         DIM(task) /= 0,
                    plot1_,
                    APPEND(plot1_,resplot_)
        )
      )
```

Ganz zu oberst erfolgt die Definition der globalen Variablen. Im Programm `hit()` wird die Aufgabe (`task`) zusammengestellt und das `par` festgelegt. Da `task` eine globale Variable ist, kann sie sofort nach dem Aufruf von `hit()` mit `task=` im Algebrafenster betrachtet werden. Die Anzahl der gewünschten Punkte wird über `pts_` gesteuert. Beachten Sie die entsprechenden Kommentarzeilen.

Welche Zufallszahlen produziert `RANDOM(23)-11`? Da `RANDOM(23)` eine ganzzahlige Zufallszahl aus [0, 22] erzeugt, ergibt sich nach Subtraktion von 11 die Menge {-11, ..., 11}.

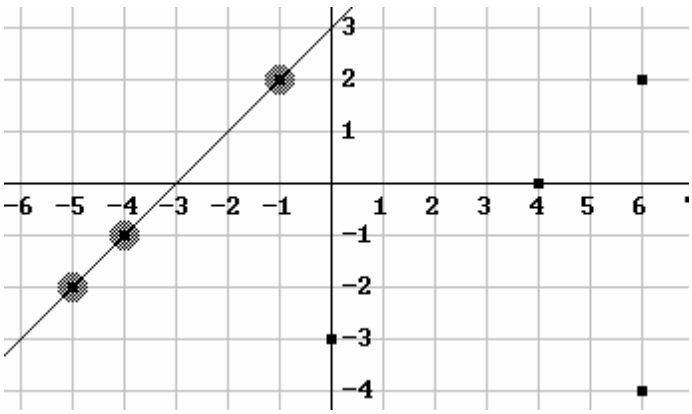
Mit `plline(Geradengleichung)` plottet man seinen ersten Versuch und erkennt, ob und wie viele Punkte getroffen wurden. Nun tritt der zentrale Teil von `plline()` in Aktion: die Verwaltung der getroffenen, bzw. der noch zu treffenden Punkte.

Ich werde das am besten an einem Beispiel erklären. Zu diesem Zweck habe ich vorher - nur zu Demonstrationszwecken - die Variable `yes_` „globalisiert“. Unser `task` möge beispielsweise neun Punkte umfassen, von denen 3 auf der Geraden $y = 3 + x$ liegen. Daher plotte ich `plline(y = 3+x)` und sehe wie erwartet meine drei Treffer schön markiert.

Im Algebrafenster inspiziere ich dann die Variablen:

```
hit(9)
plline(y = 3 + x)
yes_ = [ -4  -1 ]
       [ -1   2 ]
       [ -5  -2 ]

task = [ -2  -6 ]
       [  6  -4 ]
       [  4   0 ]
       [  6   2 ]
       [ 11   3 ]
       [  0  -3 ]
```



In ① erzeugt `yes_=` die Liste jener Punkte aus der Punktliste `task`, die getroffen wurden (weil die Koordinaten eingesetzt in die Geradengleichung eine wahre Aussage ergeben). Die Matrix `yes_` wird uns auf Abfrage auch gezeigt. Sie wird in ③ dazu verwendet, die Punkte zu markieren. In ② wird die Punktliste `task` aktualisiert. Wir finden dann auch nur mehr 6 Punkte in dieser Liste, die aus den nicht getroffenen Punkten besteht.

In `resplot_` wird ein vorläufiges Ergebnis bereitgestellt, das aber erst wirksam wird, wenn sich in ④ die Punktliste `task` als leer erweist. Der eigentliche Plotauftrag für die Gerade und für die Bälle, die in `yes_` angegeben werden, erfolgt in `plot1_`.

Ich empfehle, das Programm als Zusatz-Datei (utility file) in den Hintergrund zu laden. Damit braucht man niemanden mit dem Code zu belasten - und viel wichtiger, er wird nicht beschädigt. Wie Sie eine geladene Utility erkennen, das zeigt Ihnen Herausforderung 18.

Herausforderung 17

Bauen Sie in `hit()` eine Routine ein, die dafür sorgt, dass horizontale und vertikale Gerade nicht gezeichnet werden.

Herausforderung 18

Ändern Sie `hit()` so ab, dass Parabeln gesucht werden. Welche Änderung ist notwendig?

Herausforderung 19

Versuchen Sie, ein Trainingsprogramm für den Wurzelsatz von Vieta zu schreiben. Der PC gibt eine quadratische Gleichung – entweder immer in der gleichen Variablen, oder schwieriger mit zufälligen Variablen – vor und der Übende soll die Lösungen aus den Koeffizienten ermitteln können [26]. Wählen Sie dazu einen vernünftigen Bereich für die ganzzahligen Lösungen, dass es nicht zu schwierig wird.

Das Paket besteht aus insgesamt 3 Teilen:

`vieta()` stellt die Aufgabe, `ant(x1,x2)` überprüft die Lösung und verwaltet die Anzahl der gestellten Aufgaben und der richtigen Antworten und `results()` zeigt die Bilanz der Übungssitzung.

```
#1:   LOAD(C:\SEMINARE\Programmbuch\vieta.txt)
                    2
#2:   vieta() = (t  - 15·t + 54 = 0)
#3:   ant(9, 5) = falsch, sondern:  6, 9
                    2
#4:   vieta() = (o  - 8·o - 9 = 0)
#5:   ant(8, 1) = falsch, sondern:  -1, 9
                    2
#6:   vieta() = (l  + 9·l = 0)
#7:   ant(0, -9) = richtig!
                    2
#8:   vieta() = (r  + 18·r + 32 = 0)
#9:   ant(-16, -2) = richtig!
                      [ Aufgaben gestellt:   4 ]
#10:  results() =     [   davon richtig:     2 ]
                      [   das sind in %:    50 ]
```

Wie eine Vieta-Sitzung aussehen könnte zeigt der Bildschirmausschnitt.

`vieta.txt` (noch `-.txt`, weil direkt im Editor erstellt), wurde als Utility-file in den Hintergrund geladen.

Die Antworten von *DERIVE* auf `ant(x1,x2)=` fallen hier nicht besonders auf, allerdings ist es nicht so einfach, die Form „`falsch, sondern: -1, 9`“ zu erreichen.

Wenn Sie nur mit `APPEND` arbeiten, werden Sie ein Komma vor –1 nicht unterdrücken können. Hier greift eine Stringoperation ein (eine der vielen Neuerungen von *DfW5*):

```
APPEND("falsch, sondern:  ", STRING(w_↓1), ", ", STRING(w_↓2))
```

Oder Sie machen es noch ausgeklügelter und lassen die Übung blockweise mit einer beliebigen Anzahl von Aufgaben durchführen:

$$\text{blvieta(4)} = \begin{bmatrix} x^2 + 8\cdot x - 20 = 0 \\ m^2 + 9\cdot m - 136 = 0 \\ h^2 - 15\cdot h + 14 = 0 \\ e^2 - 3\cdot e - 10 = 0 \end{bmatrix}$$

$$\text{blant}\begin{bmatrix} -10 & 2 \\ -17 & 8 \end{bmatrix} = \text{Anzahl der Lösungen stimmt nicht!}$$

$$\text{blant}\begin{bmatrix} -10 & 2 \\ -17 & 8 \\ -1 & -14 \\ 5 & 2 \end{bmatrix} = \begin{bmatrix} \text{richtig!} \\ \text{richtig!} \\ \text{falsch, sondern: } 1, 14 \\ \text{falsch, sondern: } -2, 5 \end{bmatrix}$$

Hier müssen Sie sich natürlich die Anzahl der gestellten Aufgaben „merken“ und die Antworten müssen in der Reihenfolge der gestellten Aufgaben als Matrix eingegeben werden.

```
blant([-10,2;-17,8;-1,-14;5,2])=
```

Außerdem kann es notwendig sein, für jede einzelne Aufgabe die Variable, nach der die Gleichung zu lösen ist, zu ermitteln. Dazu hilft die `VARIABLES`-Funktion:
z.B.:

```
va:=task SUB i_,
va:=VARIABLES(va) SUB 1,
w_:=SOLUTIONS(task SUB i_,va),
```

Sie können aber auch die Matrix mit den Lösungen gleich bei der Erstellung der Aufgaben zusammenstellen und diese als globale Variable in die Beurteilung der Beantwortung „hinüberretten“.

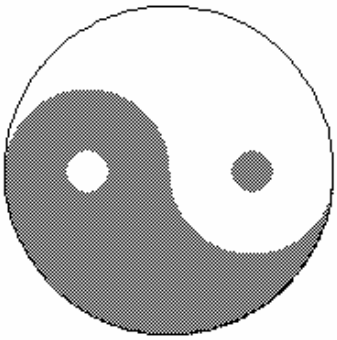
Weitere Literatur: [27], [28]

11 Zum Abschluss YIN und YANG

Als Abschluss soll ein einfaches und nicht so umfangreiches Programm verfasst werden, das das bekannte YIN/YANG-Symbol zeichnet [29]. Im Prinzip lässt sich das auch ohne Programm als eine Funktion vom Radius des umgeschriebenen Kreises formulieren, aber als Programm gewinnt die Prozedur sowohl an Deutlichkeit als auch an Lesbarkeit.

```
yinyang(r,cc,ca,serp,c1,c2):=PROG(
         "@ Kreisrand ",
         cc:=x^2+y^2=r^2,
         "@ gefüllte Kreisscheibe ",
         ca:=x^2+y^2<=r^2,
         "@ die beiden kleinen Kreisscheiben ",
         c1:=(x-r/2)^2+y^2<=(r/8)^2,
         c2:=(x+r/2)^2+y^2<=(r/8)^2,
         "@ die Schlangenlinie ",
         serp:=IF(x<-r,?,
                 IF(x<0,SQRT(r^2/4-(x+r/2)^2),
                   IF(x<r,-SQRT(r^2/4-(x-r/2)^2),?))),
         "@ logische Verknüpfung der Ungleichungen ",
         RETURN([cc,(ca AND y<=serp AND NOT(c2)) OR c1])
       )
```

```
#2:   yinyang(6)
```



Herausforderung 20

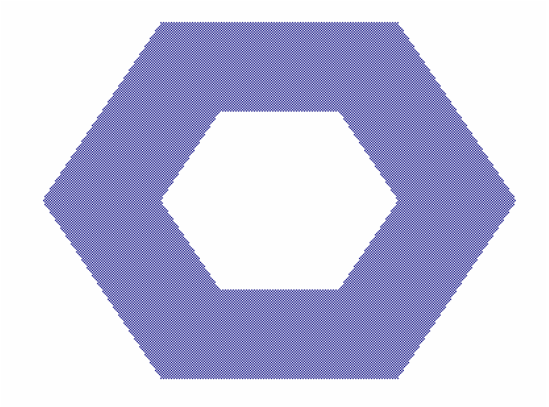
Stellen Sie das Symbol so dar, wie in es in der Abbildung am Beginn des Abschnitts zu sehen ist.

Herausforderung 21

Drehen Sie das YIN&YANG-Symbol um 90°.

Herausforderung 22

Versuchen Sie das dargestellte Sechseck zu erzeugen. (Am einfachsten wäre es, lineare Funktionen zu verwenden und mit Ungleichungen den schaffierten Bereich zu beschreiben). Das ist aber vielleicht nicht das kürzeste Programm.



Herausforderung 23

Erzeugen Sie eine Figur nach eigenem Design. (Polygone, Rosetten, Figuren, ...)

Literaturhinweise

[1] Vedic Mathematics using DERIVE, Nurit Zehavi a.o., DNL#43, 2001

[2] *The Natural Calculator*, Deepa Jeswani, Proceedings Portoroz 2000, bk-teachware

[3] *ACDCA 10 – Milton´s Problems*, Milton Lesmes & Josef Böhm, DNL#44, 2001

[4] *Mr Setif´s Treasure Box*, Robert Setif, DNL#3, 1991

[5] Schwarze Löcher im Zahlenchaos, Fr. Fricker, Spektrum d. Wissenschaft 01/2002

[6] *Dynamic Algebra*, René Hugelshofer, DNL#43, 2001

[7] *Programming with DERIVE - A Compiler*, Walter Schiller, DNL#27, 2000

[8] www.washlee.arlington.k12.va.us/DEPARTMENTS/Maths/data/content3.html

[9] *The MedMed-Regression with DERIVE*, Josef Böhm, DNL#41, 2001

[10] *Mastering the TI-92*, Larry Gilligan a.o., GILMAR Publishing, 1996

[11] *Computer Graphics with DERIVE*, Maria Koth, DNL#32-34, 1999

[12] *5000 Jahre Geometrie*, C. J. Scriba, Springer, 2001

[13] *An Introduction to Plane Geometry*, H.H.Baker, Cambridge Press, 1943

[14] *Cinderella*, Benutzerhandbuch , J.R.Gebert u.a., Springer, 2000

[15] *Maximum- and Minimum Values*, Erich Zott, DNL#12, 1993

[16] *Optimierungsaufgaben grafisch, numerisch ...*, Josef Böhm, bk-teachware, 1998

[17] *Discussion of a Curve - A Program*, Josef Böhm, DNL#15, 1994

[18] *Analysis mit DERIVE*, H.-J. Kayser, Dümmler Verlag, 1996

[19] *Reverse Discussion of a Curve*, Otto Reichel, DNL#18, 1995

[20] *Cubic Splines*, Reichel - Klingen - Böhm, DNL#18/19, 1995

[21] *Abituraufgaben mit Grafikrechnern ... Teil 1*, H.Knechtel u.a., Schroedel 2001

[22] *Mathematische Grundlagen der Computergrafik*, W.Luther, Vieweg, 1988

[23] *Numerical Analysis via DERIVE*, Steven Schonefeld, Mathware, 1994

[24] *Akima Splines Outperform Cubic Splines*, Robert Geruschkat, DNL#38, 2000

[25] *Mathe Trainer I für TI-89/92/92+*, Josef Böhm, bk-teachware, 2000

[26] *Vieta by Chance*, Jan Vermeylen, DNL#20, 1995

[27] *A Macro for DERIVE*, Johann Wiesenbauer, DNL#42, 2001

[28] *The Delayed Assignment*, Heinz Rainer Geyer, DNL#27, 1997

[29] *Mathographics*, Robert Dixon, Dover Publications, 1987

[30] *DERIVE Handbuch*, Vlasta Kokol-Volic & Bernhard Kutzler, TI-Publications

[31] *Materialien für den Einsatz von Grafikrechnern und Computeralgebra, Teil 1 : Differentialrechnung*, Heiko Knechtel u.a., westermann 2001

[32] *ACDC*, Rüdeger Baumann, DNL#47, 2002

[33] *Splines mit dem TI-92 und DERIVE*, Max-Günter Schröfel, VISIT-ME 2002 Proceedings, bk-teachware, 2002

Index