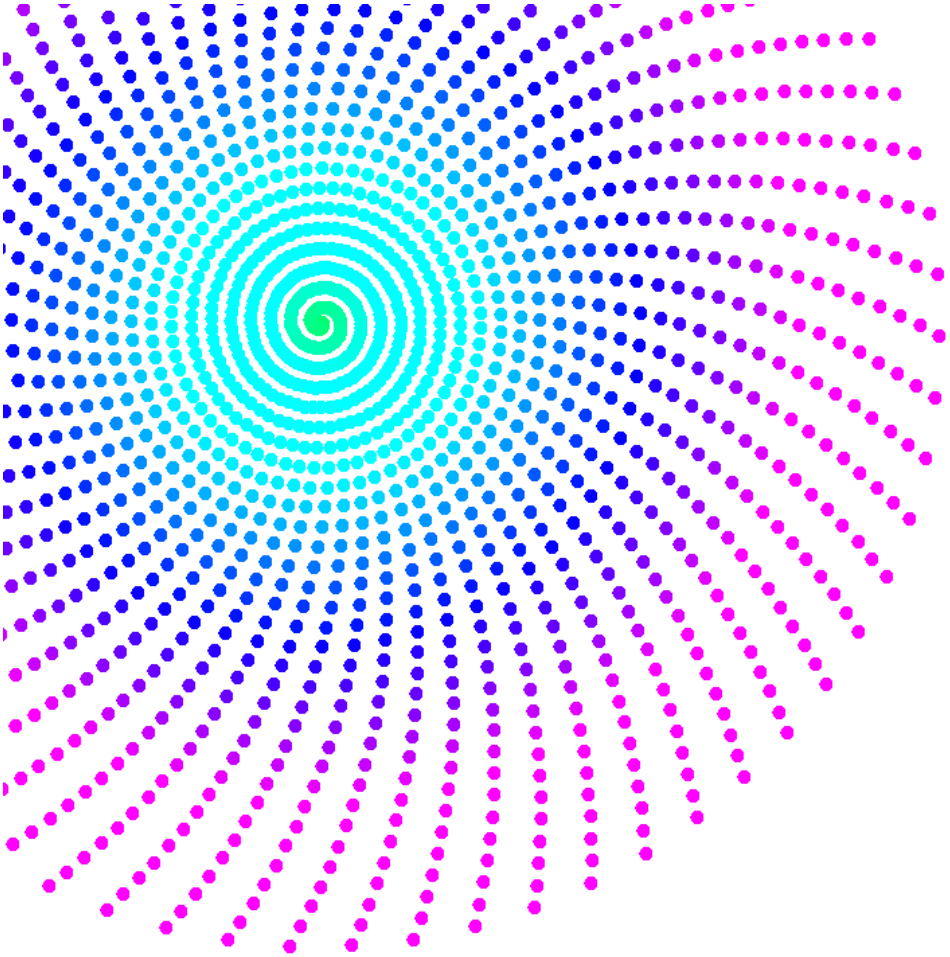
Hanfried Schlingloff

GPR-Basic

Ein Leitfaden zum Erlernen der Programmiersprache Basic



 Regensburg, Germany 2018

Basic („beginners all-purpose symbolic instruction code“) gehört (neben C und Pascal) zu den wenigen Programmiersprachen, die in den letzten fünf Jahrzehnten weltweite Verbreitung erlangten. Während im professionellen Bereich der Softwareentwicklung Pascal heute weitgehend durch C verdrängt worden ist, so konnte sich Basic wegen seiner einfachen und benutzerfreundlichen Struktur vor allem für die Anwendungs-Programmierung behaupten. Heute wird Basic in vielen Bereichen der Wissenschaft, Wirtschaft und Technik verwendet, z.B. zur Steuerung von Windows Anwendungen (Word, Excel, Access, Corel Draw etc.) oder für die Programmierung von Internet-Seiten. Aber auch professioneller Programmcode kann mit modernen Basic-Compilern erstellt werden.

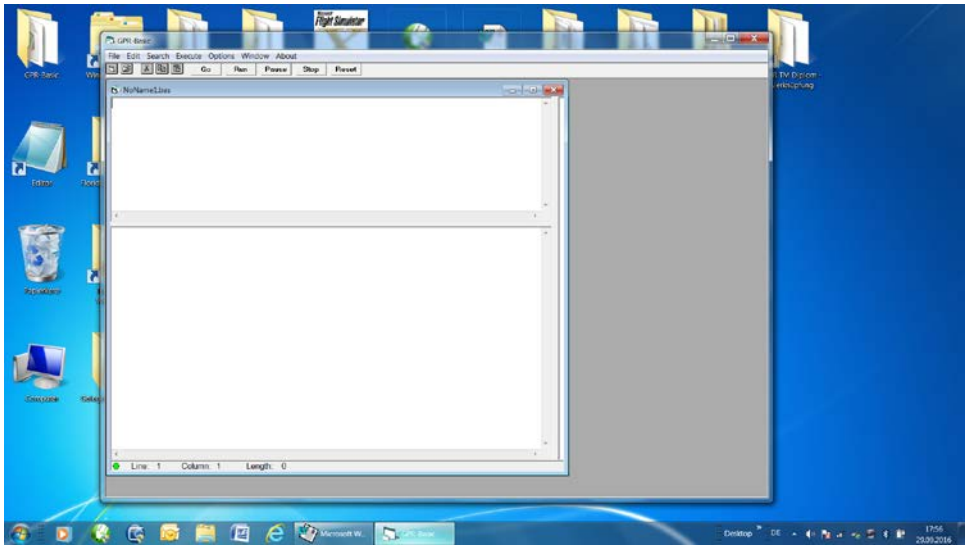
GPR-Basic ist ein Interpreter für einfachen Standard Basic Code („Übungs-Programme“).
Bei der Entwicklung dieses Programmes wurden leichte Erlernbarkeit und einfache Bedienung in den Vordergrund gestellt und dafür Abstriche in Flexibilität und Leistung in Kauf genommen. Für Anwendungen, die über das Lösen von einfachen Übungsaufgaben herausgehen (z.B. Softwareentwicklung, Steuerung von Windows-Programmen, Anwendungsprogrammierung, numerische Mathematik etc.) ist GPR-Basic also nicht geeignet. Eine dazu geeignete Programmierumgebung ist z.B. „Visual-Basic“ von Microsoft (früher Q-Basic). Die Syntax der Programmiersprache GPR-Basic orientiert an diesem heute weit verbreiteten Basic-Dialekt, ist aber aus offensichtlichen Gründen nicht immer exakt kompatibel und vor allem nicht annähernd so umfangreich.

1 Grundlagen des Programmsystems GPR-BASIC

1.1 Benutzeroberfläche

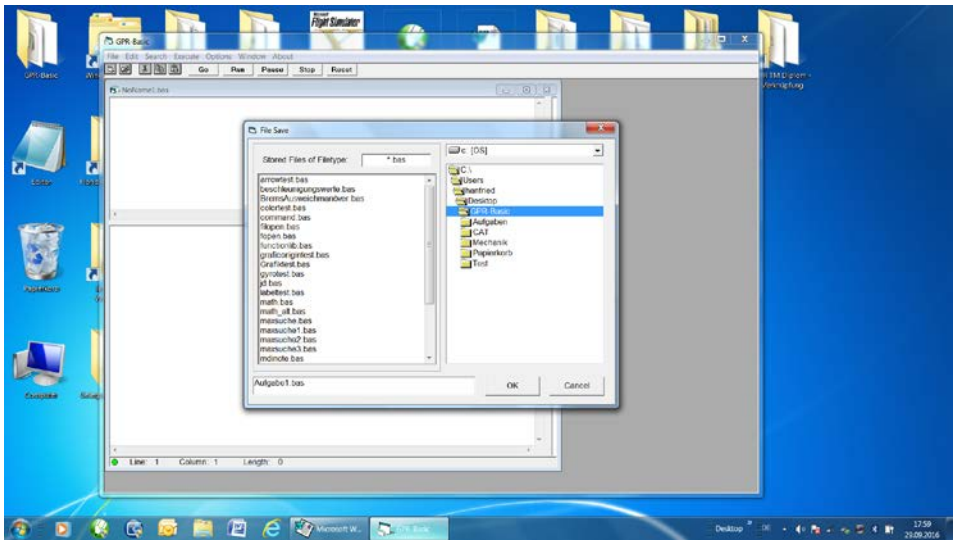
Installieren Sie das Programm GPR-Basic einfach dadurch, dass Sie einen Ordner („directory“) erzeugen, z.B. mit dem Namen GPR-Basic, in welchen Sie die Datei GPR-Basic.exe hineinkopieren („downloaden“). Wenn Sie wollen, können Sie auch gleich Unterverzeichnisse („Subdirectories“) fürs das Ordnen der Basic-Programme erzeugen (z.B. mit den Namen Aufgaben, Test, Papierkorb etc.). Verwenden Sie dazu die Funktionen des Windows Betriebssystems. Klicken Sie die Datei GPR-Basic.exe an. Es erscheint in der Bildschirmmitte ein großes graues Fenster. Wie Sie es von anderen Windows-Programmen gewohnt sind, können Sie mit dem „File-Menü“ (bzw. mit den entsprechenden Symbolen im „Toolbar“) z.B. ein vorhandenes Basic Programm öffnen oder ein neues (leeres) Programm erzeugen, oder GPR-Basic beenden. Der Menüpunkt „Options“ kann im Moment nichts weiter als den „Toolbar“ ein- und ausschalten, und im Menüpunkt „About“ erfahren Sie, dass das Programm GPR-Basic durch das Copyright geschützt ist. Autoren von Computerprogrammen nehmen im Allgemeinen ihre Eigentumsrechte sehr ernst und haben kein Verständnis für irgendwelche Verletzungen dieser Rechte. Also, bitte beachten Sie: Das Programm GPR-Basic mit dieser Dokumentation ist keine frei verfügbare „Open Domain Software“, sondern unterliegt klaren Nutzungsrechten!

Wenn Sie dann mit dem Menupunkt "New" im "File"-Menu (oder der Schaltfläche links darunter) eine neue Datei erzeugen, erscheint in etwa folgendes Fenster:



Es gibt jetzt viel mehr Menupunkte und auch der Toolbar hat sich geändert. Im Filemenu können sie die neue Datei nun auch schließen, speichern oder löschen. Außerdem gib es Menus für das editieren, durchsuchen und ablaufen lassen von einem Basic-Programm.

Probieren Sie nun mal das Erstellen eines Basic Programmes. Nachdem Sie mit dem Unterpunkt „New“ des File-Menüs ein Unterfenster erzeugt haben, ändern Sie den Namen dieses Fensters mit dem Untermenüpunkt „Save As“ vom Namen NoName1.bas z.B. auf den neuen Namen „Aufgabe1.bas“:



Das Fenster „Aufgabe1.bas“ ist in zwei Bereiche aufgeteilt (eigentlich in drei, aber der Grafik-Bereich bleibt vorerst rechts sehr schmal versteckt): Der Protokollbereich oben und der Programmbereich unten. Ebenso wie die Größe des Fensters lässt sich die Größe dieser Bereiche durch „Ziehen mit der Maus“ verändern.

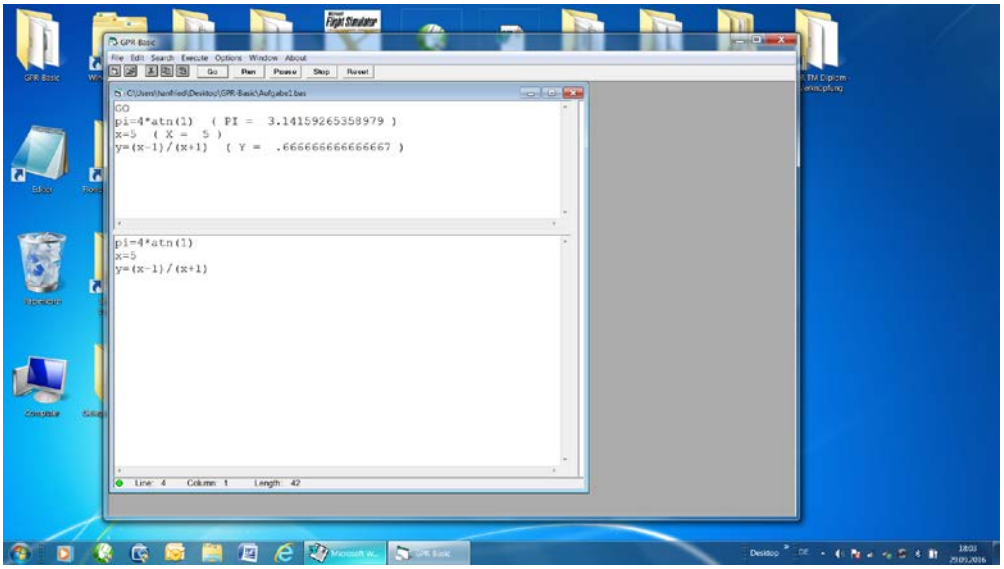
Auch der Toolbar hat sich verändert und enthält nun Symbole zum Ausschneiden, Kopieren und Einfügen von Text im Programmbereich des Fensters. Außerdem enthält er jetzt fünf Schaltflächen: „Go“, „Run“, „Pause“, „Stop“ und "Reset". Wenn Sie diese jetzt anklicken, passiert nicht viel (außer dass diese Funktionen im Protokollbereich protokolliert werden), denn Sie haben ja noch kein Programm eingetragen.

„Go“ löst die Programmausführung im Einzelschritt-Modus aus (mit Protokollierung im Protokollbereich), „Run“ löst die normale Programmausführung aus (ohne Protokollierung), „Pause“ unterbricht die normale Programmausführung und geht wieder zum Einzelschritt-Modus zurück, „Stop“ erzwingt eine Beendigung des Programmes und "Reset" beendigt den Programmlauf und löscht sämtliche gespeicherten Variablen.

Der obere Bereich ist der Protokollbereich. Man kann hier zum Beispiel so rechnen wie mit einem Taschenrechner: Wenn Sie z.B. 10/81 und dann die Taste ENTER eintippen, so erscheint: `10/81 ( = .123456790123457 ).` Man kann die Berechnungen aber auch in Variablen abspeichern. Tippen Sie z.B. x=10/81 ein, so erscheint die Meldung: `X=10/81 ( X = .123456790123457 ).` GPR-Basic merkt sich den Wert der Variablen *x*, d.h. Sie könnten *x* in einer nachfolgenden Berechnung verwenden.

Das Basic-Programm gehört aber in den Editor, d.h. in den unteren Bereich des Fensters. Probieren Sie nun einmal, Basic-Anweisungen (im Einzelschritt oder im normalen Lauf) auszuführen. Tragen Sie dazu folgende Zeilen in den "Programmbereich" (Editor) ein:

```
pi=4*atn(1)
x=5
y=(x-1)/(x+1)
```



Klicken Sie mit der Maus die Schaltfläche „Go“ dreimal an und verfolgen Sie die Protokollierung der drei Anweisungen im Protokollbereich (im oberen Teil des Fensters). Es sollte etwa folgendes ausgegeben werden:

```
Go
pi=4*atn(1)  ( PI =  3.14159265358979 )
x=5  ( X =  5 )
y=(x-1)/(x+1)  ( Y =  .666666666666667 )
```

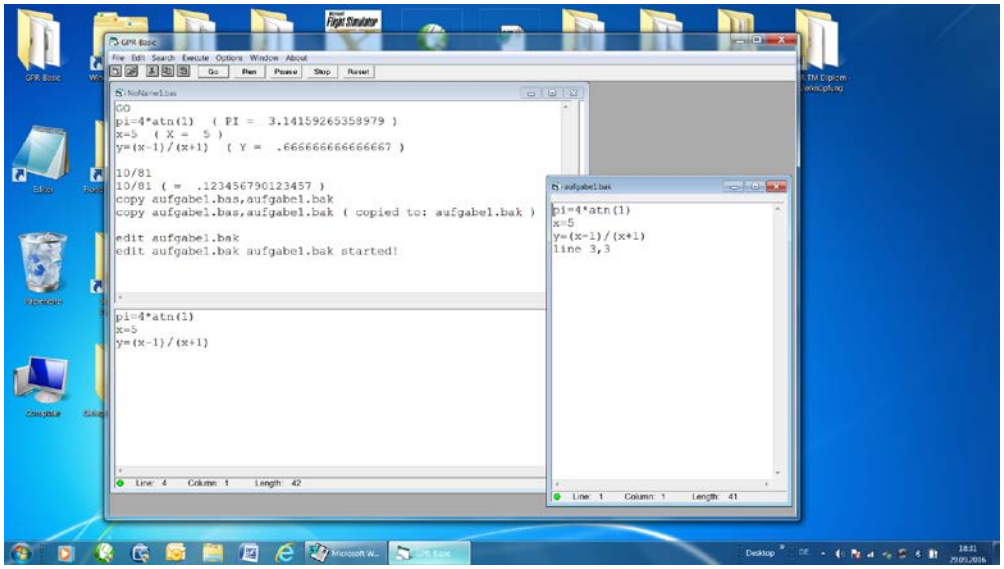
Sollte Ihnen die Schriftart nicht gefallen oder sollte Ihnen die Schrift zu klein oder zu groß erscheinen, so können Sie das mit den Untermenüpunkten „Font“ bzw. „Font Size“ (im Menü Options) ändern. GPR-Basic merkt sich Ihren Wunsch (soll heißen: Speichert diese Information in der „Registry“-Datenbank des Windows-Betriebssystems ab), sodass Sie beim nächsten Programmstart wieder Ihre gewünschten Einstellungen vorfinden.

Sie können auch einmal die Schaltfläche „Run“ anklicken, um die normale Programmausführung auszuprobieren, aber das bringt noch nichts, denn bei der normalen Programmausführung wird der Programmablauf nicht protokolliert. Das Programm wird zwar ausgeführt, aber in diesem Fall sieht man es nicht und auch sonst geschieht nicht viel.
Der Protokollbereich des Fensters ist aber nicht nur zum Rechnen oder zum Protokollieren des Programmablaufs gedacht, sondern er verfügt auch über Ein- und Ausgabefunktionen und dient zur Ablaufsteuerung.

Viele Basic-Anweisungen lassen sich auch im Protokollbereich ausführen. Zum Beispiel, tippen sie die Anweisung BEEP 2000,1 in die nächste Zeile des Protokollbereichs ein und dann die Taste ENTER. Die Anweisung wird ausgeführt; ein Signalton von 2000 Hertz und einer Sekunde Dauer ertönt. Wenn Sie diesen Nervtöter nochmals (vielleicht länger) hören wollen, tippen Sie einfach die Taste ↑, und ändern Sie den Sekundenwert. (mit den Tasten ↑ und ↓ können Sie die im Protokollbereich eingetippten Zeilen reaktivieren).

Oft möchte man während der Programmierung Betriebssystemfunktionen ausführen, z.B. Verzeichnisse erzeugen oder löschen, Dateien kopieren oder editieren. Das geht zwar auch mit den entsprechenden Befehlen im Basic-Programm, im Allgemeinen wird man es aber eher „interaktiv“ machen. Als Beispiel tippen Sie in den Protokollbereich folgendes ein:

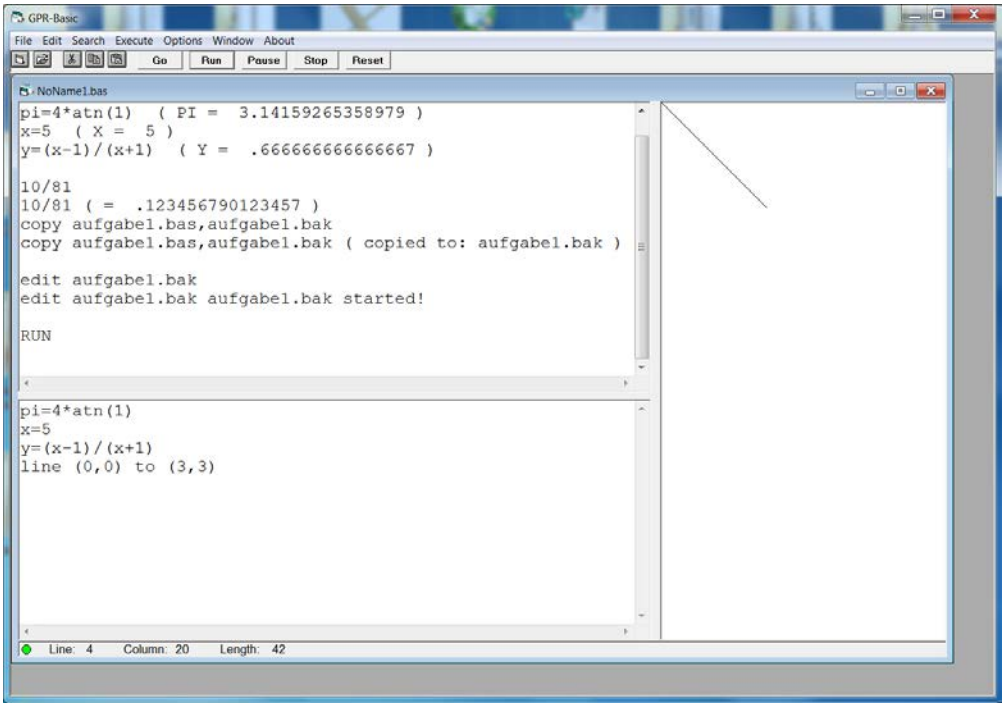
```
copy aufgabe1.bas , aufgabe1.bak
edit aufgabe1.bak
```



GPR-Basic kopiert dann die Datei „Aufgabe1.bas" und öffnet ein weiteres Fenster zum Editieren der Datei „Aufgabe1.bak“. Löschen Sie diese neue Datei wieder, z.B. mit dem Menupunkt "Delete" im File-Menu (oder schließen Sie das Fenster durch Anklicken des Kreuzes rechts oben und löschen Sie die Datei mit dem Befehl `DEL Aufgabe1.bak` im Protokollbereich des Fensters der Datei Aufgabe1.bas).

Wenn Sie mit der Maus den Mauszeiger so in die Fenstermitte ganz rechts positionieren bis der Mauszeiger die Form eines horizontalen Doppelpfeils annimmt (so etwa ↔), dann können Sie durch „Ziehen mit der Maus” den Grafikbereich des Fensters öffnen. Ziehen Sie den Mauszeiger dazu etwa in die Mitte des Fensters. Die Scrollbars im Protokollbereich oder im Programmbereich werden aktiv, sobald der darin enthaltene Text größer als der angezeigte Bereich wird.

Zum Beispiel, der Grafikbefehl `LINE (0,0) TO (3,3)` zieht eine Linie von der Position 0,0 (in der linken oberen Ecke des Grafikbereichs) zur Position 3,3 (die voreingestellte Einheit ist cm). Tippen Sie den Befehl in den Programmbereich ein und lösen Sie ihn durch „Go“ (oder durch „Run“) aus:



Natürlich hätten Sie diese Grafik-Anweisung auch im Protokollbereich ausführen können. Mit den Menupunkten "Clear Protocol" und "Grafic Delete" im "Edit"-Menu lassen sich Protokollbereich beziehungsweise Grafikbereich wieder säubern.

Die Fußzeile des Fensters enthält folgende, gelegentlich nützliche Informationen:

1.) Der grüne Punkt links unten soll eine Ampel symbolisieren (und heißt nicht etwa, dass das Programm gemäß der europäischen Verpackungsverordnung recyclebar ist ☺).
Grün: das Programm wartet auf Eingabe;
rot: das Programm läuft, und
gelb: das Programm läuft im Einzelschrittmodus.
2.) Line: die aktuell ausgeführte Programmzeile bzw. die Zeile im Programmbereich, wo der Cursor gerade positioniert ist;
3.) Column: die Spalte der Cursorpositionierung und
4.) Length: die Anzahl der Zeichen (Bytes) des Basic-Programmes.

Wenn Sie das Programm sichern wollen, so speichern Sie es mit “Save” (im Filemenü) ab.

1.2 Variablen, Konstanten und Wertzuweisungen

Ein Basic-Programm ist eine ablauffähige Liste von Basic-Anweisungen, es wird Anweisung für Anweisung aufeinanderfolgend ausgeführt; das Programm kann aber Sprünge, Verzweigungen, Schleifen, Unterprogramm-Aufrufe etc. enthalten.

In diesem Abschnitt soll nun besprochen werden, was man unter Basic-Anweisungen versteht. Basic ist „zeilenorientiert", d.h. i.A. (Abküfi ☺) enthält jede Zeile eine Anweisung. Man kann aber auch mehrere Anweisungen in dieselbe Zeile schreiben, indem man sie durch einen Doppelpunkt (gefolgt von einem Leerzeichen) voneinander trennt, z.B.:

```
A=1 : B=1 : Pi= 3.14159265358979
E=       _
Exp(1)
```

Der Underscore _ am Zeilenende bedeutet „es folgt eine Fortsetzungszeile", d.h. eine lange Anweisung kann im Programmbereich auch auf mehrere Zeilen aufgeteilt werden.

Kommentare lassen sich einfügen, entweder am Ende einer Zeile abgetrennt durch das Kommentarzeichen (Zeichen ' über dem Doppelkreuz # auf der deutschen Tastatur) oder durch eine separate Kommentarzeile welche mit diesem Zeichen beginnt. Bitte benutzen Sie Kommentare sparsam und sinnvoll. Nicht sinnvoll ist z.B. folgender Kommentar:

```
A=1  ' hier wird der Variablen A der Wert 1 zugewiesen
```

denn das sieht man ja auch ohne einen Kommentar.

Leerzeilen und (zusätzliche) **Leerzeichen** haben in Basic i.A. keine Wirkung auf den Ablauf, sie helfen aber sehr, das Programm übersichtlicher und lesbarer zu gestalten. Die Anweisungen innerhalb von For-Next Schleifen und If-Endif Konstruktionen sollten z.B. immer eingerückt werden, sonst lässt sich bei komplizierten Verschachtelungen kaum noch erkennen, welches Next zu welchem For gehört bzw. welches Endif zu welchem If.

Wertzuweisungen sind Anweisungen, bei denen Variablen Werte zugewiesen werden. Zum Beispiel, die Anweisung `A=1` ist eine Wertzuweisung, es wird der Variablen `A` der Wert 1 zugewiesen. Dabei steht rechts vom Gleichheitszeichen ein „arithmetischer Ausdruck" (in diesem Beispiel besteht der arithmetische Ausdruck nur aus einer Konstanten, dem Wert 1), und links vom Gleichheitszeichen steht der formale Name für einen Speicherplatz (in diesem Beispiel die Variable `A`).

Es muss betont werden, dass `A=1` auf keinen Fall eine Gleichung ist! Was rechts vom Gleichheitszeichen steht wird ausgerechnet und in den Speicherplatz links reingeschrieben. Also wäre z.B. `1=A` falsch und nicht etwa das gleiche (tatsächlich kann `1=A` in einem Basic-Programm schon vorkommen, z.B. in `If 1=A Then` ; dabei handelt es sich aber um einem Vergleich und nicht um eine Gleichung). Gleichungen im mathematischen Sinne gibt es in der Programmiersprache Basic nicht. Eine häufig in Programmen vorkommende Anweisung ist die sogenannte Inkrementierung `A=A+1`. Hatte die Variable `A` vor Ausführung dieser Anweisung z.B. den Wert 3, so hat sie nach der Ausführung der Anweisung den Wert 4. Als Gleichung wäre `A=A+1` einfach unsinnig.

GPR-Basic kennt drei **Typen von Variablen**:

1.) Rechenvariablen (Typ DOUBLE oder SINGLE). Sie sind geeignet um Zahlenwerte mit Vorzeichen, Mantisse, Gleitpunkt und Exponent aufzunehmen, und haben eine typische Genauigkeit von 15 Stellen. Beispiele: $1.23456789012345 \cdot 10^{32}$, -17.23, 0.0000 etc. (Zwischen Double und Single wird in GPR-Basic nicht unterschieden.)

2.) Zählvariablen (Typ LONG oder INTEGER). Es sind ganze Zahlen, ohne Gleitpunkt und ohne Exponenten, wohl aber mit Vorzeichen. Beispiele: -20016, 0, 1, 24 etc.

3.) Zeichenkettenvariablen (Typ STRING). Sie enthalten Text; man kann mit ihnen zwar nicht rechnen, aber man kann sie miteinander vergleichen, sie miteinander verknüpfen oder sie anderweitig verändern. Beispiele: "OTH-Regensburg", "GPR-Basic" etc.
String-Konstanten dürfen natürlich Leerzeichen enthalten und werden deswegen in Hochkommazeichen " eingeklammert, wobei die Hochkommata nicht Teil der Zeichenkette sind. Die Stringkonstante "GPR-Basic" besteht also aus genau 9 Zeichen (9 Bytes).

Bedenken Sie: Für den Computer ist es ein großer Unterschied, ob sie den Rechenwert 0.00000000000 oder aber die Nummer 0 oder aber das Zeichen "0" meinen. Diese Werte werden unterschiedlich abgespeichert. Für uns ist 0 zwar immer das gleiche, aber schon ein Mathematiker kann uns erklären, dass Null nicht etwa „Nichts“ bedeutet. Haben Sie sich schon einmal die Frage gestellt, ob null Birnen gleich null Äpfel sind? Warum bei "null" eine Ausnahme machen, wenn nicht einmal der verglichene Typ übereinstimmt?

Namen für Variablen unterliegen gewissen Einschränkungen:

1.) Sie dürfen Buchstaben von A-Z und Ziffern von 0-9 enthalten, müssen aber mit einem Buchstaben anfangen. Zulässige Variablennamen wären also z.B. A, A1, A01, Schub, Name, Name1, X, Y, DYDX etc. Nicht zulässig wären z.B. 1A (fängt nicht mit einem Buchstaben an) oder F(X) (enthält Sonderzeichen, die Klammern) oder DY/DX (/ ist ebenfalls ein Sonderzeichen) oder FLÄCHE (auch Ä ist ein Sonderzeichen).
Basic-Schlüsselworte (For, Next, Do, Loop, Goto etc.) sind als Namen nicht geeignet.

2.) Zwischen Groß und Kleinschreibung wird nicht unterschieden, also z.B.: FLAECHE, Flaeche oder flaeche, alle drei sind zulässige Variablennamen, sie bezeichnen aber auch exakt den gleichen Speicherplatz.

3.) Variablennamen sollten das, was die Variable enthält, gut kennzeichnen. Wenn z.B. die Variable X einen Eingabewert x und die Variable Y einen Funktionswert $y(x)$ enthalten soll, dann ist es nicht gescheit, den Ableitungswert $y'(x)$ in einer Variablen Z abzuspeichern. Nennen Sie den Ableitungswert besser DYDX oder YSTRICH, dann weiß jeder, was darin abgespeichert ist (Y' würde nicht gehen, denn der Strich ist ja auch ein Sonderzeichen – das Kommentarzeichen).

4.) Variablen sollten nicht zu kurz und nicht zu lang sein. Sind sie zu kurz (z.B. A, B etc.), weiß man bald nicht mehr was drin steht; sind sie zu lang (z.B. Donaudampfschiff...), so wird das Programm erstens lang und unübersichtlich; zweitens besteht die Gefahr von Schreibfehlern. Eine durch einen Schreibfehler entstandene neue Variable wird in den meisten Programmiersprachen mit Null initialisiert und eiskalt ohne Warnung verwendet, das Ergebnis stimmt nicht und der Fehler ist schwer zu lokalisieren.

GPR-Basic kennt zwei Formen der **Typenfestlegung** :

Variablen, deren Typ nicht festgelegt worden ist, werden in GPR-Basic automatisch als Rechenvariablen behandelt (Typ DOUBLE). Falls ein anderer Typ für eine Variable gewünscht wird, so gibt es zwei Möglichkeiten, das zu bewerkstelligen:

1.) Die explizite Typendeklaration mit der Anweisung DIM vor dem erstmaligen Benutzen der Variablen (am besten am Programmanfang). Zum Beispiel die Anweisung

```
Dim A as Double, B as Long, C as String
```

definiert die Typen für Variablen `A,B,C` explizit.

2.) Typendeklaration durch ein Sonderzeichen (Suffix) am Ende des Varablennamens. Zum Beispiel x# ist in jedem Fall eine Variable vom Typ DOUBLE (wegen dem #); x! ist SINGLE (wegen dem !-Zeichen); i& eine Variable vom Typ LONG (wegen &); i% ist vom Typ INTEGER (das %-Zeichen) und A$ ist vom Typ STRING (wegen $). Aber wie erwähnt haben in GPR-Basic Single Variablen auch Double Genauigkeit (8 Bytes), und Integer Variablen werden wie Long Variablen mit einer Stellenzahl von 4 Bytes abgespeichert.

Empfehlung: Verwenden Sie für DOUBLE-Variablen die implizite Typendeklaration (d.h. gar nichts machen) und für STRING-Variablen die Deklaration mit dem Suffix $. DOUBLE-Variablen braucht man am häufigsten, String-Variablen eher selten (z.B. ist `X` wohl eine DOUBLE-Variable und `Passwort$` eine STRING-Variable). Statt Variablen vom Typ LONG können oft auch Double-Variablen verwendet werden, andernfalls können Longvariablen entweder mit der DIM Anweisung oder mit dem Suffix & deklariert werden.

1.3 Arithmetische Ausdrücke und Funktionen

In einer Wertzuweisung steht rechts vom Gleichheitszeichen ein arithmetischer Ausdruck; aber arithmetische Ausdrücke können auch in anderen Basic-Anweisungen vorkommen, z.B. in Vergleichen (If), in Ausgabebefehlen (Print) und in Unterprogrammaufrufen (Call). Ein arithmetischer Ausdruck besteht aus Variablen, Konstanten und Funktionen, miteinander verknüpft durch Rechenoperatoren (+ , – , * , / , ^). Eine wesentliche Eigenschaft eines arithmetischen Ausdrucks ist es, dass er ausrechenbar ist und nach Berechnung einem Zahlenwert entspricht. Beispiele für arithmetische Ausdrücke sind: A+1, 4*ATN(1), X^2 etc. Aber auch einfache Variablen oder Konstanten sind strenggenommen nichts anderes als arithmetische Ausdrücke, z.B.: Pi , 2.1 , 0 etc.

Basic wertet arithmetische Ausdrücke von links nach rechts aus und beachtet dabei Prioritätsregeln, d.h. Punkt vor Strichrechnung. Runde Klammern ( ) können verwendet werden und haben dann die höchste Priorität, also $\sin\alpha^2$ zum Beispiel wird in Basic zu `SIN(ALFA^2)` und $\sin^2\alpha$ wird zu `SIN(ALFA)^2`. Die Ausführungsrichtung spielt schon eine gewisse Rolle, z.B. könnte für den Ausdruck `1/1/2` der Wert 0.5 berechnet oder aber auch der Wert 2. Klammern setzen ist also im Zweifelsfall immer sinnvoll.

GPR-Basic verfügt über viele nützliche Funktionen, eingebaute Funktionsunterprogramme. Diese Funktionen werden in arithmetischen Ausdrücken genauso wie Variablen behandelt; i.A. erkennt man sie daran, dass sie formale Parameter besitzen (meistens besitzen sie genau einen solchen Parameter).

Zum Beispiel, SIN(X) ist eine Funktion, sie berechnet den Sinus (natürlich im Bogenmaß) vom Wert X. Diese Variable X ist der formale Parameter, er ist direkt an die Funktion angehängt und in runde Klammern eingeschlossen. Werden bei einer Funktion mehrere formale Parameter benötigt, so werden diese innerhalb der runden Klammern angegeben und durch Kommata voneinander getrennt, z.B.: RGB(0, 255, 100) (die RGB-Funktion wird für Grafik verwendet und berechnet die Farbe aus rot, grün, blau).
Gelegentlich haben Funktionen aber auch gar keine Parameter und sind dann von Variablen kaum zu unterscheiden (z.B. die Funktion TIMER, sie gibt die seit Mitternacht vergangenen Sekunden zurück). An der Stelle, wo Variablen oder Konstanten als formale Parameter stehen, können natürlich auch arithmetische Ausdrücke stehen. Man kann also z.B. die Quadratwurzel von 2 als SQR(2) oder als SQR(1+1) berechnen, und es ist natürlich auch zulässig, dafür: SQR(SQR(4)) oder: SQR(SQR(SQR(16))) zu schreiben.
Wichtige mathematische Funktionen sind (x ist dabei immer der formale Parameter):

Abs(x)	Absolutbetrag
Sqr(x)	Quadratwurzel
Exp(x)	die e-Funktion
Log(x) oder Ln(x)	natürlicher Logarithmus
Sin(x)	Sinusfunktion (Winkelfunktionen immer im Bogenmaß)
Cos(x)	Cosinusfunktion
Tan(x)	Tangens
Cot(x)	Cotangens
Atn(x) oder Atan(x)	Arcustangens
Asin(x)	Arcussinus
Acos(x)	Arcuscosinus
Acot(x)	Arcuscotangens
Sign(x)	Vorzeichen (-1 für negatives x, sonst +1)
Int(x)	nächst kleinerer (bzw. gleicher) Integer-Wert
Fix(x)	Integer-Wert, x wird hinter dem Komma abgeschnitten
Max(x1,x2,x3....,xn)	Maximalwert aus den übergebenen Argumenten
Min(x1,x2,x3....,xn)	Minimalwert aus den übergebenen Argumenten
Timer	die seit Mitternacht vergangenen Sekunden

Alle Winkelfunktionen werden natürlich immer im Bogenmaß berechnet.

1.4 Strings und Stringfunktionen

Basic ist zwar keine „zeichenorientierte“ Programmiersprache (wie z.B. C), verfügt jedoch über etliche sinnvolle Funktionen zur Stringbearbeitung. Strings (oder Zeichenketten) sind nützlich zum Abspeichern von Namen, Passwörtern oder anderem Text. Man kann damit natürlich nicht rechnen, aber man kann Strings miteinander vergleichen, miteinander verketten oder „Substrings“ suchen, oder den in Strings enthaltenen Text verändern.

Zur Verkettung von zwei Zeichenketten wird das Zeichen + verwendet. Zum Beispiel A$="OTH"+"-"+"Regensburg" ergibt "OTH-Regensburg" als Inhalt der Stringvariablen A$. Mehr Operationen als + gibt es nicht, für den Rest der notwendigen Manipulationen stehen Stringfunktionen zur Verfügung. Zur Zeichenbearbeitung gibt es:

Chr(x)	erzeugt das zum Zeichencode x ($0 \leq x \leq 255$) gehörende Zeichen,
Asc(x$)	gibt den ASCII-Zeichencode eines Zeichens x$ zurück.

Zum Beispiel, Chr(97) liefert das kleine „a“, denn 97 ist nach der ASCII-Tabelle der Zeichencode dafür (siehe Anhang, die ASCII-Tabelle definiert die Zuordnung einer Bitfolge der Länge ein Byte zu einem Zeichen). Etliche Zeichen der ASCII-Tabelle sind aber nicht darstellbar, z.B. das Zeichen Escape (Code 27), CR „Carrige Return“ (Code 13) oder LF „Line Feed“ (Code 10). Die Funktion Asc gibt den ASCII Code eines Zeichens zurück, z.B. Asc("a") liefert also 97. Eine Stringvariable kann natürlich auch nicht darstellbare Zeichen enthalten (z.B. kann man einen Zeilenvorschub durch Carrige Return + Line Feed erreichen: …+Chr(13)+Chr(10)+…). Zum Umwandeln von Strings in Zahlenwerte (oder umgekehrt) gibt es die Funktionen:

Val(A$)	berechnet, wenn möglich, den Zahlenwert eines Strings A$ aus dem Inhalt.
Str(x)	wandelt den Wert der Rechen- oder Zählvariablen x in einen String um, und
Format(x,F$)	wandelt ebenfalls den Variablenwert x in einen String um, wobei jetzt allerdings ein Format-String F$ bestimmt, wie das geschehen soll.

Zum Beispiel, wenn die Zeichenkette A$ den Wert "100" enthält, so wandelt die Funktion Val(A$) diese Zeichenkette in den Zahlenwert 100 um. Die Funktion Str(x) wandelt den Wert x in einen String um, überlässt es aber Basic, zu entscheiden, wie das geschehen soll. Format dagegen verwendet als zweiten Parameter einen Formatstring, der die Art der Umwandlung bestimmt. Wenn x zum Beispiel den Wert π beinhaltet, so gibt Format(x,"#.###") den String "3.142" zurück. Die Zeichen # sind dabei Platzhalter und #.### bedeutet: „drei Zeichen nach dem Komma“. Str(x) würde dagegen "3.14159265358979" zurückgeben.
Zum Bearbeiten von Strings gibt es die folgenden Funktionen:

Len(A$)	gibt die Länge von A$ als Zahlenwert zurück (auch Strlength(A$)),
Instr(A$,B$)	sucht das erste Auftreten des Substrings B$ im String A$, gibt diese Position zurück oder aber den Wert 0, wenn B$ in A$ nicht vorkommt,
Mid(A$, I, L)	schneidet einen Substring aus A$ heraus, beginnend mit dem I-ten Zeichen und L Zeichen lang, und gibt diesen Substring zurück,
Mid(A$, I)	ein Substring beginnend mit dem I-ten Zeichen bis zum Ende ("right"),
Mid(A$, 1,L)	ein Substring vom Anfang bis zum L-ten Zeichen ("left"),
Strchr(A$,Z$,I)	schreibt das Zeichen Z$ an die Stelle I in den String A$ und gibt den Ergebnis-String zurück.

Zum Beispiel Len("0123") liefert 4 als Ergebnis; Instr("0123", "2") liefert 3 (weil 2 an der dritten Stelle gefunden wird); und Mid("0123",3,2) liefert "23".Schließlich gibt es noch:

Ucase(A$)	Upper case, formt Kleinbuchstaben im String A$ in Großbuchstaben um;
Lcase(A$)	Lower case, wandelt Großbuchstaben in A$ in Kleinbuchstaben um;
Time	Gibt die Systemzeit als String zurück, z.B. als "21:24:39";
Date	Gibt das Datum als String zurück, z.B. "29.1.2018".

1.5 Ein und Ausgabe

Die einfachste Methode, Zahlenwerte in ein Basic Programm einzugeben, ist die Wertzuweisung. Wenn man zum Beispiel der Variablen X im Programm den Wert 1 zuweisen möchte, so kann man natürlich einfach in das Programm die Anweisung `X=1` schreiben. Soll für einen erneuten Programmstart der Wert geändert werden, z.B. auf `X=2`, so kann man das leicht durch Editieren mit der Maus und der Tastatur bewerkstelligen. Die Methode ist ungeeignet, wenn sich der Eingabewert erst während des Programmablaufes ergibt, z.B. wenn man das Programm interaktiv steuern möchte. Die dazu geeigneten GPR-Basic Anweisungen heißen Input und Lineinput. Schreib man z.B. im Programm

```
Input X
```

so unterbricht das Programm an dieser Stelle seinen Ablauf, gibt `X=` in die nächste Zeile des Protokollbereichs aus und wartet darauf, dass man einen Zahlenwert eingibt (genauer gesagt: dass man die Wertzuweisung `X=` vervollständigt und mit ENTER abschließt).
Der eingegebene Wert wird der Variablen X zugewiesen (bzw. der eingegebene arithmetische Ausdruck wird ausgewertet und dann der Variablen X zugewiesen); und die Programmausführung wird fortgesetzt (der Befehl Input funktioniert auch mit mehreren Variablen getrennt durch Kommata, z.B. `Input X,Y,Z`).

Der Befehl `Lineinput A$` wartet darauf, dass man im Protokollbereich eine ganze Zeile eintippt und mit Enter abschließt. LineInput weist diesen Text dann der Stringvariablen `A$` zu, bevor die Programmausführung fortgesetzt wird (LineInput ist hauptsächlich für die Dateibearbeitung gedacht und funktioniert nur mit einer Stringvariablen, hier z.B.: A$).

Die GPR-Basic-Funktion Inputbox(String$) ist geeignet, zur Werteingabe eine Eingabebox erscheinen zu lassen. Wenn man z.B. der Variablen A während des Programmablaufes an einer bestimmten Stelle im Programm einen Zahlenwert zuweisen möchte, so kann man an dieser Stelle des Programms folgende Anweisung schreiben:

```
A=Val(Inputbox("A="))
```

Die Funktion Inputbox unterbricht die Programmausführung, erzeugt ein Fenster mit dem Prompt `A=` und wartet darauf, dass man einen Zahlenwert eingibt (und mit der Schaltfläche OK abschließt). Inputbox liest den eingegebenen Wert als String und gibt ihn dann an die Funktion Val weiter, welche den Wert in einen Zahlenwert umwandelt. Dieser Wert wird dann der Variablen `A` zugewiesen (Inputbox kann auch als Messagebox zur Nachrichtenausgabe benutzt werden, z.B `A$=Inputbox("Eingabefehler!")`

Besonders, wenn das Programm lang ist und viele Eingabewerte benötigt, empfiehlt es sich, vor die INPUT Anweisung eine PRINT Anweisung zu schreiben, z.B.:

```
Print "Bitte geben Sie einen Wert ein:"
```

Die PRINT Anweisung ist eine Ausgabeanweisung, die Ausdrücke auswertet und die Ergebnisse in den Protokollbereich des Fensters schreibt. Dabei kann mit einer PRINT Anweisung eine ganze Liste von Ausdrücken (getrennt durch Kommata) ausgegeben werden
(z.B.: `PRINT X, Y, Z` zur Ausgabe der Inhalte der Variablen X, Y und Z).

Verwendet man zur Trennung der Listenelemente in der Print-Anweisung Semikolons statt Kommata, so werden die Listenelemente bei der Ausgabe um ein Leerzeichen näher aneinandergerückt. Z.B. erzeugt `Print "A", "B", "C"` die Ausgabe `A B C` , dagegen würde `Print "A"; "B"; "C"` die Ausgabe `ABC` erzeugen.

Außerdem verursacht jede PRINT Anweisung einen Zeilenvorschub im Protokollbereich des Fensters; aber dieser Zeilenvorschub kann durch Anhängen eines Kommas oder Semikolons am Ende der Print-Anweisung unterdrückt werden. Also die folgenden zwei Befehle

```
Print "Bitte geben Sie einen Wert ein: ",
Input A
```

haben durch das an die Print Anweisung angehängte Komma die Wirkung, dass die Eingabe in derselben Zeile erfolgt wie die Ausgabe.

Sehr oft verwendet man in der Print-Anweisung die Format-Funktion. Es macht zum Beispiel keinen Sinn (Deutsch mit englischem Satzbau ☺) sondern trägt nur zur Verwirrung bei, wenn ein Programm einen Eurobetrag mit 15 Kommastellen Genauigkeit ausgibt.
Statt `PRINT Moneten` schreibt man besser `PRINT FORMAT(Moneten,"###.00")`. Die drei Hashtag-Zeichen # im Formatstring vor dem Dezimalpunkt sind dabei (unterdrückbare) Platzhalter für Vorkommastellen; die beiden Nullen hinter dem Punkt sind dabei (nicht unterdrückbare) Platzhalter für die Centbeträge.

Zum Schluss noch ein Beispiel: Wir wollen einmal ein einfaches Basicprogramm schreiben, das für den schiefen Wurf bei vorgegebenem Abwurfwinkel α und vorgegebener Abwurfgeschwindigkeit v die Wurfweite w und die Wurfhöhe h ermittelt. Die Formeln dazu lauten:

$$w = \frac{v^2}{g} \sin(2\alpha) \qquad h = \frac{v^2}{2g} \sin^2\alpha$$

mit $g = 9.8066$ [m/s²]. Es sollen jeweils ein Zahlenwert für α und v eingelesen, w und h berechnet und beide Ergebniswerte mit ihren Maßeinheiten und 4 Nachkommastellen Genauigkeit ausgegeben werden. Das zugehörige Programm könnte dann z.B. so aussehen:

```
Print "Bitte geben Sie einen Wert für alfa ein:",
Input Alfa
Print "Bitte geben Sie einen Wert für v ein: ",
Input V
Pi=4*ATN(1)
G=9.8066
Alfa=Alfa*Pi/180      ' umrechnen ins Bogenmaß
W=(V^2/G)*SIN(2*Alfa)
H=(V^2/(2*G))*SIN(Alfa)^2
Print "Weite W=", Format(W,"###.0000"), " [m] "
Print "Höhe  H=", Format(H,"###.0000"), " [m] "
End
```

Beachten Sie, dass im Programmcode von GPR-Basic zwischen Groß- und Kleinschreibung grundsätzlich nicht unterschieden wird.

2 Entscheidungen und Programmverzweigungen

2.1 Logische Ausdrücke

Arithmetische Ausdrücke und String Ausdrücke haben wir ja schon kennengelernt. In GPR-Basic gibt es noch einen anderen Typ von Ausdruck, nämlich den logischen Ausdruck (auch Boolescher Ausdruck genannt nach dem Engländer, der ihn erfunden hat). Das Ergebnis eines logischen Ausdruckes kann nur zwei Werte annehmen, und zwar TRUE oder FALSE, eigentlich *erfüllt* oder *nicht erfüllt*. Wir sollten nicht von WAHR oder FALSCH sprechen, denn FALSCH hat hier ungerechtfertigterweise einen negativen Beigeschmack.
Zum Beispiel 1 < 2 ist so ein logischer Ausdruck, und sein Ergebniswert ist natürlich TRUE.
Folgende logische Vergleichs-Operatoren stehen zur Verfügung:

<		kleiner als
<=	(oder =<)	kleiner gleich
>		größer als
>=	(oder =>)	größer gleich
=		gleich
><	(oder <>)	ungleich

Logische Ausdrücke können durch die logischen Operatoren AND und OR miteinander verknüpft werden. Zum Beispiel, für *Ausdruck1* AND *Ausdruck2* müssen die beiden Ausdrücke erfüllt (TRUE) sein, damit das Gesamtergebnis TRUE wird; dagegen für
Ausdruck1 OR *Ausdruck2* reicht es, wenn einer der beiden Ausdrücke erfüllt ist, damit das Gesamtergebnis TRUE wird.
Der logische Operator NOT dreht das Ergebnis um: NOT *Ausdruck* ist TRUE wenn *Ausdruck* FALSE war, und FALSE wenn *Ausdruck* TRUE war.

Logische Ausdrücke können weiter und komplizierter miteinander verknüpft werden, wobei runde Klammern gesetzt werden können, und bei gleicher Priorität die Auswertungsreihenfolge von links nach rechts erfolgt. Das Folgende alles ist zwar möglich aber nicht ratsam:

```
A<B AND (B>C OR C>D) OR NOT (B<C)
Y = (X >= Z OR X >= Z)
0 <= X <= 2
```

A,B,C,D und X,Y,Z seien dabei Rechenvariablen. Bei komplizierten Ausdrücken verliert man sehr schnell den Überblick, Fehler sind leicht möglich und schwer zu entdecken.
Diese logischen Ausdrücke sind der menschlichen Denkweise weit weniger angepasst als man zunächst meinen möchte (z.B. die Frage „Hast du die Wahrheit gesagt, ja oder nein?“ würde ein Computer immer mit TRUE beantworten ☺).

Es ist möglich (aber i.A. nicht sinnvoll), logische und arithmetische Ausdrücke miteinander zu verknüpfen. Der logische Wert FALSE entspricht dem (Integer) Zahlenwert 0; und der logische Wert TRUE entspricht dem Zahlenwert –1. Umgekehrt wird in jeder logischen Auswertung ein Wert, der nicht 0 ist, als TRUE aufgefasst, lediglich der Wert 0 ist FALSE.

Man kann also durchaus das Ergebnis eines logischen Vergleichs in einer Rechenvariablen abspeichern und dazu z.B. die folgende Anweisung schreiben:

```
Vergleich = A < B
```

Die Variable *Vergleich* wird dann zu –1 (falls A kleiner als B ist) oder zu 0 (falls A größer oder gleich B ist). Später kann diese Variable für Entscheidungen verwendet werden, zum Beispiel in einer IF Anweisung (IF *Vergleich* THEN).

Aus der Tatsache, dass die Ergebnisse von logischen Ausdrücken wie Integer-Werte behandelt werden, ergeben sich folgende, typische Anfängerfehler. Die beiden Anweisungen:

```
Input X
Print Y=X^2
```

sollen offensichtlich das Quadrat von einem eingegebenen Zahlenwert berechnen und ausgeben; stattdessen wird aber der Wert 0 ausgegeben. Die Print-Anweisung erwartet einen Ausdruck, und Y=X^2 ist ein logischer Ausdruck (Vergleich) mit dem Ergebniswert 0.
Richtigweise hätte man natürlich `Print "Y=", X^2` schreiben müssen.

Oft müssen für Variablen Bereiche abgefragt werden, z.B. es könnte von Bedeutung sein, ob die Variable X im Bereich zwischen 0 und 2 liegt. Man ist dann versucht, elegant zu schreiben:

```
IF 0 < X < 2 THEN .....
```

Das ist richtige Basic-Syntax, bedeutet aber ganz etwas anderes, als offensichtlich gemeint ist; und es funktioniert nicht. Zuerst wird der Vergleich 0 < X ausgewertet und dem Ergebnis –1 oder 0 zugeordnet (je nachdem ob 0 nun kleiner ist als X oder nicht); danach wird das Ergebnis (also –1 oder 0) mit dem Wert 2 verglichen. Da das Ergebnis von 0 < X immer kleiner als 2 ist, ist das Gesamtergebnis 0 < X < 2 in jedem Fall erfüllt (TRUE).
Richtigerweise hätte man schreiben müssen: `IF 0 < X AND X < 2 THEN .....`

Schließlich sei noch erwähnt, das natürlich auch die Basic Anweisung `A=B=C=1` nicht etwa bedeutet, dass die drei Variablen A und B und C gleich 1 gesetzt werden. Tatsächlich wird zuerst geprüft ob B gleich C ist, danach wird das Ergebnis (also 0 oder –1) mit 1 verglichen und schließlich der Variablen A der Wert 0 zugewiesen.

2.2 Die einzeilige IF-Anweisung

Die einfachste Form einer bedingten Programmbearbeitung ist die einzeilige IF-Anweisung. Zum Beispiel kann man schreiben:

```
If  X >= 0 Then PRINT SQR(X)
```

Die Anweisung `PRINT SQR(X)` wird nur dann ausgeführt, wenn X größer oder gleich 0 ist. Möchte man zwei Anweisungen bedingt ausführen, so kann man sie durch einen Doppelpunkt abgetrennt in dieselbe Zeile schreiben, zum Beispiel:

```
If  X >= 0 Then PRINT SQR(X) : PRINT LN(X)
```

Es ist auch möglich, in der einzeiligen IF-Anweisung das Schlüsselwort Else zu verwenden (so heißt die Schwiegermutter des Autors dieser Zeilen, mit der hat das Ganze aber überhaupt nichts zu tun ☺). Else bedeutet *anderenfalls.* Zum Beispiel kann man schreiben:

```
If  X >= 0 Then PRINT SQR(X) Else BEEP
```

Für positive X-Werte wird die Quadratwurzel ausgegeben, für negative X-Werte gibt das Programm einen Signalton aus (vorausgesetzt, der Lautsprecher wurde nicht deaktiviert).

Eine typische Form der Programmverzweigung, die von Anfängern gerne verwendet wird, ist das bedingte Springen auf eine Sprungmarke. Zum Beispiel könnte irgendwo im Programmcode eine Sprungmarke (z.B. mit dem Namen *Irgendwo*) stehen. Man könnte dann mit einer IF- und einer GOTO-Anweisung bedingt auf diese Marke springen (springen heißt hier die Programmausführung an dieser Stelle fortsetzen). Also so etwa (die Punkte stehen dabei für Anweisungen):

```
Irgendwo:
    .
    .
    .
    .
If (logischer Ausdruck) Then GOTO Irgendwo
```

Von solchem Programmierstil wird überall dringend abgeraten, da das besonders bei komplizierten Verschachtelungen zu schwer zu entschlüsselbarem Programmcode führt. Solcher Code wird allgemeinhin als Spaghetticode bezeichnet (Homo Digitalis Italianus ☺), was offenbar darauf Bezug nimmt, dass man eine Nudel auf dem Teller zwar sofort sehen kann, aber nicht wo ihr Anfang und ihr Ende ist. Für Programmverzweigungen gibt es in Basic bessere Sprachkonstruktionen, nämlich die For-Next und die Do-Loop Schleife (mehr dazu später).
Bedingte Sprünge mit GOTO sind nur dann sinnvoll, wenn man dafür eine klare Struktur vorsieht, z.B. für Programmabbrüche beim Auftreten von Fehlern etc.

2.3 Die mehrzeilige IF-Anweisung

Gerade bei komplizierteren Programmverzweigungen oder bei Verschachtelungen wird es notwendig, auf die mehrzeilige Form der IF-Anweisung zurückzugreifen. Die Anweisung:

```
If  X >= 0 Then
```

ist ganz sicher die mehrzeilige Form, denn hinter dem Schlüsselwort Then steht keine Anweisung. Während sich die einzeilige IF-Anweisung nur auf das bezieht, was in derselben Zeile hinter Then steht, so bezieht sich die mehrzeilige Form der IF-Anweisung auf alle Anweisungen, die unter ihr stehen, bis die zugehörige ENDIF-Anweisung angetroffen wird (statt ENDIF kann es auch END IF mit einem Leerzeichen heißen). Zu jeder mehrzeiligen IF-Anweisung muss es im Programm genau eine zugehörige ENDIF-Anweisung geben, andernfalls ist das Programm syntaktisch falsch und wird nicht ausgeführt.

Die Syntax einer mehrzeiligen IF-Anweisung lautet also:

```
If (logischer Ausdruck) Then
   .
   .
   .
Endif
```

Wieder stehen die Punkte für beliebig viele Anweisungen; und diese Anweisungen können natürlich auch (einzeilige oder mehrzeilige) IF-Anweisungen sein. Zum Beispiel ist die folgende verschachtelte Konstruktion möglich:

```
If (logischer Ausdruck) Then
   .
   .
       If (anderer logischer Ausdruck) Then
          .
          .
       Endif
   .
Endif
```

Ausschließlich die richtige Reihenfolge bestimmt, welches ENDIF zu welchem IF gehört (genauso wie es bei Klammern in einem arithmetischen Ausdruck der Fall ist). Die Verschachtelungstiefe ist eigentlich unbegrenzt, aber oft beginnt man ab etwa Dreifachverschachtelungen schon den Überblick zu verlieren. Auch die mehrzeilige IF-Anweisung kann Else enthalten:

```
If (logischer Ausdruck) Then
   .
   .
Else
   .
   .
Endif
```

Ist der Wert des logischen Ausdrucks erfüllt (*True*), so werden die Anweisungen zwischen If und Else ausgeführt; ist er dagegen *False*, so werden die Anweisungen zwischen Else und Endif ausgeführt. Es gibt auch die Möglichkeit der Mehrfach-Abfrage mit ELSEIF:

```
If (erster logischer Ausdruck) Then
   .
   .
Elseif (zweiter logischer Ausdruck) Then
   .
   .
Endif
```

Elseif wird aber überhaupt nur dann geprüft, wenn der erste logische Ausdruck *False* war.

Die mehrzeilige IF Konstruktion kann beliebig viele ELSEIF Abfragen aber höchstens ein ELSE enthalten, z.B.:

```
If (erster logischer Ausdruck) Then
  .
Elseif (zweiter logischer Ausdruck) Then
  .
Elseif (dritter logischer Ausdruck) Then
  .
Else
  .
Endif
```

Zu berücksichtigen ist, dass nur die zum ersten erfüllten (*True*) logischen Ausdruck gehörenden Anweisungen ausgeführt werden. Ist keiner der logischen Ausdrücke *True*, so wird das gemacht, was hinter Else steht. Wenn also z.B. der erste logische Ausdruck X > 0 lautet, der zweite Ausdruck X > 1 lautet und der dritte Ausdruck X > 2; und X tatsächlich 3 ist, so wird nur das gemacht, was zwischen den Zeilen von If und dem ersten Elseif steht.
Wäre X dagegen –3, so würde das gemacht werden, was zwischen Else und Endif steht.

Zum Schluss ein kleines Beispiel: Wir wollen ein Programm schreiben, das für eingegebene kartesische Koordinaten x und y die zugehörigen Polarkoordinaten r und φ berechnet.

Der Radius r ist immer $\sqrt{x^2 + y^2}$, aber die Formel $\varphi = \text{arctg}(y/x)$ gilt in dieser Form nur für positive x-Werte. Für negative x-Werte dagegen gilt: $\varphi = \pi + \text{arctg}(y/x)$.
Für $x = 0$ ist die Formel überhaupt nicht zu gebrauchen, aber dann muss unterschieden werden, ob der Punkt x, y nun auf der positiven oder auf der negativen y-Achse liegt.
Der folgende Programmcode löst die Aufgabe:

```
Print "Bitte X und Y eingeben: "
Input X,Y

R=SQR(X^2+Y^2)
Pi=4*ATN(1)

If X>0 Then
      Phi=ATN(Y/X)
Elseif X<0 Then
      Phi=Pi+ATN(Y/X)
Else
      If Y>=0 Then Phi=+Pi/2
      If Y< 0 Then Phi=-Pi/2
Endif

Print "Polarkoordinaten: R=",R, "; Phi=",Phi*180/Pi
End
```

Natürlich berechnet das Programm alle Winkelfunktionen im Bogenmaß; für die Ergebnisausgabe mit der letzten Anweisung ist es aber sinnvoll, den Winkel φ mit dem Faktor $180/\pi$ auf das Gradmaß umzurechnen.

3 Schleifen und Iteration

3.1 Die For-Next Schleife

Eine typische Basic-Konstruktion zum wiederholten Abarbeiten von Programmcode ist die For-Next Schleife. Die Syntax dazu ist:

```
For Variable = Anfangswert To Endwert Step Schrittweite
      .
      .
      .
Next Variable
```

Variable ist hierbei die Schleifenvariable, sie wird beim Eintritt in die Schleife mit dem *Anfangswert* besetzt und dann bei jedem Durchlaufen der Next Anweisung um den bei Step angegebenen Wert *Schrittweite* erhöht. Danach wird geprüft, ob sich die Schleifenvariable noch im durch *Endwert* gesetzten Bereich befindet. Die Angabe Step ist optional, man kann sie weglassen, wenn die Schrittweite 1 sein soll (Sie sollten sie dann aber auch weglassen, denn mit „Step 1“ outen Sie sich als Programmieranfänger).

Wir wollen nun einmal mit der For-Next Schleife ein Programm schreiben, das die Summe der Natürlichen Zahlen von Eins bis Hundert berechnet (ergibt nach Gauß 5050). Also:

```
Summe=0
For i = 1 To 100
     Summe=Summe+i
Next i
Print "Ergebnis:",Summe
```

Die Summe der Zahlen von 1 bis 100 wird dadurch berechnet, dass man auf die Summe der Zahlen von 1 bis 99 den Wert 100 draufaddiert. Die Variable `Summe` steht also in der dritten Anweisung links und rechts vom Gleichheitszeichen (und das Gleichheitszeichen bedeutet nur, dass die Zeile eine Wertzuweisung ist). Bitte beachten Sie, dass die Schleifenvariable `i` nach Abarbeitung der Schleife weiterhin existiert und in diesem Beispiel den Wert 101 enthält.

Genau wie If-Endif Konstruktionen können For-Next Konstruktionen ineinander verschachtelt werden, z.B.:

```
For I = 1 To 10
      For J = 1 To 100
            For K = 1 To 50
                .
                .
            Next K
       Next J
Next I
```

Die richtige Reihenfolge der Next Anweisungen ist dabei wesentlich (wie bei Klammern in mathematischen Formeln), sie bestimmt die Zuordnung einer Next zu ihrer For Anweisung.

Als Beispiel für ineinander geschachtelte For-Next Schleifen wollen wir einmal die Primzahlen von 1 bis 100 berechnen. Primzahlen sind natürliche Zahlen, die nur durch eins oder sich selbst teilbar sind, also 2,3,5,7,11,13,17 etc. (man benötigt sie zur Datenverschlüsselung, sonst sind sie eigentlich kaum zu etwas zu gebrauchen. Aber sollten Sie mal eine Tochter haben, die aufs Gymnasium geht, werden Sie sich wohl mit Primzahlen beschäftigen müssen ☺). Das Primzahlberechnungsprogramm könnte folgendermaßen aussehen:

```
Dim i as Long, j as Long, Schalter as Long
For i = 2 To 100
      Schalter=0
      For j = 2 To i-1
            If i/j = Int(i/j) Then
                  Schalter=1
                  Exit For
            Endif
      Next j
      If Schalter=0 Then Print "Primzahl: ",i
Next i
End
```

Die äußere For-Next Schleife zählt die Schleifenvariable `i` (Typ Long) von 2 bis 100 hoch; die innere For-Next Schleife zählt die Schleifenvariable `j` (Typ ebenfalls Long) von 2 bis fast zum aktuellen Wert von `i` hoch. Mit der inneren Schleife wird geprüft, ob es einen Wert `j` (kleiner als `i`) gibt, mit dem `i` ganzzahlig geteilt werden kann (also ob `i/j` gleich `Int(i/j)` ist). Ist das für irgendeinen Wert von `j` der Fall, so wird die Long-Variable `Schalter` mit dem Wert 1 besetzt und die innere For-Next Schleife beendet (Anweisung `Exit For`) . Wird kein Wert `j` gefunden, für den eine ganzzahlige Teilung möglich ist, so ist `i` offensichtlich eine Primzahl und die Variable `Schalter` bleibt 0. Nur dann wird der Wert `i` ausgegeben.

Im obigen Programm wird die Anweisung EXIT FOR verwendet. Diese Anweisung ist dazu da, die aktuelle For-Next Schleife abzubrechen und die Programmausführung mit der Anweisung hinter Next fortzusetzen. Solche bedingten Schleifenabbrüche werden in Basic-Programmen oft verwendet (z.B. der Art: `If` (logischer Ausdruck) `Then Exit For`).
Es gibt in GPR-Basic auch noch die Anweisung CONTINUE FOR ; mit ihr kann man von irgendeinem Punkt in der Schleife direkt auf die zugehörige NEXT Anweisung springen.

Die Schleifenvariablen (im Beispiel `i` und `j`) sind natürlich dazu da, dass man innerhalb der Schleife mit ihnen rechnen kann. Sie sollten dabei aber nur abgefragt und um Himmels Willen nicht verändert werden. Zwar gibt es in Basic nichts, was Wertzuweisungen an Schleifenvariablen innerhalb einer Schleife verbietet, aber von sowas muss unbedingt gewarnt werden: Der Ablauf des Programms wäre dann kaum noch vorherzusehen.

Es gibt bezüglich For-Next Schleifen noch eine weitere Falle vor der gewarnt werden muss. Will man z.B. einmal alle möglichen Centbeträge, die kleiner oder gleich ein Euro sind, auflisten (d.h. die Zahlenwerte 0.00, 0.01, 0.02 ... bis 1.00), so könnte man schreiben:

```
for i=0 to 1 step 0.01
    print i, "Euro"
next i
```

Das Programm funktioniert zwar, gibt aber fälschlicherweise den letzten Wert (1 Euro) nicht aus (zu geizig ☺). Der Grund dafür ist, dass der Computer nicht mit dem Zehnersystem sondern mit dem Binärsystem arbeitet. Zwar wird der Wert 1/100 mit einer Genauigkeit von etwa 15 Stellen hinter dem Komma behandelt, trotzdem entspricht er aber im Binärsystem nicht exakt dem Dezimalwert 0.01. Wenn dann dieser Wert 100-mal addiert wird, so gibt das eben nicht genau den Wert 1, sondern nur ungefähr, und die letzte Vergleichsabfrage geht schief. (Ähnlich wie im Zehnersystem mit begrenzter Stellenzahl dreimal ein Drittel nicht exakt eins sondern $3 \cdot 0.3333333333333 = 0.9999999999999$ ergeben würde).

Probieren Sie auch einmal die folgenden Vergleichsabfragen in der For-Next Schleife:

```
for i=0 to 1 step 0.01
    if i=0.09 then beep
    if i=0.10 then beep
    print i, "Euro"
next i
```

Während die Abfrage bei `i=0.09` noch korrekt abgearbeitet wird, so ist das bei `i=0.10` bereits nicht mehr der Fall. Darum merken wir uns:

- Keine „ist gleich“ Abfragen bei Rechenvariablen mit Nachkommastellen, also niemals: `If i = 0.1 Then` In diesem Beispiel könnte die Variable `i` z.B. den Wert 0.0999999999999999 haben. Ob das nun gleich ist oder nicht, ist Geschmacksache.

- Mit genau demselben Argument sollten Rechenvariablen mit Nachkommastellen nicht als Zählvariablen in For-Next Schleifen verwendet werden (außer vielleicht wenn es auf die Anzahl der Schritte nicht ankommt). `For i = 0.01 To 1.00 Step 0.01` geht zwar, ist aber schlecht, denn es führt zu einem nicht eindeutigen Programmcode. Besser also wenn wir nur Zählvariablen (oder ganzzahlige Rechenvariablen) verwenden.

Richtig geschrieben lautet das obige Programm:

```
dim i as long
for i=0 to 100
    if i= 9 then beep
    if i=10 then beep
    print i/100, "Euro"
next i
```

Die Abarbeitung dieses Programmes ist nun keine Glücksache mehr. Als Beispiel wollen wir einmal die Fakultät für vom Benutzer eingegebene Zahlen berechnen:

```
Input N
F=1
For I=1 To N
    F=F*I
Next I
Print "Die Fakultät ist:", F
End
```

Wenn wir im obigen Programm die Print Anweisung in die Schleife setzen (d.h. vor die Next Anweisung schreiben), so gibt das Programm eine Liste der Fakultäten von 1 bis N aus.

3.2 Die Do-Loop Schleife

Oft kommt es vor, dass man keine Schleifenvariable braucht oder dass man nicht weiß, wie oft eine Schleife durchlaufen werden soll, und dann ist die Do-Loop Schleife geeigneter als die For-Next Schleife. Die Syntax dazu ist:

```
Do [Until/While logischer Ausdruck ]
     ...
    [If logischer Ausdruck Then Continue Do]
     ...
    [If logischer Ausdruck Then Exit Do]
     ...
Loop [Until/While logischer Ausdruck ]
```

Alles, was in der obigen Syntaxbeschreibung in eckige Klammern gesetzt ist, kann auch weggelassen werden; und der Schrägstrich / soll heißen: entweder `Until` oder `While`.

Die Punkte sollen wie immer Anweisungen symbolisieren. Es kann also alternativ am Anfang der Schleife, irgendwo in ihrer Mitte oder an ihrem Ende überprüft werden, ob die Bedingung für das Abarbeiten der Anweisungen in der Schleife noch erfüllt ist. Wenn nicht, wird die Programmausführung mit der Anweisung fortgesetzt, welche auf die Loop-Anweisung folgt. Endlosschleifen sind leicht möglich, z.B. wenn man nur die Schlüsselworte Do und Loop verwendet (solche Endlos-Schleifen können in GPR-Basic durch Anklicken der Schaltfläche STOP im Toolbar abgebrochen werden). Im Allgemeinen wird man aber die Schleife solange ausführen wollen, bis (Until) eine Bedingung *True* wird oder solange (While) eine Bedingung *True* bleibt. Probieren wir einmal ein einfaches Beispiel. Das folgende Programm berechnet wiederholt für Eingabewerte N die Zahlenwertsumme; bricht aber ab, wenn zum ersten Mal ein negativer Wert für N eingegeben wird:

```
Do
  Input N
  If N<0 Then Exit Do
  If N=0 Then Continue Do
  Summe=0
  For i=1 To N
    Summe=Summe+i
  Next i
  Print "Die Zahlensumme von ", N, " ist ", Summe
Loop
```

Eine Do-Loop mit „`If` *logischer Ausdruck* `Then Exit Do`“ zu verlassen oder mit „`If` *logischer Ausdruck* `Then Continue Do`“ mit dem nachfolgenden Schritt fortzusetzen ist übersichtlicher als eine „`If` *logischer Ausdruck* `Then Goto` *Sprungmarke*“ Konstruktion zu verwenden (was auch gehen würde).

Do-Loop-Schleifen können ineinander geschachtelt werden, ebenso wie For-Next Schleifen. Natürlich können Do-Loop Schleifen auch innerhalb von For-Next Schleifen oder If-Endif Konstruktionen verwendet werden, und umgekehrt. Wichtig hierbei ist aber immer, dass die Verschachtelungen von innen nach außen abgearbeitet werden. Sich überlappende Schleifen (z.B. erst For dann Do dann Next dann Loop) sind unsinnig und deshalb nicht erlaubt.

Der folgende Programmcode kann als Beispiel für eine Endlos-Schleife dienen. Er soll die aktuelle Systemzeit (GPR-Basic-Funktion TIME) als Digitaluhr im Grafikbereich anzeigen:

```
Do
      Cls
      Text TIME
Loop
```

Cls („clear screen“) und Text (Textausgabe im Grafikfenster) sind beides Grafik-Befehle, um die Wirkung des Programmes zu sehen, müssen Sie das Grafikfenster öffnen (mit der Maus in die rechte Fenstermitte gehen bis der Doppelpfeil ↔ erscheint, dann mit der linken Maustaste die Leiste in die Fenstermitte ziehen). Jetzt kann sich zeigen, ob bei Ihrem Rechner am Prozessor gespart wurde: Als Folge davon, dass der Grafikbereich viele Male in der Sekunde gelöscht wird, flackert er möglicherweise. Brechen Sie die Programmausführung ab (durch Anklicken der „Stop“ Taste) und ändern Sie das Programm folgendermaßen ab:

```
Do
      T$=TIME
      Do
      Loop Until T$ >< TIME
      Cls
      Text TIME
Loop
```

Und da haben wir auch gleich ein Beispiel für zwei ineinander geschachtelte Do-Loop Schleifen: Die innere (Warte-) Schleife ruft die Systemzeit `TIME` solange auf, bis sie sich vom vorherigen Wert unterscheidet (welcher zuvor in der Stringvariablen `T$` abgespeichert worden ist). Das Flackern der Ausgabe ist beseitigt, denn erst, wenn sich die Werte unterscheiden, wird der Grafikbereich gelöscht und die Zeit ausgegeben.

Schließlich noch ein Beispiel aus der Mathematik: Die Quadratwurzel wird in Computern durch eine Iterationsvorschrift berechnet (das lateinische Wort "iterum" heißt wiederum, und also erkennen wir auch sogleich den Ursprung des deutschen Wortes "wiederum" ☺).

Die Iterationsvorschrift für Quadratwurzel $r = \sqrt{x}$ lautet: $r_{i+1} = 0.5 \cdot (r_i + x / r_i)$;
darin ist r_i der aktuelle Wert für die Wurzel und r_{i+1} der verbesserte Wert.
Das folgende Programm berechnet mit wenigen Schritten den genauen Wurzelwert:

```
Input x
r=x                ' x ist erster Schätzwert für r
Do
   r=0.5*(r+x/r)
Loop until ABS(r*r-x)<0.0000000000001
Print "r= ",r, " zum Vergleich: ",SQR(x)
```

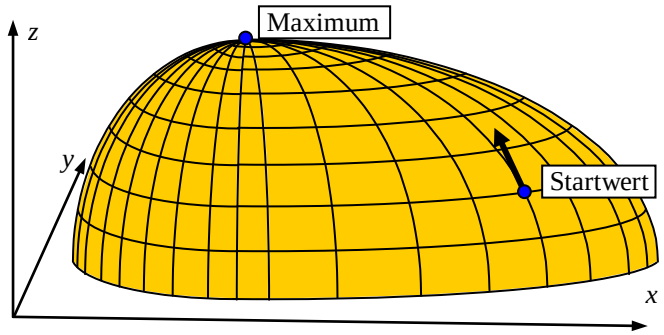
Bitte beachten Sie, dass innerhalb der Do-Loop Schleife nur eine einzige Anweisung notwendig ist. Der rechts vom Gleichheitszeichen stehende Ausdruck $0.5 \cdot (r_i + x / r_i)$ wird ausgerechnet und in die links stehende Variable reingeschrieben. Praktischerweise ist das ja genau der Wert r_{i+1} , der beim nächsten Durchlaufen der Schleife im rechten Ausdruck wieder gebraucht wird.

3.3 Numerische Iterationen

For-Next oder Do-Loop Schleifen sind hervorragend dazu geeignet, mathematische Probleme numerisch (d.h. „zahlenwertmäßig“) zu lösen. Betrachten wir einmal als Beispiel, dass das Maximum einer nichtlinearen Funktion berechnet werden soll:

$$z(x, y) = -y^2 \cdot x^2 - 2x^2 - y^2 + 10x + 12y - 20$$

Analytisch geht das nun nicht mehr, also muss der Computer helfen. Die gegebene Beispielfunktion ist für numerische Optimierungsverfahren besonders unangenehm, denn wenn wir uns $z(x, y)$ als eine Art Gebirge über der x, y Ebene vorstellen, so hat dieses Gebirge in etwa die Form einer Banane (lecker ☺). Das Maximum kann man dann so finden, indem man ausgehend von einem Schätzwert x, y (Benutzereingabe!) die Richtung des steilsten Aufstiegs berechnet und dann ein Stückchen in diese Richtung geht. Von dem neuen Punkt aus berechnet man eine neue Richtung, geht weiter usw. Geht es in alle Richtungen nur noch bergab, so hat man ein lokales Maximum gefunden. Falls es noch weitere, eventuell höhere Gipfel gibt, kann man diese durch Starten desselben Verfahrens mit anderen Startwerten berechnen (glücklicherweise gibt es für unsere Beispielfunktion nur ein Maximum).



Gradient ($\partial z/\partial x$, $\partial z/\partial y$) heißt der Vektor in der x-y Ebene, der in Richtung des stärksten Anstiegs der Funktion $z(x, y)$ zeigt. Der Gradient hat den Betrag der Steigung; man kann ihn einfach numerisch (und in diesem Fall sogar analytisch) berechnen:

$$\partial z/\partial x = -2y^2 \cdot x - 4x + 10$$
$$\partial z/\partial y = -2y \cdot x^2 - 2y + 12$$

Ist der Gradient in alle Richtungen Null, so könnte man statt eines lokalen Maximums auch einen sogenannten Sattelpunkt erreicht haben. Eigentlich müsste man also auch die zweiten Ableitungen (d.h. die Krümmungen) berechnen, in der Praxis aber begnügt man sich damit, anzunehmen dass das Verfahren nicht in einen Sattelpunkt hineinläuft (warum sollte es auch?). Besondere Beachtung verdient dabei die Schrittweite h. Wird sie zu groß gewählt, so fängt das Verfahren an zu schwingen, es überschreitet das Maximum und erreicht keine Genauigkeit. Wird die Schrittweite dagegen zu klein gewählt, so rechnet das Verfahren ewig lang. Probieren wir einmal, h folgendermaßen anzunehmen:

$$h = 0.01 \cdot \sqrt{(\partial z/\partial x)^2 + (\partial z/\partial y)^2}$$

So berechnet hängt die Schrittweite h von der Länge des Gradientens ab, geteilt durch 100.

Der folgende GPR-Basic Programmcode könnte dann das Problem lösen:

```
input x,y     ' Schätzwerte x und y
for i=1 to 100
   z=-y^2*x^2-2*x^2-y^2+10*x+12*y-20
   dzdx=-2*x*y^2-4*x+10
   dzdy=-2*y*x^2-2*y+12
   h=0.01*sqr(dzdx^2+dzdy^2)
   x=x+h*dzdx
   y=y+h*dzdy
   print "z(";format(x,"0.000");",";_
              format(y,"0.000");") =",z
next i
end
```

Das Programm funktioniert zwar und findet z.B. für die Startwerte $x = 1$, $y = 1$ das Ergebnis $z(0.150,5.610) = 16.593024376767$, aber in der Nähe der Lösung schwingt das Verfahren stark und dieses Ergebnis ist keinesfalls genau. Eigentlich bräuchten wir eine bessere Schrittweitensteuerung und eine Kriterium für die Beendigung des Programms bei erreichter Endgenauigkeit (z.B. könnte eine verbesserte Schrittweitensteuerung die Krümmung der Funktion in x und y-Richtung berücksichtigen). Aber lassen wir es für jetzt einmal so stehen, denn mit genügend Rechenleistung erzielen wir auf jeden Fall die Lösung.

Wenn die zu maximierende Funktion $z(x,y)$ explizit angegeben worden ist wie in unserem Beispiel, so können wir natürlich auch die Extremwerte einfach dadurch berechnen, dass wir die partiellen Ableitungen analytisch bilden und Null setzen:

$$\partial z/\partial x = -2\,y^2 \cdot x - 4\,x + 10 = 0$$
$$\partial z/\partial y = -2\,y \cdot x^2 - 2\,y + 12 = 0$$

Die Maximumsuche ist so in ein anderes mathematisches Problem transformiert worden, nämlich in das Problem, die Lösung von einem nichtlinearen Gleichungssystem zu finden:

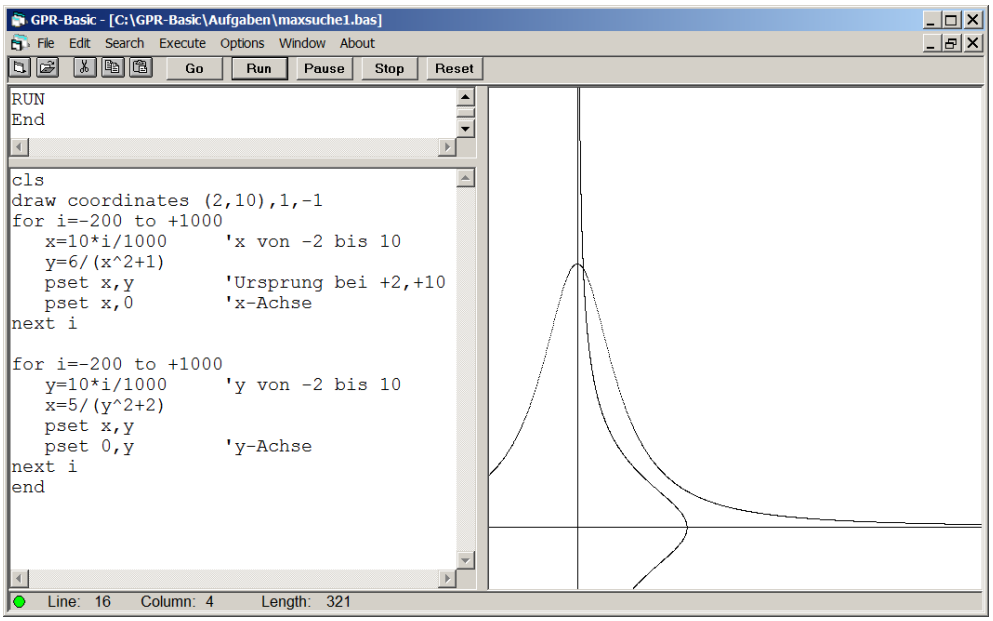
$$(y^2 + 2) \cdot x - 5 = 0$$
$$(x^2 + 1) \cdot y - 6 = 0$$

Wegen der Nichtlinearität des Gleichungssystems ist eine analytische Lösung leider nicht möglich; zufällig aber lässt sich die erste Gleichung nach x auflösen und die zweite nach y:

$$x = 5 / (y^2 + 2) \qquad y = 6 / (x^2 + 1)$$

Die zugehörigen Kurven sind einander ziemlich ähnlich, wenn man einmal davon absieht, dass bei der jeweils anderen Kurve x und y vertauscht sind: Sie sind achsensymmetrisch, die beiden rechten Seiten haben positive Wertebereiche und streben für $y \to \infty$ (bzw. $x \to \infty$) gegen Null. Schnittpunkte der beiden Kurven gibt es also höchstens im ersten Quadranten.

Wir können die beiden Kurven leicht von einem GPR-Basic Programm zeichnen lassen. Dazu benötigen wir den Grafikbefehl `PSET x,y` der an der Position x,y einen Punkt setzt. Der GPR-Basic Grafikbefehl Draw Coordinates stellt das Koordinatensystem ein (gerechnet von der oberen linken Ecke im Grafikbereich eines GPR-Basic Fensters), die voreingestellte Einheit ist Zentimeter. Zwei For-Next Schleifen bewältigen die Ausgabe :



Es gibt also nur einen Schnittpunkt (d.h. nur ein Maximum), etwa bei $x = 0.2$, $y = 5.8$. Um diesen Schnittpunkt nun genau zu berechnen, können wir die beiden Gleichungen als Iterationsansatz interpretieren und schreiben:

$$x_{neu} = 5 / (y^2 + 2)$$
$$y_{neu} = 6 / (x_{neu}^2 + 1)$$

Mit der ersten Vorschrift wird zu einem Schätzwert y ein neuer x-Wert berechnet, mit der zweiten Vorschrift zum neu berechneten x-Wert ein neuer y-Wert berechnet. Falls das Verfahren konvergiert (d.h. zur Lösung hinläuft), so wird es solange fortgesetzt, bis eine ausreichende Genauigkeit erreicht worden ist. Dies ist z.B. der Fall, wenn sich die verbesserten Werte x, y von den vorherigen nicht mehr unterscheiden. Wir können also schreiben:

```
Input y
do
    x=5/(y^2+2)
    y=6/(x^2+1)
    print x,y
loop until abs(x-5/(y^2+2))<0.00000001
```

Tatsächlich konvergiert das Verfahren und findet nach einigen Schritten die genaue Lösung $x = .136240206158811$, $y = 5.8906611193729$.

Dieses sehr einfache Verfahren ist aber weder sonderlich schnell noch genau, und statt zu konvergieren hätte es auch genauso gut divergieren können. Aus diesen Gründen wird das Verfahren in der Numerik nicht oft eingesetzt.

In der Praxis gibt es weitaus bessere Verfahren zur Lösung von nichtlinearen Gleichungssystemen; und das am meisten verbreitetste ist wohl das sogenannte Newton-Verfahren.

Es lässt sich mit Worten etwa so beschreiben: Um die Lösung eines nichtlinearen Gleichungssystems zu finden, schätzt man zunächst eine Näherungslösung. Danach linearisiert man das Gleichungssystem um diese geschätzte Lösung, d.h. man ersetzt es in der Nähe der Lösung durch ein linearisiertes System. Das linearisierte System hat nun den einzigen Vorteil, dass man seine Lösung berechnen kann. Man tut das und hofft, dass die Lösung des linearisierten Systems eine verbesserte Schätzung der Lösung des nichtlinearen Systems darstellt. Das Verfahren wird dann solange fortgesetzt, bis die Lösung des nichtlinearen Systems mit ausreichender Genauigkeit gefunden worden ist.

Wir wollen das Newton-Verfahren einmal am Beispiel des uns bekannten nichtlinearen Gleichungssystems ausprobieren. Dabei werden wir besonderen Wert auf eine formale Betrachtung des Problems legen, da sich dieses Verfahren ja auch zur Lösung komplizierterer Gleichungssysteme höherer Ordnung eignet (und nicht nur zur Lösung eines einfachen Gleichungssystems mit lediglich zwei Unbekannten).

Es seien zwei nichtlineare Funktionen $f(x,y) = 0$, $g(x,y) = 0$ gegeben. Die Funktionswerte $f(x,y)$ und $g(x,y)$ lassen sich für jedes Wertepaar x und y berechnen, aber für eine geschätzte Lösung x_0, y_0 werden diese Funktionen wohl nicht genau, sondern nur ungefähr zu Null werden (sonst hätten wir die Lösung ja schon gefunden). Wir ersetzen diese beiden Funktionen in der Umgebung der geschätzten Lösung durch ihre linearisierten Repräsentanten:

$$\begin{aligned} f(x, y) &\approx f(x_0, y_0) + \partial f/\partial x \cdot (x - x_0) + \partial f/\partial y \cdot (y - y_0) \\ g(x, y) &\approx g(x_0, y_0) + \partial g/\partial x \cdot (x - x_0) + \partial g/\partial y \cdot (y - y_0) \end{aligned}$$

oder mittels der "Jacobi"-Matrix in Vektor-Schreibweise :

$$\begin{pmatrix} f(x, y) \\ g(x, y) \end{pmatrix} \approx \begin{pmatrix} f(x_0, y_0) \\ g(x_0, y_0) \end{pmatrix} + \begin{pmatrix} \partial f/\partial x & \partial f/\partial y \\ \partial g/\partial x & \partial g/\partial y \end{pmatrix} \bullet \begin{pmatrix} x - x_0 \\ y - y_0 \end{pmatrix}$$

Darin sind $\partial f/\partial x$, $\partial f/\partial y$, $\partial g/\partial x$, $\partial g/\partial y$ die partiellen Ableitungen der Funktionen $f(x,y)$ und $g(x,y)$ nach x und y an der Stelle x_0, y_0. Geometrisch interpretiert heißt das, dass wir die nichtlinearen Funktionen $f(x,y)$ und $g(x,y)$ als „Berge“ über der x-y-Ebene betrachten, welche wir im geschätzten Punkt x_0, y_0 durch Ebenen annähern. Der gemeinsame Schnittpunkt dieser beiden Ebenen mit der x-y Ebene ist dann unser verbesserter Schätzpunkt. Diesen Lösungspunkt berechnen wir einfach durch Nullsetzen der Vektorgleichung:

$$\begin{pmatrix} f(x_0, y_0) \\ g(x_0, y_0) \end{pmatrix} + \begin{pmatrix} \partial f/\partial x & \partial f/\partial y \\ \partial g/\partial x & \partial g/\partial y \end{pmatrix} \bullet \begin{pmatrix} x - x_0 \\ y - y_0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Auflösen der Vektorgleichung nach den Variablen x, y erfordert ein Invertieren der Matrix; dabei wird die invertierte Matrix von links mit allen Vektoren der Gleichung multipliziert.

Die invertierte Matrix wird (eigentlich nicht ganz korrekt) mit dem Exponenten –1 gekennzeichnet. Definitionsgemäß ergibt die invertierte Matrix multipliziert mit der Matrix selbst die Einheitsmatrix :

$$\begin{pmatrix} \partial f/\partial x & \partial f/\partial y \\ \partial g/\partial x & \partial g/\partial y \end{pmatrix}^{-1} \cdot \begin{pmatrix} \partial f/\partial x & \partial f/\partial y \\ \partial g/\partial x & \partial g/\partial y \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Und die Einheitsmatrix multipliziert mit einem Vektor ergibt denselben Vektor. Die so invertierte Vektorgleichung lautet dann:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} - \begin{pmatrix} \partial f/\partial x & \partial f/\partial y \\ \partial g/\partial x & \partial g/\partial y \end{pmatrix}^{-1} \cdot \begin{pmatrix} f(x_0, y_0) \\ g(x_0, y_0) \end{pmatrix}$$

Die Werte *x* und *y* sind dann unsere neuen (hoffentlich verbesserten) Schätzwerte. Das Problem bei der Aufgabe ist eigentlich nur das Invertieren der Matrix. Für Matrizen höherer Ordnung ist das eine aufwendige Sache (schauen Sie mal in einer Mathe-Formelsammlung nach ☹); bei zweidimensionalen Matrizen geht es mittels der "Cramerschen" Regel noch einigermaßen einfach:

$$\begin{pmatrix} \partial f/\partial x & \partial f/\partial y \\ \partial g/\partial x & \partial g/\partial y \end{pmatrix}^{-1} = \frac{1}{\partial f/\partial x \cdot \partial g/\partial y - \partial f/\partial y \cdot \partial g/\partial x} \begin{pmatrix} \partial g/\partial y & -\partial f/\partial y \\ -\partial g/\partial x & \partial f/\partial x \end{pmatrix}$$

Darin repräsentiert der Ausdruck unter dem Bruchstrich den Wert der Determinanten (berechnet nach der Regel von Sarrus). Die Determinante einer Matrix kann Null werden, und ist dies der Fall, so heißt das, dass die Matrix nicht invertierbar ist.

Aber auch wenn die Determinante sehr klein wird, ist das Newton-Verfahren an einem kritischen Punkt angelangt: Die „verbesserten“ Schätzwerte unterscheiden sich dann stark von den eben noch geschätzten Werten, und es ist wahrscheinlich, dass das Verfahren „aussteigt“. Hoffen wir, dass sowas bei uns nicht passiert.

Die Iterationsvorschrift für unser zweidimensionales Gleichungssystem ergibt sich also zu

$$x = x_0 - \frac{\partial g/\partial y \cdot f(x_0, y_0) - \partial f/\partial y \cdot g(x_0, y_0)}{\partial f/\partial x \cdot \partial g/\partial y - \partial f/\partial y \cdot \partial g/\partial x}$$

$$y = y_0 - \frac{-\partial g/\partial x \cdot f(x_0, y_0) + \partial f/\partial x \cdot g(x_0, y_0)}{\partial f/\partial x \cdot \partial g/\partial y - \partial f/\partial y \cdot \partial g/\partial x}$$

Darin sind x_0 , y_0 der ursprüngliche Schätzwert und *x*, *y* der hoffentlich verbesserte Wert.

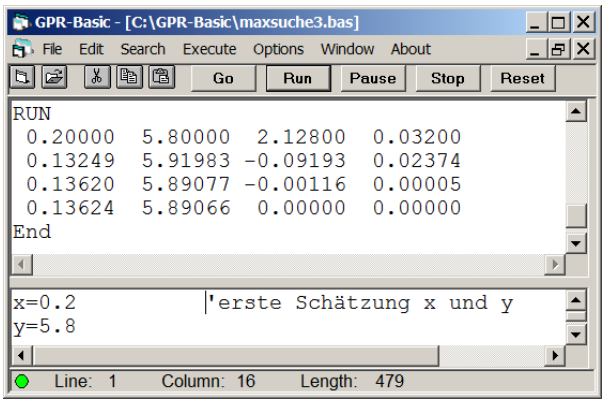
Erinnern Sie sich: Wir haben die theoretische Ableitung dieser Iterationsvorschrift wegen ihres hohen praktischen Wertes absichtlich mit allgemeingültigen Formeln durchgeführt (und das war geistig anstrengend ☺). Das Ergebnis jedoch ist eigentlich recht einfach, und die Umsetzung der Iterationsvorschrift für ein praktisches Beispiel ist fast noch einfacher.

Zurück also zu unserem Ausgangsbeispiel: Für die Lösung der Beispiel-Gleichungen können wir die partiellen Ableitungen für jede beliebige Stelle x, y analytisch berechnen:

$$\left.\begin{aligned} f(x,y) &= (y^2+2)\cdot x - 5 = 0 \\ g(x,y) &= (x^2+1)\cdot y - 6 = 0 \end{aligned}\right\} \quad \begin{aligned} \partial f/\partial x &= y^2+2 \\ \partial f/\partial y &= 2y\cdot x \\ \partial g/\partial x &= 2x\cdot y \\ \partial g/\partial y &= x^2+1 \end{aligned}$$

Natürlich könnten wir die Ableitungen auch numerisch berechnen; und wären die Gleichungen nicht explizit gegeben, so müssten wir das auch. So ein Fall würde zum Beispiel vorliegen, wenn sich die Funktionswerte f(*x*,*y*) und g(*x*,*y*) nur durch Unterprogramme berechnen lassen würden, welche uns vielleicht nicht näher bekannte Iterationen und Integrationen beinhalten könnten. Das Newton-Verfahren würde in solchen Fällen natürlich genauso funktionieren. Mit den analytischen Ableitungen aber lautet der Programmcode:

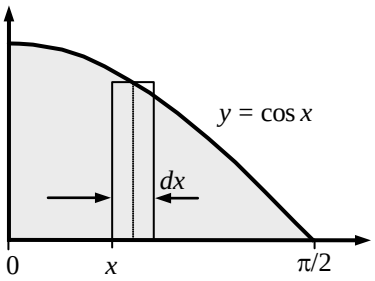
```
x=0.2              'erste Schätzung x und y
y=5.8
do
  f=(y^2+2)*x-5    'Funktionswerte f und g
  g=(x^2+1)*y-6
  print format(x,"00.00000"), format(y,"00.00000"), _
        format(f,"00.00000"), format(g,"00.00000")
  dfdx=y^2+2
  dfdy=2*y*x
  dgdx=2*x*y
  dgdy=x^2+1
  det=dfdx*dgdy-dfdy*dgdx    'Determinante
  x=x-(+dgdy*f-dfdy*g)/det   'verbessertes x
  y=y-(-dgdx*f+dfdx*g)/det   'verbessertes y
loop until abs(f)+abs(g)<0.000000001
end
```



Das Programm konvergiert sehr schnell zu hoher Genauigkeit, vorausgesetzt die erste Schätzung ist gut. Ist die Schätzung dagegen schlecht, so „schwingt“ das Programm hin und her und findet die Lösung eher zufällig erst nach etlichen Schritten. Dieses Verhalten ist für das Newton-Verfahren typisch und kann bei stark nichtlinearen Gleichungssystemen höherer Ordnung zu einem Problem werden: Das Verfahren konvergiert nur mit guten Anfangswerten. Diese richtig schätzen zu können hängt dann oft von der genauen Kenntnis der technischen Problemstellung ab und ist deshalb eher ein ingenieurmäßiges als ein mathematisches Problem.

3.4 Numerische Integration

Oftmals lassen sich mathematische Funktionen nicht mehr analytisch sondern nur numerisch integrieren. Die Cosinusfunktion gehört sicher nicht dazu, aber sie kann uns hier gut als Beispiel dienen, denn so können wir leicht die Genauigkeit der numerischen Rechnung mit dem exakten analytischen Wert vergleichen.
Berechnen wir einmal die Fläche F zwischen der Funktion $y = \cos x$ und der x-Achse im Bereich zwischen 0 und $\pi/2$. Dazu zerschneiden wir den Bereich in viele (z.B.: 1000) kleine Rechteckflächen der größe $dF = \cos x \cdot dx$, die wir dann einfach aufsummieren:




```
pi=4*atn(1)
F=0
dx=(pi/2)/1000
For i=0 To 999
    x=(pi/2)*i/1000
    dF=COS(x+0.5*dx)*dx              'Rechteckflächen
   'dF=0.5*(COS(x)+COS(x+dx))*dx     'Trapezflächen
    F=F+dF
Next i
Print "Ergebnis: ", F
```

Das Ergebnis 1.00000010280839 weicht nur wenig vom genauen Wert 1 ab (ein ähnlich gutes Ergebnis erhalten wir, wenn wir statt Rechteckflächen Trapezflächen hernehmen).
Ein sog. "uneigentliches Integral" liegt vor, wenn sich für einen Bereich eine Fläche berechnen lässt, obwohl der Funktionswert in diesem Bereich eigentlich unendlich groß wird. Das ist zum Beispiel der Fall für das Integral der folgenden Funktion:

$$F = \int_0^1 \frac{dx}{\sqrt{1-x}} = \left[-2\sqrt{1-x}\right]_0^1 = 2$$

Da der Funktionswert am Intervallende gegen unendlich strebt, würde nun die numerische Berechnung von F mit konstanter Schrittweite ungenau bzw. rechenaufwendig werden. Wir variieren dann besser die Schrittweite, z.B. indem wir ein kleines konstantes „Flächenstreifchen“ definieren (z.B. $dF = 0.001$) dessen Breite sich mit x ändert: $dx = dF \cdot \sqrt{1-x}$
Dann müssen wir nur zählen, wie oft dieses Flächenstreifchen in den Bereich hineinpasst:

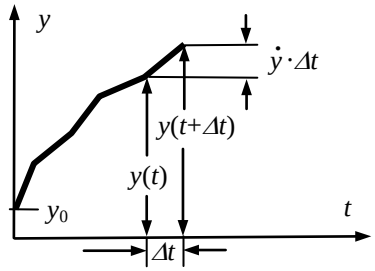
```
F=0
x=0
Do
   dx=0.001*sqr(1-x)
   F=F+dx/sqr(1-x)
   x=x+dx
Loop Until x>=1
Print "Ergebnis: ", F
```

Es gibt also vielfältige numerischen Methoden Integrale zu berechnen; Genauigkeit und Geschwindigkeit der Methoden hängen aber von der Art der zu berechnenden Funktion ab.

Auch Differentialgleichungen begegnen uns sehr oft in Natur und Technik, und leider lassen sich auch deren Lösungen in den meisten Fällen nicht mit uns bekannten Funktionen beschreiben. Das einfachste Verfahren zur numerischen Lösung ist das sog. "Eulerverfahren", es lässt sich leicht mittels For-Next (oder Do-Loop) Schleife in GPR-Basic realisieren.

Betrachten wir einmal eine nichtlineare Differentialgleichung erster Ordung $\dot{y} = f(y, t)$ mit dem Anfangswert $y(0) = y_0$ (es könnte auch ein System von Gleichungen sein, dann ist y ein Vektor und $f(y, t)$ eine Vektorfunktion). Für die Änderung von y beim Übergang von einem Punkt t zum unendlich nahen Nachbarpunkt $t + dt$ gilt dann: $y(t+dt) = y(t) + \dot{y} \cdot dt$. Ersetzen wir nun den unendlich kleinen Intervall dt durch einen kleinen aber endlich kleinen Intervall Δt, so haben wir näherungsweise:

$$y(t+\Delta t) \approx y(t) + \dot{y} \cdot \Delta t \,, \quad \text{mit } \dot{y} = f(y, t) \,.$$



Somit lässt sich also eine Reihe aufstellen:

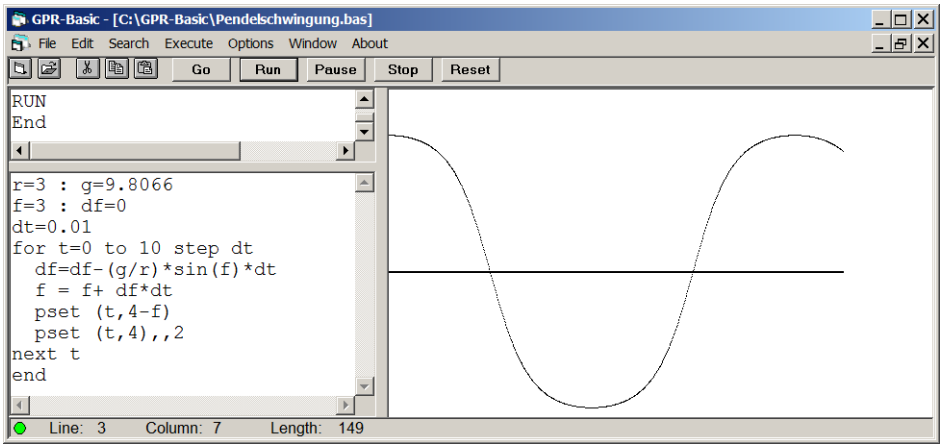
$$y_{i+1} = y_i + f(y_i, t_i) \cdot \Delta t$$

die ausgehend vom Punkt y_0 (für den Punkt t_0) alle nachfolgenden Werte y_i (jeweils mit dem Abstand Δt) Schritt für Schritt berechnet.

Dazu ein Beispiel: Bei einem schwingenden Pendel (eine Punktmasse m aufgehängt im Erdschwerefeld $g = 9.8066$ m/s^2 an einem reibungsfrei drehbaren masselosen Draht der Länge r) wird der Auslenkungswinkel $\varphi(t)$ durch die Differentialgleichung $\ddot{\varphi} + (g/r) \cdot \sin\varphi = 0$ beschrieben. Die Näherungslösung für kleine Auslenkungen beinhaltet die uns gut bekannten harmonischen Funktionen (Sinus und Cosinus); die genaue Lösung beinhaltet uns weniger gut bekannte sog. elliptische Integralfunktionen. Durch die Einführung der Variablen $\dot{\varphi}$ können wir die DGL zweiter Ordnung in zwei DGL erster Ordnung transformieren:

$$d\dot{\varphi}/dt = -(g/r) \cdot \sin\varphi$$
$$d\varphi/dt = \dot{\varphi}$$

Das folgende Programm integriert dann dieses Gleichungssystem für die Anfangswerte $\varphi = 3$; $\dot{\varphi} = 0$ (im Bogenmaß) mit einer Schrittweite $\Delta t = 0.01$ s und zeichnet die Integralfunktion (im Maßstab $r = 3$ m $\rightarrow$ 3 cm) in den Grafikbereich des GPR-Basic Fensters:

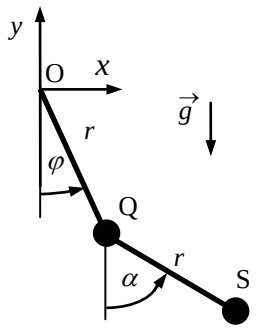


Numerische Integrationen sind zur "Simulation" von Bewegungsabläufen sehr gut geeignet. Wenn wir das Punktpendel schwingen sehen möchten, müssen wir einfach nur die grafische Ausgabe ein kleines bisschen abändern:

```
draw coordinates (6,6),1,-1
r=3 : g=9.8066
f=3 : df=0    ' Anfangswerte
dt=0.01       ' Schrittweite
for t=0 to 100 step dt
  df=df-(g/r)*sin(f)*dt
  f = f+ df*dt
  cls
  Qx= r*sin(f) : Qy=-r*cos(f)
  line (0,0) to (Qx,Qy),,5
  pset (Qx,Qy),,25
next t
```

Die Grafikanweisung `draw coordinates (6,6),1,-1` steht am Programmanfang, sie versetzt den Ursprung des *x*-*y*-Koordinatensystems in etwa in den Mittelpunkt des Grafikbereichs und stellt die *x*-Koordinate nach rechts und die *y*-Koordinate nach oben ein.
Clear screen `cls` löscht den Grafikbereich; `line (0,0) to (Qx,Qy),,5` zieht eine 5-Punkte starke Linie vom Koordinatenursprung O zum aktuellen Aufenthaltsort der pendelnden Masse (Punkt Q: $x = r \cdot \sin\varphi$, $y = -r \cdot \cos\varphi$); und `pset (Qx,Qy),,25` zeichnet dort einen 25 Punkte großen Kreis. Durch geeignet Wahl der Schrittweite *dt* können wir das Pendel schnell oder langsam schwingen lassen.

Das Schwingungsverhalten des Pendels ist aber gut vorhersehbar, und so ist diese Simulation auch nicht besonders spektakuär. Das ändert sich aber sobald wir ein zweites gleichartiges Pendel S an den bewegten Massepunkt Q anhängen.
Die Ableitung der Bewegungsgleichungen für das Doppelpendel ist eine (etwas komplizierte) Aufgabe aus der Technischen Mechanik:



$$\ddot{\alpha} + \ddot{\varphi} \cdot \cos(\alpha-\varphi) + \dot{\varphi}^2 \cdot \sin(\alpha-\varphi) + (g/r) \cdot \sin\alpha = 0$$

$$2\ddot{\varphi} + \ddot{\alpha} \cdot \cos(\alpha-\varphi) - \dot{\alpha}^2 \cdot \sin(\alpha-\varphi) + 2(g/r) \cdot \sin\varphi = 0$$

Wenn wir diese gekoppelten nichtlinearen Gleichungen mittels der Euler-Methode numerisch integrieren wollen, müssen wir sie zunächst umstellen, d.h. nach $\ddot{\alpha}$ und $\ddot{\varphi}$ auflösen.
Dies geschieht am übersichtlichsten in Matrix-Vektor Schreibweise:

$$\begin{pmatrix} 1 & \cos(\alpha-\varphi) \\ \cos(\alpha-\varphi) & 2 \end{pmatrix} \cdot \begin{pmatrix} \ddot{\alpha} \\ \ddot{\varphi} \end{pmatrix} = \begin{pmatrix} -\dot{\varphi}^2 \sin(\alpha-\varphi) - (g/r)\sin\alpha \\ +\dot{\alpha}^2 \sin(\alpha-\varphi) - 2(g/r)\sin\varphi \end{pmatrix}$$

Dabei werden die Winkelbeschleunigungen als ein Vektor aufgefasst, welcher mit einer Koeffizientenmatrix multipliziert eine nichtlineare Vektorfunktion ergibt.

Für die Invertierung der 2×2 Matrix gilt die Cramersche Regel (die Elemente in der Hauptdiagonalen vertauschen, die Elemente in der Nebendiagonalen mit –1 multiplizieren, und schließlich alle Elemente durch die Determinante der Matrix teilen). Also:

$$\begin{pmatrix} \ddot{\alpha} \\ \ddot{\varphi} \end{pmatrix} = \frac{1}{2 - \cos^2(\alpha-\varphi)} \begin{pmatrix} 2 & -\cos(\alpha-\varphi) \\ -\cos(\alpha-\varphi) & 1 \end{pmatrix} \cdot \begin{pmatrix} -\dot{\varphi}^2 \sin(\alpha-\varphi) - (g/r)\sin\alpha \\ +\dot{\alpha}^2 \sin(\alpha-\varphi) - 2(g/r)\sin\varphi \end{pmatrix}$$

Schließlich können wir durch Einführen der "Substitutionen" $\dot{\varphi}$ und $\dot{\alpha}$ aus diesen zwei Gleichungen zweiter Ordnung ein System von vier Gleichungen erster Ordnung bilden.

$$\left.\begin{array}{l} d\dot{\alpha}/dt = \ddot{\alpha} \\ d\dot{\varphi}/dt = \ddot{\varphi} \\ d\alpha/dt = \dot{\alpha} \\ d\varphi/dt = \dot{\varphi} \end{array}\right\}$$

Das folgende Basic Programm integriert diese Gleichungen und simuliert so die Bewegung:

```
draw coordinates (6,6),1,-1
r=3 : g=9.8066
dt=0.01                          'Schrittweite
df=0             : da=0          'Anfangswerte
f=3              : a=3           'a=2.9, a=3.1
'f=-1+sqr(2)     : a=+2-sqr(2)       'langsam
'f=(-1-sqr(2))/5 : a=(+2+sqr(2))/5   'schnell

for t=0 to 100 step dt
   f1=-df^2*sin(a-f)-  (g/r)*sin(a)
   f2= da^2*sin(a-f)-2*(g/r)*sin(f)
   det=2-cos(a-f)^2
   dda=(   2*f1      -cos(a-f)*f2)/det
   ddf=(-cos(a-f)*f1     +f2     )/det
   da=da+dda*dt
   df=df+ddf*dt
   a=a+da*dt
   f=f+df*dt
   Qx=   r*sin(f) : Qy=  -r*cos(f)
   Sx=Qx+r*sin(a) : Sy=Qy-r*cos(a)
   cls
   line (0,0) to (Qx,Qy),,5
   pset (Qx,Qy),,25
   line (Qx,Qy) to (Sx,Sy),,5
   pset (Sx,Sy),,25
next t
```

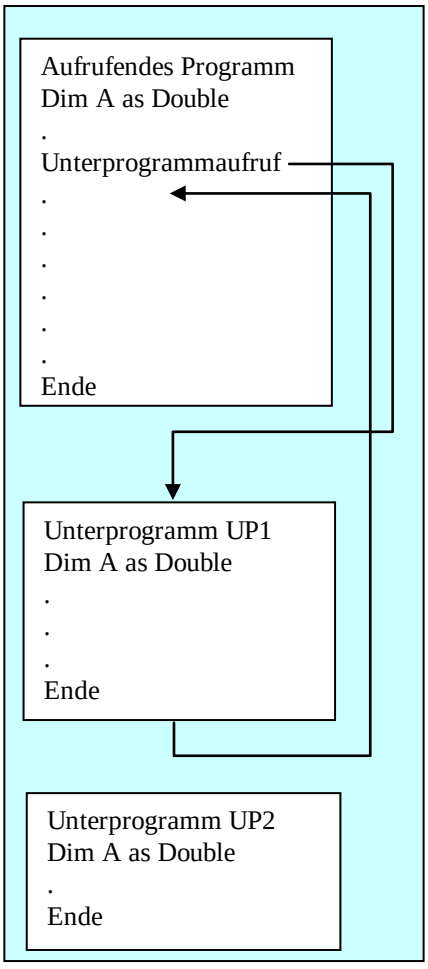
Bei kleineren Auslenkungswinkeln ist eine analytische Berechnung der Bewegung durch eine Linearisierung des Gleichungssystems noch möglich (z.B. erhält man so die Anfangswerte für die "langsame Schwingung miteinander" und die "schnelle Schwingung gegeneinander"); bei größeren Auslenkungswinkeln jedoch ist man auf eine numerische Simulation des nichtlinearen Gleichungssystems angewiesen. Dabei zeigt das Doppelpendel ein recht chaotisches Verhalten: Sich nur wenig unterscheidende Anfangsbedingungen führen nach kurzer Zeit zu sehr unterschiedlichen Bewegungen (probieren Sie einmal α_0 oder φ_0 = 2.9, 3.0, 3.1, etc.).

4 Unterprogrammtechniken

4.1 Segmentierte Programmierung

Programme werden oft lang und kompliziert, und besonders dann beginnt man schnell, den Überblick zu verlieren (es kommt natürlich auch vor, dass man den Überblick bei einem kurzen Programm verliert ☺). Notwendig und hilfreich ist dann die Unterprogrammtechnik. Unterprogramme sind separate Programmsegmente, die i.A. eine klar definierte Aufgabe erledigen, welche immer wieder vorkommt, und welche über eine einfache Schnittstelle zu dem aufrufenden Hauptprogramm verfügen.

Natürlich können auch Unterprogramme andere Unterprogramme (und sogar sich selbst) aufrufen. Aufruf und Rückkehr zum aufrufenden Programm lässt sich etwa folgendermaßen veranschaulichen:



Unterprogramme werden ausgeführt (im Gegensatz zu sogenannten Makros, die nur als Text vor der Programmausführung in das aufrufende Programm eingefügt werden). Der Aufruf eines Unterprogrammes ist für den Computer vergleichsweise viel Arbeit (Variableninhalte müssen erst gesichert und später wieder hergestellt werden). Man sollte deswegen nicht den Ehrgeiz entwickeln, jede Kleinigkeit in der Programmausführung als Unterprogramm zu gestalten.

Ein ganz wesentlicher Gesichtspunkt ist, dass Unterprogramme über eigene Variablen verfügen. Diese werden als „lokale Variablen“ bezeichnet, im Gegensatz zu „globalen Variablen“, die in allen Programmsegmenten eines Fenster vorhanden sind. Zum Beispiel, eine lokal definierte Variable mit dem Namen A im Unterprogramm UP1 bezeichnet einen anderen Speicherplatz als eine lokale Variable mit demselben Namen A im anderen Unterprogramm UP2 oder im aufrufenden (Haupt-) Programm. Diese Variablen-Kapselung macht es möglich, dass Unterprogramme unabhängig voneinander entwickelt werden können; außerdem ist die Gefahr von Fehlern so erheblich geringer. Besonders bei der Entwicklung von großen Programmen spielt das oft eine entscheidende Rolle.

4.2 Subroutinen

Probieren wir doch gleich mal ein Beispiel. Am besten dazu ist Grafik geeignet, denn da sieht man was passiert (und außerdem erfreut so ein buntes Bildchen jeden gestressten Programmierer ☺). Das folgende Programm gibt im Grafikbereich ein Feld von verschieden-farbigen Kästchen (bzw. Quadraten) mit kontinuierlichen Farbübergängen aus. Es verwendet dazu einige Grafik-Befehle: `cls` löscht den Grafikbereich, `draw scale pixel` setzt die Skala von cm (voreingestellt) auf pixel, `draw color` setzt die individuelle Farbe eines Kästchens (unter Zuhilfenahme der Funktion rgb) und `brick` gibt ein Kästchen aus.

Ziehen Sie den Grafikbereich eines GPR-Basic Fensters auf, tippen sie das Programm in den Programmbereich des Fensters ein und starten Sie es:

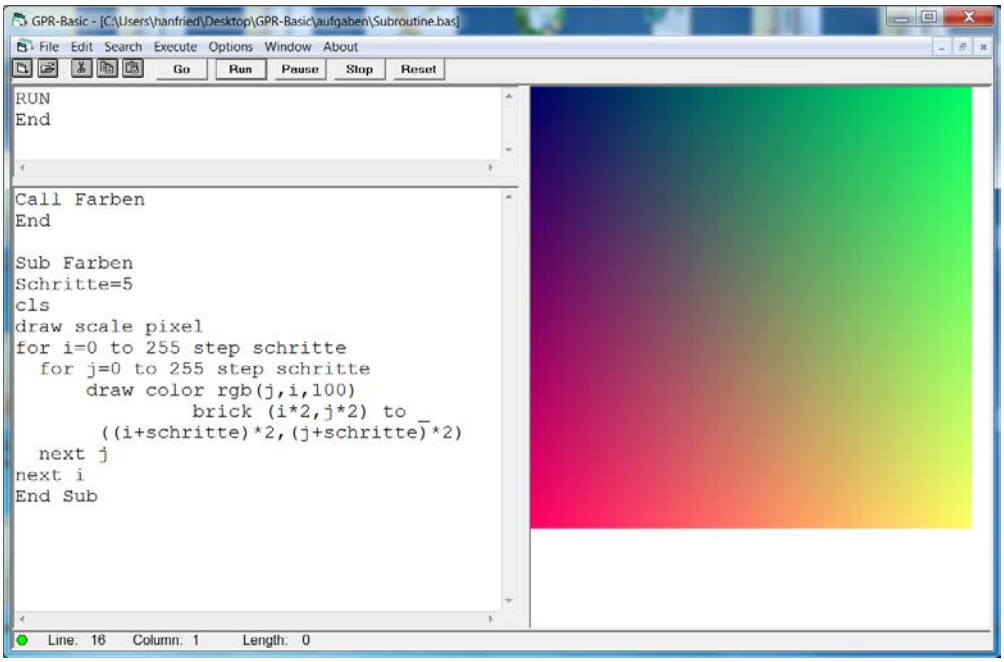
```
Schritte=5
cls
draw scale pixel
for i=0 to 255 step schritte
  for j=0 to 255 step schritte
      draw color rgb(j,i,100)
      brick (i*2,j*2) to _
      ((i+schritte)*2,(j+schritte)*2)
  next j
next i
```

Sie können das Programm abspeichern, zum Beispiel unter dem Namen „Subroutine.bas“. Das Programm ist natürlich noch kein Unterprogramm. Wir können es aber in ein solches abändern, dazu fügen wir vor den Code eine Zeile mit dem Text `Sub Farben` („Farben“ sei hier der Name des Unterprogramms) ein, und nach dem Code eine Zeile mit dem Text `End Sub`. Jetzt ist das Programm ein richtiges Unterprogramm. Um es zu starten, brauchen wir noch ein Hauptprogramm. Dieses können wir einfach vor das Unterprogramm einfügen:

```
Call Farben
End

Sub Farben
Schritte=5
cls
draw scale pixel
for i=0 to 255 step schritte
  for j=0 to 255 step schritte
      draw color rgb(j,i,100)
      brick (i*2,j*2) to _
       ((i+schritte)*2,(j+schritte)*2)
  next j
next i
End Sub
```

Das Hauptprogramm besteht hierbei nur aus zwei Zeilen, nämlich dem Unterprogramm-Aufruf `Call Farben` und dem Befehl `End` (den man auch weglassen könnte).

Wenn Sie das Programm starten, sollte etwa folgendes Fenster auf Ihrem PC erscheinen:

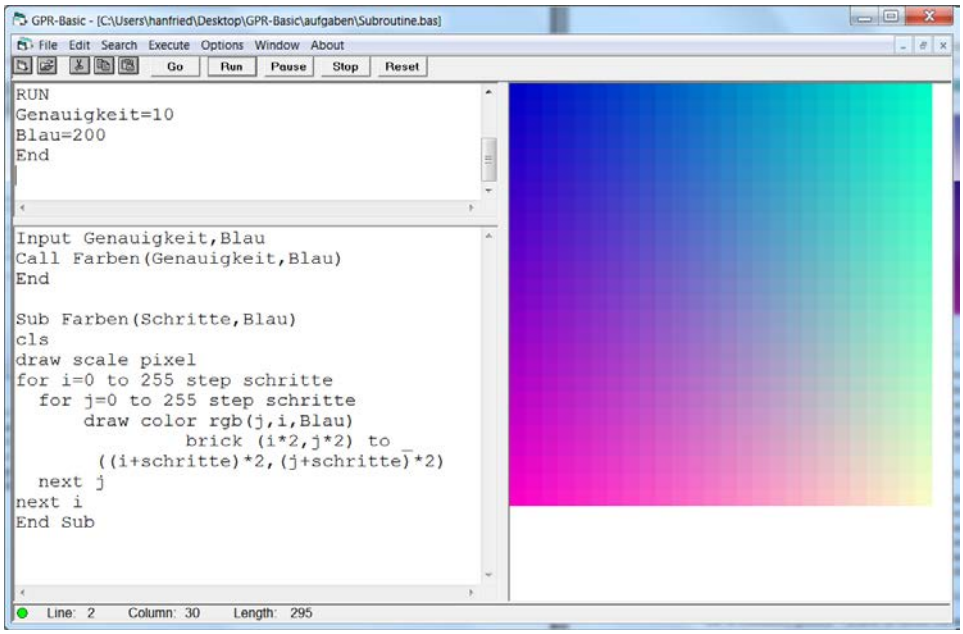


Um das Unterprogramm flexibler zu gestalten wollen wir als Beispiel einmal die Wahl der Variablen „Schritte“ und die Wahl der dritten Farbe in der rgb-Funktion (also den Wert 100 für blau, denn rgb steht für rot-grün-blau) dem Hauptprogramm überlassen. Diese zwei Werte müssen dann vom Hauptprogramm an das Unterprogramm übergeben werden, und das geschieht in der Liste der "formalen Parameter" (oder "Argumente").
Ändern wir das Programm also folgendermaßen ab:

```
Input Genauigkeit
Input Blau
Call Farben(Genauigkeit,Blau)
End

Sub Farben(Schritte,Blau)
cls
...
          draw color rgb(j,i,Blau)
...

End Sub
```

Beim Programmaufruf (Call) werden die im Hauptprogramm bekannten Werte für die Variablen `Genauigkeit` und `Blau` an das Unterprogramm übergeben und dort in den lokalen Variablen `Schritte` und `Blau` abgespeichert. Auch wenn Variablen im Unterprogramm den gleichen Namen haben wie im Hauptprogramm (im Beispiel den Namen `Blau`), so sind es doch zwei ganz verschiedene Speicherplätze. Im Unterprogramm lassen wir die Anweisung `Schritte=5` dann natürlich weg, und in der rgb-Funktion ändern wir `100` auf `Blau`.

Sie können jetzt nach Programmstart im Hauptprogramm Werte für `Genauigkeit` und `Blau` eingeben. `Genauigkeit=10` und `Blau=200` liefert z.B. etwa das folgende Bild:



Zum Abschluss noch ein kleines Experiment: Fügen Sie einmal ins Unterprogramm eine Zeile ein, und zwar vor die End Sub Anweisung die Wertzuweisung: `Schritte=1000`
Unmittelbar nachdem der Variablen „Schritte“ dieser neue Wert zugewiesen worden ist, wechselt die Programmausführung wieder ins Hauptprogramm zurück. Wir können uns den Wert der Variablen „Genauigkeit“ im Hauptprogramm nach der Unterprogramm-Ausführung ansehen, indem wir vor „End“ eine Print-Anweisung einfügen:

```
Print "Genauigkeit=",Genauigkeit
```

Die Variable „Genauigkeit“ im Hauptprogramm hat also nach Unterprogrammaufruf den Wert 1000, egal, was wir vorher mit der Input-Anweisung dafür eingegeben hatten. Dies ist also eine Methode, wie man Information vom Unterprogramm an das aufrufende Programm zurückgeben kann. Diese „Rückgabe des neuen Wertes“ funktioniert aber nur, wenn im Unterprogrammaufruf eine nackte Variable und kein arithmetischer Ausdruck steht. In der Call-Anweisung könnte nämlich in diesem Beispiel auch stehen:

```
Call Farben((Genauigkeit),Blau)
```

Die in der formalen Parameterliste jetzt in Klammern gesetzte Variable `(Genauigkeit)` ist bereits ein arithmetischer Ausdruck (nicht mehr splitternackt ☺): bei Übergabe an das Unterprogramm hat der Ausdruck `(Genauigkeit)` natürlich denselben Wert wie die Variable `Genauigkeit`, aber die Änderung der Variablen ist nun blockiert, und der ursprüngliche Wert der Variablen im aufrufenden Programm wird beibehalten.

4.3 Lokale und Globale Variablen

Fassen wir wegen der großen Wichtigkeit noch einmal zusammen: Übergabe von Information an Unterprogramme geschieht mit Hilfe von sogenannten formalen Parametern; das sind (lokale!) Variablen, die in der Sub Anweisung hinter dem Prozedur-Namen aufgelistet werden (abgetrennt vom Prozedur-Namen durch runde Klammern und in diesen Klammern separiert durch Kommata). Diesen lokalen Variablen werden beim Unterprogramm-Aufruf Werte zugewiesen, und zwar in Form einer Liste von Variablen oder arithmetischen Ausdrücken, die der Call Anweisung nach dem Prozedurnamen angehängt wird (ebenfalls abgetrennt vom Prozedur-Namen durch runde Klammern und in diesen Klammern separiert durch Kommata). Wenn die Ausführung des Unterprogramms beendet wird, werden die Werte der *Variablen* in der Parameterliste des Aufrufs mit den aktuellen Werten der entsprechenden Variablen des Unterprogramms besetzt. Werden dagegen *arithmetische Ausdrücke* oder *Konstanten* im Aufruf verwendet, so kann an der Stelle natürlich kein Informations-Rückfluss erfolgen. Es ist dabei also wesentlich, dass Anzahl und Typ der im Aufruf aufgelisteten Werte mit Anzahl und Typ der Variablen in der Sub Anweisung übereinstimmen. Die Namen der Variablen brauchen nicht übereinzustimmen.

Bei den meisten Programmiersprachen und also auch bei GPR-Basic kann (und sollte) man den Typ der zu übergebenden Variablen in der Sub Anweisung explizit deklarieren.
Zum Beispiel, im obigen Programm hätten wir auch schreiben können:

```
Sub Farben(Schritte as Double, Blau as Long)
```

Alternativ können wir diese Parameter-Variablen natürlich auch implizit (als Double) oder durch ein angehängtes Sonderzeichen (Suffixe #,& oder $) deklarieren. Es gibt drei Methoden zur Typdeklaration: Erstens implizit als Double, zweitens durch das angehängte Suffix und drittens explizit durch eine Dimensionierungs-Anweisung:

```
Dim x as Double, y as Long, z as String
```

Eine solche explizite Dimensionierung kann überall im Programm erfolgen, allerdings notwendigerweise vor dem ersten Benutzen der Variablen. Die so deklarierten Variablen sind „lokale Variablen“, d.h. ihr Gültigkeitsbereich ist auf das Unterprogramm (bzw. das Hauptprogramm) beschränkt, in welchem sie deklariert worden sind. Man kann auch schreiben:

```
Local x as Double, y as Long, z as String
```

`Local` hat die gleiche Wirkung wie `Dim`. Schreibt man aber stattdessen:

```
Global x as Double, y as Long, z as String
```

so sind diese Variablen im gesamten Bereich (d.h. im Hauptprogramm und in allen Unterprogrammen) gültig. Die Variable `x` im Unterprogramm ist dann der gleiche Speicherplatz wie die Variable `x` im Hauptprogramm.

Im Gegensatz zu globalen Variablen sind lokale Variablen "flüchtig", d.h. ihre Inhalte gehen bei der Rückkehr ins aufrufende Programm verloren. Nur die Information, die über die Parameterliste übergeben wird oder in globalen Variablen gespeichert ist, bleibt erhalten.

Unterprogramme werden i.A. bis zum Ende abgearbeitet, d.h. bis zur Anweisung `End Sub` (bzw. `End Function`). Diese Anweisung bewirkt die Rückkehr ins aufrufende Programm und die Wiederherstellung der Variablen im aufrufenden Programm. Will man vor dem Ende des Unterprogramm-Codes in das aufrufende Programm zurückkehren, so kann man dazu die Anweisung `Exit Sub` (bzw. `Exit Function`) verwenden.

Die Verwendung von globalen Variablen erscheint besonders dem Programmieranfänger als eine elegante Methode, Probleme bei der Variablenübergabe zu vermeiden. Besonders dann wenn viele Variablen übergeben werden müssen ist die Versuchung groß, stattdessen lieber globale Variablen zu verwenden. Vor der übermäßigen Verwendung von globalen Variablen aber muss gewarnt werden: Gerade die „Variablen-Kapselung“ mit einer klar definierten Schnittstelle (der Parameterliste) ist der große Vorteil der Unterprogrammtechnik.
Bei einem Geometrie-Programm könnte es sich zum Beispiel anbieten, die Variable π mit `Global Pi=4*Atn(1) as Double` im Hauptprogramm als globale Variable zu dimensionieren, um sie so nur einmal im Hauptprogramm berechnen zu müssen. Wenn dann aber nun ein Ausländer im Programmierteam ist, und wenn der gern Variablen mit Begriffen aus seiner Muttersprache bezeichnet, und wenn dann in dessen Muttersprache Pi etwas ganz anderes heißt als bei uns, und wenn der das nicht weiß...

Es stellt sich die Frage, ob es möglich ist, in einem Unterprogramm eine Variable als Local zu definieren, wenn es schon eine globale Variable mit genau demselben Namen gibt?
Antwort: In GPR-Basic geht das. Die globale Variable wird dann einfach ignoriert aber nicht überschrieben. Zuerst wird überprüft, ob es eine lokale Variable mit diesem Namen gibt, dann erst wird geprüft, ob es eine so bezeichnete globale Variable gibt. Es ist aber nicht möglich, den Typ einer existierenden Variablen einfach neu zu deklarieren. Ist also z.B. eine Double Variable mit dem Namen `x` im Programm bereits verwendet worden (vorhanden), so führt die Anweisung `Dim x as Long` zu einer Fehlermeldung. Die Anweisung RESET löscht alle aktuellen lokalen oder globalen Variablen.

Wenn man ausnahmsweise einmal möchte, dass ein Unterprogramm nicht mit eigenen Variablen sondern stattdessen mit den Variablen des aufrufenden Programms arbeitet, so kann man das in GPR-Basic durch die Verwendung des Schlüsselwortes `"Inline"` vor einem Sub (oder Function) Statement erreichen. Solche InlineSubs oder InlineFunctions verhalten sich dann exakt genauso wie normale Subs oder Functions, sie sind aber keine „echten Unterprogramme“, denn sie kennen und verwenden ja die lokalen Variablen des aufrufenden Programms (und sie gehören nicht zum Standard Basic!). Falls man beim ihrem Aufruf mit einer Parameterliste Variablen erzeugt, so sind auch das lokale Variablen des aufrufenden Programms. Und obwohl die Verwendung von InlineSubs oder InlineFunctions etwas Rechenzeit und etwas Speicherplatz spart, so ist i.A. die Verwendung von echten Unterprogrammen mit einer sauberen Schnittstelle durch eigene lokale Variablen die richtige Programmiertechnik.

4.4 Funktions-Unterprogramme

In der Programmiersprache Basic gibt es außer der Sub noch einen anderen Typ von Unterprogramm: Die Function. Wir haben ja Funktions-Unterprogramme schon ein paarmal erwähnt; und eigentlich kennen wir sie schon, denn die eingebauten Funktionen wie Sin, Sqr, Atn, Rgb etc. sind nichts anderes. Allerdings sind diese eingebauten Funktionen bereits im Basic-System vorhanden, so dass man dafür keinen Programmcode schreiben muss.
Man kann aber auch eigene Funktionen schreiben.

Function-Unterprogramme unterscheiden sich von Sub-Unterprogrammen durch die Art ihres Aufrufs und durch die Tatsache, dass sie einen Wert an das aufrufende Programm zurückgeben. Zum Beispiel, die eingebaute Funktion SQR(X) wird nicht durch eine Call-Anweisung aufgerufen, sondern sie wird eher ähnlich wie eine Variable benutzt (sie steht in einem arithmetischen Ausdruck an einer Stelle, wo auch eine Variable stehen könnte). Deswegen muss es für die Funktion auch einen Rückgabewert geben, ebenso wie eine Variable an dieser Stelle einen Wert haben müsste. Die Funktion übernimmt die Werte aus der Parameterliste (im Beispiel nur einen Wert, nämlich den der Variablen X), berechnet das Ergebnis (im Beispiel die Wurzel aus X), und gibt den Ergebniswert wie eine Variable an die Stelle des aufrufenden Programmes zurück, von welcher aus sie gestartet worden ist.

Wir können einmal probieren, das Funktions-Unterprogramm SQR(X) als eigene Function nachzuprogrammieren. Das soll natürlich nur eine Übung sein, denn normalerweise verwendet man zur Wurzelberechnung selbstverständlich die (viel schnellere) eingebaute Funktion, deswegen gibt es sie ja. Für das eigene Unterprogramm wollen wir die folgende Iterationsvorschrift verwenden:

$$r_{i+1} = 0.5 \cdot (r_i + x / r_i),$$

die wir weiter oben im Text ja schon einmal kennengelernt haben (darin war r_i der aktuelle Wert für die Wurzel und r_{i+1} der verbesserte Wert). Das Programm lautete damals:

```
Input x
r=x                ' x ist erster Schätzwert für r
Do
   r=0.5*(r+x/r)
Loop until ABS(r*r-x)<0.0000000000001
Print "r= ",r, " zum Vergleich: ",SQR(x)
```

Wenn wir dieses Programm in ein Funktions-Unterprogramm umschreiben wollen, dann sollten wir die Anweisungen Input und Print im aufrufenden Hauptprogramm verwenden, denn Ein- und Ausgabe-Operationen gehören i.A. nicht in Funktions-Unterprogramme (obwohl sie da natürlich auch stehen könnten). Hier sind Ein- und Ausgabeoperationen typische Aufgaben des Hauptprogramms. Dieses könnte also lauten:

```
Input x
Print "r= ",Wurzel(x), " zum Vergleich: ",SQR(x)
End
```

Wurzel ist der Name unseres eigenen Funktions-Unterprogrammes; und Wurzel(x) in der Print Anweisung der Aufruf. Der Code des Unterprogramms Wurzel ist dann:

```
Function Wurzel(x as double) as double
      r=x
      Do
         r=0.5*(r+x/r)
      Loop until ABS(r*r-x)<0.0000000000001
      Wurzel=r
End Function
```

Die erste Zeile hätten wir auch kürzer schreiben können als `Function Wurzel(x)`, denn sowohl `x` wie auch `Wurzel` sind ja wie alle Variablen implizit als Double deklariert.

`Wurzel` ist nicht nur der Name des Unterprogrammes sondern auch ist eine im Bereich des Unterprogrammes definierte lokale Variable, die den Rückgabewert aufnimmt. Bei ihrer Deklaration (beim Unterprogramm-Aufruf) wird ihr der Wert Null zugewiesen. Bitte beachten Sie, dass der Variablen `Wurzel` in der zweitletzten Zeile des Unterprogrammes ein Ergebniswert zugewiesen wird. Vergessen wir diese Wertzuweisung, so gibt das Programm statt des iterierten Wertes einfach den Wert Null zurück. Bitte beachten sie auch, dass die Variable `r` im aufrufenden Hauptprogramm selbstverständlich unbekannt ist.

Natürlich können wir das Programm Wurzel jetzt überall im Hauptprogramm aufrufen, d.h. an jeder Stelle, wo auch eine Variable stehen könnte. Probieren sie einmal, die vierte Wurzel zu berechnen, indem sie schreiben:

```
Print "r=",Wurzel(Wurzel(x)), " zum Vergleich: ",SQR(SQR((x))
```

Natürlich können Funktionen auch Zeichenkettenvariablen (Strings) als Parameter übernehmen oder Zeichenketten zurückgeben. Wenn man z.B. in einem Programm die eingebauten Funktion TIME immer zusammen mit einem beliebigen Datum ausgeben möchte, kann man für diese Aufgabe leicht eine eigene Funktion schreiben.

```
Function Timedate(datum as string) as string
Timedate="Tag: " + datum + ", Uhrzeit: " + TIME
End Function
```

Die Variable `datum` wird dabei als (einziger) Parameter übergeben, die Zeit wird über die Systemfunktion `TIME` berechnet. `Timedate` übernimmt die Aufgabe einer String-Variablen. Wir können diese Funktion also nun z.B. mit den beiden folgenden
(Hauptprogramm-) Anweisungen aufrufen:

```
Print Timedate("Heute")
Print Timedate(date)
End
```

Beim ersten Aufruf übergeben wir `"Heute"` als eine Zeichenketten-Konstante an das Unterprogramm; beim zweiten Aufruf übergeben wir `date`, eine Systemfunktion, an das Unterprogramm. Tatsächlich wird aber `date` nicht übergeben, sondern vor der Übergabe als ein arithmetischer Ausdruck betrachtet und ausgewertet. Das Ergebnis dieser Auswertung ist eine Zeichenkette, die dann übergeben wird (der Variablen `datum` zugewiesen wird).

Schließlich sollte noch erwähnt werden, dass im GPR-Basic Programm zwischen einer Function und einer Sub überhaupt nicht unterschieden wird, d.h. eine Function kann wie eine Sub mit einer Call-Anweisung aufgerufen werden; und in einem arithmetischen Ausdruck kann an der Stelle einer Function auch eine Sub aufgerufen werden (dabei kann auch einer Sub Prozedur über ihren Namen ein Rückgabewert zuordnet werden). Eigentlich gibt es also in GPR-Basic (ähnlich wie in der Programmiersprache C) nur einen einzigen Unterprogramm-Typ. Die Aufteilung in zwei unterschiedliche Unterprogramm-Typen ist aber charakteristisch für Basic (und Fortran) wurde wohl ursprünglich aus didaktischen Gründen so vorgenommen, d.h. es wurde erwartet, dass dies logischer und für Anfänger leichter zu erlernen sei. Allerdings meint der Autor dieser Zeilen (also ich ☺), dass die Erfahrung wohl eher dagegen spricht.

4.5 Rekursive Programmierung

Wie schon erwähnt, können Unterprogramme auch andere Unterprogramme aufrufen, und sie können sogar sich selbst aufrufen. Dies führt zu einem Programmierstil, der i.A. gerne als „rekursiv“ bezeichnet wird, und den man eigentlich tunlichst lieber unterlassen sollte. Beim Unterprogrammaufruf müssen die aktuellen Werte der lokalen Variablen des aufrufenden Programmes nämlich in einem Stapelspeicher (dem sogenannten „Stack“) bis zu ihrer Wiederherstellung zwischengespeichert werden, und wenn das Unterprogramm sich ohne Ende immer wieder selbst aufruft, so läuft dieser Stack zwangsläufig über („Stack-Overflow“). Man muss bei der rekursiven Programmierung also immer genau darauf achten, dass die Bedingung für die Beendigung der Rekursion auch sicher erreicht wird.

Als Beispiel dazu wollen wir einmal ein Funktions-Unterprogramm `F(N)` schreiben, dass die Fakultät für einen Parameter `N` berechnet und an das aufrufende Hauptprogramm zurückgibt. Dieses Unterprogramm könnte dann z.B. folgendermaßen aufgerufen werden:

```
Input N
Print F(N)
End
```

Und der Code für das Unterprogramm könnte lauten:

```
Function F(N)
  F=1
  For i=1 To N
    F=F*i
  Next i
End Function
```

Wir haben diesen Lösungsansatz mittels einer For Next Schleife ja schon kennengelernt; und genau so und nicht anders sollte man die Berechnung der Fakultät auch programmieren. Der Ausdruck `F` im Code des Unterprogrammes ist eine lokale Variable, deren Wert beim Beenden des Unterprogrammes an das aufrufende Hauptprogramm zurückgegeben wird.
Es geht aber auch ohne eine For Next Schleife! Man könnte z.B. auch schreiben:

```
Function F(N)      : Print "N=",N
  If N=1 then
       F=1
  Else
       F=N*F(N-1)
  Endif            : Print "F=",F
End Function
```

Jetzt wird im Programmcode des Unterprogramms der Ausdruck `F` wiederum als eine lokale Variable betrachtet, der Ausdruck `F(N-1)` dagegen als ein erneuter Aufruf desselben Unterprogrammes, und zwar diesmal mit dem Parameter `N-1`. Insgesamt wird das Unterprogramm dabei dann genau *N*-mal aufgerufen, beginnend mit dem Parameter `N` und dann mit einem jeweils um 1 verringerten Wert, bevor es dann *N*-mal wieder beendet wird und dabei jeweils (beginnend mit 1) die Ergebniswerte der Fakultätsberechnung 1, 2, 6, 24, 120, ... zurückgibt.

Und obwohl diese Variablen allesamt `N` (bzw. `F`) heißen, nehmen sie doch bei jedem Unterprogrammaufruf (bzw. bei jedem Ende) einen anderen Wert an, und natürlich handelt es sich dabei nicht um dieselben Speicherplätze. Der Bedarf an Speicherplatz und Rechenzeit ist also beim rekursiven Lösungsansatz erheblich höher als bei der konventionellen Lösung, bei der das Unterprogramm nur einmal aufgerufen wird. Der größte Nachteil der rekursiven Programmierung ist aber die Gefahr von Systemabstürzen, z.B. wenn das Programm aus Versehen einmal mit einer negativen Zahl für N aufgerufen werden würde.
Die Methode ist also weder effizient noch sicher (dafür aber „sehr elegant“ ☺).

4.6 Eltern-Kind Prozesse

Lassen Sie uns an dieser Stelle noch eine andere Art der segmentierten Programmierung besprechen, die eigentlich keine richtige Unterprogrammtechnik ist, und die nicht zum Standard-Basic gehört. Sie ist aber trotzdem gelegentlich sehr nützlich.

Wie Sie schon gesehen haben, kann man mit dem Befehl Run die Ausführung eines GPR-Basic Programmes starten. Sie können den Befehl Run als Anweisung auch in einem Basic-Programm verwenden, dann natürlich nicht, um das Basic-Programm selbst zu starten (denn es läuft ja schon), sondern um ein anderes Basic-Programm (abgespeichert in einer anderen Datei) an genau dieser Stelle auszuführen. Die Ausführung des aufrufenden Programmes (parent) wird dabei unterbrochen, bis die Ausführung des aufgerufenen Programmes (child) abgeschlossen worden ist. Dabei können im aufgerufenen Programm (lokale und globale) Variablen des aufrufenden Programmes verwendet werden (das „child“ kennt also seinen „parent“). Später, wenn das aufgerufene Programm beendet worden ist und Basic mit der Ausführung des aufrufenden Programmes weitermacht, können die Variablen des ausgeführten Programmes verwendet werden (der „parent“ kennt also auch das „child“).

Probieren wir dazu einmal ein Beispiel. Der sogenannte Julianische Kalender wird in der Astronomie oft verwendet, dabei werden die Tage fortlaufend gezählt und haben Nachkommastellen, welche die Zeit in UTC („coordinated universal time“) darstellen.
Das Julianische Datum (Jd) ist also ein Wert, der überall auf der Welt einen Zeitpunkt exakt darstellt. Es kann mit folgender Formel berechnet werden:

Jd = (367*jahr-INT(7*(jahr+INT((monat+9)/12))/4)+INT(monat*275/9)) + tag + 1721013.5

Machen Sie jetzt ein Basic Fenster auf, tippen sie diese Formel in den Programmbereich ein, und speichern Sie die Datei unter dem Namen „c:\GPR-Basic\jd.bas“ ab. Machen Sie jetzt ein weiteres Fenster auf und tippen sie in den Programmbereich folgende Zeilen ein:

```
Input tag, monat, jahr
Run "c:\GPR-Basic\jd.bas"
Print "Julianisches Datum",jd
end
```

Das Programm berechnet für die eingegebenen Daten den Julianischen Wert genauso, als ob statt `Run "c:\GPR-Basic\jd.bas"` in derselben Zeile die zugehörige Formel stehen würde. Run erwartet als Argument einen String, der den Namen des zu startenden Programmes enthält. Wenn dieser String eine Konstante ohne Leerzeichen ist, so kann man die Anführungszeichen auch weglassen. Vorausgesetzt, dass das aktuelle Arbeitsverzeichnis also C:\GPR-Basic ist, könnten wir auch vereinfacht `Run jd.bas` schreiben.

Natürlich können in der aufgerufenen Programmdatei statt einem auch mehrere (sehr viele) Programmschritte (und sogar Unterprogramme) stehen. Trotzdem ist diese Methode nur für ausgereifte Programmteile zu empfehlen: Bei Fehlerabbrüchen im „Child"-Prozess kehrt die Programmausführung nicht mehr ins aufrufende „Parent"-Programm zurück, und ein Neuladen dieses Programmes wird erforderlich.

Eine andere Methode einen „Kind"-Prozess auszuführen stellt der Befehl `Execute` dar.
Im Gegensatz zu Run führt Execute eine Basic-Datei aber nicht in demselben Fenster aus, sondern erzeugt dafür ein eigenes Fenster und startet dort das Programm. Da das aufgerufene Programm in einem eigenen Fenster abläuft, verfügt es über eigene (lokale und globale) Variablen. Information kann vom aufrufenden Programm zum aufgerufenen Programm also nicht fließen (es sei denn über externe Kanäle wie Dateien). Der Befehl Execute "Dateiname" ist ganz ähnlich wie der Befehl Edit "Dateiname"; nur, dass Edit den Programmbereich des erzeugten Fensters maximal vergrößert und das Programm nicht startet. Der Befehl ExecuteGrafic "Dateiname" startet ebenfalls eine Basic-Datei in einem neuen Fenster, wobei diesmal der Grafikbereich des Fensters maximal vergrößert wird.

Man könnte also meinen, dass mit Execute, Executegrafic oder Edit aufgerufene „Kind"-Prozesse ihren aufrufenden „Eltern" Prozessen gleichwertig gegenüberstehen; und dass GPR-Basic diese Prozesse sozusagen im „Multitasking-Betrieb" parallel ablaufen lassen kann. Das ist aber nicht der Fall. Das Aufrufen eines „Kind"-Prozesses unterbricht den „Eltern"-Prozess solange, bis der „Kind"-Prozess abgearbeitet (oder abgebrochen) worden ist. Danach kehrt die Programmausführung automatisch zum aufrufenden „Eltern"-Prozess zurück.
GPR-Basic kann zu einem Zeitpunkt immer nur ein Programm abarbeiten.

Wenn keine ausführbaren Befehle mehr vorgefunden werden, beendet sich das Programm. Explizit beenden kann man das Programm durch die Anweisung `END` (oder `STOP`), welche nicht nur am Ende des Hauptprogrammes sondern überall vorkommen kann. Ein Prozess hingegen wird entweder durch Anklicken der Schaltfläche × (ganz rechts oben im Fenster) oder aber im Programm durch die Anweisung `Terminate` beendet.

Befehle wie Run, Execute oder Edit gehören zu den fortschrittlichen Programmiertechniken und werden i.A. innerhalb von Basic-Programmen nur gelegentlich mal benötigt. Dasselbe gilt für die in GPR-Basic implementierten Befehle zur Datei- und Verzeichnisbearbeitung:

`Copy "Name1","Name2"`	kopiert die Datei *Name1* auf die Datei *Name2*.
`Rename "Name1","Name2"`	nennt die Datei *Name1* jetzt *Name2*.
`Del "Name"`	löscht die Datei *Name*.

`Cd "Name"`	wechselt das Arbeitsverzeichnis zu *Name*, „change directory";
`Md "Name"`	erzeugt ein Arbeitsverzeichnis *Name*, „make directory";
`Rd "Name"`	löscht das Arbeitsverzeichnis *Name*, „remove directory".

Diese Befehle können nicht nur im Protokollbereich eines GPR-Basic Fensters eingegeben werden, sondern sie stehen auch im Programmbereich zur Verfügung (wo sie aus verständlichen Gründen mit Vorsicht zu gebrauchen sind). Sie erwarten als Argumente Strings (Variablen oder Konstanten), welche natürlich auch vom Programm berechnet werden können.
Das Basic-Programm kann also z.B. den Namen einer Datei selbst erzeugen.

5 Dateibehandlung

5.1 Öffnen und Schließen von Dateien

Daten-Dateien (im Gegensatz zu Programm-Dateien) eignen sich sehr gut für die Ein- und Ausgabe von Information. Die Anweisungen dazu heißen Lineinput und Print. Bevor wir aber diese Anweisungen auf Daten-Dateien anwenden können, müssen wir die gewünschten Dateien „öffnen“. Dazu müssen wir uns die Anweisungen Open und Close ansehen.

Betrachten wir zunächst einmal das Protokollieren von Programmergebnissen. Die Anweisung Print kennen Sie ja schon, wir haben sie oft verwendet. Print gibt eine Liste von Ergebnissen arithmetischer Ausdrücke (getrennt durch Kommata) in jeweils eine Zeile des Protokollbereichs eines GPR-Basic Fensters aus und führt einen Zeilenvorschub durch (welcher durch Anhängen eines weiteren Kommas unterdrückt werden kann). Möchten wir nun ein Programm schreiben, das diese Ausgabedaten statt in den Protokollbereich des Fensters in eine Daten-Datei schreibt, so müssen wir vor der Verwendung der Print Anweisungen die gewünschte Ausgabedatei im Programm mit einer Open Anweisung „öffnen“. Dann erst kann das Programm durch Print Anweisungen in diese Datei schreiben. Erfolgen im Programmverlauf keine weiteren Schreiboperationen mehr auf diese Datei, so sollte sie mit der Close Anweisung wieder „geschlossen“ werden. Die Syntax dazu lautet:

```
Open Dateiname For Zugriffsart As #Dateinummer
      .
      .
      .
      .
      .
Close #Dateinummer
```

Die Punkte stehen dabei wieder für viele Anweisungen, die nun Aus- oder Eingabeoperationen in der erlaubten *Zugriffsart* auf die Datei namens *Dateiname* enthalten können. *Dateiname* ist ein String oder ein String-Ausdruck, der den Namen der zu öffnenden Datei beinhaltet; *Zugriffsart* ist keine Variable sondern ein Schlüsselwort der Art:

`Input`	nur zum Lesen, Schreibbefehle auf die Datei nicht möglich,
`Output`	zum Scheiben aber alles, was in der Datei vorher drin stand wird gelöscht,
`Append`	zum Schreiben, aber das neu Geschriebene wird nur angehängt.

Die Zugriffsart festzulegen ist ein sinnvoller Schutzmechanismus: Wenn man versehentlich auf eine zur Dateneingabe (Lesen oder Input) geöffnete Datei eine Schreiboperation (Print) anwendet, so führt das zu einer Fehlermeldung und nicht zur Zerstörung der Datei.

Für *Dateinummer* muss eine Nummer von 1 bis 100 angegeben werden (das Zeichen # ist dabei ein Steuerzeichen, welches in der Open und in der Close Anweisung auch weggelassen werden kann). Mit dieser „Kanalnummer“ nehmen die Schreib- und Leseoperationen Bezug auf die geöffnete Datei.

Die Open Anweisung verbindet nun die Dateinummer mit dem Dateinamen und setzt den Dateizeiger auf das erste Zeichen in der Datei (außer, wenn die Datei mit der Zugriffsart *Append* geöffnet wurde). Eine Print Anweisung würde also hier anfangen zu schreiben, und eine LineInput Anweisung würde hier anfangen zu lesen. Print, z.B., hängt i.A. Zeilenvorschubzeichen hinten an (CR und LF), und die LineInput Anweisung liest bis zu den nächsten Zeilenvorschubszeichen in der Datei. Diese Operationen schieben den Dateizeiger also vor bis zur nächsten Zeile. Wenn man also eine Datei mit mehreren Zeilen schreiben oder lesen möchte, so darf man um Himmels Willen nicht vor jeden Zugriff eine Open Anweisung setzten, sonst würde man jeweils immer nur die erste Zeile der Datei bearbeiten.

Man lässt die Datei also „offen“ bis alle Schreib- (oder Lese) Vorgänge durchgeführt worden sind. Dann erst schließt man die Datei mit Close. Die Anweisung Close gibt die Dateinummer wieder frei (d.h. dieselbe Zahl kann dann z.B. für das Öffnen einer anderen Datei wiederverwendet werden), sie schreibt alle eventuell noch im Zwischenspeicher (cache) stehenden Daten auf die Platte, und sie aktualisiert die Datei-Information auf dem Laufwerk (Erstellungsdatum, wann geändert, wie viele Bytes etc.).

5.2 Ausgabe in eine Daten-Datei

Die Ausgabe in eine geöffnete Datei geschieht mit der Print-Anweisung, und zwar fast genauso, als wollte man in den Protokollbereich ausgeben. Der Unterschied ist lediglich das Kontrollzeichen # direkt gefolgt von der Dateinummer und abgetrennt durch ein Komma, welches der Ausgabeliste vorangestellt werden muss. Wollen wir also z.B. einmal die Summe aller Natürlichen Zahlen für Endwerte von ein bis hundert in die Datei „ENDSUMME.DAT“ schreiben, so können wir folgendermaßen vorgehen:

```
Open "endsumme.dat" For Output As #1
Summe=0
For i = 1 to 100
     Summe=Summe+i
     Print #1, "Für Wert:",i,"ergibt sich:",Summe
Next i
Close #1
End
```

Wir sollten `End` an das Programm anhängen, damit wir die Meldung „End“ kriegen, wenn das Programm mit dem Datei-Schreiben fertig ist. Wenn wir uns die fertige Datei ansehen wollen, können wir einfach in den Protokollbereich tippen:

```
edit endsumme.dat
```

Da die Ausgabedatei „ENDSUMME.DAT“ vor Programmlauf noch nicht existierte, so wurde sie durch die Open Anweisung angelegt (hätte sie bereits existiert, so wäre sie nicht angelegt sondern ohne Warnung überschrieben worden). Die Datei wurde mit der Dateinummer 1 geöffnet (das Steuerzeichen # kann in der Open Anweisung auch weggelassen werden).

Die Argumente (oder Parameter) der Print Anweisung beginnen mit #1 und lenken so die Ausgabe vom Protokollbereich in die Datei um. Würden wir hier das Steuerzeichen weglassen, so würde 1 eiskalt als arithmetischer Ausdruck interpretiert werden, und damit als Wert in den Protokollbereich des Fensters ausgegeben werden.

Fast immer möchte man die Ausgabewerte formatiert in die Ausgabedatei schreiben. Die dazu erforderliche Funktion heißt Format. Wir haben sie ja schon kennengelernt, sollten sie uns hier an dieser Stelle aber noch einmal genauer anschauen. Format wandelt den Wert einer Rechenvariablen (bzw. eines arithmetischen Ausdrucks) benutzerdefiniert in einen String um.
Die Funktion benötigt dazu zwei Parameter: erstens den umzuwandelnden Ausdruck und zweitens einen „Formatstring“. Die Zeichen in diesem Formatstring sind Platzhalter für Ziffern und für den Dezimalpunkt (und, eventuell, für den Exponenten im Falle der Ausgabe im wissenschaftlichen Format). In GPR-Basic sind überhaupt nur fünf Zeichen für den Formatstring zugelassen, nämlich Platzhalter für:

`#`	eine Ziffer (führende oder nachlaufende Nullen werden unterdrückt)
`0`	ebenfalls eine Ziffer (nachlaufende Nullen werden nicht unterdrückt)
`.`	den Dezimalpunkt
`E`	E im Exponentialformat (+ und – im Exponenten werden ausgegeben)
`e`	E im Exponentialformat (nur – im Exponenten wird ausgegeben).

Probieren wir einmal ein paar Beispiele: Für eine negative Zahl z.B. a = –.5 wird ausgegeben:

```
a=-0.5
Print Format(a,"###.###")          ' ergibt: -.5
Print Format(a,"#.#")              ' ergibt: -.5
Print Format(a,"###.000")          ' ergibt: -.500
Print Format(a,"000000.000")       ' ergibt:     -0.500
Print Format(a,"#.###E##")         ' ergibt: -5.e-1
Print Format(a,"#.###e##")         ' ergibt: -5.e-1
Print Format(a,"#.000E00")         ' ergibt: -5.000e-01
Print Format(a,"#.000e00")         ' ergibt: -5.000e-01
```

Ist dagegen a = +500 wird ausgegeben:

```
a=+500
Print Format(a,"###.###")          ' ergibt: 500.
Print Format(a,"#.#")              ' ergibt: 500.
Print Format(a,"##0.000")          ' ergibt: 500.000
Print Format(a,"#.###E##")         ' ergibt: 5.00e+2
Print Format(a,"#.###e##")         ' ergibt: 5.00e2
Print Format(a,"#.000E00")         ' ergibt: 5.000e+02
Print Format(a,"#.000e00")         ' ergibt: 5.000e02
```

Passt die Zahl nicht in das spezifizierte Format herein, z.B.: `Format(12345,"#.#"),` so ignoriert die Format Funktion die Spezifikation weitgehend und gibt den Wert trotzdem richtig aus (der im Beispiel ausgegebene Wert hat dann natürlich mehr als drei Stellen). Natürlich sollte man aber bestrebt sein, für die Ausgabe genügend Platzhalter bereitzustellen. Besser wäre also, für das obige Beispiel: `Format(12345,"#####.#")` zu schreiben.

Wir wollen dazu abschließend unter Verwendung der Format-Funktion in der Print Anweisung das Programm oben zur Erzeugung der Datei „ENDSUMME.DAT“ so umschreiben, dass es alle Ausgabezeilen nun schön geordnet mit richtig positionierten Werten untereinander schreibt. Da die Print Anweisung durch die zwei zusätzlichen Format Funktionen unpraktisch lang werden würde, machen wir vom Zeilenfortsetzungszeichen Gebrauch (der Underscore _ über dem Minuszeichen auf der deutschen Tastatur). Alternativ hätten wir natürlich auch zwei Print Anweisungen schreiben können, und bei der ersten den Zeilenvorschub durch ein angehängtes Komma unterdrücken können. Also:

```
Open "endsumme.dat" For Output As #1
Summe=0
For i = 1 to 100
     Summe=Summe+i
     Print #1, "Für Wert:",Format(i,"0000"), _
      " ergibt sich:",Format(Summe,"00000")
Next i
Close #1
Edit "endsumme.dat"
Pause
End
```

Besonders dann wenn wir später wieder auf diese Datei mit Leseoperationen durch ein anderes Basic-Programm zugreifen wollen, ist richtiges Formatieren bei der Ausgabe wesentlich.

5.3 Lesen aus einer Daten-Datei

Oft benötigen Programme sehr viele Eingabedaten, und dann wird es einem verständlicherweise schnell zu mühselig, diese immer wieder bei Programmstart neu eintippen zu müssen. Empfehlenswert wird dann die Verwendung einer Eingabe-Datendatei.

Probieren wir das Datei-Lesen vielleicht aber mal am Beispiel der Datei Endsumme.dat aus, denn die haben wir ja schon (und brauchen sie nicht erst anzulegen). Es könnte ja sein, dass man vielleicht das Summenergebnis für 50 wissen möchte, ohne es berechnen zu wollen. Dann könnte man ja in der Datei Endsumme.dat mal nachschauen. Schreiben wir dazu ein Programm:

```
Open "endsumme.dat" for Input as #1
For i = 1 to 50
     LineInput #1,a$
Next i
Close #1
Print a$
End
```

Das Programm liest 50-mal jeweils aufeinanderfolgende Zeilen der Datei und speichert die gelesenen Zeilen jedes Mal als Zeichenkette in der Variablen `a$` ab, wobei diese andauernd wieder überschrieben wird. Schließlich, nach 50-mal lesen hat die Variable aber den Wert: `Für Wert:  50 ergibt sich:  1275` Die For-Next Schleife wird dann beendet, die Datei „endsumme.dat“ wird geschlossen und der Inhalt der Variablen `a$` wird in den Protokollbereich des Fensters geschrieben.

Die Anweisung LineInput benötigt hierbei zwei Parameter: erstens die Dateinummer der Datei, auf welche zugegriffen werden soll (diese muss vorher mit Open in der Zugriffsart Input mit einer existierenden Datei verbunden worden sein); zweitens eine Stringvariable, welcher die aktuelle Zeile zugewiesen werden soll. Möchte man also die fünfzigste Zeile einer Datei lesen, so muss man 50 mal lesen, denn es gibt keine Anweisung, mit welcher der Dateizeiger direkt auf eine bestimmte Zeile positioniert werden kann.

Wenn man nun Zahlenwerte aus der Datei lesen und im Programm verwenden möchte, so muss man die Stringvariable, welche die gelesene Zeile enthält, sinngemäß zerlegen und dann die entstandenen „Sub-Zeichenketten“ in Zahlenwerte transformieren. Dies geschieht mit den String-Funktionen Mid und Val, die wir ja schon kennengelernt haben.

Wir könnten einmal die Aufgabe stellen, dass ein Programm unter Verwendung der Datei „Endsumme.dat“ die kleinste Natürliche Zahl herausbekommen soll, für welche der Summenwert bereits größer als 1000 ist. So könnte man es machen:

```
Open "endsumme.dat" for Input as #1
For i = 1 to 100
     LineInput #1,a$
     N=Val(Mid(a$,12,3))
     Summe=Val(Mid(a$,30,5))
     If Summe>1000 Then Exit For
Next i
Close #1
Print "Ergebnis= ",N
End
```

Das Programm findet: `Ergebnis = 45.` Beachten Sie, dass die Funktion Mid(A$, I, L) aus einem String A$ (erstes Argument) einen Sub-String herausschneidet, beginnend an der Position I im String A$ (zweites Argument) und L Zeichen lang (drittes Argument). Die Funktion Val(B$) wandelt einen als Zeichenkette definierten Zahlenwert (im String B$) in einen Rechenwert (Double) um. Das Programm oben berechnet also den Wert der Variablen `N` für die aktuelle Natürliche Zahl und den Wert der Variablen `Summe` für den zugehörigen Summenwert. Der Summenwert wird dann mit 1000 verglichen. Wenn er größer als 1000 ist, so wird die For-Next Schleife beendet.

Daten-Dateien können natürlich auch als Aus- und Eingabekanäle zur Datenübertragung zwischen verschiedenen Programmen (oder Prozessen) benutzt werden. Man kann solche Dateien aber auch dazu benutzen, um Information zur Konfiguration von Programmen zu hinterlegen (sogenannte INI-Dateien). Wesentlich ist dabei immer, dass das Programm die Struktur der zu lesenden Datei genau kennt (bzw. genau analysiert), andernfalls führen falsch gerichtete Lesebefehle leicht zu Laufzeitfehlern.

Abschließend dazu sei noch die GPR-Basic Funktion `A$ = FILE("Dateiname")` erwähnt, die eine komplette Textdatei „Dateiname“ direkt in einen String `A$` rein lädt. Diese Funktion ist gelegentlich sehr nützlich, gehört aber nicht zu den Standard Basic-Funktionen.

6 Indizierte Variablen

6.1 Dimensionierung von indizierten Variablen

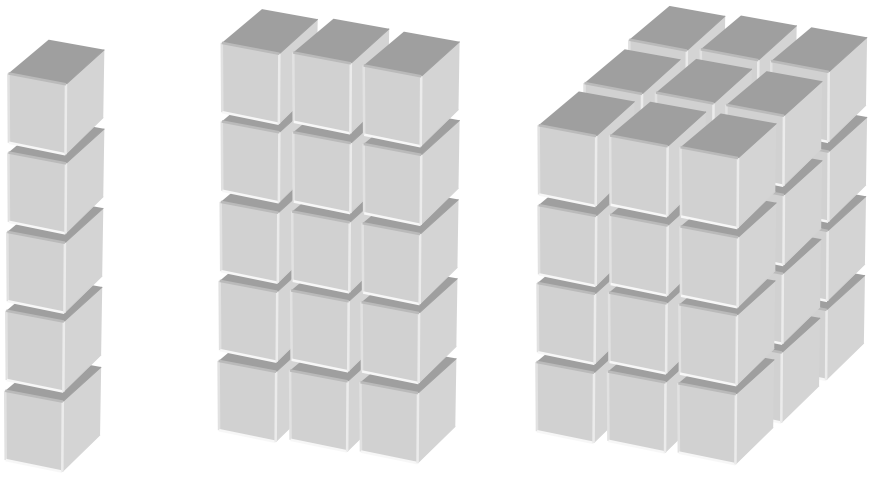
Meist eröffnet erst die Verwendung von indizierten Variablen in einem Programm die Möglichkeit ein kompliziertes Problem zu lösen. Dabei weist der Variablenname nicht zu einer einzigen Variablen, sondern gleich zu einer ganzen Gruppe von Variablen mit demselben Namen. Die einzelnen Gruppenmitglieder werden durch ihre Indizes unterschieden, wobei in GPR-Basic bis zu maximal drei Indizes möglich sind.
Bevor man aber mit indizierten Variablen arbeiten kann, muss man sie dimensionieren. Dies geschieht mit der Anweisung Dim (oder analog mit Local bzw. Global, wenn man lokale oder globale Variablen dimensionieren möchte). Dimensionieren wir also zum Beispiel einmal:

```
Dim A(5) as Double, B(3,5) as Long, C(3,4,3) as String
```

Diese Anweisung reserviert Speicherplätze für drei indizierte Variablen: 5 Plätze vom Typ Double für die eindimensionale Variable A, 15 vom Typ Long für die zweidimensionale Variable B, und 36 vom Typ String für die Variable C (eigentlich sogar 6 Speicherplätze für A, 24 für B und 80 für C, denn der Index 0 wird automatisch mit-initialisiert). Wir können diesem reservierten Speicherplatz bei der Initialisierung mit der Dim Anweisung auch gleich Werte zuweisen, aber jeweils nur einen Wert für jeden Variablennamen, z.B.:

```
Dim A(5)=1.23, B(3,5)=2 as Long, C(3,4,3)=" " as String
```

Alle 5 (bzw. 6) Elemente von `A` haben dann den Wert 1.23 (`as Double` braucht nicht angegeben zu werden, denn alle nicht anders spezifizierten Variablen sind ja sowieso Double); alle Elemente von `B` haben den Wert 2 und alle Elemente von `C` sind Leerzeichen.
Wir können uns die Speicherplätze dieser Variablen etwa so vorstellen:



Eindimensional als „Vektor", zweidimensional als „Feld" und dreidimensional als „Raum".
Ob im mittleren obigen Beispiel das Feld `B` nun drei Zeilen und fünf Spalten hat oder fünf Zeilen und drei Spalten, ist Geschmacksache.

Nur unsere Vorstellung entscheidet, ob der erste Index die Zeile bestimmt und die zweite die Spalte oder umgekehrt (denn die grafische Abbildung existiert nur in unserer Vorstellung). Der Computer verlässt sich ausschließlich auf die Reihenfolge der Indizes. Wenn in einem Programm die obige DIM Anweisung ausgeführt worden ist, so gibt es in diesem Programm dann die folgenden Speicherplätze:

A(0), A(1), A(2), A(3), A(4) und A(5) (alle Double)

B(0,0)..., B(1,1), B(1,2), B(1,3), B(1,4), B(1,5),
B(2,1), B(2,2),.., B(2,5), B(3,1), B(3,2),.., B(3,5) (alle Long)

C(0,0,0)..... , C(1,1,1), C(1,1,2), C(1,1,3), C(2,1,1),...C(2,1,3),
C(3,1,1)..... C(3,4,3) (alle String)

Komplexität und Speicherplatzbedarf steigen mit der Zahl der Indizes überproportional an.

6.2 Arbeiten mit indizierten Variablen

Für die Berechnung des zugehörigen Speicherplatzes im Computer sind nur Reihenfolge und Wert der Indizes maßgeblich. Also, z.B. `B(2,3)` bezeichnet eindeutig ein ganz bestimmten Speicherplatz, in unserem obigen Beispiel deklariert als vom Typ Long. Wie bei einer ganz normalen Variablen können wir nun einen Wert dort reinschreiben, z.B. den Wert 100. Und dieser Wert kann dann, ebenfalls wie der Wert einer ganz normalen Variablen, in darauffolgenden Anweisungen verwendet werden. Also z.B.:

```
B(2,3)=100
...
If B(2,3)>99 then Print B(2,3)
```

Das ist aber bislang noch kein großer Vorteil im Vergleich zu ganz normalen Variablen, denn das hätten wir mit einer Variablen namens B23 zum Beispiel auch machen können.

Der große Unterschied bei indizierten Variablen ist aber: Man kann die Werte der Indizes auch vom Programm berechnen lassen. An den Stellen, wo im obigen Beispiel also 2 und 3 stehen, könnten überall auch arithmetische Ausdrücke stehen. Die Ergebniswerte dieser arithmetischen Ausdrücke müssen natürlich (Integer-) Werte im zulässigen (dimensionierten) Bereich sein (Index-Grenzen zu überschreiten ist ein typischer Programmierfehler). Statt `B(2,3)=100` hätten wir also z.B. auch schreiben können:

```
I=2
B(I,I+1)=100
...
If B(2,3)>99 then Print B(2,3)
```

Die große Mächtigkeit des Arbeitens mit indizierten Variablen besteht darin, dass man während des Programmlaufes die aktuellen Werte der Indizes berechnen kann.

Dazu wollen wir ein paar Beispiele machen. Das folgende Programm dimensioniert ein Feld A mit genau 100 Elementen (ignorieren wir mal die Null-Indizes), besetzt dieses Feld mit aufsteigenden Nummern und gibt diese zeilenweise im Protokollbereich des Fensters aus. Dazu werden zweimal zwei ineinander geschachtelte For-Next Schleifen verwendet:

```
Dim a(10,10) as Long
k=0
For i=1 to 10
      For j=1 to 10
            k=k+1
            A(i,j)=k
      Next j
Next i
For i=1 to 10
      For j=1 to 10
            Print Format(A(i,j),"000"),
      Next j
      Print
Next i
```

(Natürlich könnte man dieses Ergebnis auch ohne die Verwendung eines Variablenfeldes erzeugen, aber die Verwendung von Variablenfeldern zu zeigen ist ja gerade der Sinn dieses Beispiels.)

Nach der Ausgabe bleibt das Feld natürlich erhalten, und wir können weiter damit arbeiten. Zum Beispiel, wir könnten es einmal als Matrix betrachten und Transponieren (Transponieren heißt Zeilen mit Spalten vertauschen). Folgender Code, angehängt an des obige Programm würde dies bewerkstelligen:

```
For i=1 to 9
      For j=i+1 to 10
            C=A(j,i)
            A(j,i)=A(i,j)
            A(i,j)=C
      Next j
Next i
For i=1 to 10
      For j=1 to 10
            Print Format(A(i,j),"000"),
      Next j
      Print
Next i
```

Beachten Sie, dass die Feldelemente hier „zyklisch“ vertauscht werden, dazu wird die Hilfsvariable `C` benötigt. Der Inhalt vom Feldelement `A(j,i)` wird erstmal in `C` gesichert, dann wird `A(j,i)` mit `A(i,j)` überschrieben, und schließlich wird das Element `A(i,j)` mit dem in `C` gesichertem, vorherigen Inhalt von `A(j,i)` überschrieben.

Die Aufgabe mag immer noch als nicht besonders sinnvoll erscheinen (sondern als konstruiert für das Arbeiten mit „Feldern“). Deswegen stellen wir jetzt einmal eine andere anspruchsvolle Aufgabe, die sich ohne die Verwendung von Index-Variablen wohl nicht lösen lassen würde.

Betrachten wir einmal die Aufgabe als gestellt, die Eulersche Zahl e mit 100 Nachkommastellen Genauigkeit berechnen zu müssen. Die Formelsammlung sagt uns, dass e sich z.B. über die unendliche Reihe 1 + 1/1! + 1/2! + 1/3! + 1/4! + 1/5! + ... darstellen lässt.
Die folgende Rechenvorschrift („Algorithmus“) löst also das Problem:

```
A=1
E=1
For i=1 to 10000
      A=A/i
      E=E+A
      If A<0.0000000000001 then Exit For
Next i
Print "E=", E
```

Dabei simuliert `For i=1 to 10000` eine Art „Endlosschleife“, die mit `Exit For` bei erreichter Genauigkeit abgebrochen wird. Wenn wir dieses Programm eintippen und rechnen, liefert es natürlich keine 100 Nachkommastellen Genauigkeit (sondern nur vielleicht 14 bei der Verwendung von „Double“-Genauigkeit).

Wir müssen das Programm also abändern und statt der beiden Variablen A und E „Vektoren“ (eindimensionale indizierte Variablen) verwenden. Wir können z.B. am Programmanfang die Anweisung `Dim a(100) as Long, e(100) as Long` schreiben und so Speicherplatz für neue indizierte Variablen festlegen. Die 100 Nachkommastellen der Werte A und E in der oben stehenden Rechenvorschrift können dann in den Elementen dieser neuen indizierten Variablen abgespeichert werden. Zum Beispiel, das Element `a(0)` kann dann die erste (und einzige) Stelle der Variablen A vor dem Komma enthalten, `a(1)` die erste Nachkommastelle, `a(2)` die zweite Nachkommastelle etc.; und für die Variable E gilt sinngemäß das Gleiche.

Wenn wir die Variablen A und E jetzt durch „Vektoren“ `a(100)` und `e(100)` ersetzen, müssen wir fast alle Anweisungen des Programmes umschreiben. Für die ersten beiden Anweisungen (`A=1, E=1`) und für die letzte Print Anweisung ist das noch vergleichsweise einfach. Wir öffnen am besten ein neues Fenster (dasselbe Fenster zu verwenden ist unverträglich, denn wir haben es mit neuen indizierten Variablen `a` und `e` zu tun) und schreiben:

```
Dim a(100)=0 as Long, e(100)=0 as Long
a(0)=1
e(0)=1


Print "E=";
For j=0 to 100
      Print Format(e(j),"0");
      If j=0 then Print ".";
Next j
Print
End
```

Die Elemente der Vektoren `a(100)` und `e(100)` werden durch die Dim Anweisung mit Nullen vorbesetzt; `a(0)` und `e(0)` ändern wir auf 1, damit es heißt: 1.0000000000000000000.....

Für die Ausgabe verwenden wir eine For-Next Schleife. Das Listentrennzeichen am Ende der Print-Anweisung ist jetzt ein Semikolon; es hat die gleiche Wirkung wie ein Komma, nur, das jetzt die Ausgabe eines Leerzeichens zwischen den Elementen der Ausgabeliste unterdrückt wird. Das Semikolon am Ende der Print Anweisung bewirkt, das der Zeilenvorschub unterdrückt wird (alle Zeichen werden also in derselben Zeile ausgegeben). Wir brauchen also noch eine zusätzlich `Print` Anweisung, um am Ende der Ausgabeliste einen Zeilenvorschub anzuhängen (die Zeichen CR und LF). Starten wir das Programm, so gib es jetzt noch aus:
`E=1.000000000000000000000000000000000000000.....`

Nun müssen wir die Anweisungen innerhalb der For-Next Schleife des Iterations-Algorithmus korrekt nachprogrammieren. Beginnen wir mit der Anweisung `A=A/i`: Der Inhalt von `A` soll durch `i` (einen Integer Wert) dividiert werden, und danach soll das Ergebnis wieder in `A` drinstehen. Zum Beispiel, im dritten Durchlauf der Schleife ist A=0.5 und soll durch 3 geteilt werden. Wir verwenden für `A` einen Vektor vom Typ Long, der die Nachkommazahlen der Zahl beinhaltet. Wenn `A` also 0.5 ist, so sind `a(0)=0`, `a(1)=5`, `a(2)=0`, `a(3)=0` etc. Also 0.50000 durch 3 (und das ohne Taschenrechner, oweiowei ☺)! Wissen Sie noch wie's geht?

```
0.5000000000000.....   /  3  = .166...
 .3
 .20
  18
   20
    18
     ..
```

Erstmal teilen wir den ersten Betrag (die Ziffer 5) durch 3, bilden den Integer Wert und notieren das Ergebnis (ergibt 1). Danach rechnen wir 1 mal 3 (ergibt 3), die 3 schreiben wir unter den Betrag (die 5) und ziehen sie ab (ergibt den Rest 2). Den Rest multiplizieren wir mit 10 und addieren darauf die Ziffer in der nächsten Stelle (die 0 nach der 5). Ergibt den neuen Betrag 20. Diesen teilen wir ganzzahlig durch 3, notieren Ergebnis (6) und den Rest (2), multiplizieren den Rest mit 10 (wieder 20), addieren die nächste Stelle (wieder 0) etc. Ist doch eigentlich ganz einfach. Programmieren wir es:

```
Rest=0
For j=0 to 100
      Betrag=Rest*10+a(j)
      Ergebnis=Int(Betrag/i)
      Rest=Betrag-Ergebnis*i
      a(j)=Ergebnis
Next j
```

Wenn Sie diesen Code testen möchten und ihn dazu vor die `Print "E=";` Anweisung im Programmfenster einfügen, bedenken Sie, dass Sie der Variablen `i` einen Wert zuweisen müssen (z.B.: `i=3`). Außerdem müssen dann natürlich die Ausgabe ändern und statt Werten für `e` die Werte für `a` ausgeben.

Vergessen Sie nicht, diese Änderungen später wieder rückgängig zu machen.

Kommen wir nun zu der Anweisung E=E+A: Der Inhalt von A soll auf den Inhalt von E draufaddiert werden, und danach soll das Ergebnis wieder in E drinstehen. Die Rechenvorschrift dazu ist vergleichsweise einfach. Beim dritten Schleifendurchlauf haben wir z.B.:

```
2.50000000000.....00000
0.16666666666.....66666
                      6
```

Man schreibt beide Zahlen e und a untereinander, fängt die Ziffern von hinten an zu addieren. Ist das Ergebnis größer oder gleich 10 (in diesem Beispiel ist das aber nicht der Fall), so zieht man 10 ab und addiert beim nächsten Mal den Wert 1 als Übertrag für die nächst höhere Stelle dazu. Das Ergebnis schreibt man dann wieder in die Variable e rein. Also:

```
Uebertrag=0
For j=100 to 0 Step -1
      e(j)=e(j)+a(j)+Uebertrag
      If e(j) >= 10 then
            e(j)=e(j)-10
            Uebertrag=1
      Else
            Uebertrag=0
      Endif
Next j
```

Schließlich brauchen wir die Überprüfung, ob die Rechnung noch fortgesetzt werden muss oder nicht (entsprechend der Anweisung `If A<0.000000000001 Then Exit For`).

Der Algorithmus ist fertig, wenn die indizierte Variable a nichts anders mehr als Nullen enthält. Wir können das zum Beispiel so programmieren:

```
Schalter=0
For j=100 to 0 step –1
      if a(j) >< 0 then Schalter=1 : Exit For
Next j
If schalter=0 then Goto Ausgabe
```

Der bedingte Sprung auf die Sprungmarke `Ausgabe` (die wir vor die Print-Anweisung einzufügen haben: `Ausgabe: Print "E=";`) erfolgt nur dann, wenn alle Zahlenwerte (Elemente) der indizierten Variablen a gleich Null sind. Nur dann behält die Variable `Schalter` ihren Wert 0 (andernfalls wird sie 1 gesetzt). Mit der Überprüfung der Elemente von a fangen wir aus Gründen der schnelleren Rechnung am besten hinten an.

Alle Segmente des Programmes sind nun fertig, und wir können das Gesamt-Programm zusammensetzen. Da es recht langsam rechnet, ist es zweckmäßig, die Stellenzahl variabel zu halten (z.B. mit einer Variablen `Stellenzahl`, die durch eine Input-Anweisung eingelesen werden kann). Außerdem sollte eine Print Anweisung in der Haupt For-Next Schleife Auskunft über den Fortschritt des Programm-Ablaufs geben.

Das Programm zur Berechnung der Eulerzahl mit 100 Stellen Genauigkeit lautet dann:

```
Dim a(1000)=0 as Long, e(1000)=0 as long
e(0)=1
a(0)=1

Input Stellenzahl

For i=1 to 10000
      Print i
      Rest=0
      For j=0 to Stellenzahl
            Betrag=Rest*10+a(j)
            Ergebnis=Int(Betrag/i)
            Rest=Betrag-Ergebnis*i
            a(j)=Ergebnis
      Next j
      Uebertrag=0
      For j=Stellenzahl to 0 step -1
            e(j)=e(j)+a(j)+Uebertrag
            If e(j) >= 10 then
                  e(j)=e(j)-10
                  Uebertrag=1
            Else
                  Uebertrag=0
            Endif
      Next j
      Schalter=0
      For j=Stellenzahl to 0 step –1
            If a(j) >< 0 then Schalter=1 : Exit For
      Next j
      If schalter=0 then Goto Ausgabe
Next i

Ausgabe: Print "E=";
For j=0 to Stellenzahl
      Print Format(e(j),"0");
      If j=0 then Print ".";
Next j
Print
Print "Zum Vergleich:",Exp(1)
```

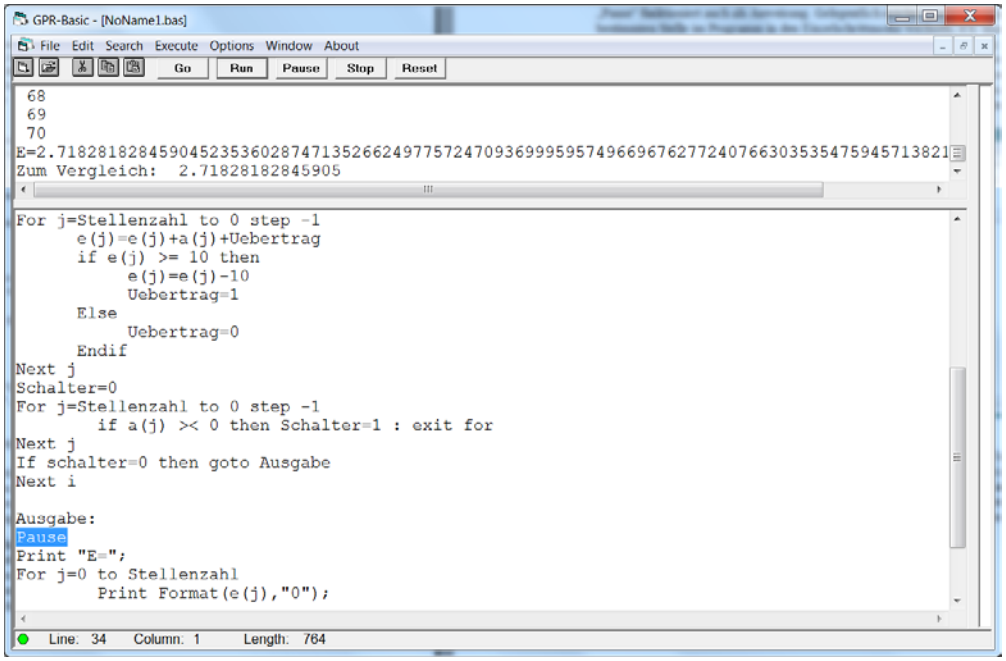
Bleibt noch zu erwähnen, dass die letzten Stellen des Ergebnisses nicht ganz genau sind: Die danach folgenden Ziffern wurden ja einfach abgeschnitten; und hätten wir sie addiert, so hätte sich daraus schon ein Wert ergeben, den wir hätten berücksichtigen sollen (oder muss das heißen: den wir berücksichtigt haben sollten? Oder vielleicht: den wir hätten berücksichtigt haben sollen? Fragen Sie mal Ausländer ☺) .

Das Programm oben ist wohl selbst für erfahrene Programmierer etwas zu komplex, als das man es auf einmal hätte „runterschreiben“ können. Die eigentliche Kunst besteht darin, dass man es erst blockweise schreiben und dann zusammensetzen muss.

Diese Blöcke müssen natürlich dabei einzeln getestet werden, wobei bei GPR-Basic die Möglichkeit der Einzelschritt-Abarbeitung mit der Schaltfläche „Go“ eine große Hilfe sein kann.

Möchte man während der Programmausführung mit „Run“ in den Einzelschrittmodus wechseln, so drückt man einfach die Schaltfläche „Pause“. Die automatische Abarbeitung wird unterbrochen, und man kann durch wiederholtes Drücken der „Go“ Taste das Programm im Einzelschrittmodus fortsetzen. Drücken der Schaltfläche „Run“ dagegen wechselt wieder zur automatischen Programmausführung zurück. „Stop“ beendet die Programmausführung.

„Pause“ funktioniert auch als Anweisung. Gelegentlich möchte man an einer ganz bestimmten Stelle im Programm in den Einzelschrittmodus wechseln, d.h. man möchte einen „Breakpoint“ setzen, um das Programm zu "debuggen". Die Anweisung PAUSE schaltet während der Programmausführung zum Einzelschrittmodus, die Taste „Run“ schaltet zur automatischen Ausführung zurück. Seit einmal in einem Röhrencomputer ein durch einen Käfer verursachter Kurzschluss als Fehlergrund gefunden worden ist, werden Programme debugged, „entlaust“ (der Käfer hat den Kurzschluss nicht überlebt ☹).



Einen wichtigen Unterschied zwischen indizierten Variablen und normalen Variablen gibt es noch zu erwähnen: Indizierte Variablen können auch in Unterprogrammen als lokale Variablen dimensioniert werden, aber sie können nicht in der Parameterliste an Unterprogramme übergeben werden.

Verwenden Sie zur Dimensionierung von Vektoren oder Feldern die Anweisung Global, wenn sie diese auch in Unterprogrammen verwenden wollen.

7 Grafik

7.1 Die Grafik Werkzeugsammlung

Das Programmiersystem GPR-Basic ist im Wesentlichen zum Erlernen der Programmiersprache Basic gedacht (es ist kein Grafik-Programm); und so beschränken sich die Grafik Befehle natürlich nur auf das Notwendigste. Außerdem gehört Grafik nicht unbedingt zum Standard Befehlssatz von Basic. Wir haben ja schon einige Male mit dem Grafik-Bereich eines GPR-Basic Fensters gearbeitet. Dieser Bereich speichert eine Grafik als speicher-residente Bitmap ab. Mit dem Untermenü-Punkt Grafic-Copy (im Menü Edit) können Sie die Bitmap in die Zwischenablage (Clipboard) kopieren, wenn Sie die erzeugte Grafik in anderen Programmen verwenden wollen. Mit Grafic-Paste können Sie eine Bitmap aus der Zwischenablage als Hintergrundbild in den Grafikbereich eines GPR-Basic Fensters laden. Grafic-Delete hat dieselbe Funktion wie die Anweisung CLS und löscht die von Programm in den Grafik-Bereich geschriebene Information.

Voreingestellt als Ursprung des Koordinatensystems ist die linke obere Ecke des Grafikbereichs; die x-Koordinate geht nach rechts, die y-Koordinate nach unten. Auf dem Bildschirm ist cm die passendste Einheit. Man kann diese Voreinstellung durch die Anweisung

```
DRAW Coordinates (x,y),xscale,yscale
```

ändern, z.B. die Anweisung `Draw Coordinates (7,7)` bewirkt, dass der Koordinaten-Ursprung jetzt etwa in der Mitte des Grafikbereichs liegt. Die Skalierungs-Faktoren sind optional, müssen also nicht angegeben werden, sind aber gelegentlich ganz nützlich, zum Beispiel wenn man z.B. will, dass die y-Koordinate nach oben zeigt:

```
Draw Coordinates (7,7),1,-1
```

DRAW ist eine wichtige Grafik-Anweisung, sie zeichnet aber nicht, sondern sie stellt die Grafik Werkzeuge nur ein. DRAW wird erst gefolgt von einem Schlüsselwort und dann dem zugehörigen Wert:

`DRAW Scale X`	`X` kann sein cm (voreingestellt), mm, inch oder pixel;
`DRAW Width X`	`X` bezeichnet die Breite des Zeichenstiftes in pixels;
`DRAW Color X`	`X` kann sein black (voreingestellt), eine andere Farbe oder die RGB-Funktion;
`DRAW Fillcolor X`	`X` kann sein transparent (voreingestellt) oder eine Farbe;
`DRAW Linestyle X`	`X` ist normal (voreingestellt), dash, dot, dashdot oder dashdotdot;
`DRAW Textfont A$`	`A$` ist ein String, der die Schriftart kennzeichnet;
`DRAW Textstyle X`	`X` kann sein normal (voreingestellt), bold, italic, oder underline;
`DRAW Textsize X`	`X` bezeichnet die Schriftgröße (in Punkte).

Bei GPR-Basic sind eine ganze Reihe an Farben vordefiniert, so gibt es außer black and white noch grey, red, green, blue, yellow, cyan, magenta, pink, brown, orange und violet.

Bei allen diesen Farben kann man auch light oder dark anwählen, es gibt also auch z.B. lightblue oder darkred. Zudem gib es noch die Farben aquamarin, azure, rosy, beige, olive, amber, salmon, coral und purple (ohne light oder dark). Wem das nicht reicht, der kann sich seine Farbe mit der RGB-Funktion selbst zusammenmischen.

Schauen wir uns einmal an, was es an Zeichenobjekten so gibt. Einige werden an einen bestimmten Ort (x,y) platziert, andere werden zwischen Koordinatenpaare (x1,y1) und (x2,y2) gezeichnet:

`PSET (x,y)`	zeichnet einen Punkt;
`LINE (x1,y1) to (x2,y2)`	zeichnet eine Linie
`BOX (x1,y1) to (x2,y2)`	zeichnet eine horizontales Rechteck
`BRICK (x1,y1) to (x2,y2)`	wie BOX, nur mit der Zeichenfarbe ausgefüllt;
`ELLIPSE (x1,y1) to (x2,y2)`	wie BOX, nur statt Rechteck eine Ellipse;
`ARROW (x1,y1) to (x2,y2)`	wie LINE, nur als Pfeil;
`CIRCLE (x,y),r`	ein Kreis mit Mittelpunkt (x,y) und Radius r
`BALL (x,y),r`	wie CIRCLE, nur ausgefüllt;
`TEXT (x,y),A$`	gibt einen Text `A$` aus.

Allen diesen Anweisungen können optional noch zwei weitere Parameter angehängt werden, einen für die Farbe und einen für die Zeichenstiftbreite. Dabei beziehen sich diese Parameter nur auf die aktuelle Anweisung, sie ändern jedoch die Voreinstellung nicht. Zum Beispiel, die Anweisung `LINE (1,1) to (2,2), darkgreen, 5` zeichnet eine dicke, dunkelgrüne, schräge Linie, ändert aber die mit DRAW *Color* voreingestellte Farbe und die mit DRAW *Width* voreingestellte Zeichenstiftbreite nicht. Wenn man für diese Anweisung unter Verwendung der voreingestellten Farbe nur die Zeichenstiftgröße auf 5 einstellen möchte, so kann man schreiben: `LINE (1,1) to (2,2),, 5`

Selbstverständlich kann man auch nur die Farbe individuell einstellen und die Zeichenstiftbreite voreingestellt verwenden: `LINE (1,1) to (2,2),darkgreen`

Die Anweisung LINE hat noch eine besondere Eigenschaft: Sie kann auch mit nur einen Koordinatenpaar geschrieben werden, und sie verwendet und versetzt eine Zeichenmarke. Schreibt man z.B. innerhalb einer For-Next Schleife nur `LINE (x,y)`, so wird beim ersten Schleifendurchlauf gar keine Linie gezeichnet sondern die zugehörige Schreibmarke auf die aktuelle Position x,y gesetzt. Beim zweiten und jedem folgenden Schleifendurchlauf jedoch wird eine Linie von der vorherigen Position der Zeichenmarke zur nun neuen Position der Zeichenmarke gezogen. Weil man in einer For-Next Schleife öfters mehr als nur eine Linie zeichnen möchte, gibt es bis zu 10 Zeichenmarken, spezifiziert durch die Befehle `LINE0, LINE1, LINE2` usw. (`LINE0` ist dabei geichbedeutend mit `LINE`).

Natürlich kann man auch dabei für jede Linie eine eigene Farbe und Zeichenstiftstärke einstellen (beispielsweise `LINE3 (x,y),red,2`).

Voreingestellt für die Grafik-Textausgabe mittels der Anweisung TEXT ist die Schriftart und die Schriftgröße, welche aktuell gerade im Protokollbereich und im Programmbereich verwendet wird (man kann sie auch durch DRAW *Textfont* und DRAW *Textsize* ändern).

Auch die TEXT-Anweisung verwendet eine Zeichenmarke, die "Schreibmarke". Sie wird an das Ende des ausgegeben Textes gesetzt, und wenn man die Anweisung TEXT A$ ohne Koordinatenangabe spezifiziert, so wird die Ausgabe des Textstrings A$ nahtlos an den vorher ausgegebenen Text angefügt. Schreibt man z.B. erst `TEXT (1,1), "Hallo!"` und dann `TEXT " Wie geht's?"` so steht im Grafikbereich an der Stelle 1,1 der Text "Hallo! Wie geht's?". Natürlich kann man auch dabei für jeden Text eine eigene Farbe und Textgröße einstellen (beispielsweise `Text "Wie geht's?",blue,12`).

Der ausgegebene Text kann durch sogenannte "Escape-Sequenzen" modifiziert werden. Escape-Sequenzen beginnen mit dem Backslashzeichen \ gefolgt von einem Steuerzeichen:

`\_`	das nachfolgende Zeichen wird verkleinert und tiefergestellt (Index);
`\^`	das nachfolgende Zeichen wird verkleinert und höhergestellt (Exponent);
`\<`	"Backspace", die Schreibmarke wird ein Zeichen zurückversetzt;
`|`	"New Line", Zeilenvorschub und Schreibmarke zurück zum Textanfang;
`\(  \)`	Fettschrift ein bzw. ausschalten;
`\{  \}`	Kursivschrift ein bzw. ausschalten;
`\[  \]`	Unterstrichene Schrift ein bzw. ausschalten.

Wenn man Text mit einem tiefer gestellten Index und einem höhergestellten Exponenten ausgeben möchte, z.B. $A_1 = 1\ m^2$ so kann man schreiben `TEXT " A\_1 = 1 m\^2"`
Die Escapesequenzen können auch miteinander kombiniert werden, zum Beispiel gibt `TEXT "\(\{\[Hallo!"` das fette, kursive, unterstrichene Wort ***Hallo!*** aus.

Einfache Grafik, wie sie als Ausgabemittel von einem guten Basic-Programm erwartet wird, ist z.B. das Zeichen und Beschriften von Diagrammen und Kurven. Für solch einfache Grafik reicht die Grafik-Werkzeugsammlung von GPR-Basic i.A. aus. Sie können ja einmal das Programm der Digital-Uhr, welches wir schon bei der Do-Loop Schleife besprochen haben, verwenden, um ein bisschen mit den Grafik-Funktionen zu spielen:

```
Draw Color green
Draw Textfont "Times Roman"
Draw Textstyle Bold
Draw Textsize 50
Do
      T$=TIME
      Do
      Loop Until T$ >< TIME
      Cls
      Text (1,5),TIME
Loop
End
```

Die Anweisung Cls löscht den im Grafik-Bereich ausgegebenen Text und setzt die Schreibmarke auf den Ursprung zurück; die anderen Grafik-Einstellungen werden dabei aber beibehalten.

7.2 Arbeiten mit dem Grafik-Bereich

Statt die Werkzeuge der Grafik-Sammlung nun einzeln auszuprobieren, wollen wir einmal zwei etwas anspruchsvollere grafische Programmieraufgaben ausprobieren. Wir können zuerst einmal versuchen, die Uhrzeit nicht als Digitaluhr, sondern als Analoguhr im Grafikbereich eines Fensters auszugeben. Die zweite Aufgabe, aus dem Themenbereich numerische Mathematik, berechnet und zeichnet die Wiedereintrittsbahn einer ballistischen Raumkapsel.

Zur Zeitausgabe in Form einer Analog-Uhr gehen wir von folgendem Programm aus:

```
Cls
Do
      T$=TIME
      .
      .
      .
      .
      Do
      Loop Until T$ >< TIME
      Cls
Loop
End
```

An den Stellen wo Punkte sind sollen Anweisungen stehen, welche die Uhrzeiger zeichnen. Die innere Do-Loop Schleife bewirkt als „Warteschleife“, dass die Zeiger nur jede Sekunde einmal neu ausgegeben werden; und vor jeder Neuausgabe löscht Cls den Zeichenbereich. Die Basic-Funktion TIME gibt die aktuelle (System-) Zeit als String zurück, z.B.: 11:35:41 Wenn wir die Zeit aber nicht digital (mit `Text TIME`) im Grafikbereich ausgeben wollen, sondern analog mit Zeigern, so benötigen wir die Sekunden, Minuten und Stunden als Zahlenwerte. Folgende drei Anweisungen erreichen das zum Beispiel:

```
Std=val(mid(T$,1,2))
Min=val(mid(T$,4,2))
Sek=val(mid(T$,7,2))
```

Die Mid-Funktion schneidet aus dem String `T$` einen Sub-String heraus, welchen die Val-Funktion dann in einen Zahlenwert umwandelt. Nun müssen wir die Koordinaten für die Uhr festlegen (die voreingestellte Einheit ist cm). Wählen wir z.B. die Position 5, 5 für den Mittelpunkt der Uhr und 4 für die Zeigerlänge (der Stundenzeiger sollte etwas kürzer sein):

```
draw coordinates (5,5)
rsek=4 : rmin=4 : rstd=2
```

Die Mittelposition der Zeiger ist soweit klar, die Endposition muss aber unter Verwendung der Werte für `Std`, `Min` und `Sek` mittels „Polarkoordinaten“ (r und φ) berechnet werden.

Dazu brauchen wir π und die jeweils zugehörigen Winkel φ (natürlich im Bogenmaß):

```
pi=4*ATN(1)
phisek=(sek/60)*2*pi-pi/2
phimin=(min/60)*2*pi-pi/2
phistd=((std+min/60)/12)*2*pi-pi/2
```

Die Berechnung des Winkels φ für den Stundenzeiger unterscheidet sich etwas von der für den Minuten- und den Sekundenzeiger. Erstens geht der Stundenzeiger in 24 Stunden zweimal rum, zweitens springt er nicht von einer auf die andere Stunde (eine Uhr mit springenden Stundenzeiger wäre vielleicht gar nicht so schlecht ablesbar; wir können sie ja bei Gelegenheit mal erfinden und patentieren lassen – muss aber nicht heute sein ☺) .

Das Zeichnen der Zeiger geschieht mit folgenden Grafik-Anweisungen:

```
Draw Color Blue       ' oder eine andere Farbe
Line (0,0) to (rsek*cos(phisek),rsek*sin(phisek)),,1
Line (0,0) to (rmin*cos(phimin),rmin*sin(phimin)),,2
Line (0,0) to (rstd*cos(phistd),rstd*sin(phistd)),,4
```

Wenn wir alle diese Anweisungen hintereinander in das Programm einfügen, dort wo die Punkte stehen, und das Programm starten, so läuft es zwar schon, aber es „flackert“ noch, bewirkt durch die Anweisung Cls, welche den Ausgabebereich jede Sekunde löscht. Dieses Flackern lässt sich beseitigen, wenn wir statt Cls auszuführen die Zeiger weiß übermalen (aber nur dann, wenn sich ihre Lage geändert hat). Also ersetzen wir Cls durch folgendes:

```
'cls
Draw Color White
Line (0,0) to (rsek*cos(phisek),rsek*sin(phisek)),,1
If min >< val(mid(Time,4,2)) then
    Line (0,0) to (rmin*cos(phimin),rmin*sin(phimin)),,2
    Line (0,0) to (rstd*cos(phistd),rstd*sin(phistd)),,4
Endif
```

Die Lage des Sekundenzeigers ändert sich jede Sekunde, also müssen wir ihn jedes Mal weiß übermalen. Die Lagen von Minuten- und Stundenzeiger aber brauchen wir nur jede Minute neu zu zeichnen. Das Flackern ist nun weitgehend beseitigt.

Besonders schön wird unsere Uhr noch durch einen Minutenkranz:

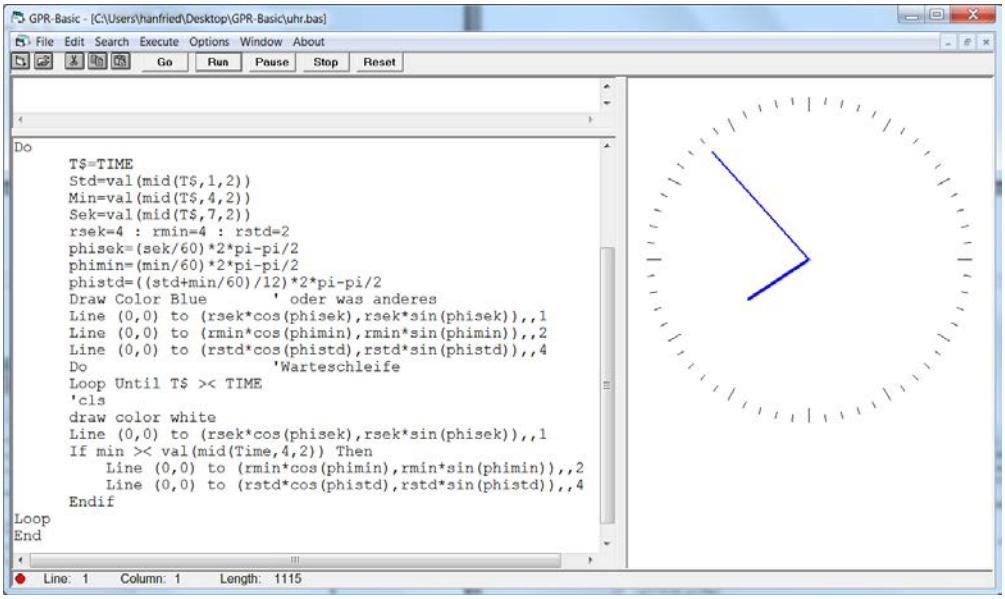
```
Pi=4*ATN(1)
For i=1 to 60
   phi=(i/60)*2*Pi
   If i/5=int(i/5) then r=4.1 else r=4.3
   Line (r*cos(phi),r*sin(phi)) to _
        (4.5*cos(phi),4.5*sin(phi))
Next i
```

Wir müssen diese Anweisungen in den Programmcode einfügen. Die beste Stelle dazu ist der Programmanfang gleich nach der ersten Cls Anweisung.

Wenn wir aber π gleich am Anfang des Programmes berechnen, können wir die Anweisung `pi=4*ATN(1)` weiter unten im Programm natürlich weglassen.

Das fertige Programm schaut dann wie folgt aus. Wenn wir es unter dem Namen "Uhr.bas" im aktuellen Arbeitsverzeichnis abspeichern, können wir es mit dem Befehl (bzw. der Anweisung) `Executegrafic "Uhr.bas"` von einem anderen Fenster aus aufrufen.

```
Cls                        ' Programm Analoguhr
draw coordinates (5,5)
rsek=4 : rmin=4 : rstd=2  'Zeigerlänge


Pi=4*ATN(1)
For i=1 to 60              ' Zahlenkranz
      phi=(i/60)*2*Pi
      If i/5=int(i/5) then r=4.1 else r=4.3
      Line (r*cos(phi),r*sin(phi)) to _
           (4.5*cos(phi),4.5*sin(phi))
Next i
Do
      T$=TIME
      Std=val(mid(T$,1,2))
      Min=val(mid(T$,4,2))
      Sek=val(mid(T$,7,2))
      phisek=(sek/60)*2*pi-pi/2
      phimin=(min/60)*2*pi-pi/2
      phistd=((std+min/60)/12)*2*pi-pi/2

      Draw Color Blue       ' oder eine andere Zeigerfarbe
      Line (0,0) to (rsek*cos(phisek),rsek*sin(phisek)),,1
      Line (0,0) to (rmin*cos(phimin),rmin*sin(phimin)),,2
      Line (0,0) to (rstd*cos(phistd),rstd*sin(phistd)),,4

      Do                    'Warteschleife
      Loop Until T$ >< TIME

      'cls
      draw color white      'Zeiger weiß übermalen
      Line (0,0) to (rsek*cos(phisek),rsek*sin(phisek)),,1
      If min >< val(mid(Time,4,2)) then
          Line (0,0) to (rmin*cos(phimin),rmin*sin(phimin)),,2
          Line (0,0) to (rstd*cos(phistd),rstd*sin(phistd)),,4
      Endif
Loop
End
```



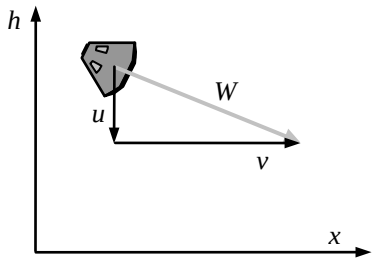
Als zweite Aufgabe zum Thema Grafik wollen wir die Wiedereintrittsbahn einer Raumkapsel numerisch integrieren und grafisch ausgeben. Ein vom Mond zurückkommendes Raumschiff vom Typ Apollo 11 unterliegt beim Wiedereintritt in die Erdatmosphäre folgenden Bewegungs-Differentialgleichungen (etwas vereinfacht, Auftrieb und Erdrotation werden vernachlässigt):

$dh/dt = u$

$dx/dt = v$

$du/dt = v^2/r - \gamma/r^2 - \frac{1}{2}\,\rho\, u \sqrt{u^2+v^2}\;\; (c_d A/M)$

$dv/dt = u\, v/r \;- \frac{1}{2}\,\rho\, v \sqrt{u^2+v^2}\;\; (c_d A/M)$



Darin sind h die Flughöhe über Meeresspiegel und x die Flugstrecke [in m], u und v sind vertikale bzw. horizontale Komponente des Geschwindigkeitsvektors W [in m/s]. Der Abstand zum Erdmittelpunkt ergibt sich als $r = 6378000 + h$ [m] . Der Staudruck Q mit der Luftdichte ρ [kg/m^3] in Abhängigkeit der Flughöhe h [m] kann dann folgendermaßen angenähert werden:

$$Q = \tfrac{1}{2}\,\rho\, W^2 \quad ; \quad \rho = 1.225\, e^{-0.00012\, h} \quad ; \quad W^2 = u^2 + v^2$$

Der „ballistische Faktor“ $c_d A/M$ (Beiwert mal Fläche durch Masse) kann für Apollo mit dem Wert 0.0005 m²/kg angenommen werden; und die Erdgravitations-Konstante beträgt $\gamma = 3.9865 \cdot 10^{14}$ m^3/s^2. Die Ableitung der Bewegungsgleichungen ist ein Problem aus der Technischen Mechanik, wir wollen uns hier aber nur mit dem mathematischen Problem der Lösung beschäftigen. Diese erfordert eine numerische Integration des Gleichungssystems.

Das folgende Programm integriert die Bewegungsgleichungen nach der „Eulermethode“:

```
cls
draw coordinates (2,12),1,-1
box (0,0) to (16,11),,5

r0=   6378000        'Erdradius in [m]
h0=    100000        'in 100 km Höhe beginnt das Manöver
w0=     11000        'Geschwindigkeit [m/s] in 100 km Höhe
rho0=   1.225        'Luftdichte am Boden in kg/Kubikmeter
gm= 3.9865*10^14     'Gravitationskonstante der Erde [m3/s²]
bf=     .0005        'Ballistischer Faktor [m²/kg]
dt=       1          'Integrationsschrittweite (1 Sekunde)

PRINT "Eintrittswinkel in Grad: ",
input alfa               'Eintrittswinkel alfa ungefähr 4.65°
alfa=alfa*acos(-1)/180   'Umrechnung alfa ins Bogenmaß
h= h0
x= 0
u= -w0*SIN(alfa)
v=  w0*COS(alfa)

Do
    W = SQR(u^2 + v^2)
    Q = 0.5*rho0 * EXP(-.00012*h)*W^2
    r = r0 + h
    h = h + u * dt
    x = x + v * dt
    u = u + (v^2/r - gm/r^2 - Q*bf*u/W) * dt
    v = v + (-u*v/r -         Q*bf*v/W) * dt
    line1 (x/200000,h/10000),red,2
Loop Until h<=0 or h>200000
End
```

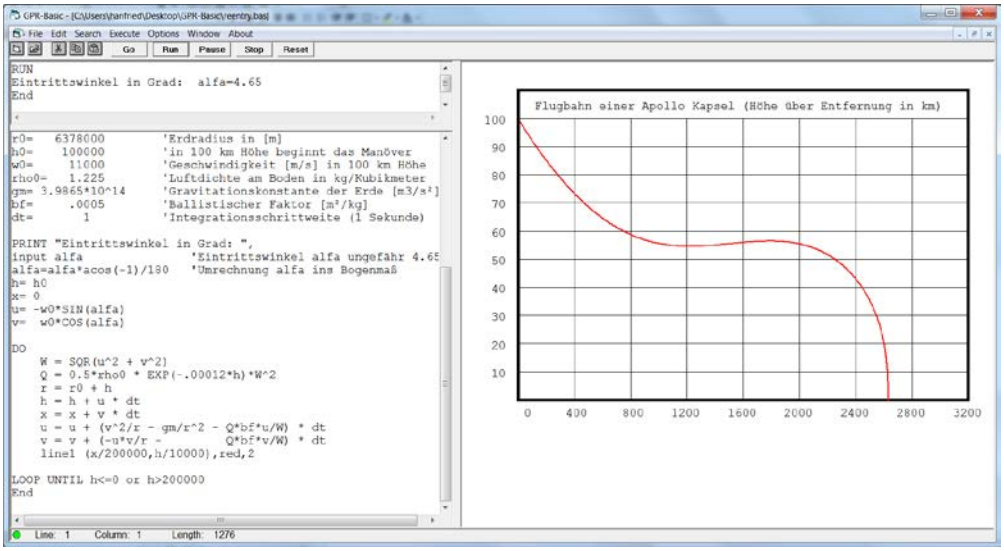
Das Programm ist nun geeignet, für Ingenieure sehr wichtige Fragestellungen zu beantworten, wie zum Beispiel:

a.) Ab welchem minimalen Eintrittswinkel α wird die Kapsel nicht mehr in den Weltraum zurückgeschleudert, in welcher Flughöhe erfolgt dabei der Hauptteil der Abbremsung?
b.) Wie groß ist bei $\alpha = 4.65°$ der maximale Wärmefluß $Flux = \sqrt{\rho}\, W^3$, wann tritt er auf?
c.) Wie groß wird dabei der gesamte (integrale) Wärmeübergang?
d.) Wie ändern sich Flugzeit und Flugweite, wenn der Luftdruck um 1% steigt?
e.) Welche maximale Abbremsung (in g) müssen die Astronauten aushalten?

Natürlich muss das Programm zur Beantwortung dieser Fragen noch entsprechen modifiziert werden, d.h. zur Lösung der Fragen müssen die notwendigen Anweisungen noch ermittelt und eingefügt werden. Hier geht es aber im Moment nur um die grafische Ausgabe, und die ist wohl noch etwas dürftig (es wird gerade einmal ein Rechteck zur Aufnahme der Flugbahn gezeichnet).

Ergänzen wir also die ersten drei Anweisungen durch folgendes Grafik-Programmsegment:

```
Cls
Draw Coordinates (2,12),1,-1
Box (0,0) to (16,11),,5

Text (0.6,10.2),"Flugbahn einer Apollo Kapsel" _
              + " (Höhe über Entfernung in km)",,12
For i=1 to 10
    Line (0,i) to (16,i)
    Text (-1.2,i-0.3), Format(10*i,"000")
Next i
For i=0 to 8
    Line (2*i,0) to (2*i,10)
    Text (2*i-0.8,-0.7), Format(400*i,"00000")
Next i
```



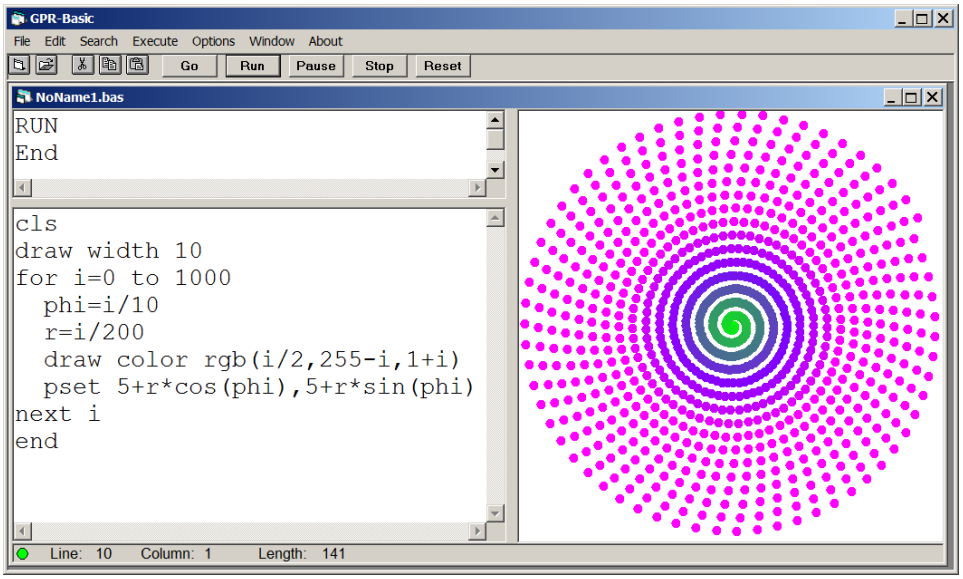
Jetzt schaut die Grafik schon erheblich professioneller aus.

Wenn wir diese Grafik in einem anderen Programm benötigen, z.B. in einem Textverarbeitungsprogramm zum Zwecke einer Masterarbeit oder einer Veröffentlichung, so können wir sie mit dem Menüpunkt „Edit→Grafic copy“ in die Zwischenablage transferieren.

8 Übungsaufgaben

8.1 Oberfläche, Variablentypen, Anweisungen, Ein- und Ausgabe

a.) Das folgende Programm zeichnet eine Spirale in den Grafik-Bereich eines Fensters:



Geben Sie das GPR-Basic Programm ein und starten Sie es. Untersuchen Sie die Einflüsse von Änderungen der Parameter in der FOR Anweisung, in der PSET Anweisung und in der RGB-Funktion. Versuchen Sie dabei Ihr künstlerisches Talent unter Beweis zu stellen.

b.) Das folgende Basic Programm berechnet für beliebige Eingabewerte von x die Funktionswerte $y = x^2$. Geben Sie das Programm ein und starten Sie es.

```
DIM x as double, y as double, antwort as String
antwort = "Ergebnis: y= "
INPUT x
y=x^2
PRINT antwort, y
END
```

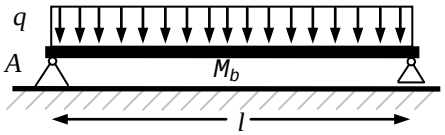
Welchen Variablentyp haben die Variablen `x` und `y`? Wodurch unterscheiden sie sich von Zählvariablen (Typ Long). Kann man mit der Zeichenkette `antwort` (Typ String) auch rechnen? Entfernen Sie die Stringvariable `antwort` und ersetzen Sie sie in der Print-Anweisung durch die Stringkonstante `"Ergebnis: y= "`. Ändern Sie dann das Programm jetzt so ab, dass es die folgenden Werte ermittelt und zusammen ausgibt:

$$y = \frac{x-1}{x+1} \qquad\qquad y' = \frac{2}{(x+1)^2}$$

Probieren Sie Beispiele und überprüfen Sie diese. Was passiert für `x` = – 1 ?

c.) Bei der Dimensionierung von Trägern in der Statik erfolgt die Berechnung der Auflagerkraft A [N] und des maximalen Biegemomentes M_b [Nm] für den Fall gleichförmiger Streckenlast q [N/m] bei einer Gesamtlänge von l [m] nach den Gleichungen:

$$A = \frac{q \cdot l}{2} \qquad M_b = \frac{q \cdot l^2}{8}$$



Schreiben Sie ein Basic Programm, das jeweils einen Zahlenwert für q und l einliest, A und M_b berechnet und die beiden Ergebnisse mit ihren Maßeinheiten ausgibt.

d.) Umfang U und Fläche F einen Kreises mit dem Radius r ergeben sich nach den Formeln:

$$U = 2\,\pi\,r \quad ; \quad F = \pi\,r^2 \qquad (\,\pi = 3.14159265358979 \rightarrow \quad \text{PI} = 4*\text{ATN}(1)\,)$$

Schreiben Sie ein Programm, das Umfang und Fläche für eingegebene Werte von r berechnet und mit 4 Nachkommastellen Genauigkeit (mittels der Format-Funktion) ausgibt.

e.) Mit dem Erdradius R = 6378.388 km berechnet sich die Entfernung D zwischen zwei Positionen (φ_1 , λ_1) und (φ_2 , λ_2) auf der Erdkugel als:

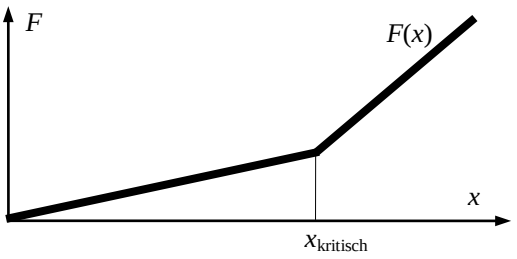
$$D = R \cdot x \;, \quad \text{mit} \quad \cos x = \sin\varphi_1\,\sin\varphi_2 + \cos\varphi_1\,\cos\varphi_2\,\cos(\lambda_2 - \lambda_1)$$

Die OTH-Regensburg befindet sich an der geographischen Postition: 49.0° Nördliche Breite, – 12.1° Östliche Länge. Schreiben Sie bitte ein Basic Programm, das die Entfernung einer anderen Position (Benutzereingabe) zur OTH-Regensburg berechnet und ausgibt.

f.) Zusatzaufgabe: Schreiben Sie ein Basic Programm, das die insgesamt zurückgelegte Strecke beim simulierten Startabbruch eines Flugzeuges berechnet:
Das Flugzeug beschleunigt zunächst ganz normal mit konstant 4.3 m/s², dann fällt zu einem bestimmten Zeitpunkt (Benutzereingabe!) der Motor aus; danach behält das Flugzeug für genau 1.2 Sekunden die erreichte Geschwindigkeit bei, bevor der Pilot das Flugzeug mit einer Bremsverzögerung von konstant 8.5 m/s² abbremst

8.2 Arithmetische Ausdrücke, Bedingte Programmausführung

a.) An der Hinterachsaufhängung von schweren Nutzfahrzeugen werden häufig Zusatzfedern angebracht. Dadurch wird bei voller Beladung ein zu starkes Einfedern verhindert.
Die Gesamtfederkraft lässt sich durch eine abschnittsweise lineare Funktion darstellen:

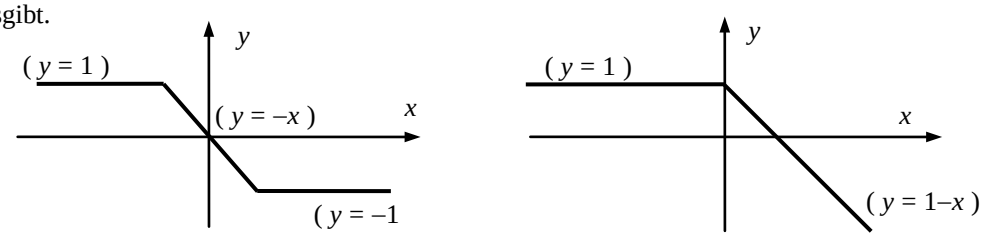


Die Steigung der Kraft $F(x)$ ist dabei für $x < x_{kritisch}$ durch C_1 und für $x > x_{kritisch}$ durch $C_1 + C_2$ gegeben.

Zahlenwerte: C_1 = 4OO [kN/m];
C_2 = 2OO [kN/m];
$x_{kritisch}$ = O.2 [m]

Schreiben Sie ein Programm, das die Federkraft F als Funktion des Federweges x berechnet. Überprüfen Sie das Programm durch die Berechnung kritischer Funktionswerte!!!

b.) Schreiben Sie ein Basic-Programm, das einen Funktionswert *y* für einen Eingabewert *x* von einer der beiden unten dargestellten abschnittsweise linearen Funktionen *y*(*x*) berechnet und ausgibt.



Das Programm soll interaktiv (d.h. durch Benutzereingabe) entweder zu der einen oder zu der anderen Funktion geschaltet werden können (verwenden Sie bei dieser Aufgabe nicht die Anweisung Goto, sondern ineinander-geschachtelte If..Else..Endif Konstruktionen).

c.) In den USA werden Höhenwerte oft in Fuß (1 ft = 30.48 cm) und Zoll (1" = 2,54 cm) angegeben. Schreiben Sie ein Basic Programm, das in einer Endlosschleife (Anweisung GOTO) solange Höhenwerte in Zentimeter anfragt und umrechnet, bis der Benutzer die Endlosschleife durch die Eingabe eines negativen Höhenwertes abbricht.
Für Höhenwerte kleiner als 30.48 cm sollen die Ergebniswerte ausschließlich als einziger Zoll-Wert angegeben werden; für größere Höhenwerte sollen zwei Werte (Fuß and Zoll) ausgegeben werden (also z.B. 10 cm => 3.937" ; dagegen 100 cm => 3 ft + 3.370" usw.).

d.) Zusatzaufgabe: Die Lösung einer quadratischen Gleichung $y = x^2 + px + q = 0$ ist gegeben durch:

$$x_{1,2} = -p/2 \pm \sqrt{(p/2)^2 - q}$$

Schreiben Sie ein Basic Programm, welches die Werte *p* und *q* als Benutzereingabe einliest, den Scheitelpunkt der Parabel berechnet und ausgibt. Nur falls vorhanden, sollen auch die Schnittstellen x_1 und x_2 der Parabel mit der *x*-Achse berechnet und ausgegeben werden.

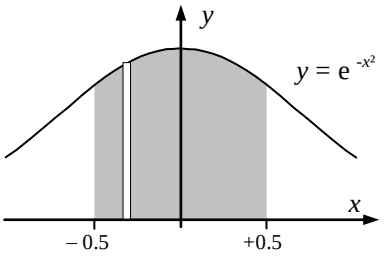
8.3 Die For-Next Schleife

a.) Schreiben Sie ein Basic Programm, das eine Tabelle für die Umrechnung der Temperaturen von Celsius in Fahrenheit berechnet und im Protokollbereich des Fensters ausgibt. Die Tabelle soll 101 Zeilen (für 101 Temperaturwerte von 0° bis 100° Celsius) und zwei Spalten (Temperaturen in Celsius und Fahrenheit) enthalten. Verwenden Sie die Format Funktion, um die Tabelle ordentlich zu formatieren. Fahrenheit = (Celsius · 9/5) + 32

b.) Ändern Sie das obige Programm jetzt so ab, dass es die Tabelle in eine Datei "Temp.txt" schreibt. Öffnen Sie dazu vor der For-Next Schleife die Datei mit der Anweisung
`Open "Temp.txt" for Output as #1` ; lenken Sie die Ausgabe mittels der Anweisung `Print #1,` ..... vom Protokollbereich in die Datei um, und schließen sie die Datei nach der For-Next Schleife wieder mit der Anweisung `Close #1`.
Tippen Sie `Edit "Temp.txt"` in den Protokollbereich ein, um die Datei anzuzeigen.

c.) Schreiben Sie bitte ein Basic Programm, welches den Wert *F* des bestimmten Integrals der Gauß-Funktion

$$F = \int_{-0.5}^{+0.5} e^{-x^2}\, dx$$

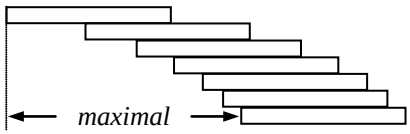


im Bereich $x = -0.5$ bis $x = +0.5$ numerisch berechnet. Das Integral ist durch Aufsummieren von Rechteckflächen (genau 1000 „Streifchen" der Breite 0.001) zu bilden.
Zusatzaufgabe: Berechnen Sie auch die *y*-Koordinate des Schwerpunktes dieser Fläche.

d.) Für den Ausbau des Dachbodens Ihres Ein-Familienhauses benötigen Sie (leider) einen Bankkredit von 50000 €, welcher von Ihrer Hausbank für eine Laufzeit von 5 Jahren für 0.5% Zinsen pro Monat zur Verfügung gestellt wird. Als Rückzahlung wollen Sie konstante monatliche Raten von 800 € vereinbaren (dieser Betrag enthält Zinsen und Rückzahlung!). Wie viel Geld schulden Sie der Bank noch am Ende der Laufzeit, und wie viel Zinsen haben Sie in dem Zeitraum dann insgesamt gezahlt? Wie hoch müssten die monatlichen Raten vereinbart werden, damit sie nach der Laufzeit von 5 Jahren gerade schuldenfrei sind, und wie wirkt sich die erhöhte Rückzahlung auf die insgesamt zu zahlenden Zinsen aus?

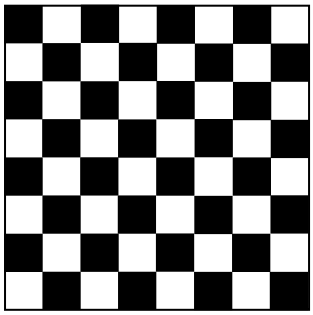
e.) Ihnen wird eine Kapital Lebensversicherung angeboten, bei der Sie über einen Zeitraum von 30 Jahren jeweils am Ersten eines Monats 100 € einzahlen sollen. Am Ende der Laufzeit würde Ihnen dann ein Betrag von 100000 € ausgezahlt werden. Alternativ könnten Sie das Geld natürlich auch auf einem Konto bei Ihrer Hausbank anlegen, welche für Sparguthaben einen monatlichen Zinssatz von 0.2 % (für Guthaben unter 25000 €) und 0.3% (für Guthaben über 25000 €) zahlt. Ist die Kapital-Lebensversicherung eine gute Geldanlage?

f.) Zusatzaufgabe: Eine bestimmte Anzahl gleichartiger Münzen (Radius $r = 1$ cm) soll so aufeinandergestapelt werden, dass der horizontale Kantenabstand von oberster und unterster Münze maximal wird. Schreiben Sie ein Programm, dass diesen maximalen Kantenabstand in Abhängigkeit von der Münzenzahl (Benutzereingabe) berechnet.



8.4 Zahlenwert-Iteration

a.) Schreiben Sie ein Basic Programm, welches das Gesamtgewicht der Reiskörner auf einem Schachbrett berechnet, wenn man auf das erste Feld ein Körnchen legt, auf das zweite zwei, auf das dritte vier, dann acht und immer so weiter (d.h. die Menge für das jeweils folgende Feld verdoppelt). Eine Tonne Reis enthält etwa 33 Millionen Körner, die Weltjahresproduktion betrug 1994 etwa 539 Millionen Tonnen...
(Reis ist das Hauptnahrungsmittel der Menschheit!)



b.) Das folgende Programm berechnet für die Werte `i` (von 1 bis 20) die Fakultät `a = i!`

```
a=1
For i=1 To 20
    a=a*i
    Print a
Next I
```

Ändern Sie das Programm nun so ab, dass es gemäß der Vorschrift $a_0 = 1$; $a_i = \frac{a_{i-1}}{i}$ jetzt die Werte `a = 1/i!` berechnet. Summieren Sie anschließend diese Werte `a` unter Verwendung einer weiteren Variablen `e` auf (initialisiert mit `e = 1`). (Die Eulersche Zahl e = 1 + 1/1! + 1/2! +1/3! +1/4!+ 1/5! + ... = 2.7182818284590.....)

c.) Die Exponentialfunktion läßt sich durch eine Reihe berechnen, dabei ergeben sich die einzelnen Glieder nach der Vorschrift $a_0 = 1$; $a_i = a_{i-1} \cdot \frac{x}{i}$; `i = 1,2,3,`...

$$e^x = 1 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \frac{x^6}{6!} + \frac{x^7}{7!} + ...$$

Schreiben Sie ein Basic Programm, dass für eingegebene Werte von *x* diese Reihe berechnet (ohne das Exponentiationszeichen ^ zu verwenden!) und mit der Funktion exp(*x*) vergleicht.

d.) Welche Iterationsvorschrift kann man verwenden um sin(*x*) und cos(*x*) zu berechnen?

$$\sin(x) = + \frac{x^1}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!} - ....$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - ....$$

Anleitung: Eine Möglichkeit ist es, das Programm so ähnlich wie die Iteration von e^x ablaufen zu lassen; nur, dass man zwar mittels einer For-Next Schleife jedes Glied der Reihe berechnet aber nur jedes zweite Glied der Reihe zum Summenwert hinzugefügt; und dass man außerdem beim Hinzufügen jedes Mal das Vorzeichen wechselt.
Wie kann man das erreichen? Man kann z.B. eine Long Variable „Schalter“ definieren und diesen Schalter (z.B. mit `schalter = – schalter`) in jedem Schritt der Schleife „umschalten“. In Abhängigkeit der „Schalterstellung“ kann man dann ein Glied auslassen bzw. nicht auslassen. Und ob dieses der Reihe zugefügte Glied dann addiert oder subtrahiert wird, kann man durch Multiplikation des Gliedes mit einer Long-Variablen „Vorzeichen“ bewerkstelligen, welche man jedesmal (z.B. mit `vorzeichen = – vorzeichen`) von +1 auf –1 bzw. von –1 auf +1 umschaltet.

Dies ist aber nicht die einzige Möglichkeit: z.B. der Ausdruck `i/2=int(i/2)` ist nur für gerade Werte von `i` erfüllt (true); der Ausdruck `i/4=int(i/4)` ist nur für ganzzahlig durch 4 teilbare Werte von `i` erfüllt (true). Auch damit lässt sich die Aufgabe lösen.

e.) Zusatzaufgabe: Für die Kreiszahl π gilt u. A. die folgende Reihenentwicklung:

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \frac{4}{13} - \frac{4}{15} + \frac{4}{17} - +.....$$

Schreiben Sie bitte ein Basic-Programm, dass den Wert dieser Reihe für 1000 Glieder berechnet und zusammen mit dem Wert $4 \cdot \text{arctg}(1)$ ausgibt.

8.5 Lösung nichtlinearer Gleichungen, die Do-Loop Schleife

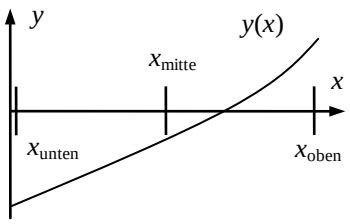
a.) Zur Lösung nichtlinearer Gleichungen werden in der Mathematik Iterationsverfahren eingesetzt. Zum Beispiel hat die Gleichung $x^2 = \cos x$ (oder umgeformt: $x = \sqrt{\cos x}$) etwa bei $x \approx 0.8$ eine reelle Lösung. Genauer lässt sich diese Lösung aber nur numerisch berechnen:

$$x_{\text{verbessert}} = \sqrt{\cos x_{\text{geschätzt}}}$$

Die einfache oben stehende Iterationsvorschrift kann dazu benutzt werden, den genauen Schnittpunkt der Graphen x^2 und $\cos x$ iterativ zu berechnen. Schreiben Sie dazu ein Iterationsprogramm, das diesen Wert auf mindestens 10 Stellen genau berechnet und ausgibt. (Anmerkung: der Ansatz $x_{\text{verbessert}} = \arccos x^2_{\text{geschätzt}}$ würde dagegen nicht konvergieren!)

b.) Schreiben Sie nun bitte ein Basic Programm, dass die positive Nullstelle der Gleichung $y = x^2 - \cos x$ mit dem Verfahren der Intervallschachtelung (Intervallhalbierung) auf mindestens 10 Stellen Genauigkeit berechnet und ausgibt.

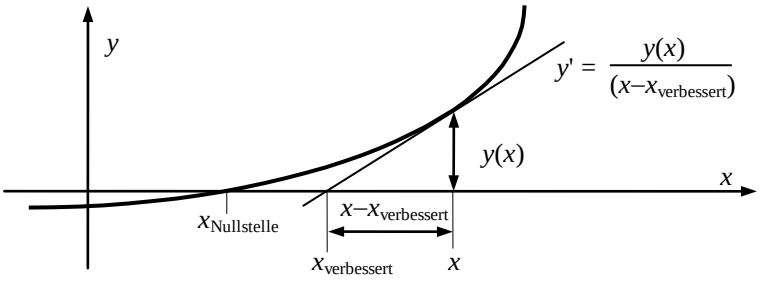
Anleitung: Die Funktion $y(x)$ sieht etwa wie in der rechts nebenstehenden Skizze aus und ist beim Wert x_{unten} (Anfangswert z.B. $x_{\text{unten}} = 0$) negativ und bei x_{oben} (Anfangswert z.B. $x_{\text{oben}} = 1$) positiv. Halbiert man nun das Intervall durch Berechnung des Mittelwertes x_{mitte}, so kann man durch Berechnung des Funktionswertes $y(x_{\text{mitte}})$ entscheiden, ob die Nullstelle nun rechts oder links von x_{mitte} liegen muss. Damit können die Werte x_{unten} bzw x_{oben} aktualisiert werden. Das Verfahren wird so bis zu ausreichender Genauigkeit fortgesetzt.



c.) Zu den schnelleren Methoden zählt das Newton-Verfahren: Zur Lösung einer Gleichung vom Typ $y(x) = 0$ berechnet man ausgehend von einem Startwert (Schätzwert) über die Iterationsvorschrift

$$x_{\text{verbessert}} = x_{\text{geschätzt}} - \frac{y(x_{\text{geschätzt}})}{y'(x_{\text{geschätzt}})}$$

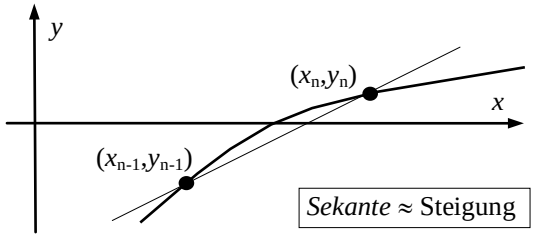
verbesserte Schätzwerte. $y'(x)$ ist dabei die Ableitung der Funktion. Bestimmen Sie die Lösungen der Gleichung $y = x^2 - \cos x = 0$.



d.) Zusatzaufgabe: Beim Sekanten-Verfahren berechnet man ein neues Wertepaar x_{n+1}, y_{n+1} aus den beiden vorherigen Wertepaaren (x_n , y_n und (x_{n-1} , y_{n-1}). Die Iterationsvorschrift lautet jetzt:

$$x_{n+1} = x_n - \frac{y(x_n)}{Sekante}$$

$$Sekante = \frac{(y_n - y_{n-1})}{(x_n - x_{n-1})}$$



Das Verfahren benötigt zwei Start-Wertepaare, z.B. für die Werte $x_0 = 0.7$ und $x_1 = 0.9$. Schreiben Sie ein Basic-Programm, das die Gleichung $y(x) = x^2 - \cos x = 0$ löst. Was passiert, wenn der Funktionswert $y(x)$ sehr klein wird?

8.6 Sub- und Function- Unterprogramme

a.) Das Subroutine-Unterprogramm `Tausch(X,Y)` vertauscht die Variablenwerte *x* und *y*. Rufen Sie das Unterprogramm vom Hauptprogramm aus mit verschiedenen Parametern auf.

```
X=1 :  Y=2 :  Z=3 :  A=4 :  B=5 :  C=6
CALL Tausch(X,Y)
PRINT X,Y,Z                    ' druckt :
CALL Tausch(A,B)
PRINT A,B,C                    ' druckt :
CALL Tausch(Y,(X))
PRINT X,Y,Z                    ' druckt :
CALL Tausch(X*X,X)
PRINT X,Y,Z                    ' druckt :
CALL Tausch(C,C*A)
PRINT A,B,C                    ' druckt :
CALL Tausch(C,123)
PRINT C                        ' druckt :
END

SUB Tausch(A,B)
     C=A
     A=B
     B=C
END SUB
```

Anmerkung: Dies ist nur eine Übungsaufgabe, um die Syntax von Unterprogrammen und die Variablenübergabe prinzipiell zu studieren. Niemand käme in der Praxis auf die Idee, zum Variableninhalt-Vertauschen ein Unterprogramm zu schreiben!

b.) Schreiben Sie ein Subroutine-Unterprogramm `Polarkoordinaten(x,y,r,phi)`, das kartesische Koordinaten in Polarkoordinaten umrechnet. (Nur die Umrechnung soll im Unterprogramm erfolgen, Koordinaten Ein- und Ausgabe erfolgt im Hauptprogramm.)

c.) Schreiben Sie bitte ein Basic Funktions-Unterprogramm, das von einem als formalen Parameter "Argument") übergebenen Wert *x* den Areacosinushyperbolicus gemäß

$$y = \text{arcosh}\,(x) = \ln\left(x + \sqrt{(x^2-1)}\right) \qquad \text{mit } x \geq 1$$

berechnet und an das aufrufende Programm zurückgibt. Falls Werte kleiner als 1 übergeben werden, soll der Funktionswert Null zurückgegeben werden.

d.) Die Kubikwurzel aus einer Zahl kann man iterativ berechnen, und zwar folgendermaßen

$$r_{i+1} = \frac{1}{3}\left(2\,r_i + \frac{x}{r_i \cdot r_i}\right)$$

Hierin sind `x` der Wert, aus dem die Kubikwurzel zu ziehen ist, r_i der alte Näherungswert, und r_{i+1} der neue Näherungswert. Ein geeigneter Startwert für r_i wäre z.B. der Wert `x` selbst. Schreiben Sie ein Funktions-Unterprogramm genannt `W3(x)`, das die Kubikwurzel entsprechend auf mindestens 10 Stellen genau bestimmt und zurückgibt.

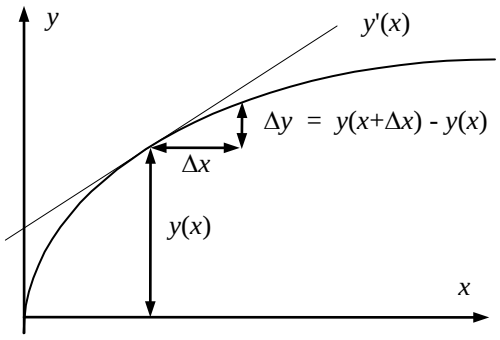
e.) Zusatzaufgabe: Die Steigung der Funktion $y(x)$ ist durch ihre Ableitung $y'(x) = dy(x)/dx$ gegeben. Numerisch kann man die Ableitung der Funktion über Differenzen-Quotienten bestimmen. Dabei kann man die mathematische Definition der Ableitung

$$y'(x) = \lim_{h \to 0} \frac{y(x+h) - y(x)}{h}$$

näherungsweise durch den Vorwärts-Differenzenquotienten ersetzen:

$$y'(x) \approx \frac{y(x+\Delta x) - y(x)}{\Delta x}$$

(mit $\Delta x > 0$, $\Delta x << 1$)



Berechnen Sie durch Vorgabe eines kleinen Wertes Δx an einer Stelle *x* (z.B. $x = 2$) die numerische Ableitung der Funktion $y = \sqrt[3]{x}$.

Danach vergleichen Sie für eingegebene *x*-Werte den numerischen Ableitungswert mit dem exakten analytisch berechneten Ableitungswert. Wie beeinflusst die Größenordnung von Δx die Ergebnisse? Zeigen Sie, dass durch Verkleinerung von Δx die numerisch berechneten Ableitungswerte nur bis zu einer bestimmten Grenze hin immer genauer werden. Warum ist das so? Wo liegt diese Grenze und wovon hängt sie ab?

Bessere Näherungswerte für die numerische Ableitung der Funktion erhält man über die Berechnung des zentralen Differenzenquotientens:

$$y'(x) \approx \frac{y(x + \Delta x) - y(x - \Delta x)}{2\ \Delta x}$$

Um wie viel ist der so berechnete Wert der Ableitung besser als der Wert des Vorwärts-Differenzenquotientens? Wie beeinflusst hier die Größenordnung von Δx die Ergebnisse?

8.7 Indizierte Variablen, Grafik

a.) Das folgende Programm erzeugt im Bereich von 0 bis 2 π genau 1001 äquidistante Datenpaare x, $y(x)$ gemäß der Funktion $y = e^{-x} \cdot \sin(x)$ und speichert sie in den Datenfeldern ("Vektoren") x(1000) und y(1000) ab:

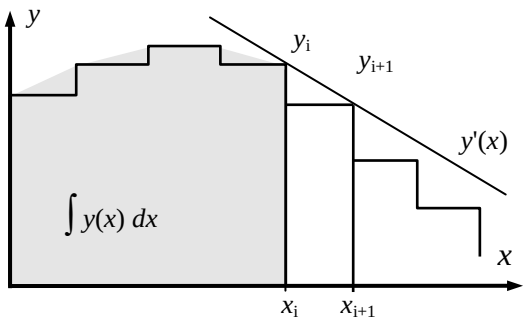
```
dim x(1000) as double, y(1000) as double
pi=4*atn(1)
for i=0 to 1000
   x(i)=2*pi*i/1000
   y(i)=exp(-x(i))*sin(x(i))
next i
```

ergänzen Sie dieses Programm durch Grafikanweisungen innerhalb der For-Next Schleife, welches die Kurve in geeignetem Maßstab (ausprobieren) im Grafikbereich darstellt;
und ergänzen Sie dieses Programm durch ein nachfolgendes Segment, welches das Datenpaar x, y_{max} mit dem größten (positivsten) y-Wert und das Datenpaar x, y_{min} mit dem kleinsten (negativsten) y-Wert heraussucht und beide Wertepaare ausgibt.

b.) Die Funktion $y(x)$ soll nun numerisch differenziert und integriert werden. Dabei soll die Steigungsfunktion durch die folgende Formel:

$$y'(x_i) \cong \frac{y_{i+1} - y_i}{x_{i+1} - x_i} \quad , \; i = 0,1,2,..,999$$

und die Integralfunktion durch Summieren der Trapezflächen berechnet werden:



$$\int_0^{x_i} y(x)\,dx \cong \sum \frac{1}{2}\,(y_{i+1} + y_i) \cdot (x_{i+1} - x_i)\,, \quad i = 0,1,2,..,999$$

Ergänzen Sie das obige Programm so, dass es diese Funktionen berechnet und in je einem Datenfeld vom Typ Double `Ydif(999)` bzw. `Yint(999)` abgespeichert. Ergänzen Sie diesen Programmteil durch Grafikanweisungen, welche auch diese beiden Kurve im entsprechenden Maßstab in ansprechender Farbe im Grafikbereich darstellen.

c.) Zusatzaufgabe:Die Eulerzahl e läßt sich z.B. durch den „Tröpfelalgorithmus“ berechnen:

$$e = 1 + \frac{1}{2}\,(1 + \frac{1}{3}\,(1 + \frac{1}{4}\,(1 + \frac{1}{5}\,(1 + \frac{1}{6}\,(1 + \frac{1}{7}\,(....)))))$$

Ausgehend von dem Wert der Pünktchen (z.B: für `n = 75: a`$_n$ `= 2`) ergibt sich e also als Endwert a_1 einer „rückläufigen“ Reihe:

$$a_n = (....)\,; \quad a_{i-1} = 1 + \frac{a_i}{i} \;\; ; \;\; (\; i = n,\; n-1,\; n-2\; ...,\; 1\;)$$

Schreiben Sie ein Programm, das die Zahl e unter Verwendung einer indizierten Variablen (`Dim e(100) as Long`) auf 100 Stellen Genauigkeit berechnet und ausgibt.

9 Zusatzaufgaben

9.1 Numerischen Integration

a.) Das Wachstum einer Population y als Funktion der Zeit t sei durch eine Differentialgleichung gegeben:

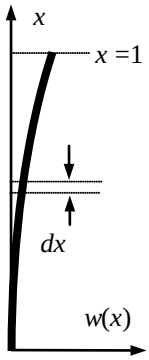
$$dy / dt = 0.76 \cdot y - 0.295 \cdot \ln(y) \qquad \text{Anfangswert:} \quad y(t = 0) = 1$$

Durch Zerlegen des gesamten Zeitintervalls in 1000 kleine Zeitintervalle $dt = 0.001$ lässt sich der jeweils nachfolgende Wert $y(t+dt)$ aus dem jeweils aktuellen Wert $y(t)$ berechnen, indem man einfach die Steigung dy/dt mit dem kleinen Zeitinterval dt multipliziert und darauf-addiert: $y(t + dt) = y(t) + (dy/dt) \cdot dt$

Schreiben Sie ein Basic Programm, das die Population y zum Zeitpunkt $t = 1$ berechnet.

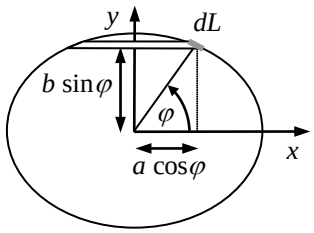
b.) Die Differentialgleichung der Biegelinie $w(x)$ eines unter Eigengewicht ausgeknickten homogenen Stabes lautet: $w'''(x) = -7.91\,(1-x) \cdot w'(x)$;
mit den Substitutionen $u(x) = v'(x) = w''(x)$ umgeformt:

$u'(x) = -7.91\,(1-x) \cdot v(x)$ Anfangswerte: $u(0) = w''(0) = 1$
$v'(x) = u(x)$ $v(0) = w'(0) = 0$
$w'(x) = v(x)$ $w(0) = 0,$



Schreiben Sie ein Programm, dass $w(x)$ numerisch integriert und die Werte zeilenweise in 1001 Schritten (für x von 0 bis 1 in Schritten $dx = 0.001$) in eine Datei "Biegelinie.txt" schreibt. (Durch Zerlegen des Bereichs x in viele kleine Intervalle dx lässt sich jeweils aus dem aktuellen Wert einer Funktion $f(x)$ der nachfolgende Wert $f(x+dx)$ berechnen, indem man einfach die Ableitung $f'(x)$ mit dem Intervall dx multipliziert und darauf-addiert, also: $f(x+dx) = f(x) + f'(x) \cdot dx$)

c.) Die Form der abgeplatteten Erde entlang eines Längenkreises wird in etwa durch die Gleichungen $x = a\cos\varphi$, $y = b\sin\varphi$ beschrieben, mit φ als Parameter. Die Erdoberfläche F ergibt sich als Summe aller horizontalen Ringflächen $dF = 2\pi x\, dL$:



$$F = \int_{-\pi/2}^{+\pi/2} 2\pi x\, dL = \int_{-\pi/2}^{+\pi/2} 2\pi\, a\cos\varphi \sqrt{(a\sin\varphi)^2+(b\cos\varphi)^2}\; d\varphi$$

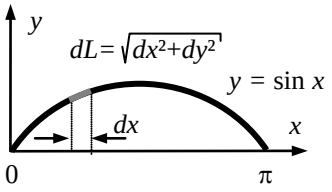
(wegen $dL = \sqrt{dx^2+dy^2} = \sqrt{(dx/d\varphi)^2+(dy/d\varphi)^2}\,d\varphi = \sqrt{(a\sin\varphi)^2+(b\cos\varphi)^2}\; d\varphi$)

Unterteilen Sie den Winkelbereich φ von $-\pi/2$ bis $+\pi/2$ in 10000 Elemente $d\varphi = \pi/10000$ und summieren Sie mit einem Programm die Ringflächen dF für die Werte $a = 6378.137$ km und $b = 6357.752$ km auf. Um wie viele Quadratkilometer ist die so berechnete Erdoberfläche F geringer als die Kugeloberfläche $4\pi a^2$?

d.) Für eine Knick-Aufgabe aus der Technischen Mechanik soll der Wert der Bogenlänge der Sinusfunktion im Bereich 0 bis π numerisch berechnet werden

(also $L = \int dL = \int \sqrt{dx^2+dy^2}$ für $y = \sin x$).

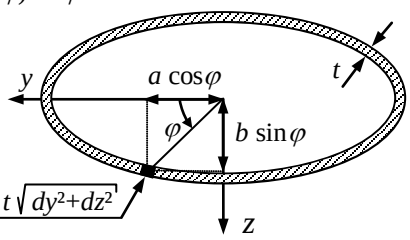
Dazu wird der Bereich in 10000 kleine Streifchen der Breite $dx = \pi/10000$ unterteilt, dabei ist dann $dy = y(x+dx) - y(x)$, und die Bogenlänge L ist einfach die Summe aller Teillängen dL.
Schreiben Sie ein Basic Programm zur Berechnung der Bogenlänge der Sinusfunktion.



e.) Das Flächenträgheitsmoment $I_y = \int z^2\, dA$ eines dünnwandigen Rohres mit elliptischer Querschnittsfläche ergibt sich wegen $y = a\cos\varphi$, $z = b\sin\varphi$ zu

$$I_y = \int_0^{2\pi} z^2\, t\sqrt{dy^2+dz^2} = \int_0^{2\pi} (b\sin\varphi)^2\, t\sqrt{(dy/d\varphi)^2 + (dz/d\varphi)^2}\; d\varphi$$

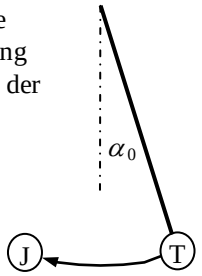
$$= b^3 t \int_0^{2\pi} \sin^2\varphi \sqrt{(a/b)^2\sin^2\varphi + \cos^2\varphi}\; d\varphi$$



Der Term b^3t muss also noch mit einem Faktor multipliziert werden, welcher, als „elliptische Integralfunktion“ des Parameters φ dargestellt, nur noch vom Verhältnis (a/b) der beiden Halbachsen abhängt. Schreiben Sie ein Basic-Programm, das den Wert dieses Integrals für Benutzereingaben (a/b) berechnet.

f.) Wenn sich Tarzan (T) zu Jane (J) an einer 9.81 Meter langen Liane hinüberschwingt und man der Bewegung eine harmonische Schwingung zugrundelegt, so dauert das $t = \pi$ Sekunden. Tatsächlich aber hängt der genaue Wert t von der Anfangsauslenkung α_0 ab und lautet:

$$t = \sqrt{2} \cdot \int_0^{\alpha_0} \frac{d\alpha}{\sqrt{\cos\alpha - \cos\alpha_0}}$$



Schreiben Sie ein Basic-Programm, das die Zeit t für $\alpha_0 = 25°$ hinreichend genau (z.B. auf drei Nachkommastellen genau) berechnet und ausgibt

Anleitung: Da der Ausdruck $\sqrt{\cos\alpha - \cos\alpha_0}$ für $\alpha \to \alpha_0$ gegen Null strebt, wird die Integralberechnung bei konstanter Schrittweite $d\alpha$ ungenau bzw. rechenaufwendig. Man definiert dann besser ein kleines „Flächenstreifchen“ $df = d\alpha / \sqrt{\cos\alpha - \cos\alpha_0}$ konstanter Fläche (z.B.: $df = 0.0001$), also $d\alpha = df \cdot \sqrt{\cos\alpha - \cos\alpha_0}$ und zählt, wieoft dieses Flächenstreifchen df mit der sich ändernden Breite $d\alpha$ in den Bereich zwischen 0 und α_0 hineinpasst.

9.2 Iterationen

a.) Wenn man am Taschenrechner für irgendeine positive Zahl in der Anzeige wiederholt die Wurzeltaste drückt, sieht man, dass der angezeigte Wert recht schnell gegen 1.000 konvergiert. Schreiben Sie bitte ein Basic-Programm, dass für einen vom Benutzer eingegebenen Anfangswert x den Produktwert P der so entstehenden Wurzelreihe iterativ berechnet und ausgibt. Zum Beispiel, für den eingegebenen Wert $x = 16$ ist P:

$P = 16 \cdot 4 \cdot 2 \cdot 1.4142135623731 \cdot 1.18920711500272 \cdot 1.09050773266526 \cdot .....$

Die Iteration des Wertes P soll beendet werden, wenn der nachfolgende Faktor nahe bei Eins liegt. (Also sich z.B. nicht mehr als um 0.000000000001 von Eins unterscheidet.)

b.) Ein sog. "Chaotischer Prozess" wird z.B. durch folgende mathematische Reihe dargestellt:

$$x_{i+1} = 4 \cdot (1 - x_i) \cdot x_i \qquad \text{Anfangswert: } x_o = 0.4 + \Delta x$$

Für kleinste Änderungen von Δx (z.B. $\Delta x = 0.0000001$, $\Delta x = 0.0000002$ etc.) variiert der Wert x der Reihe schon nach wenigen Schritten erheblich. Schreiben Sie bitte ein Basic Programm, welches den Wert der Reihe nach 30 Schritten für eingegebene Werte von Δx berechnet und anzeigt.

c.) In der Mathematik da gibt es sog. „unendliche Kettenbrüche“, man findet z.B. in der Formelsammlung nebenstehendes: Um die Richtigkeit dieser Aussage zu überprüfen, schätzen wir einen Wert für die Pünktchen (z.B. 3), berechnen den Kehrwert und addieren 2 dazu usw., bilden also die Reihe:

$$\sqrt{2} + 1 = 2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{2 + \cfrac{1}{\ddots}}}}}$$

$$a_n = 3 \; ; \quad a_{i-1} = 2 + \frac{1}{a_i} \; ; \; (i = n, n-1, \ldots 3, 2, 1)$$

Schreiben Sie ein Basic-Programm, das den Wert der Reihe für $n = 20$ berechnet und mit dem Ergebniswert (2.41421356237309) vergleicht.

d.) Auch bei einem anderen Kettenbruch kann man die Nenner rekursiv von unten beginnend berechnen. Schreiben Sie ein Basic-Programm, das den Wert dieser Reihe für $n = 20$ berechnet und mit dem Ergebniswert vergleicht.

$$\pi = \cfrac{4}{1 + \cfrac{1^2}{3 + \cfrac{2^2}{5 + \cfrac{3^2}{7 + \cfrac{4^2}{\ddots}}}}}$$

$$a_n = 2.4 \cdot n \; ; \; a_{i-1} = (2i-1) + \frac{i^2}{a_i} \; ; \; (i = n, n-1, \ldots 2, 1)$$

e.) Schreiben Sie bitte ein Basic-Programm, das den Wert der folgenden unendlichen Reihe

$$\pi = \sqrt{\frac{6}{1^2} + \frac{6}{2^2} + \frac{6}{3^2} + \frac{6}{4^2} + \frac{6}{5^2} + \frac{6}{6^2} + \frac{6}{7^2} + \ldots\ldots\ldots}$$

für 1000 Glieder berechnet berechnet und mit dem Wert $4 \cdot \text{arctg}(1)$ vergleicht.

9.3 Die Kreiszahl π lässt sich durch viele Reihen berechnen

a.) z.B. gibt es da den sog. „Tröpfelalgorithmus“:

$$\pi = 2 + \frac{1}{3}\,(\,2 + \frac{2}{5}\,(2 + \frac{3}{7}\,(\,2 + \frac{4}{9}\,(2 + \frac{5}{11}\,(\,2 + \frac{6}{13}\,(....)))))$$

Ausgehend von dem Wert der Pünktchen könnte man π also als den Endwert a_1 einer „rückläufigen“ Reihe berechnen:

$$a_n = (....)\,; \quad a_{i-1} = 2 + \frac{i}{2\cdot i + 1}\cdot a_i \;;\; (\; i = n,\; n-1,\; n-2\;...,\; 1\;)$$

Scheiben Sie ein Programm, das π so gemäß für die Werte $n = 50$ und $a_n = 2$ berechnet.

b.) oder findet man da die folgende Vorschrift:

$$\pi = 2 + 2\cdot\frac{1}{1\cdot 3} + 2\cdot\frac{1\cdot 2}{1\cdot 3\cdot 5} + 2\cdot\frac{1\cdot 2\cdot 3}{1\cdot 3\cdot 5\cdot 7} + 2\cdot\frac{1\cdot 2\cdot 3\cdot 4}{1\cdot 3\cdot 5\cdot 7\cdot 9}\;\cdots$$

Die Bruchterme entstehen dabei also als $a_i = a_{i-1}\cdot\frac{i}{2i+1}$ mit $a_0 = 1$ und $i = 1, 2, .. \propto$.
Schreiben Sie ein Basic-Programm, das den Wert dieser unendlichen Reihe für 50 Glieder numerisch berechnet und ausgibt.

c.) oder diese:

$$\pi = 3 + \frac{4}{2\cdot 3\cdot 4} - \frac{4}{4\cdot 5\cdot 6} + \frac{4}{6\cdot 7\cdot 8} - \frac{4}{8\cdot 9\cdot 10} + \cdots$$

Schreiben Sie ein Programm, das den Wert dieser unendlichen Reihe für 1000 Glieder numerisch berechnet und mit dem Wert $\pi = 4\,\mathrm{arctg}(1)$ vergleicht (d.h. zusammen ausgibt).

d.) oder eine unendlichen Produktreihe

$$\pi = 2\cdot\frac{2}{1}\cdot\frac{2}{3}\cdot\frac{4}{3}\cdot\frac{4}{5}\cdot\frac{6}{5}\cdot\frac{6}{7}\cdot\frac{8}{7}\cdot\frac{8}{9}\cdot\;\cdots$$

Dabei entstehen die Bruchterme als $a_i = \frac{i+1}{i}$ (für ungerade i) bzw. als $a_i = \frac{i}{i+1}$ (für gerade i), mit den Zählwerten $i = 1,2,3,4,5,6,7,8\;...\;\propto$
Schreiben Sie ein Basic-Programm, das den Wert von π für 1000 Glieder der Reihe berechnet und ausgibt.

e.) oder eine andere:

$$\pi = 2\cdot\left(\frac{2}{\sqrt{2}}\cdot\frac{2}{\sqrt{2+\sqrt{2}}}\cdot\frac{2}{\sqrt{2+\sqrt{2+\sqrt{2}}}}\cdot\;\cdots\right)$$

deren Nenner durch die Iterationsvorschrift bestimmt werden:

$$a_1 = \sqrt{2}\;; \quad a_{i+1} = \sqrt{2 + a_i} \quad ; \quad i = 1,\; 2,\; 3\;...\;\; n$$

Schreiben Sie ein Basic Programm, das den Wert dieser Reihe für 1000 Glieder berechnet.

9.4 Zahlenwert-Transformationen

a.) Schreiben Sie ein Basic Programm, das für eine vom Benutzer eingegebene Natürliche Zahl (z.B. "107324" oder "213" oder "10") den Quersummenwert berechnet und ausgibt.
Zum Beispiel, die Quersumme der Zahl 107324 ist 17.

b.) Schreiben Sie bitte ein Basic Programm, das alle ganzzahligen Teiler einer vom Benutzer eingegebenen Natürlichen Zahl berechnet und ausgibt.
(z.B. für 12 sollte das Programm 1, 2, 3, 4, 6 ausgeben).

c.) Schreiben Sie ein Funktions-Unterprogramm genannt Dezimal(A$), dass für eine als String übergebene Binärzahl (z.B. A$ könnte sein "10110010110" oder "1101" oder "10") den zugehörigen Dezimalwert berechnet und an das aufrufende Programm zurückgibt.

Zum Beispiel, der String mit der Binärzahl "1011" hat den Dezimalwert 13;
also "1101" (binär) ergibt $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13$ (dezimal).

d.) Bei der Transformation einer Ganzzahl vom Zehnersystem ins Binärsystem geht man folgendermaßen vor: Man teilt die Zahl ganzzahlig durch 2, notiert Ergebnis und Rest.
Der Rest (0 oder 1) ergibt die erste (niedrigste) Stelle der Binärzahl; das Ergebnis wird wiederum ganzzahlig durch 2 geteilt; wobei der Rest dann die nächsthöhere Stelle der Binärzahl ergibt.
Mit dem Ergebnis wird solange weitergemacht bis das Ergebnis zu 0 wird. Schreiben Sie bitte ein Funktions-Unterprogramm `Function Binaer(I as Long) as string` , das für eine Ganzzahl I einen String mit ihrer Binärform berechnet und zurückgibt.

Beispiel:	13 / 2 = 6 Rest 1
dezimal 13	6 / 2 = 3 Rest 0
ergibt	3 / 2 = 1 Rest 1
binär 1101	1 / 2 = 0 Rest 1

e.) Welche Zahlenwerte werden die folgenden Basic Programme ausdrucken? Überprüfen Sie Ihre Antworten durch Eingeben des Programmcodes in den Programmbereich eines GPR-Basic Fensters. Führen Sie die Programme dazu im Einzelschritt-Modus aus.

```
X = 1
Y = 2
Z = 3
A = 4
B = 5

PRINT F(X,Y)
PRINT X, Y
PRINT A,B,F(A,B)
PRINT F(10,20)
PRINT F(300,F(10,20))
PRINT Z

END

FUNCTION F(X,Y)
      Y = 100
      Z = 200
      F = X + Y
END FUNCTION
```

```
A=0 : B=1 : C=2 : D=3
For I=1 to 5 Step 2
    A=A+B
    B=B*I
    C=C*C
    If Int(I/2) <> I/2 Then D=D-1
Next I
Print A, B, C, D, I
End
```

```
X=2 : B=1 : C=1 : D=1 : E=1
For A=1 to 20
    B=B*A
    C=C*X
    D=0.5*(D+X/D)
    E=E+C/B
Next A
Print A, B, C, D, E
End
```

Anhang

Anweisungen (Befehle) zur Ablaufsteuerung:

run "file"	startet ein Basic-Programm "File" im aktuellen Fenster,
execute "file"	startet ein Basic-Programm in einem neuen Fenster,
edit "file"	started ein Fenster zum Editieren der Datei "File",
executegrafic "file"	startet ein Basic-Programm in einem neuen Fenster mit maximaler Vergrößerung des Grafik-Bereichs,
pause	schaltet in den Einzelschrittmodus
end	beendet das Basic-Programm (auch stop),
reset	beendet das Programm und löscht alle Variablen,
terminate	beendet das Programm und schließt das Fenster,
copy "file1", "file2"	kopiert "file1" auf "file2",
rename "file1", "file2"	ändert den Namen "file1" zu "file2",
del "file"	löscht die Datei "file" (auch delete),
cd "directory"	wechselt das Arbeitsverzeichnes zu "directory" (cdir),
md "directory"	erzeugt das neue Verzeichnes "directory" (auch mdir),
rd "directory"	löscht das Verzeichnes "directory" (auch rdir),
beep [frequenz, dauer]	gibt einen Signalton aus, optional frequenz in Hz,
wait "dauer"	pausiert die Programmausführung (dauer in sec.)

Anweisungen (Funktionen) zur Ein und Ausgabe:

input *variable* [,...]	liest vom Protokollbereich einen Wert für die *variable*;
a$=inputbox("prompt")	öffnet ein Eingabefenster, gibt dort den Wert "prompt" aus und weist der Stringvariablen a$ den Eingabewert zu
print *ausgabeliste* [,]	gibt eine Ausgabeliste (= Ergebniswerte von arithmetischen Ausdrücken) im Protokollbereich aus.

Anweisungen für bedingte Programmbearbeitung:

if *bedingung* then *anweisung*	die Anweisung wird nur bearbeitet, wenn die Bedingung erfüllt ist;
if *bedingung* then *a1* else *a2*	die Anweisung a1 wird bearbeitet, wenn die Bedingung erfüllt ist, sonst wird die Anweisung a2 bearbeitet,

if *bedingung1* then
.....
elseif *bedingung2* then
.....
else
.....
endif
	wenn Bedingung1 erfüllt ist, wird der erste Anweisungsblock abgearbeitet und die if-Verzweigung beendet;
	Bedingung 2 wird nur geprüft (und der zweite Block eventuell abgearbeitet), wenn Bedingung1 „false“ war;
	ist keiner der elseif-Blöcke „true“, so wird der else-Block abgearbeitet;
	die Anweisungen elseif und else sind optional, endif dagegen nicht.

Anweisungen zur Steuerung von Schleifen:

for *variable=anfangswert* to *endwert* [step *schritte*] ... [exit for] ... [continue for] ... next *variable*	Die Schleifenvariable wird mit dem Anfangswert besetzt und die Schleife solange fortgesetzt, bis die Schleifenvariable außerhalb des durch Endwert bestimmten Bereichs ist. exit for: bricht Schleife ab continue for: springt auf Next
do [until/while *bedingung*] ... [exit do] ... [continue do] ... loop [until/while *bedingung*]	Die Schleifenvariable wird fortgesetzt bis (until) oder solange (while) eine Bedingung erfüllt ist continue do: sprint auf Loop exit do: springt hinter Loop (exit do und continue do sind optional).
goto *sprungmarke* ... *sprungmarke*:	Einfacher Sprungbefehl,die Programm-Ausführung wird an der durch Sprungmarke definierten Stelle fortgesetzt.

Anweisungen zur Unterprogrammtechnik:

call *up*(*liste*)	Das Unterprogramm Up wird mit den Parametern der Liste aufgerufen.
sub *up*(*liste*) ... [exit sub] ... end sub	Hier beginnt der Code des Unterprogramms Up, der bei end sub endet. Die Parametervariablen Liste sind untereinander abgetrennt durch Kommata (exit sub ist optional).
function *fup*(*liste*) ... [exit function] ... end function	Hier beginnt der Code des Funktions-Unterprogramms fup, der bei end function endet. Funktionen haben eine Liste formaler Parameter (exit function ist optional).

Anweisungen zur Variablendimensionierung:

dim *v1* as typ, *v2(i,j,k)* as typ,.... (local *v1* as typ, *v2(i,j,k)* as typ)	Legt Speicherplatz für die Variable v1 oder die indizierten Variable v2(i,j,k) fest. Typ kann sein Double (voreingestellt), Long oder String.
global *v1* as typ, *v2(i,j,k)* as typ,....	Legt globalen Speicherplatz fest.

Anweisungen (Funktionen) zur Dateibearbeitung:

open "file" for *zugriffsart* as *#nummer*	öffnet die Datei "file" und weist ihr eine Datei-Nummer *nummer* zu. *Zugriffsart* kann sein: Input, Output oder Append.
print #nummer, *ausgabeliste* [,]	gibt eine Ausgabeliste (= Ergebniswerte von arithmetischen Ausdrücken) in die Datei aus.
lineinput #nummer, *Stringvariable*	liest die aktuelle Zeile der Datei und weist sie der Stringvariablen zu.
close #nummer	schließt die Datei wieder.
A$ = file("file")	lädt die Textdatei "file" direkt in den String A$

Grafikanweisungen (Funktionen):

draw coordinates (x,y)	setzt den Koordinatenursprung auf die Position (x,y);
draw scale x	x kann sein cm (voreingestellt), mm, inch oder pixel;
draw width x	x bezeichnet die Breite des Zeichenstiftes (voreingestellt 1 pixel);
draw color x	x kann sein black (voreingestellt), Farbe, oder die rgb-Funktion;
draw fillcolor x	x transparent (voreingestellt), eine Farbe oder die rgb-Funktion;
rgb(red,green,blue)	gibt eine Mischfarbe zurück (red, green, blue zwischen 0-255);
draw linestyle x	x ist normal (voreingestellt), dash, dot, dashdot oder dashdotdot;
draw textstyle x	x kann sein normal (voreingestellt), bold, italic, oder underline;
draw textfont A$	A$ ist ein String, der die Schriftart kennzeichnet;
draw textsize x	x bezeichnet die Schriftgröße (in Punkte).
line (x1,y1) to (x2,y2)	zeichnet eine Linie von der Position (x1,y1) zur Position (x2,y2)
lineX (x,y)	zieht die LinieX (X=0,1,2...9) weiter zur Position (x,y)
pset (x,y)	zeichnet einen Punkt an der Position (x,y);
box (x1,y1) to (x2,y2)	zeichnet eine horizontales Rechteck
brick (x1,y1) to (x2,y2)	zeichnet ein mit Zeichenfarbe ausgefüllte horizontales Rechteck;
ellipse (x1,y1) to (x2,y2)	zeichnet eine Ellipse;
arrow (x1,y1) to (x2,y2)	zeichnet einen Pfeil;
circle (x,y),r	zeichnet ein Kreis mit Mittelpunkt (x,y) und Radius r
ball (x,y),r	zeichnet einen mit der Zeichenfarbe ausgefüllten Kreis;
text (x,y),A$	gibt einen Text A$ an der Position (x,y) aus.

Mathematische Funktionen:

abs(x)	Absolutbetrag
sqr(x)	Quadratwurzel
exp(x)	e-Funktion,
log(x) (oder ln(x))	natürlicher Logarithmus
sin(x), cos(x), tan(x), cot(x)	Winkelfunktionen (im Bogenmaß!)
atn(x), asin(x), acos(x), acot(x)	Arcus-Funktionen (auch im Bogenmaß)
sign(x)	Vorzeichen (plus oder minus Eins),
int(x), fix(x)	nächst kleinerer Integer-Wert, bzw. Wert abgeschnitten
max(x1,x2...), min(x1,x2...)	Maximal bzw Minimalwert aus den Argumenten
timer	die seit Mitternacht vergangenen Sekunden

String-Funktionen:

chr(x)	erzeugt das zum Zeichencode x (0≤ x ≤ 255) gehörende Zeichen,
asc(Zeichen$)	gibt den ASCII-Zeichencode eines Zeichens Zeichen$ zurück,
val(A$)	berechnet, wenn möglich, den Zahlenwert eines Strings A$ aus dem Inhalt
str(x)	wandelt die Rechen- oder Zählvariable x in einen String um,
format(x,F$)	wandelt ebenfalls die Variable x in einen String um, wobei ein Format-String F$ bestimmt, wie das geschehen soll,
len(A$)	gibt die Länge (Anzahl der Bytes) von A$ als Integer-Zahlenwert zurück,
instr(A$,B$)	sucht das erste Auftreten des Substrings B$ im String A$, gibt diese Position zurück oder aber den Wert 0, wenn B$ in A$ nicht vorkommt,
mid(A$, I, L)	schneidet Substring aus A$ heraus, beginnend mit dem I-ten Zeichen und L Zeichen lang (auch mid(A$, I) vom Zeichen I bis zum Ende) ,
strchr(A$,Z$,I)	schreibt das Zeichen Z$ an die Stelle I in den String A$,
ucase(A$)	upper case, wandelt Kleinbuchstaben eines Strings in Großbuchstaben um,
lcase(A$)	lower case, wandelt Großbuchstaben in A$ in Kleinbuchstaben um,
time	gibt die Systemzeit als String zurück, z.B. als "21:24:39",
date	gibt das Datum als String zurück, z.B. "30.9.2006".

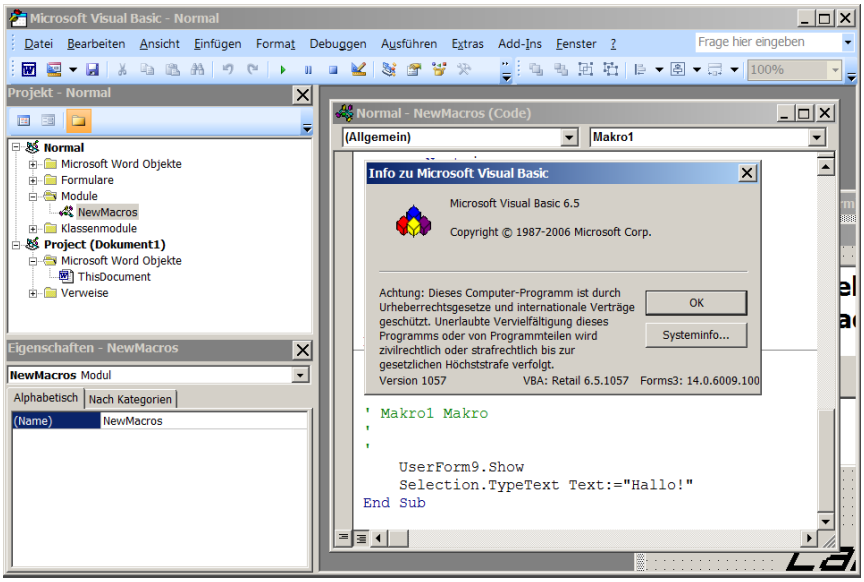
Standard ASCII Zeichen (Zeichencodes 0 - 127):

000	(nul)	016	(dle)	032	sp	048	0	064	@	080	P	096	`	112	p
001	(soh)	017	(dc1)	033	!	049	1	065	A	081	Q	097	a	113	q
002	(stx)	018	(dc2)	034	"	050	2	066	B	082	R	098	b	114	r
003	(etx)	019	(dc3)	035	#	051	3	067	C	083	S	099	c	115	s
004	(eot)	020	(dc4)	036	$	052	4	068	D	084	T	100	d	116	t
005	(enq)	021	(nak)	037	%	053	5	069	E	085	U	101	e	117	u
006	(ack)	022	(syn)	038	&	054	6	070	F	086	V	102	f	118	v
007	(bel)	023	(etb)	039	'	055	7	071	G	087	W	103	g	119	w
008	(bs)	024	(can)	040	(	056	8	072	H	088	X	104	h	120	x
009	(tab)	025	(em)	041	)	057	9	073	I	089	Y	105	i	121	y
010	(lf)	026	(eof)	042	*	058	:	074	J	090	Z	106	j	122	z
011	(vt)	027	(esc)	043	+	059	;	075	K	091	[	107	k	123	{
012	(np)	028	(fs)	044	,	060	<	076	L	092	\	108	l	124	|
013	(cr)	029	(gs)	045	-	061	=	077	M	093	]	109	m	125	}
014	(so)	030	(rs)	046	.	062	>	078	N	094	^	110	n	126	~
015	(si)	031	(us)	047	/	063	?	079	O	095	_	111	o	127	□

Inhaltsverzeichnis

Umsteigen auf Visual Basic

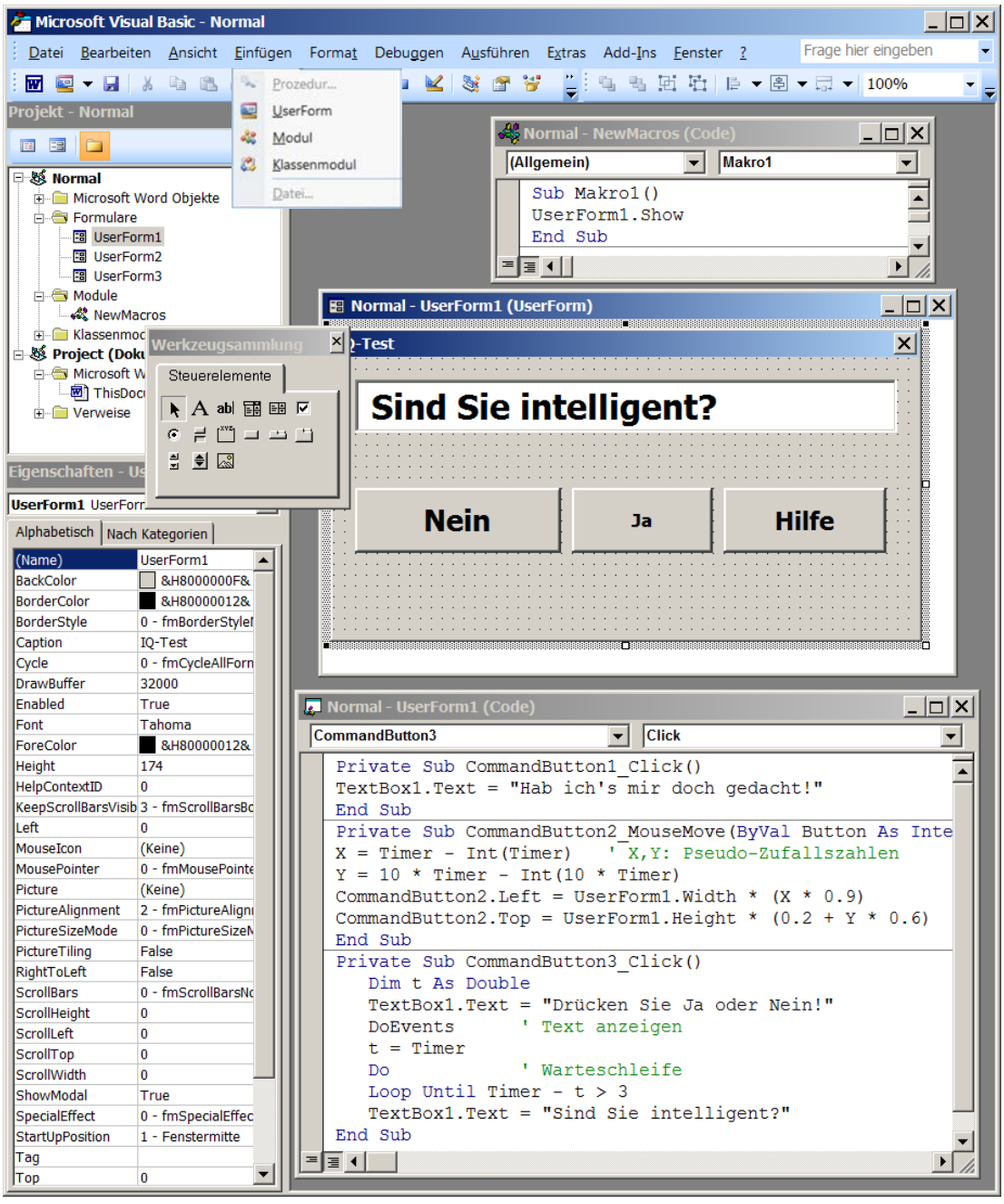
Arbeitsmaterialien zur Vorlesung und Übung
Prof.Dr.-Ing.H.Schlingloff



VBA: Zwei Beispiele zum Kennenlernen

GPR-Basic als prozedurale Programmiersprache "an sich" ist sehr ähnlich zu Visual Basic (VBA und VB-Classic); jedoch ist Visual Basic nicht nur prozedural, sondern auch noch "objektorientiert" (Objekte sind Instanzen von Klassen, sie haben Eigenschaften, verwenden Methoden und reagieren auf Ereignisse). Der VBA-Editor wird aus einer MS-Office Anwendung (z.B. Word) heraus gestartet: Menü Ansicht → Makros, Makro aufzeichnen, einer Schaltfläche zuordnen, Normal.NewMakros.Makro1 → Hinzufügen , Makro → Aufzeichnung beenden, dann Makros → anzeigen; Bearbeiten. Es erscheint der VBA-Editor (Visual Basic for Applications).

Beispiel 1: IQ-Test. Menü Einfügen → Form (Userform1.Caption = "IQ-Test"), Textbox (Textbox1.Text = "Sind Sie intelligent?") sowie drei Schaltflächen (CommandButton's) anordnen (.Font.Size und .Caption → ändern). Dann Code anzeigen, die Codezeile für das Commandbutton1_Click Ereignis ("Nein") einfügen, für den CommandButton2 ("Ja") aber das MouseMove Ereignis auswählen (im Code mittels der Timer Funktion Pseudo-Zufallszahlen erzeugen die den Commandbutton2 auf der Form immer neu positionieren). Für das CommandButton3_Click Ereignis erst sinnlose Hilfe anzeigen, dann wieder nach einer Warteschleife von 3 Sekunden in der Textbox1 "Sind Sie intelligent?" anzeigen.

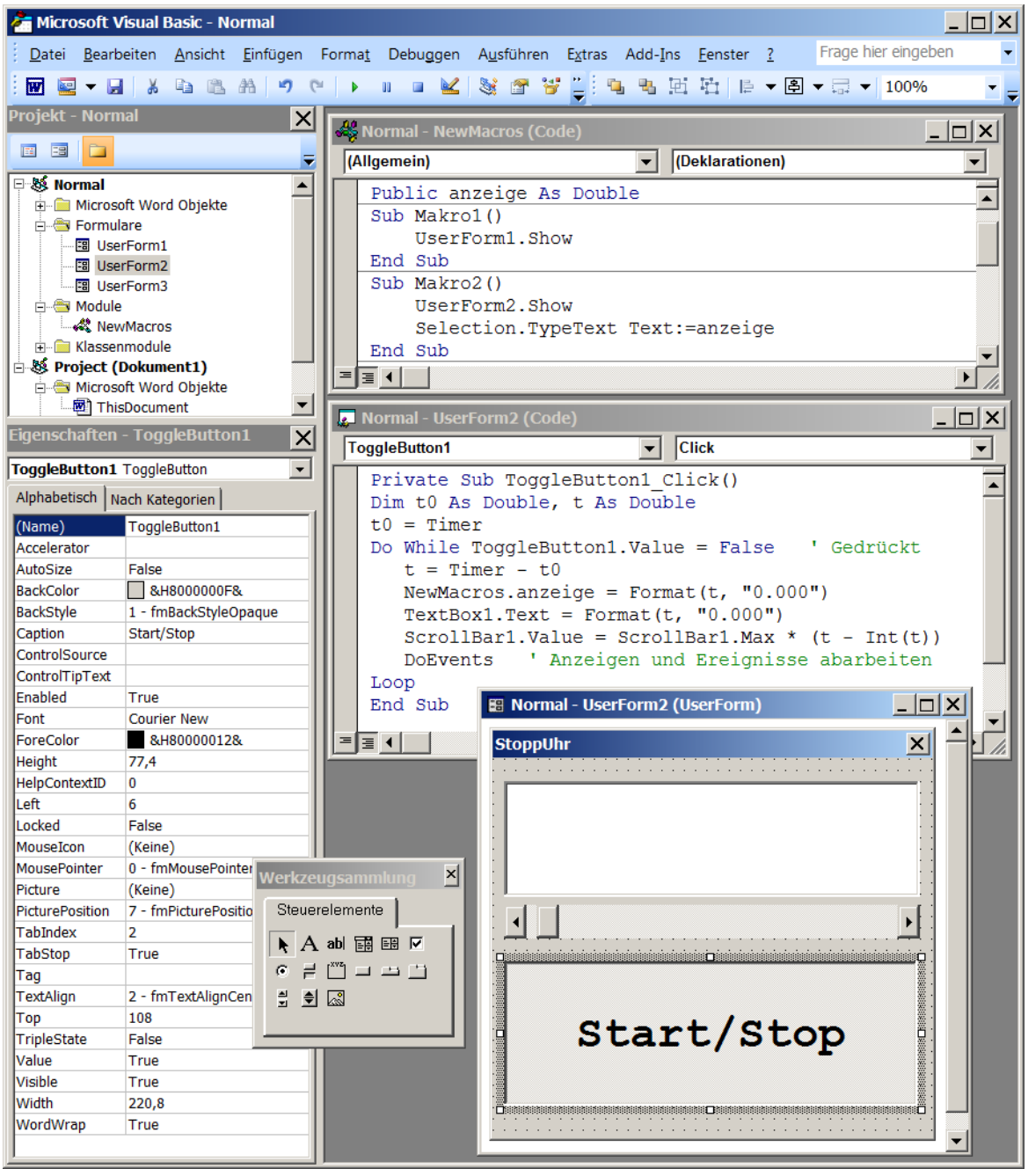


Beispiel 2: Etwas sinnvoller ist die Stoppuhr. Makros→ Aufzeichnen, dem Makro2 die Schaltfläche mit dem Bildchen "Uhr" zuordnen, "wrzlbrrrmffft" eingeben, dann Makros →Aufzeichung beenden, Makros →anzeigen , Bearbeiten. Den Code im Modul NewMakros ändern (im Objekt "Allgemein" die Public-Variable "Anzeige" deklarieren, welche den gemessenen Zeitwert an die Office Anwendung zurückgeben soll):

```
Public anzeige As Double
Sub Makro2()
    UserForm2.Show
    Selection.TypeText _
       Text:=anzeige
End Sub
```

Dann Menü Einfügen → Form (Userform2.Caption = "StoppUhr"), eine Textbox (Textbox1) einen ScrollBar (ScrollBar1) sowie eine Umschaltfläche (ToggleButton1) anordnen (.Font.Size und .Caption → ändern). Dann Code anzeigen (rechte Maustaste) und die Anweisungen für das ToggleButton1_Click Ereignis einfügen.
Im Eigenschaften-Fenster ToggleButton1.Value auf True setzen.

Die Anweisung DoEvents aktualisiert die Anzeige in der Textbox und überprüft ob ein weiteres Click-Ereignis vorliegt. Das zweite Clicken schaltet die Value-Eigenschaft des ToggleButtons wieder auf True.



VBA: Hochgenaue Berechnung der Kreiszahl π

Die Microsoft-Office Makro-Programmiersprache VBA ist "objektorientiert" und wegen der hohen Anzahl an Objekten für den Programmieranfänger durchaus kompliziert. Umsteigen von GPR-Basic auf VBA ist jedoch unproblematisch, da sich der prozedurale (nicht objektorientierte) Teil nicht wesentlich unterscheidet, wie nachfolgendes Beispiel zeigt. In der Formelsammlung findet man u.A. für π die folgende unendliche Reihe (den sogenannten Tröpfelalgorithmus):

$$\pi = 2 + \frac{1}{3}\,(2 + \frac{2}{5}\,(2 + \frac{3}{7}\,(2 + \frac{4}{9}\,(2 + \frac{5}{11}\,(2 + \frac{6}{13}\,(....)))))$$

Ausgehend von dem Wert der Pünktchen berechnet man dabei π als den Endwert a_1 einer „rückläufigen" Reihe:

$$a_n = (....);\quad a_{i-1} = 2 + \frac{i}{2 \cdot i + 1} \cdot a_i \;;\; (\; i = n,\ n-1,\ n-2\ ...,\ 1\;)$$

z.B. mit dem Anfangswert $a_n=2$ und der Genauigkeit `n = 4 * stellen.`

Algorithmus in GPR-Basic:

```
stellen=15
a=2
for i=4*stellen to 1 step -1
   a=a/(2*i+1)
   a=a*i
   a=a+2
next i
pi=a
print pi
```

mit indizierter Variable a(100) modifiziert für hohe Stellenzahl:

```
stellen = 100
Dim a(100) As Long, pi As String
a(0) = 2
For i = 4 * stellen To 1 Step -1
  rest = 0
  For j = 0 To stellen
     betrag = rest * 10 + a(j)
     a(j) = Int(betrag / (2 * i + 1))
     rest = betrag - a(j) * (2 * i + 1)
  Next j
  rest = 0
  For j = stellen To 0 Step -1
    wert = a(j) * i + rest
    a(j) =((wert/10) - Int(wert/10)) * 10
    rest = Int(wert / 10)
  Next j
    a(0) = a(0) + 2
Next i
pi = Format(a(0), "0") + "."
For i = 1 To stellen
  pi = pi + Format(a(i), "0")
  If i/10 = Int(i/10) Then pi = pi + " "
Next i
Print pi
```

Für die Ausgabe wird die indizierte Variable `a` in eine Stringvariable `pi` umgewandelt.

Division (a durch 2i+1 teilen)

```
1.234567.. : 5 = 0.24..
0___
 12
 10_
  23
```

Vorne anfangen, Betrag = 1, ganzzahlig geteilt durch 5 ergibt 0 (vorne hinschreiben), Rest = 1; dann neuer Betrag = 10*Rest + 2, also 12/5 ergibt 2 (Rest 2); dann neuer Betrag = 10*Rest + 3, ergibt 23, 23/5 ergibt 4 (Rest 3), etc...

Multiplikation (a mit i malnehmen)

```
...9987 * 11 =       77
                    88
                   99
                  99___
                 ...9857
```

Hinten anfangen, Wert = 7 mal 11 ergibt 77, die letzte Stelle von Wert (also 7) unten hinschreiben, Rest=7, dann die zweithinterste Stelle 8 mit 11 multiplizieren, ergibt 88,+Rest 7 ergibt Wert=95, 5 hinschreiben, Rest 9, dann 9*11=99, 99+9=108 → 8 (hinschreiben) Rest 10, Wert= 9*11+10=109 → 9 Rest 10, etc...

Addition (zu a Wert 2 zuzählen):

```
1.234... + 2 = 3.234...
```

einfach, weil die Zahl a wegen der Multiplikation mit dem Faktor i/(2i+1) sich praktischerweise immer im Bereich zwischen 1 und 2 bewegt.

GPR-Basic rechnet nicht besonders schnell, man kann aber den Code direkt in VBA übernehmen und so π z.B. auf 1000 oder sogar auf 10000 Stellen genau berechnen.

Word aufrufen: Menü Ansicht→ Makros, Makro aufzeichnen, Schaltfläche zuordnen: Hinzufügen → Ändern, Symbol π auswählen, Makro Aufzeichnung beenden.

Dann Makro → Bearbeiten:
Vor Sub Makro2() zum Objekt Allgemein

```
DefDbl A-H, L-Z
DefLng I-K
```

zufügen, Code einfügen zwischen

```
Sub Makro3()      und
End Sub
```

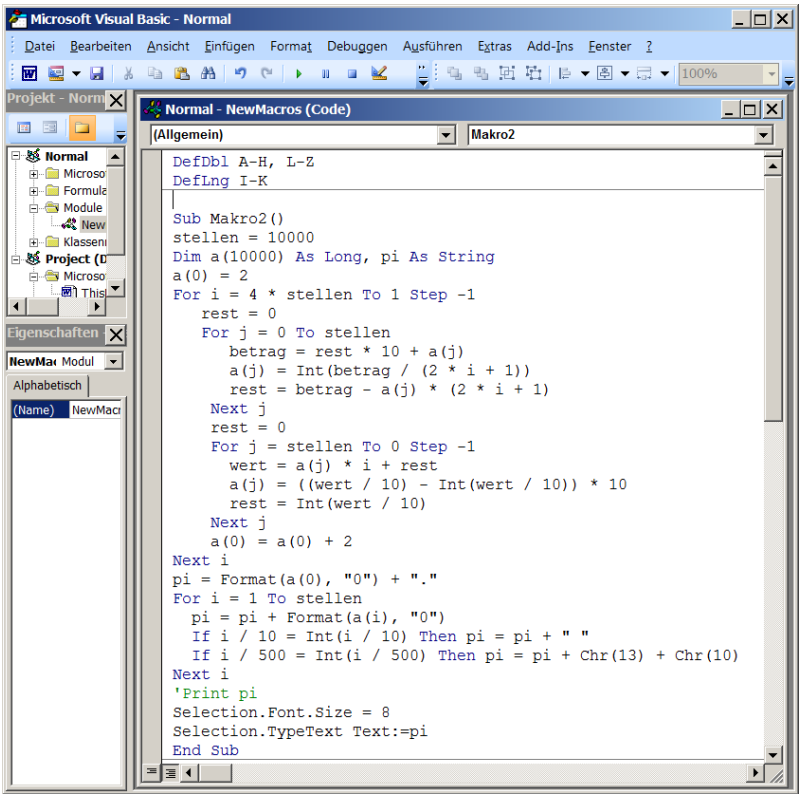
dabei Variable "stellen" und "a" ändern

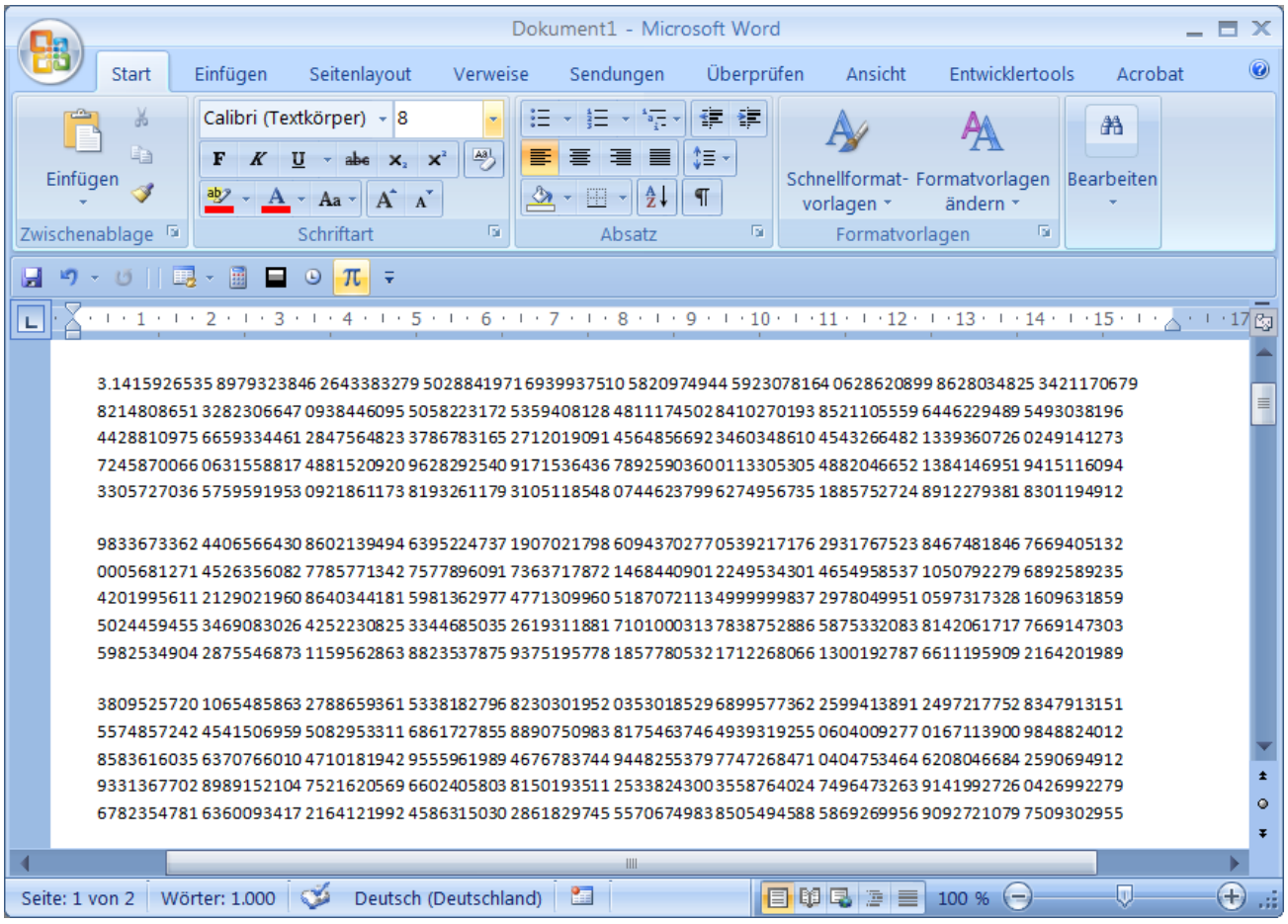
```
stellen=10000
Dim a(10000) As Long
```

und statt "Print pi":

```
Selection.Font.Size = 8
Selection.Typetext Text:= pi
```

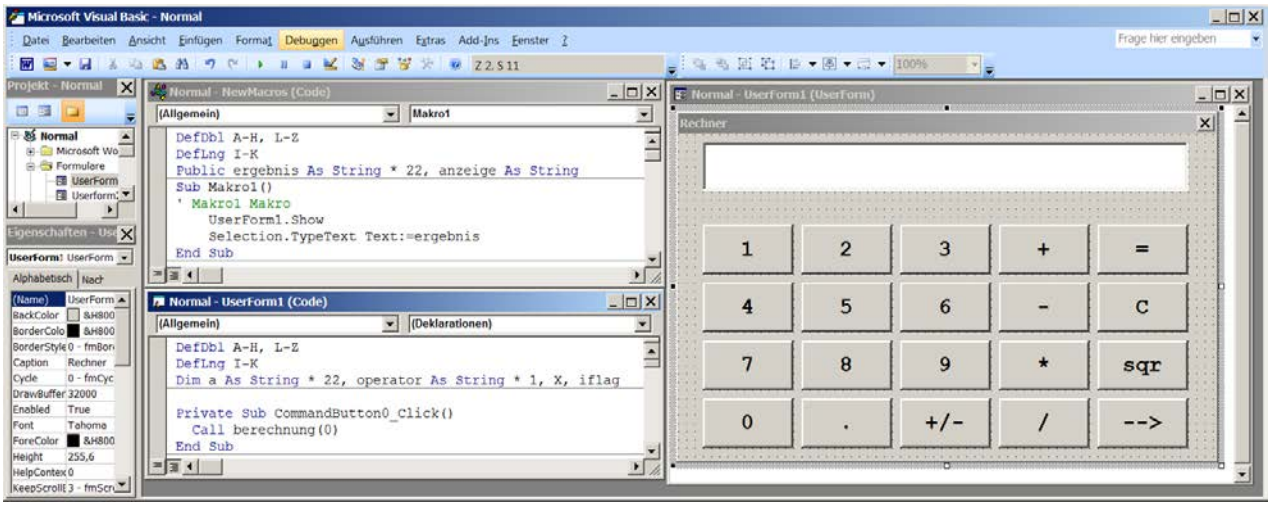
Der Berechnung von π mit 10000 Nachkommastellen dauert jetzt etwa 3 Minuten.





VBA: Der Taschenrechner als MS-Word Makro

Eine nützliche VBA-Anwendung ist z.B. ein Taschenrechner, den man aus Word heraus als "Makro" starten kann, und dessen Ergebnis man beim Beenden in Word wiederfindet. Als Objekte benötigt man hier die Form (Objekt UserForm1) die Anzeige (Objekt TextBox1), und 20 Schaltflächen (die Objekte CommandButton0 bis CommandButton19).




```
Private Sub berechnung(taste As Long)
On Error GoTo fehler
If taste >= 0 And taste <= 9 Then      ' Ziffern
      If iflag = 1 Then a = "                          ": iflag = 0
      If Mid(a, 1, 1) = " " Then
         For i = 1 To 21
            Mid(a, i, 1) = Mid(a, i + 1, 1)
         Next i
         Mid(a, 22, 1) = taste
      Else
         Beep
      End If
ElseIf taste = 10 Then                 '  Dezimalpunkt
      If iflag = 1 Then a = "                          ": iflag = 0
      If Mid(a, 1, 1) = " " And InStr(a, ".") = 0 Then
         For i = 1 To 21
            Mid(a, i, 1) = Mid(a, i + 1, 1)
         Next i
         Mid(a, 22, 1) = "."
      Else
         Beep
      End If
ElseIf taste >= 11 And taste <= 14 Then  ' + - * /
      If taste = 11 Then operator = "+"
      If taste = 12 Then operator = "-"
      If taste = 13 Then operator = "*"
      If taste = 14 Then operator = "/"
      X = Val(a)
      'a = "                        "
      iflag = 1
ElseIf taste = 15 Then                    ' ist gleich
      Y = Val(a)
      If operator = "+" Then a = Str(X + Y)
      If operator = "-" Then a = Str(X - Y)
      If operator = "*" Then a = Str(X * Y)
      If operator = "/" Then a = Str(X / Y)
ElseIf taste = 16 Then               ' clear
      Call UserForm_Activate
ElseIf taste = 17 Then               ' plusminus
      a = Str(-Val(a))
ElseIf taste = 18 Then               ' Wurzel
      a = Str(Sqr(Val(a)))
ElseIf taste = 19 Then               ' übernehmen/beenden
      NewMacros.ergebnis = a
      Unload Me
End If

anzeige:
```

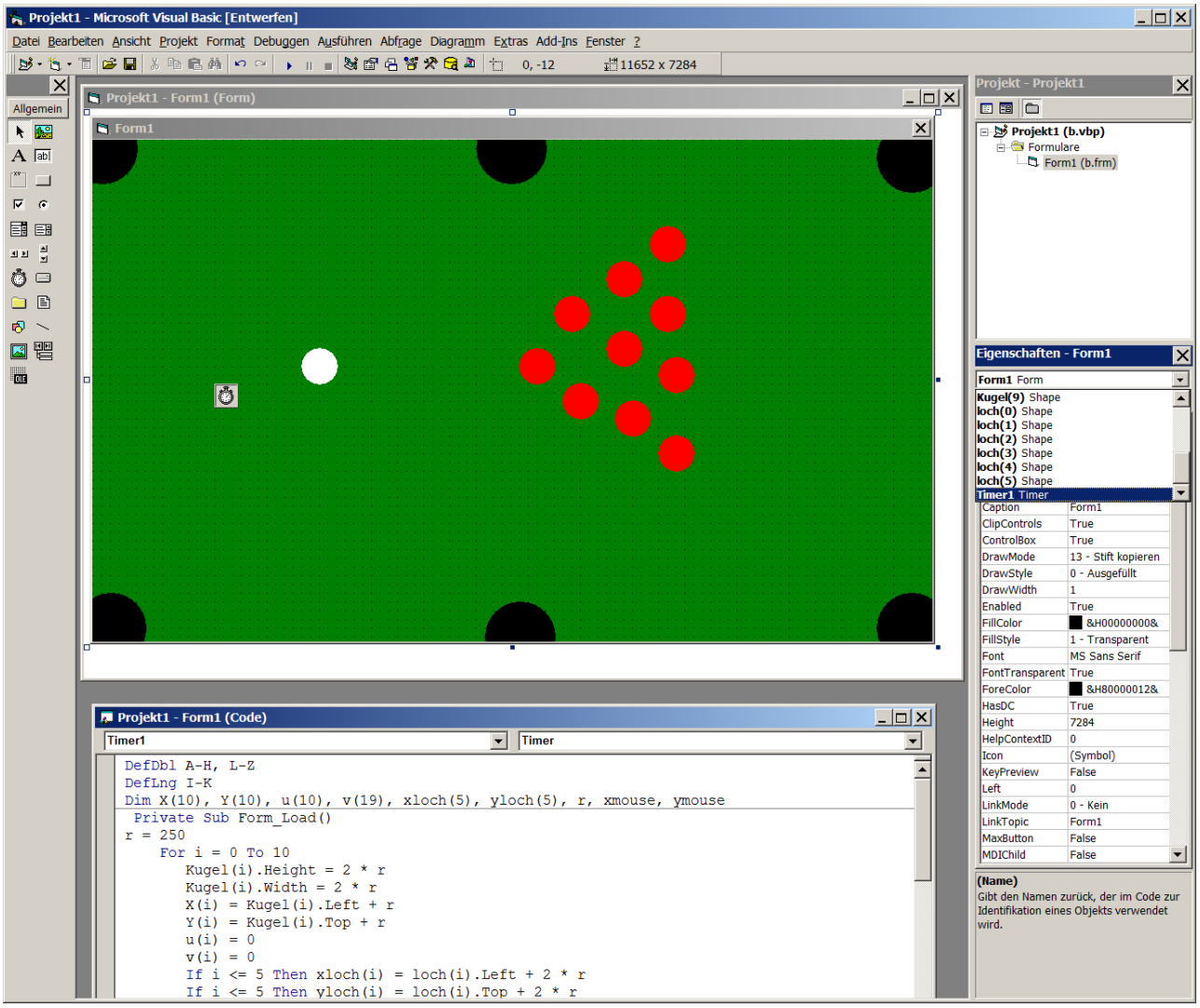
Für jeden CommandButton muss in der Click-Ereignis-Prozedur nur die Subroutine "berechnung" mit der zugehörigen Tasten-Nummer aufgerufen werden:

```
Sub CommandButton0_Click()
    Call berechnung(0)
End Sub
Sub CommandButton1_Click()
    Call berechnung(1)
End Sub
        ...
```

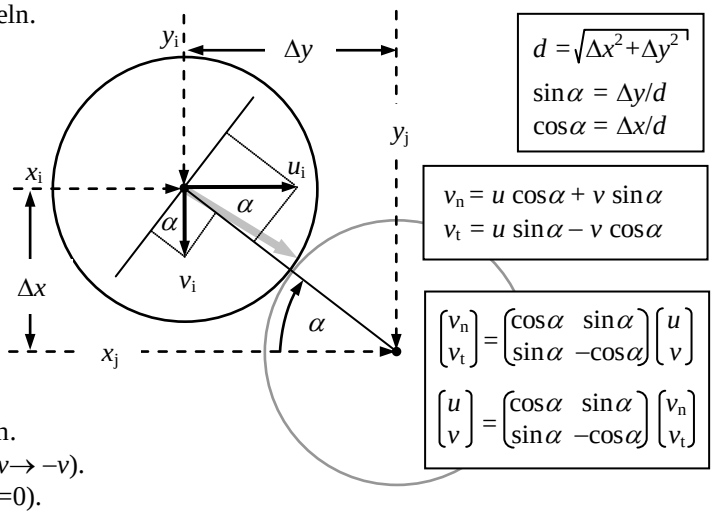
```
anzeige:

For j = 1 To 22
  If Mid(a, 22, 1) = " " Then
   For i = 22 To 2 Step -1
    Mid(a, i, 1) = Mid(a, i - 1, 1)
   Next i
   Mid(a, 1, 1) = " "
  Else
   Exit For
  End If
Next j
  TextBox1.Text = a
  Exit Sub

fehler:
  Call UserForm_Activate
  a = "                    error"
  iflag = 1
  GoTo anzeige
End Sub

Private Sub UserForm_Activate()
  a = "                                "
  operator = " "
  iflag = 0
End Sub
```

VB-Classic (VB-6): Billard als Computer-Game

VBA ist aus dem Programmiersystem VB-6 abgeleitet worden, welches heute als Visual Basic Classic bezeichnet wird. Leider wurde VB-Classic eingestellt um den Erfolg des (viel schlechteren) Nachfolgers VB.net nicht zu gefährden. Trotzdem eignet sich VB Classic gut um Anhand eines Beispiels die Entwicklung von Software zu demonstrieren.



1.) Form einrichten: Farbe, Größe, BorderStyle,
Objekte anordnen: erst 6 Löcher, dann 11 Kugeln.
2.) Objekt Allgemein: r = 250, Positionen x, y
Geschwindigkeiten u, v für alle 11 Kugeln,
Positionen x_{loch}, y_{loch} für die 6 Löcher.
3.) Routine Form_Load:
Kugel/Loch-Mittelpunke messen,
Geschwindigkeiten =0 setzen.
4.) 25msek TimerObjekt einfügen:
Kugelpositionen aus x, y und u, v
berechnen, Kugeln bewegen sich.
5.) Sub Form_MouseMove:
Cursorposition x_{mouse}, y_{mouse} messen.
6.) Sub Form_Click:
Kugel/Cursor-Abstand messen, u, v ändern sich.
7.) Kugeln prallen von der Wand ab ($u \to -u$ bzw $v \to -v$).
8.) Kugeln fallen ins Loch (.visible=false, u=0, v =0).
9.) Kugeln stoßen sich: v_t bleibt erhalten, v_n wird getauscht.



```
DefDbl A-H, L-Z
DefLng I-K
Dim x(10), y(10), u(10), v(19), xloch(5), yloch(5), r, xmouse, ymouse

Private Sub Form_Load()
r = 250
    For i = 0 To 10
       Kugel(i).Height = 2 * r
       Kugel(i).Width = 2 * r
       x(i) = Kugel(i).Left + r
       y(i) = Kugel(i).Top + r
       u(i) = 0
       v(i) = 0
       If i <= 5 Then xloch(i) = loch(i).Left + 2 * r
       If i <= 5 Then yloch(i) = loch(i).Top + 2 * r
    Next i
End Sub

Private Sub Form_MouseMove(Button As Integer, Shift As Integer, x As Single, y As Single)
  xmouse = x
  ymouse = y
End Sub

Private Sub Form_Click()
   For i = 0 To 10
      d = Sqr((xmouse - x(i)) ^ 2 + (ymouse - y(i)) ^ 2)
      If d > r And d < 2 * r Then
          u(i) = 10000 * (x(i) - xmouse) / (d - r) ^ 2
          v(i) = 10000 * (y(i) - ymouse) / (d – r) ^ 2
      End If
   Next i
End Sub

Private Sub Timer1_Timer()
For i = 0 To 10
   'Kugeln prallen an der Wand ab:
   If x(i) - r < 0 Or x(i) + r > Form1.ScaleWidth Then u(i) = -u(i)
   If y(i) - r < 0 Or y(i) + r > Form1.ScaleHeight Then v(i) = -v(i)
   'Kugeln ins Loch:
   For j = 0 To 5
      d = Sqr((x(i) - xloch(j)) ^ 2 + (y(i) - yloch(j)) ^ 2)
      If d < 1.5 * r Then Kugel(i).Visible = False: x(i) = 0: y(i) = 0: u(i) = 0: v(i) = 0
   Next j
   'Kugeln prallen voneinander ab:
   For j = 1 To 10
     d = Sqr((x(i) - x(j)) ^ 2 + (y(i) - y(j)) ^ 2)
     dneu = Sqr((x(i) + u(i) - x(j) - u(j)) ^ 2 + (y(i) + v(i) - y(j) - v(j)) ^ 2)
     If i <> j And d <= 2 * r And dneu < d And d > 0 Then
       sina = (y(j) - y(i)) / d
       cosa = (x(j) - x(i)) / d
       vnj = cosa * u(i) + sina * v(i)
       vti = sina * u(i) - cosa * v(i)
       vni = cosa * u(j) + sina * v(j)
       vtj = sina * u(j) - cosa * v(j)

       u(i) = vni * cosa + vti * sina
       v(i) = vni * sina - vti * cosa
       u(j) = vnj * cosa + vtj * sina
       v(j) = vnj * sina - vtj * cosa
     End If
   Next j
   'Kugeln bewegen sich:
    u(i) = u(i) * 0.99
    v(i) = v(i) * 0.99
    x(i) = x(i) + u(i)
    y(i) = y(i) + v(i)
    Kugel(i).Left = x(i) - r
    Kugel(i).Top = y(i) - r
Next i
End Sub
```