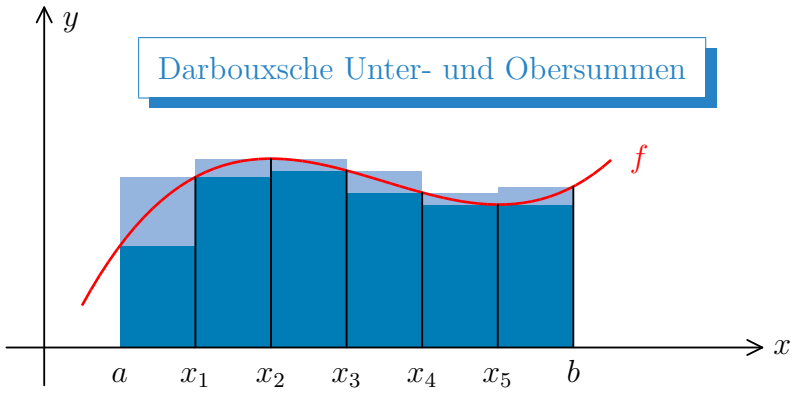
Dokumentation zu MGA-TEX

Version 2.4

– ausgedruckt am 30. Juni 2013 –

K. Fritzsche – März 2010

Graphische Darstellungen mit LATEX und pdfLATEX



Inhaltsverzeichnis

Kapitel 1

Einführung und Installation

§ 1 Was ist MGA-TEX ?

MGA-TEX steht für „**M**odular **G**raphic **A**pplications for TEX“.

Ausgangspunkt war der Wunsch, mit der `picture`-Umgebung in LATEX etwas bequemer arbeiten zu können. Es lag nahe, Makros zu entwerfen, die automatisch Koordinatensysteme zeichnen und Beschriftungen vornehmen. Das löste aber nicht das Problem, dass nur Linien mit bestimmter Steigung und Kreise mit bestimmten, kleinen Radien gezeichnet werden konnten. Damals arbeitete ich auf einem Atari mit `cs-tex` von Chr. Strunk, das ein paar `\special`-Befehle für einfache Graphik-Routinen zur Verfügung stellte. Diese sogenannten `CS`-Befehle wurden innerhalb des Systems natürlich vom Previewer und den Druckertreibern interpretiert. Also bastelte ich eine neue Umgebung, innerhalb der komfortablere Graphiken gezeichnet werden konnten. Intern wurde die LATEX-Picture-Umgebung aufgerufen, als `\unitlength` wurde `1pt` gewählt. Der Benutzer konnte eine beliebige Einheit benutzen und alle Längenangaben auf diese Einheit beziehen. Das System rechnete alles automatisch um. Damit war die erste MGA-TEX-Version geboren, die schnell weiterentwickelt wurde.

Als nächstes stand ich vor dem Problem, das Paket auf einem UNIX-Rechner einzusetzen. Es musste ein Ersatz für die `CS`-Befehle gefunden werden. Als Previewer stand Xdvi, als Druckertreiber Dvips zur Verfügung. Zu der Zeit erwies sich der `eepic`-Style als geeignetes Hilfsmittel. Die Umsetzung war nicht ganz einfach, weil der Vorrat an `\special`-Befehlen in den beiden Systemen unterschiedlich war. Es entstand das Konzept der Interface-Dateien, die je nach Hardware-Umgebung schnell ausgetauscht werden konnten.

Im dritten Schritt musste ich mein System auf einem PC unter emTEX zum Laufen bringen. Hier stand nur das sehr spartanische System der emTEX-Specials zur Verfügung, mit denen man im Wesentlichen nur eine beliebige gerade Linie zeichnen kann. Wie also beliebige Kreise zeichnen? Ich brauchte eine Routine, die beliebige Kurven aus Streckenstücken zusammensetzen konnte, und für den Kreis mussten Sinus- und Cosinus-Funktionen implementiert werden. Als Nebeneffekt standen nun auch Polarkoordinaten für verschiedene Zwecke zur Verfügung. Außerdem lag es nahe, nun auch weitere elementare Funktionen einzubauen.

Unbefriedigend blieb bisher der Import externer Grafiken. Dafür waren die Systeme zu unterschiedlich. Gebraucht wurde ein Standard. Der einzige graphische Standard, der für TEX zur Verfügung stand, war Postscript und die Verwendung von EPS-Grafiken. Zum Glück hatte ich inzwischen gelernt, wie man `\special`-Befehle

mit Postscript-Code schreibt. Gleichzeitig ergab sich die Notwendigkeit, MGA-TEX unter MikTEX (mit dem Previewer Yap) einzusetzen, was mit Postscript zwar nicht optimal, aber relativ gesehen doch am besten funktionierte. Zwar kann Yap angeblich emTEX-Specials interpretieren, in der Praxis funktioniert das aber nicht richtig. Mit Dvips und Ghostview war MGA-TEX nun auch unter MikTEX einsetzbar. Und mit Hilfe des `graphicx`-Paketes konnten eps-Grafiken geladen werden.

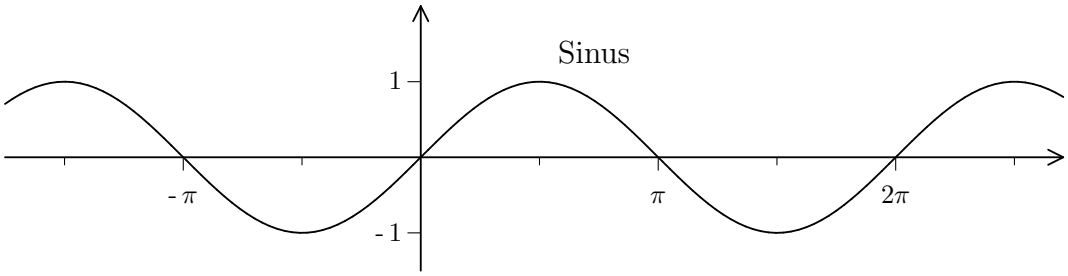
Zwei Probleme traten nun aber wieder auf: Zum einen war es lästig, unter MikTEX immer den Arbeitszyklus LATEX-Dvips-Ghostview zu benutzen, zum anderen bestand die Notwendigkeit, Vorlesungs-Skripten mit Grafiken als PDF-Dateien ins Netz zu stellen. Aber PdfLATEX konnte mit den Postscript-Befehlen nichts anfangen. Nun hatte ich die Idee, eine Programm-Version herzustellen, die unabhängig von der Hardware war. Ein Vorbild war das PicTEX-Paket, von dem ich sowieso schon viel gelernt und manche Routine übernommen hatte. Allerdings hatte ich es immer vermieden, PicTEX selbst zu verwenden, weil da alle Linien aus einzelnen Punkten zusammengesetzt werden, was zeitaufwendig ist und leicht zu Kapazitäts-Überschreitungen führt. Stattdessen griff ich auf das alte `epic`-Paket zurück, das ich früher schon benutzt hatte, etwa um auf einfache Weise gestrichelte Linien zu erstellen. In `epic` werden gerade Linien beliebiger Steigung auf geniale Weise aus den Zeichen des LATEX-Linien-Fonts zusammengesetzt. Das ist auch aufwendig, geht aber doch schneller als das Zusammensetzen von Linien aus einzelnen Punkten. Und damit ließ sich tatsächlich eine rechnerunabhängige Version erstellen. Schräge Geraden waren manchmal etwas gezackt, aber ansonsten war das Ergebnis gar nicht so übel. Und für einen sauberen Ausdruck stand ja noch immer die Postscript-Version zur Verfügung. Lediglich bei sehr komplexen Grafiken bestand die Gefahr einer Kapazitätsüberschreitung.

Die Geschichte war immer noch nicht zu Ende. Immer drängender wurde das Problem, auch PDF-Ausgaben bester Qualität zu ermöglichen. Es war etwas mühsam, bei Adobe die Dokumentation für PDF zu finden, und die ist auch nicht allzu verständlich. Während Postscript noch eine richtige Programmiersprache ist, ist PDF eine recht primitive - aus Tabellen aufgebaute - Seitenbeschreibung. Immerhin gibt es einfache Entsprechungen der Postscript-Grafik-Primitiven, sogar Bezier-Kurven, aber leider keine Befehle für Kreise und Ellipsen. Trotzdem gelang nun auch die Übertragung nach PDF. In der Version 2.3 ist diese Möglichkeit komplett integriert, zusammen mit einem komfortablen und umfangreichen Farbmanagement. Auch die Einbindung externer Grafiken ist nun leicht, die Grafiken müssen nur in zwei Versionen vorliegen, als eps-Grafik und als jpg-Grafik (oder PDF-Grafik), mit gleichem Namen. Das System sucht sich selbst an Hand der Endung die richtige Version. Jetzt kann MGA-TEX vieles, was PicTEX kann und manches, was mit PSTricks möglich ist. Darüber hinaus zeichnet es sich durch besonders einfache Bedienung und problemlose Integration in PdfLATEX aus. Es gibt eine komfortable Routine zum Zeichnen von Kurven, und es besteht die Möglichkeit, das System jederzeit durch weitere Moduln zu erweitern. Gerade das Letztere war immer meine Absicht und bisher noch etwas zu kurz gekommen.

Die Versionsnummern zeigen, dass sich das System auf eine Version 3.0 zubewegt. Geplant sind dafür vor allem Änderungen im Innern zu Gunsten einer besseren Performance. Die vielen Anpassungen an neue Situationen haben die Programmstruktur zum Teil recht schwerfällig gemacht. Insbesondere muss das Variablen-Management gründlich überarbeitet werden, so dass bessere Schnittstellen für die gewünschten Erweiterungen geschaffen werden können.

§ 2 Kurzeinführung

Mit MetaGraphA-TEX können Grafiken direkt in LATEX erstellt werden, ohne das System zu verlassen. Um zu zeigen, wie einfach das geht, zeichnen wir hier zunächst eine Sinuskurve:



Dazu sind die folgenden Befehle nötig:

```
\InitGraph{14}{3.5}{5.5}{1.5}{1cm}
\UserLine[1](2,1){$\pi$}
\UserLine[3](2,1){$\pi$}
\Coordinates(\EinPi,2)(1,1)
\Sinus(1,1,0,0)(-5.5,8.5)[140]
\TextAt(\PiHalbe,1)[tr]{Sinus}
\CloseGraph
```

Die erste Zeile öffnet eine Graphik-Umgebung, sie entspricht der folgenden Kombination von LATEX-Befehlen:

```
\setlength{\unitlength}{1cm} \begin{picture}(14,3.5)(-5.5,-1.5)
```

Universelle Einheit ist 1 cm. Die Zahlen 14 und 3.5 geben Breite und Höhe des Bildes an, die Zahlen 5.5 und 1.5 sind x- und y-Koordinate des Nullpunktes. In LATEX wird das durch Angabe eines Offsets geregelt, d.h. man gibt Zahlen an, die künftig von allen Benutzer-Eingaben zu subtrahieren sind, so dass z.B. die Angabe der Koordinaten $(0, 0)$ zu den Systemkoordinaten $(0-(-5.5), 0-(-1.5)) = (5.5, 1.5)$ führt.

Der Befehl `\Coordinates(\EinPi,2)(1,1)` zeichnet ein Koordinatensystem. Dabei erhält die x-Achse bei allen Vielfachen von π eine große Marke. An Stelle der Zahl 3.14159... kann man `\EinPi` schreiben. Der Abschnitt zwischen zwei großen Marken wird jeweils in zwei Hälften geteilt, was durch eine kleine Marke sichtbar gemacht wird. Die y-Achse erhält große Marken bei jeder ganzen Zahl, der Abschnitt dazwischen wird nicht weiter unterteilt.

Im Prinzip werden die Koordinaten auch automatisch beschriftet (mit ±1, ±2, ±3 usw.). Eine davon abweichende Beschriftung erhält man hier durch die Befehle `\UserLine[1](2,1){$\pi$}` und `\UserLine[3](2,1){$\pi$}`, die vor dem `\Coordinates`-Befehl stehen müssen. Dabei gibt die Ziffer in der eckigen Klammer die jeweilige Halbachse an ([1] für die positive x-Achse, [2] für die positive y-Achse, usw. im Gegen-Uhrzeigersinn). Die beiden folgenden Ziffern geben an, ab welcher Marke Zahlen erscheinen sollen und ab welcher Marke der Text (hier „π“) erscheinen soll. Um die Beschriftung „π, 2π, 3π, ...“ zu erreichen, gibt man also $(2, 1)$ an.

Der Befehl `\Sinus(`a, b, c, d`)(`x_1, x_2`)[`n`]` zeichnet die Funktion $y = a \cdot \sin(bx+c)+d$, im Intervall $[x_1, x_2]$, mit Hilfe von n Strecken. Also liefert

```
\Sinus(1,1,0,0)(-5.5,8.5)[140]
```

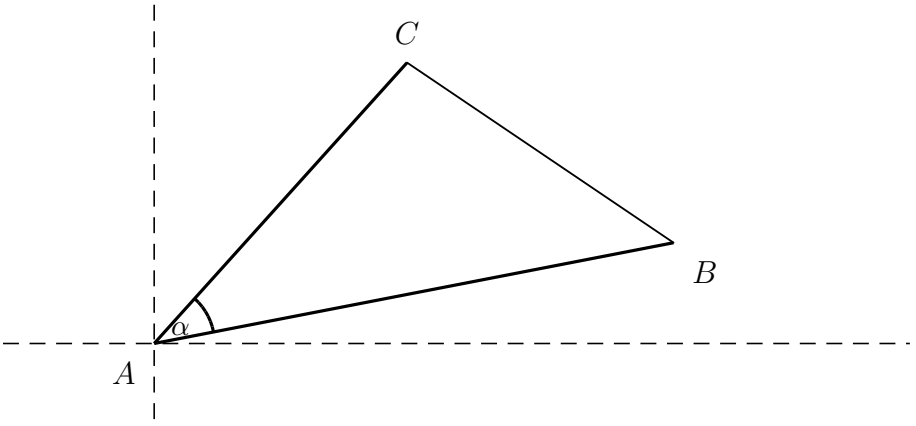
eine Standard-Sinuskurve über die gesamte Bildbreite, zusammengesetzt aus 140 Strecken.

Der Befehl `\TextAt(\PiHalbe,1)[tr]{Sinus}` liefert schließlich noch eine Beschriftung, und zwar rechts oberhalb (r und t) der Position $(\pi/2, 1)$. Der Text ist der Schriftzug „Sinus“.

Schließlich wird mit `\CloseGraph` die Graphik-Umgebung wieder geschlossen. Das entspricht dem LaTeX-Befehl `\end{picture}`.

Im zweiten Beispiel soll ein Dreieck aus zwei gegebenen Seiten und dem von ihnen eingeschlossenen Winkel konstruiert werden.

Die Seite AB (der Länge 7) soll um 11° gegen die x-Achse geneigt sein, der Winkel BAC soll 37° betragen, die Seite AC soll die Länge 5 haben.



Das geht folgendermaßen:

```
\InitGraph{12}{5.5}{2}{1}{1cm}
\SetDashed
\NoArrowAxes
\Axes
```

```
\SetNormal
\MoveTo(0,0)
\SetThick{1.2pt}
\NamedAngle{$\alpha$}of(37)At(0,0)(11)
\LineDirection(11,7) \Store(PunktB)
\MoveTo(0,0)
\LineDirection(48,5) \Store(PunktC)
\SetNormal
\LineToLoc(PunktB)
\TextAt(0,0)[bl]{$A$}
\MoveToLoc(PunktB) \Text[br]{$B$}
\MoveToLoc(PunktC) \Text[t]{$C$}
\CloseGraph
```

Zur Orientierung wird ein gestricheltes Koordinatensystem (`\SetDashed` und `\Axes`) eingezeichnet. Mit dem Befehl `\NoArrowAxes` werden die Pfeilspitzen an den Koordinatenachsen unterdrückt. Mit `\SetNormal` wird wieder auf normale (nicht gestrichelte) Linien zurückgeschaltet. Ein Teil des Dreiecks wird mit dickeren Linien gezeichnet (`\SetThick{1.2pt}`).

Nun zur Konstruktion des Dreiecks:
`\MoveTo(0,0)` bewegt einen imaginärer Cursor auf die Position $(0, 0)$.
`\LineDirection(11,7)` zeichnet eine Gerade von der Position des Cursors aus (also von $(0, 0)$ aus) zu dem Punkt mit den Polarkoordinaten $r = 7$, $\varphi = 11°$.
Mit `\Store(PunktB)` wird die Position des nun erreichten Punktes B unter dem Namen „PunktB“ gespeichert.
Nochmal wird der Cursor zum Nullpunkt bewegt und dann eine Gerade zum Punkt mit den Polarkoordinaten $r = 5$, $\varphi = 48°$ gezeichnet.
Mit `\Store(PunktC)` wird die Position von C gespeichert.
Dann wird mit `\LineToLoc(PunktB)` die dritte Seite gezeichnet, dafür war die Speicherung von B nötig.

Alle anderen Befehle dienen der Beschriftung. Der Befehl `\Text` bzw. `\TextAt` schreibt den angegebenen Text. Die Positionsbefehle c, b, t, r, l bedeuten „zentriert“, „oben“, „unten“, „rechts“, „links“.
Der Befehl `\NamedAngle{`α`}of(37)At(0,0)(11)` beschriftet den Winkel und zeichnet den Winkelbogen.

§ 3 Installation

Die folgenden Angaben beziehen sich auf den Gebrauch der Version MG A-TEX.2.3 unter LaTeX2e.

Kopieren Sie die folgenden Dateien in Ihr Arbeitsverzeichnis:

`mgtex_23.sty, mg23_pre.sty, mg23_cor.sty,`

`mg23i_ep.sty, mg23i_ps.sty, mg23i_pd.sty,`
`mga_col.sty, mga_turt.sty,`
`mg23_3d.sty,`

sowie die Datei `epic.sty` (die nur unter der Option `ep` geladen wird), falls sie in Ihrem TEX-Sytem fehlt. **Damit ist die Installation abgeschlossen!**

Die Dokumentation ist in `mg23_doc.pdf` enthalten. Will man sie selbst erstellen (etwa als PS-Datei), so braucht man außerdem:

`mg23_doc.tex`, `mg23_d01.tex` bis `mg23_d04.tex`,
`mga_math.tex`, `mga_stil.sty`,
`Fourier2.csg`, `Hund.eps`, `Hund.jpg`,
`Torus3.eps` und `Torus3.jpg`.

`mgtex_23.sty` enthält fast alle Benutzer-Macros (nur die Koordinaten-Befehle sind in `mg23_cor.sty` ausgelagert) und stellt damit den Kern des Grafik-Paketes dar. Intern werden weitere Pakete geladen: `mg23_pre.sty` enthält mathematische Hilfsroutinen. Je nach eingestellter Option wird ein Interface geladen, in dem die Grafik-Primitiven (Punkt, gerade Linie, Kreis, Ellipse, Bezier-Kurve) definiert werden (`mg23i_ep`, `-.ps` oder `-.pd`). Vom jeweiligen Interface aus wird intern auch noch das Package „graphicx“ geladen! Mit Hilfe weiterer Optionen werden die Dateien `mga_col.sty` (Farb-Management), `mga_turt.sty` (Turtle-Graphik) und `mg23_3d` (3d-Graphik) aufgerufen.

Im Vorspann Ihres Files (also vor `\begin{document}`) sollte stehen:

```
%
\usepackage[ps,col,3d,turt]{mgtex_23} % MGA-Kern
%
% Optionen fuer mgtex_23:
%    1. Interface:
%         keine Option oder [ep] - epic (laeuft mit allen Treibern)
%         oder [pd] - PDF (pdflatex und acrobat-reader)
%         oder [ps] - Postscript (dvips und ghostview)
%    2. Zusatzoptionen:
%         [3d] - 3d-Modul
%         [turt] - Turtle-Graphik
%         [col] - Farb-Management
%
```

Für die Wahl des Interface ermöglicht Version 2.3 die folgenden drei Optionen:

1. **Keine Options-Angabe** oder `[ep]` lädt das Interface `mg22i_ep.sty` und intern den Epic-Style von Sunil Podar (also das File `epic.sty`). Dadurch arbeitet MGA-TEX völlig rechner-unabhängig. Jedes System und jeder Treiber, der die LATEX-Picture-Umgebung beherrscht, kann dann die erzeugten Grafiken darstellen bzw. ausdrucken. Die Ausgabe-Qualität ist überraschend gut,

aber nicht optimal, und auf langsamen Rechnern muss man sich auf lange Rechenzeiten gefasst machen. Außerdem muss man auf die Einbindung von externen Grafiken verzichten.

2. Die Option `[ps]` lädt das Interface `mg22i_ps.sty`. Dann benötigt man PS-fähige Treiber, hat aber den vollen Funktionsumfang von MGA-TEX zur Verfügung. Die Ausgabequalität ist sehr gut, und die Rechenzeiten sind vernünftig. Außerdem können zusätzliche Tools genutzt werden, z.B. die Einbettung von PS- und EPS-Grafiken und die Möglichkeit, Inline-POSTSCRIPT-Befehle einzufügen. Der Treiber DVIPS von Thomas Rokicki kann die Ergebnisse ausdrucken oder in ein PS-File schreiben, mit Ghostview kann man sie sich am Bildschirm ansehen.

   Unter WINDOWS hat sich mittlerweile das mikTEX-System eingebürgert. Der Previewer Yap verarbeitet die POSTSCRIPT-Makros, zeigt die Grafiken aber leider immer noch nicht richtig an (ehrlich gesagt tut es meine Version 0.99g sogar schlechter als einige ältere Versionen). Das macht aber nichts. Wenn Sie Ihr System richtig eingerichtet haben, stehen Ihnen die Buttons „Make PS“ (Aufruf von DVIPS) und „View PS“ (Aufruf von GSview) zur Verfügung. Damit können Sie alles am Bildschirm korrekt sehen und auch ausdrucken. Wenn Sie eine hinreichend neue Version von Ghostview installiert haben, kann über WINDOWS gedruckt werden und es sollte dann auch keine Treiber-Probleme geben. Ist Ihnen der Zyklus „LATEX–MakePS–ViewPS“ zu umständlich, so arbeiten Sie im Entwurf-Stadium zunächst mit der Option `[ep]` und ändern das erst ganz am Schluss in `[ps]` um. Wenn Sie allerdings externe Grafiken laden wollen, kommen Sie sowieso nicht um POSTSCRIPT herum.

   Unter UNIX arbeite ich mit TeTEXund Consolen-Kommandos. Der Previewer XDVI (in einer einigermaßen modernen Version) zeigt auch bei der Option `[ps]` alles korrekt an, und gedruckt wird sowieso mit DVIPS. Unter LINUX sollte es genauso gehen.

3. Die Option `[pd]` lädt das Interface `mg22i_pd.sty`. Es dient der Erstellung von PDF-Dokumenten, was in der letzten Zeit immer mehr an Bedeutung gewinnt. Aber aufgepaßt! Jetzt entfällt der TEX-Durchgang (er würde sogar zu Fehlermeldungen führen), er wird ersetzt durch den Aufruf von `pdflatex`. Das Ergebnis ist ein PDF-File, das z.B. mit dem Acrobat-Reader angesehen und ausgedruckt werden kann. Externe Grafiken müssen im jpeg-Format oder als PDF-File vorliegen.

   Bei mikTEX(WINDOWS) brauchen Sie die Buttons „Make PDF“ und „View PDF“, unter UNIX/LINUX verwenden Sie direkt die Befehle „pdflatex *File-Name*“ und „acroread *File-Name.pdf*“.

Die gute Nachricht: Sie müssen nur die Option im Vorspann ändern, die eigentlichen Grafik-Befehle bleiben unverändert. Die Einbettung externer Grafiken ist

nur bei den Optionen `[ps]` und `[pd]` möglich, funktioniert dann aber in beiden Fällen gleich (vorausgesetzt, die Grafiken liegen doppelt vor, als EPS-File und als jpeg-File, mit gleichem Namen).

Wie der Name des Paketes sagt, ist es als modulares Paket gedacht. Daher können jederzeit Zusatzmodule angefügt werden (für deren Programmierung natürlich bestimmte Vorgaben zu beachten sind). Als Beispiel dienen die Zusatzmodule `mga_turt.sty` und `mga_3d.sty`, mit deren Hilfe eine einfache Turtle-Grafik bzw. einfache 3-dimensionale Bilder in Zentralperspektive erzeugt werden können.

Die Grafiken werden nun wie folgt innerhalb des Textes erzeugt:

```
\begin{center}
\InitGraph{...}{...}{...}{...}{...}
... Grafik-Befehle
\CloseGraph
\end{center}
```

Kapitel 2

Die elementaren Grafik-Routinen

§ 1 Die Grafik-Umgebung

`\InitGraph{`b`}{`h`}{`x_0`}{`y_0`}{Einheit}` leitet eine Grafik der Breite b und der Höhe h ein. Der Nullpunkt für die Benutzer-Koordinaten wird an die Stelle (x_0, y_0) gelegt, bezogen auf die linke untere Ecke des Bildes. Alle Zahlenangaben werden mit der angegebenen Einheit multipliziert. Ich empfehle, möglichst immer mit der gleichen Einheit (etwa 1cm) zu arbeiten. In Ausnahmefällen kann man durch Wahl einer anderen Einheit schnell Bilder vergrößern oder verkleinern. Allerdings bezieht sich das nicht auf Schriftgrößen!

Mit der Initialisierung wird auch ein gedachter Grafik-Cursor bereitgestellt und auf die Position $(0, 0)$ gesetzt (in Benutzer-Koordinaten).

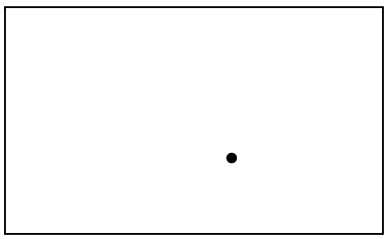
`\DrawBoundary` erzeugt einen Rahmen um das Bild.

`\CloseGraph` beendet die Grafik.

So erhält man z.B. mit

```
\InitGraph{5}{3}{3}{1}{1cm}
\DrawBoundary
\BigPointAt(0,0)
\CloseGraph
```

folgendes Bild:



Nach der Initialisierung sind alle Koordinaten-Angaben als Benutzer-Koordinaten zu verstehen. Im Beispiel dürfen sie von $x = -3$ bis $x = +2$ und von $y = -1$ bis $y = +2$ gehen, und es sind jeweils Zentimeter gemeint.

Vorsicht!! MGA-TEX merkt nicht, wenn Ihre Zeichnung über den Bild-Rand oder gar über den Blatt-Rand hinausragt. Wenn der Bildschirm-Treiber den Fehler nicht abfängt, kann es zu katastrophalen Folgen kommen! Das war z.B. bei einer früheren Version der Fall, die unter emTEX (DOS) lief. Außerdem produziert MGA-TEX (bis jetzt noch) keine sinnvollen Fehlermeldungen. Wundern Sie sich also

nicht, wenn ein einfacher Syntax-Fehler irgendwelche kryptischen Meldungen wie „`Missing Number, treated as zero ...`“ erzeugt.

§ 2 Punkte und Strecken

`\PointAt`(x, y), `\MedPointAt`(x, y) und `\BigPointAt`(x, y) zeichnet jeweils an der Stelle (x, y) einen kleineren, mittleren oder dicken Punkt. Der Cursor wird auf diese Position gesetzt.

`\Point` , `\MedPoint` und `\BigPoint` tun das Gleiche, aber jeweils an der momentanen Position des Cursors.

Mit `\SetPointSymbol`(n) kann ein anderes Punktymbol eingestellt werden, der Befehl `\ResetPointSymbol` entspricht dem Befehl `\SetPointSymbol`(0). Der Punkt sollte dann mit `\Point` oder `\PointAt`(x, y) aufgerufen werden. Es folgt eine Tabelle der möglichen Punkt-Typen:

$t =$	0	1	2	3	4	5	6	7	8	9	10	11	12
Punkt:	.	•	•	×	+	○	□	◇	⊙	⊕	⊗	●	·

Dabei entsprechen die Nummern 0, 1 und 2 den Befehlen `\Point`, `\MedPoint` und `\BigPoint`, Nummer 12 liefert einen noch kleineren Punkt. Die Nummer 11 ist für spätere Zwecke reserviert.

`\MoveTo`(x, y) bewegt den Cursor an die Stelle (x, y), `\MoveDirection`(α, r) bewegt ihn von der momentanen Cursorposition (x_0, y_0) nach $(x_0 + r\cos(\alpha), y_0 + r\sin(\alpha))$.

`\Store`(Name) speichert die momentanen Cursor-Koordinaten unter dem angegebenen Namen. `\MoveToLoc`(Name) bewegt den Cursor auf die unter dem angegebenen Namen gespeicherte Position.

`\LineAt`(x_1, y_1, x_2, y_2) zeichnet eine gerade Linie von (x_1, y_1) nach (x_2, y_2).
`\LineTo`(x, y) zeichnet eine solche Linie von der momentanen Cursorposition nach (x, y). In beiden Fällen steht der Cursor anschließend bei dem Endpunkt der Linie.

`\LineDirection`(α, r) zeichnet eine gerade Linie von der momentanen Cursor-Position (x_0, y_0) nach $(x_0 + r\cos(\alpha), y_0 + r\sin(\alpha))$. Dabei muß der Winkel α in Grad angegeben werden.

`\LineToLoc`(Name) zeichnet eine Linie von der momentanen Cursor-Position zu der unter dem angegebenen Namen gespeicherten Position.

`\RelativeLine`$(\Delta x, \Delta y)$ zeichnet eine Linie von der momentanen Cursor-Position (x_0, y_0) nach $(x_0 + \Delta x, y_0 + \Delta y)$.

`\LatexLine`$(\Delta x, \Delta y)(l)$ zeichnet eine Linie (von der momentanen Cursor-Position aus), mit der durch $(\Delta x, \Delta y)$ gegebenen Steigung. Die Länge der Linie ist durch

ihre Projektion l auf die x-Achse festgelegt, genau wie beim `\line`-Befehl in der LaTeX-Picture-Umgebung.

Mit den Befehlen `\SetDotted`, `\SetDashed` und `\SetThick{`x`pt}` erreicht man, daß alle nachfolgenden Linien gepunktet, gestrichelt oder dick (in der Dicke x`pt`) gezeichnet werden. Mit `\SetNormal` hebt man diese Befehle wieder auf. Die Dicke kann übrigens auch in einer anderen Maßeinheit (z.B. mm) angegeben werden.

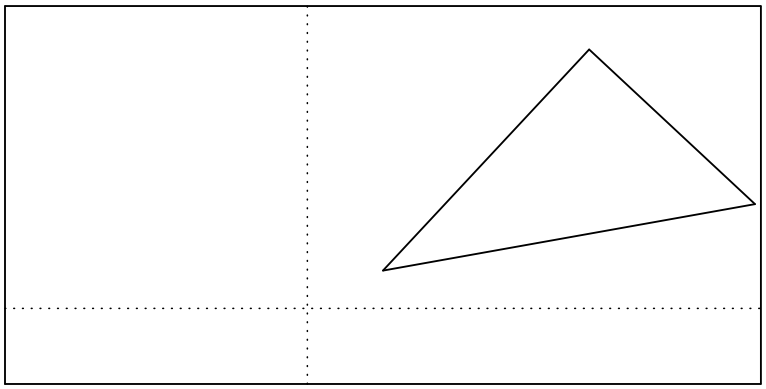
Es gibt noch eine andere Methode, Linien in einem besonderen Stil zu zeichnen:

`\DottedLineTo`(x_0, y_0),
`\DottedLineAt`(x_1, y_1, x_2, y_2),
`\DottedLineToLoc`(Name),
und genauso mit `Dashed...` Dabei wird nach dem Befehl automatisch wieder in den vorher gültigen Modus zurückgeschaltet. Bei dicken Linien gibt es eine Besonderheit: Die Befehle `\ThickLineTo`(x_0, y_0), `\ThickLineAt`(x_1, y_1, x_2, y_2) und `\ThickLineToLoc`(Name) verwenden automatisch die Dicke 1.2`pt`.

Beispiel: (rechtwinkliges Dreieck, mit Seiten 5, 4 und 3, um ca. 10° geneigt)

Der Winkel an der längeren Kathete wird vorher berechnet, er beträgt ungefähr 37°. Also ist diese Kathete um etwa 47° gegen die x-Achse geneigt, die andere dann um $180 + 47 + 90 = 317°$.

```
\InitGraph{10}{5}{4}{1}{1cm}
\DrawBoundary
\SetDotted
\LineAt(-4,0,6,0)
\LineAt(0,-1,0,4)
\SetNormal
\MoveTo(1,0.5)
\Store(Anfang)
\LineDirection(47,4)
\LineDirection(317,3)
\LineToLoc(Anfang)
\CloseGraph
```

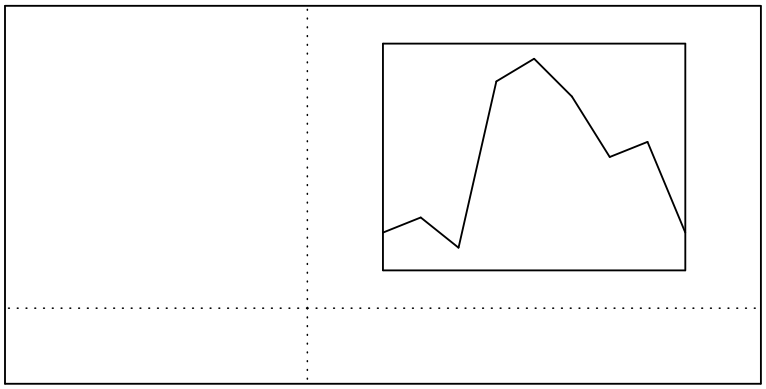


Der Befehl \PolyLine$(x_1, y_1)(x_2, y_2) \ldots (x_n, y_n)$ liefert einen Streckenzug mit den angegebenen Stützpunkten. Spezialfälle sind \Rectangle$(x_1, y_1, x_2, y_2)$ (achsenparalleles Rechteck mit linker unterer Ecke bei (x_1, y_1) und rechter oberer Ecke bei (x_2, y_2)) und \Triangle$(x_1, y_1)(x_2, y_2)(x_3, y_3)$ (Dreieck mit den angegebenen Ecken).

Beispiel: (Linienzug in einem Rechteck)

```
\InitGraph{10}{5}{4}{1}{1cm}
\DrawBoundary
\SetDotted
\LineAt(0,0,6,0)
\LineAt(0,0,-4,0)
\LineAt(0,0,0,4)
\LineAt(0,0,0,-1)
\SetNormal
\Rectangle(1,0.5,5,3.5)
\PolyLine(1,1)(1.5,1.2)(2,0.8)(2.5,3)%
(3,3.3)(3.5,2.8)(4,2)(4.5,2.2)(5,1)
\CloseGraph
```

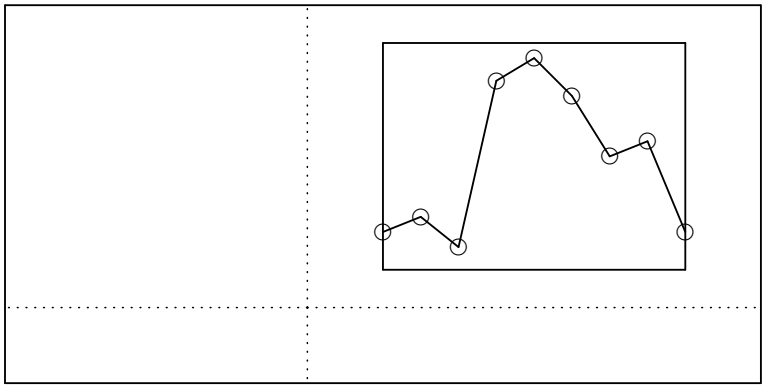
ergibt folgendes Bild:



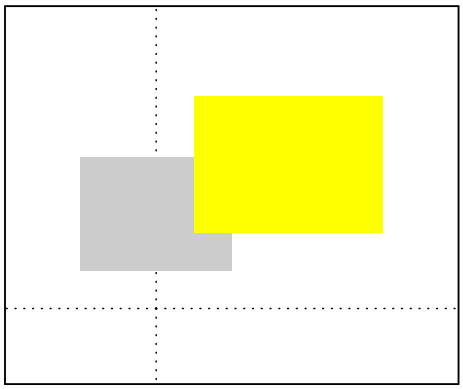
Benutzt man statt \PolyLine die Befehlskombination

```
\SetPointSymbol(5)
\JoinPoints(1,1)(1.5,1.2)(2,0.8)(2.5,3)%
(3,3.3)(3.5,2.8)(4,2)(4.5,2.2)(5,1)
\ResetPointSymbol
```

so erhält man folgendes Bild:



`\PaintBox`(b,h) zeichnet ein grau (oder nach einem entsprechenden Befehl farbig) gefülltes (achsenparalleles) Rechteck der Breite b und der Höhe h, dessen linke untere Ecke an der momentanen Cursor-Position liegt.
`\PaintBoxAt`$(x_0,y_0)(b,h)$ bewegt den Cursor zuvor nach (x_0,y_0).



```
\InitGraph{6}{5}{2}{1}{1cm}
...
\PaintBoxAt(-1,0.5)(2,1.5)
\SetYellow
\PaintBoxAt(0.5,1)(2.5,1.8)
\CloseGraph
```

§ 3 Kreise und Ellipsen

`\Circle`(r) zeichnet einen Kreis um die momentane Cursor-Position mit Radius r,
`\CircleAt`$(x_0,y_0)(r)$ bewegt den Cursor zuvor nach (x_0,y_0).

`\PaintDisk`(r) bzw. `\PaintDiskAt`$(x_0,y_0)(r)$ zeichnet einen grau (oder farbig) gefüllten Kreis.

`\EllipticArc`$(a,b)(\alpha,\delta)$ zeichnet einen Ellipsenbogen um die Cursor-Position herum, mit den Halbachsen a und b. α ist der Winkel-Parameter des Anfangspunktes, δ die **Differenz** zum Winkel-Parameter des Endpunktes (also nicht der Endwinkel). Beides muß im Gradmaß angegeben werden. Man beachte, daß der Punkt $(x_0 + a\cdot\cos(\alpha), y_0 + b\cdot\sin(\alpha))$ auf einer Ellipse mit $a \neq b$ vom Mittelpunkt aus nicht unter dem Winkel α gesehen wird, sondern unter dem Winkel β mit

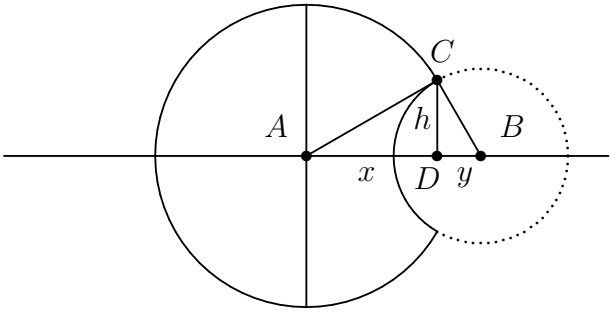
$$\tan(\beta) = \frac{b}{a}\cdot\tan(\alpha).$$

Ist $a = b$, so liegt ein Kreisbogen vor, und der Winkel-Parameter gibt den tatsächlichen Winkel gegen die positive x-Achse wieder, d.h. in diesem Fall ist $\beta = \alpha$.

`\Ellipse`(a, b) zeichnet eine komplette Ellipse mit den Halbachsen a und b.

Die Befehle `\EllipticArcAt`$(x_0, y_0)(a, b)(\alpha, \delta)$ und `\EllipseAt`$(x_0, y_0)(a, b)$ verstehen sich nun von selber.

Beispiel: (Kreise und Ellipsen) Es soll u.a. ein „angeknabberter“ Kreis gezeichnet werden. Dazu müssen einige Größen berechnet werden. Ein großer Kreis mit Radius $= 2$ soll mit einem kleineren Kreis geschnitten werden, dessen Mittelpunkt außerhalb des großen Kreises liegt. Die Mittelpunkte A und B mögen auf einer Geraden parallel zur x-Achse liegen, der obere Schnittpunkt C soll von A aus unter 30° gesehen werden, und von B aus unter 60°. Die Höhe h im (rechtwinkligen) Dreieck ABC auf die Hypotenuse treffe diese im Punkt D und schneide zwei Abschnitte x und y aus ihr aus.



Hier muss zunächst etwas gerechnet werden: Mit $\angle DAC = 30°$ und $\angle DBC = 60°$ ergibt sich: $x = 2 \cdot \cos(30°) \approx 1.732$, $h = 2 \cdot \sin(30°) = 1$ und $y = h \cdot \tan(30°) \approx 0.5773$, also $x + y = 2.3093$.

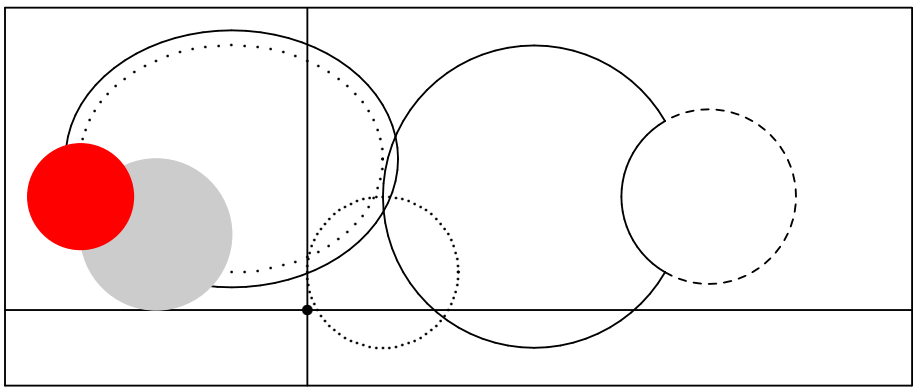
Das Bild lässt sich nun folgendermaßen erzeugen:

```
\InitGraph{8}{4}{4}{2}{1cm}
\LineAt(-4,0,4,0)
\LineAt(0,-2,0,2)
\BigPointAt(0,0) \Text[tl]{$A$}
\EllipticArcAt(0,0)(2,2)(30,300)
\EllipticArcAt(2.3093,0)(1.155,1.155)(120,120)
\SetDotted
\EllipticArcAt(2.3093,0)(1.155,1.155)(0,120)
\EllipticArcAt(2.3093,0)(1.155,1.155)(240,120)
\SetNormal
\BigPointAt(2.3093,0) \Text[tr]{$B$}
\BigPointAt(1.732,1) \TextAt(1.8,1)[t]{$C$}
\LineAt(1.732,1,1.732,0) \BigPoint
\MoveTo(1.6,0.1) \Text[b]{$D$}
\LineAt(0,0,1.732,1) \LineTo(2.3093,0)
\TextAt(0.8,0.1)[b]{$x$}
\TextAt(2.1,0.1)[b]{$y$}
\TextAt(1.9,0.5)[l]{$h$}
```

```
\CloseGraph
```

Ein anderes Bild ergibt die Befehlsfolge

```
\InitGraph{12}{5}{4}{1}{1cm}
\DrawBoundary
\LineAt(-4,0,8,0)
\LineAt(0,-1,0,4)
\BigPointAt(0,0)
\EllipticArcAt(3,1.5)(2,2)(30,300)
\EllipticArcAt(5.3093,1.5)(1.155,1.155)(120,120)
\SetDashed
\EllipticArcAt(5.3093,1.5)(1.155,1.155)(0,120)
\EllipticArcAt(5.3093,1.5)(1.155,1.155)(240,120)
\SetDotted
\CircleAt(1,0.5)(1)
\EllipseAt(-1,2)(2,1.5)
\SetNormal
\EllipseAt(-1,2)(2.2,1.7)
\PaintDiskAt(-2,1)(1)
\SetRed
\PaintDiskAt(-3,1.5)(0.7)
\CloseGraph
```



§ 4 Pfeile

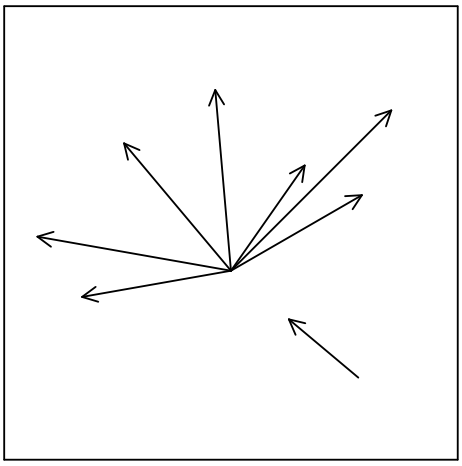
`\ArrowAt`(x_1, y_1, x_2, y_2) (bzw. `\DrawArrow`(x_1, y_1, x_2, y_2) in Versionen vor 2.1) zeichnet einen Pfeil von (x_1, y_1) nach (x_2, y_2).
`\ArrowHeadAt`(x_1, y_1, x_2, y_2) (bzw. `\DrawArrowHead`(x_1, y_1, x_2, y_2) in Versionen vor 2.1) zeichnet nur die Spitze des Pfeils.

Die Befehle `\ArrowDirection`(α, r), `\ArrowToLoc`(Name) und `\RelativeArrow`$(\Delta x, \Delta y)$ verstehen sich nun von selbst.

Bei \ReverseArrowToLoc(Name) und \ReverseArrowDirection(α, r) zeigt der Pfeil in die entgegengesetzte Richtung.

Beispiel: (Pfeile)

```
\InitGraph{6}{6}{3}{2.5}{1cm}
\DrawBoundary
\ArrowDirection(30,2)
\MoveTo(0,0)
\ArrowDirection(45,3)
\MoveTo(0,0)
\ArrowDirection(55,1.7)
\MoveTo(0,0)
\ArrowDirection(95,2.4)
\MoveTo(0,0)
\ArrowDirection(130,2.2)
\MoveTo(0,0)
\ArrowDirection(170,2.6)
\MoveTo(0,0)
\ArrowDirection(190,2)
\MoveTo(0,0)
\MoveDirection(340,1)
\ReverseArrowDirection(340,1.2)
\CloseGraph
```



§ 5 Beschriftung

\TextAt(x_0, y_0)[*pos*]{*Zeile*} setzt den Text der Zeile an die angegebene Stelle. Der Parameter *pos* gibt an, wie der Text dort positioniert wird:

pos-Parameter	Richtung
c	zentriert
r	rechts
tr	rechts oben
t	oben

pos-Parameter	Richtung
tl	links oben
l	links
bl	links unten
b	unten
br	rechts unten

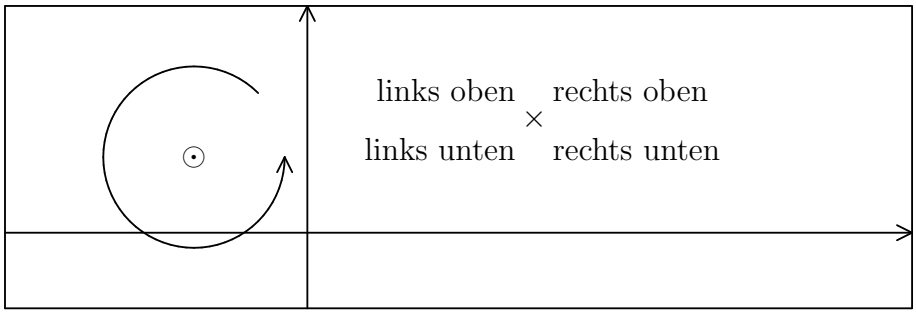
Dabei ist es egal, ob man z.B. **tr** oder **rt** schreibt. Standard ist **c**. Sind mehrere Zeilen zu schreiben, so arbeitet man am besten mit einer Minipage.

\Text[*pos*]{*Zeile*} setzt den Text an die momentane Cursor-Position.

Beispiel: (Beschriftung)

```
\InitGraph{12}{4}{4}{1}{1cm}
\DrawBoundary
\ArrowAt(-4,0,8,0) \ArrowAt(0,-1,0,3)
\EllipticArcAt(-1.5,1)(1.2,1.2)(45,315)
```

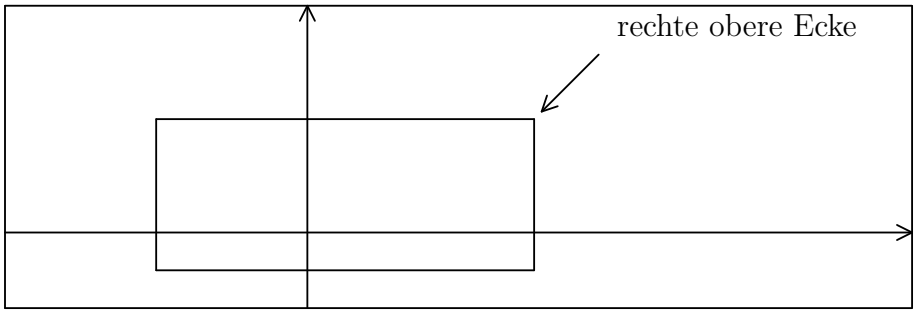
```
\ArrowHeadAt(-0.3,-1,-0.3,1)
\SetPointSymbol(8) \PointAt(-1.5,1)
\SetPointSymbol(3) \PointAt(3,1.5)
\Text[tr]{rechts oben}
\Text[br]{rechts unten}
\Text[tl]{links oben}
\Text[bl]{links unten}
\CloseGraph
```



`\ArrowText`[*pos*](*d*){*Zeile*} zeichnet einen Pfeil mit Spitze an der momentanen Cursur-Position. Der Parameter *pos* gibt die Richtung an, aus der der Pfeil kommt, *d* beeinflußt die Länge des Pfeils (bezogen auf die Benutzer-Einheit, bei Diagonalen wird die Projektion gemessen) und der Text erscheint am Schaft-Ende des Pfeils.

Beispiel: (Text mit Pfeil)

```
\InitGraph{12}{4}{4}{1}{1cm}
\DrawBoundary
\ArrowAt(-4,0,8,0) \ArrowAt(0,-1,0,3)
\Rectangle(-2,-0.5,3,1.5)
\MoveTo(3.1,1.6)
\ArrowText[tr](1){rechte obere Ecke}
\CloseGraph
```



Bemerkung: `\SetTextDist`{*d*pt} legt den Abstand der Beschriftung vom Punkt fest, `\ResetTextDist` stellt den Standardwert her (7pt).

Kapitel 3

Farbe

§ 1 Die Farbräume

Mit Version 2.2/2.3 kann nun endlich auch farbig gezeichnet werden. Voraussetzung ist die Wahl der Option „col“.

Es stehen drei Farbsysteme zur Verfügung:

1. `\GRAYColorspace` stellt Grautöne ein, mit `\SetGRAYColor`(g) wird ein spezieller Grauton gewählt. Dabei kann g Werte zwischen 0 (Schwarz) und 1 (Weiß) annehmen.

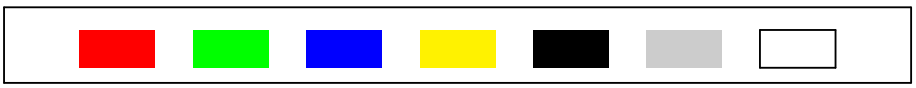
2. `\RGBColorspace` stellt den RGB-Farbraum ein. Grundfarben sind Rot, Grün und Blau. Die Farben werden „additiv“ gemischt, d.h. alle drei Grundfarben zusammen ergeben Weiß. Das RGB-System ist besonders gut für die Darstellung auf dem Bildschirm geeignet.

   Mit `\SetRGBColor`(r, g, b) wird ein spezieller RGB-Farbton gewählt. Die Werte r, g und b müssen zwischen 0 und 1 liegen. Mit $r = 1$, $g = 0$ und $b = 0$ erhält man z.B. die Grundfarbe Rot. Sind alle drei Werte gleich, so ergibt sich ein Grauton.

   Beispiele:

   `\SetRGBColor`$(1, 0, 0)$ liefert die Grundfarbe Rot,
   `\SetRGBColor`$(0, 1, 0)$ liefert die Grundfarbe Grün,
   `\SetRGBColor`$(0, 0, 1)$ liefert die Grundfarbe Blau,
   `\SetRGBColor`$(1, 0.95, 0)$ liefert einen Gelb-Ton,
   `\SetRGBColor`$(0, 0, 0)$ liefert Schwarz und `\SetRGBColor`$(0.8, 0.8, 0.8)$ einen hellen Grau-Ton. Mit `\ResetColor` schaltet man die Farbe wieder aus.

   Die Farben sehen dann folgendermaßen aus:

3. `\CMYKColorspace` stellt den CMYK-Farbraum ein. Grundfarben sind Cyan (ein grünlicher Blauton), Magenta (spätestens seit den Erfolgen des Rad-Teams „Telekom“ bekannt), Yellow (Gelb) und Black (Schwarz). Die Farben werden „subtraktiv“ gemischt, so wie man es vom Schulmalkasten kennt. Deshalb eignet sich das CMYK-System besonders gut für den Druck.

Hier ergeben sich die bekannten Mischungen „$M + Y$ →Orange“, „$M + C$ →Violett“, „$C + Y$ →Grün“. Mischungen gleicher Anteile von Cyan, Magenta und Yellow sollten theoretisch Grautöne ergeben. Reale Druckfarben verhalten sich aber nicht so, man erhält eher Brauntöne. Deshalb wird ein Schwarz-Anteil dazugemischt. Jeder kennt das von seinem Tintenstrahldrucker.

Theoretisch können die verschiedenen Farbsysteme ineinander umgerechnet werden. Praktisch braucht man bei der Umrechnung von RGB nach CMYK noch zwei Zusatzfunktionen, genannt „black generation“ und „undercolor removal“, die vom jeweiligen Ausgabegerät und den Druckfarben abhängen. Diese Funktionen konnte ich natürlich nicht berücksichtigen. Eine Rolle spielen sie nur, wenn man einen der vordefinierten (und an den RGB-Farben orientierten) Farbnamen verwenden, aber im CMYK-System arbeiten möchte. Wer genaue Kontrolle über die Farben bei der Druckausgabe braucht, sollte die Farben mit `\SetCMYKColor`(c, m, y, k) direkt definieren.

Die Befehle

```
\InitGraph{12}{1}{0}{0}{1cm}
\DrawBoundary
\SetCMYKColor(1,0,0,0) \PaintBoxAt(1,0.2)(1,0.5)
\SetCMYKColor(0,1,0,0) \PaintBoxAt(2.5,0.2)(1,0.5)
\SetCMYKColor(0,0,1,0) \PaintBoxAt(4,0.2)(1,0.5)
\SetCMYKColor(0,1,1,0) \PaintBoxAt(5.5,0.2)(1,0.5)
\SetCMYKColor(0,0,0,1) \PaintBoxAt(7,0.2)(1,0.5)
\SetCMYKColor(0,0,0,0.2) \PaintBoxAt(8.5,0.2)(1,0.5)
\ResetColor
\Rectangle(10,0.2,11,0.7)
\CloseGraph
```

liefern das Bild:

Zur Erleichterung gibt es einige Kurz-Befehle für gewisse Standardfarben, wie `\SetRed`, `\SetYellow` usw. Wenn zuvor kein Farbraum festgelegt wurde, wird automatisch RGB gewählt, denn die Farbtöne sind am Bildschirm getestet worden, wirken also nach Anwahl des RGB-Farbraumes am besten. Man kann aber auch die Farbräume GRAY oder CMYK festlegen. Dann werden entsprechend umgerechnete Farben dargestellt. Ob das im CMYK-Fall den Erwartungen entspricht, hängt vom jeweiligen Ausgabegerät ab.

Hier kommen die vordefinierten Farben im RGB-Farbraum:

\SetYellow		\SetLemon	
\SetRed		\SetDarkgreen	
\SetBlue		\SetOlive	
\SetGreen		\SetPurple	
\SetViolet		\SetMagenta	
\SetOrange		\SetAqua	
\SetDarkbrown		\SetCyan	
\SetLightbrown		\SetSalmon	
\SetWhite		\SetWheat	
\SetBlack		\SetPink	
\SetDarkgrey			
\SetLightgrey			

§ 2 Farbige Zeichnungen

Nachdem eine Farbe festgelegt wurde, erscheinen alle Linien in dieser Farbe. Für farbige Beschriftung gibt es eigene Befehle, die im nächsten Abschnitt besprochen werden.

Neben den Befehlen \PaintBox$(b,h)$, \PaintBoxAt$(x_0,y_0)(b,h)$, \PaintDisk$(r)$ und \PaintDiskAt$(x_0,y_0)(r)$ stehen noch die folgenden Befehle zur farbigen Gestaltung von Flächen zur Verfügung:

\PaintTrapez$(x_1,y_1,z_1)(x_2,y_2,z_2)$ produziert ein gefülltes Trapez mit den Ecken (x_1,y_1), (x_2,y_2), (x_2,z_2) und (x_1,z_1).

\PaintTriangle$(x_1,y_1)(x_2,y_2)(x_3,y_3)$ produziert ein gefülltes Dreieck mit den angegebenen drei Ecken. Damit lässt sich fast jedes gefüllte Polygon erzeugen.

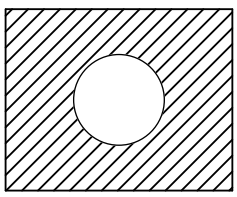
Die Ausführung dieser genialen Zeichnung sei dem Leser überlassen.

Mit dem Befehl `\PaintSektorAt`$(x_0, y_0)(r)(t_1, t_2)$ wird ein (gefüllter) Kreissektor gezeichnet, mit dem Mittelpunkt (x_0, y_0), dem Radius r und zwischen den Winkeln t_1 und t_2. Damit lassen sich Tortendiagramme erstellen.

Dazu braucht man folgende Befehle:

```
\InitGraph{10}{4}{5}{2}{1cm}
\SetDarkgrey
\PaintSektorAt(0,0)(1.2)(0,95)
\SetPurple
\PaintSektorAt(0,0)(1.2)(95,110)
\SetRed
\PaintSektorAt(0,0)(1.2)(110,170)
\SetGreen
\PaintSektorAt(0,0)(1.2)(170,230)
\SetYellow
\PaintSektorAt(0,0)(1.2)(230,278)
\SetWheat
\PaintSektorAt(0,0)(1.2)(278,360)
\CloseGraph
```

Die Farbe Weiß scheint überflüssig zu sein, ist es aber nicht. Unter den Optionen `[ps]` und `[pd]` überdecken sich die Farben in der Reihenfolge, in der sie aufgerufen werden (bei `[ep]` ist die Reihenfolge leider etwas willkürlich). Man kann damit z.B. erreichen:



(Der Befehl `\ShadedRectangle` wird später erklärt)

```
\InitGraph{5}{3}{1}{0}{1cm}
\ShadedRectangle(0,0,3,2.4)[4]
\SetRGBColor(1,1,1)
\PaintDiskAt(1.5,1.2)(0.6)
\ResetColor
\Rectangle(0,0,3,2.4)
\CircleAt(1.5,1.2)(0.6)
\CloseGraph
```

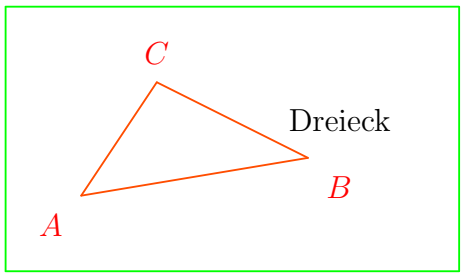
§ 3 Farbiger Text

Die Schrift im Befehl `\Text{...}` bleibt zunächst schwarz. Soll sie auch farbig erscheinen, so muss man das mit `\TextColor{...}` angeben. Bei der Beschriftung von Koordinatenachsen (vgl. nächster Abschnitt) steuert man dies mit `\SetColoredLabels` und `\NoColoredLabels`

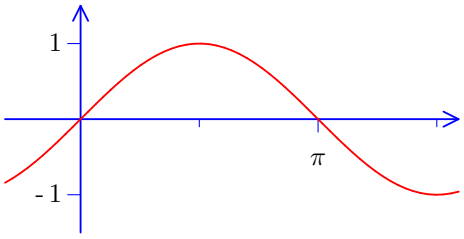
Letzteres ist der Standard. `\SetColoredLabels` sorgt für farbige Beschriftung der Koordinatenachsen, `\NoColoredLabels` schaltet wieder auf Schwarz um.

Hier ist zunächst ein einfaches Beispiel:

```
\InitGraph{6}{3.5}{0}{0.5}{1cm}
\SetGreen\DrawBoundary
\SetOrange
\Triangle(1,0.5)(4,1)(2,2)
\SetRed
\TextAt(1,0.5)[bl]{\TextColor{$A$}}
\TextAt(4,1)[br]{\TextColor{$B$}}
\TextAt(2,2)[t]{\TextColor{$C$}}
\TextAt(3.5,1.5)[r]{Dreieck}
\CloseGraph
```

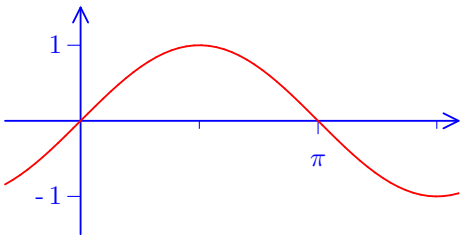


Hier ist eine Zeichnung mit schwarzer Beschriftung der Koordinatenachsen:




```
\InitGraph{6}{3}{1}{1.5}{1cm}
\SetBlue
\UserLine[1](2,1){$\pi$}
\Coordinates(\EinPi,2)(1,1)
\SetRed
\Sinus(1,1,0,0)(-1,5)[60]
\CloseGraph
```

Und hier ein Beispiel mit farbiger Beschriftung der Achsen:




```
\InitGraph{6}{3}{1}{1.5}{1cm}
\SetBlue
\SetColoredLabels
\UserLine[1](2,1){$\pi$}
\Coordinates(\EinPi,2)(1,1)
\SetRed
\Sinus(1,1,0,0)(-1,5)[60]
\CloseGraph
```

Kapitel 4

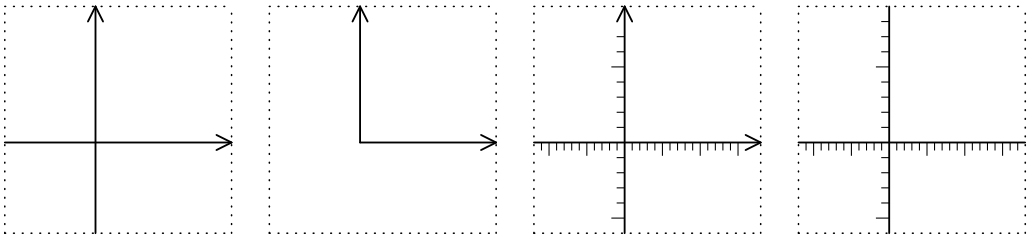
Koordinaten

§ 1 Koordinatenachsen

Koordinatenkreuze werden folgendermaßen erzeugt:

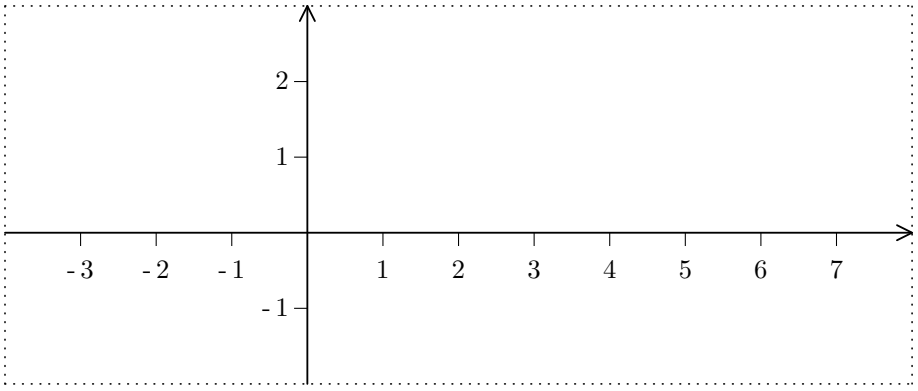
`\Axes` zeichnet ein Achsenkreuz, `\HalfAxes` nur die positiven Halbachsen. `\StickAxes`$(\Delta_x, n_x)(\Delta_y, n_y)$ versieht die Achsen zusätzlich mit mit großen Markierungen (im Abstand Δ_x bzw. Δ_y) und dazwischen mit $n_x - 1$ bzw. $n_y - 1$ kleinen Markierungen. Auch `\HalfStickAxes` ist möglich.

Der Befehl `\NoArrowAxes` sorgt dafür, dass die Pfeilspitzen weggelassen werden. Mit `\ResetArrowAxes` kann man sie wieder herstellen.

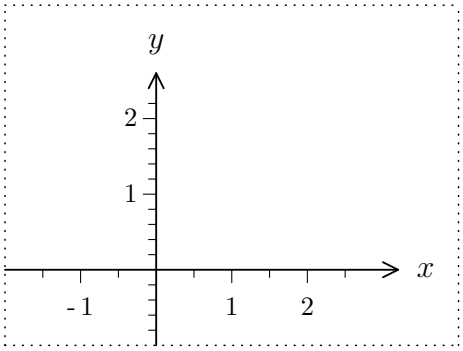


`\Coordinates`$(\Delta_x, n_x)(\Delta_y, n_y)$ arbeitet wie `\StickAxes`, schreibt aber zusätzlich Zahlen an die großen Markierungen. Auch `\HalfCoordinates`$(\ldots)(\ldots)$ ist möglich.

`\StandardCoordinates` entspricht `\Coordinates`$(1,1)(1,1)$ und ergibt:



Alle sechs Befehle (`\Axes`, `\HalfAxes`, `\StickAxes`, ..., `\HalfCoordinates`) können auch vorne mit einem `B` und hinten mit dem Zusatz $< b_x, b_y >$ versehen werden. Dann enden die Pfeilspitzen der Achsen nicht an den Bildgrenzen, sondern schon bei $(b_x, 0)$ und $(0, b_y)$. Damit ist folgendes möglich:




```
\InitGraph{6}{4.5}{2}{1}{1cm}
...
\BCoordinates(1,2)(1,5)<3.2,2.6>
\TextAt(3.2,0)[r]{$x$}
\TextAt(0,2.6)[t]{$y$}
...
\CloseGraph
```

§ 2 Veränderung der Beschriftung

Der Befehl \UserLine$[n](z,t)\{Text\}$ verändert die Achsenbeschriftung. Dabei bedeuten die Parameter:

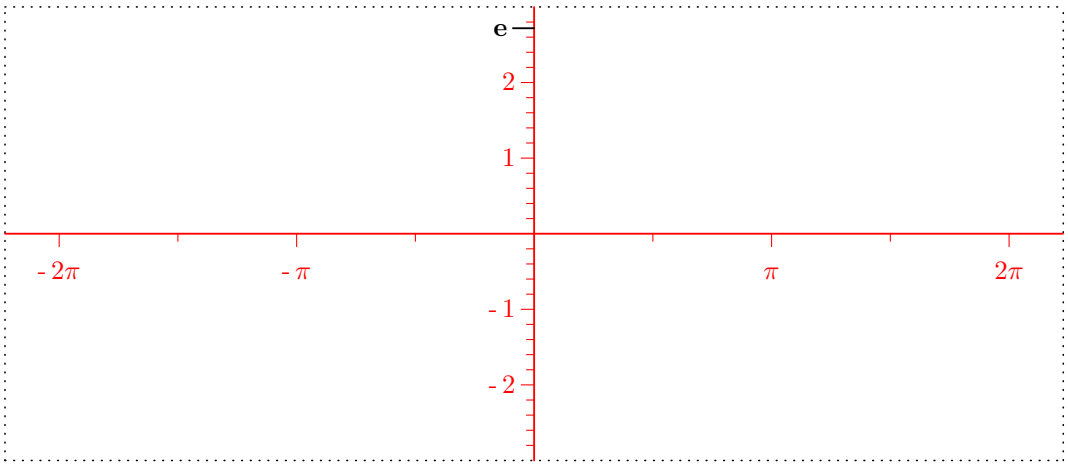
n	1	2	3	4
Bedeutung	pos. x-A.	pos. y-A.	neg. x-A.	neg. y-A.

z ist die Nummer der ersten Position, an der eine Zahl ausgedruckt werden soll, t ist die Nummer der ersten Position, an der der folgende Text angefügt werden soll. Dadurch sind Beschriftungen wie 1cm, 2cm, ... oder π, 2π, ... möglich.

\AllUserLines$(z,t)\{Text\}$ behandelt alle 4 Halbachsen auf die gleiche Weise.

\XLabelAt$(x_0)\{Text\}$ bzw. \YLabelAt$(y_0)\{Text\}$ erlaubt es, einzelne Markierungen mit Text an der x- bzw. y-Achse anzubringen.

Hier ist ein Beispiel:



Folgendermaßen wird das verwirklicht:

```
\InitGraph{14}{6}{7}{3}{1cm}
...
\SetRed
\SetColoredLabels
\NoArrowAxes
```

```
\UserLine[1](2,1){$\pi$} \UserLine[3](2,1){$\pi$}
\SetLabelTicks{8pt} \SetLabelDist{10pt}
\Coordinates(\EinPi,2)(1,5)
\ResetColor
\YLabelAt(\Euler){$\mathbf{e}$}
...
\CloseGraph
```

Die meisten Größen können vom Benutzer verändert werden:

`\Set...`{*d*pt} setzt jeweils die Größe auf *d* `pt` fest, `\Reset...` stellt den Standardwert her. Statt `pt` kann man natürlich auch andere in LaTeX erlaubte Einheiten benutzen. An Stelle der Pünktchen kann man einsetzen:

1. `BigTicks` (Standard=5pt) : Länge der großen Markierungen.
2. `SmallTicks` (Standard=3pt) : Länge der kleinen Markierungen.
3. `LabelTicks` (Standard=5pt) : Länge der Extra-Markierungen.
4. `LabelDist` (Standard=7pt) : Abstand der Schrift bei Extra-Markierungen.

`\SetLabelTicks`{1cm} setzt z.B. die Länge der Extra-Markierungen auf `1cm`, mit `\ResetLabelTicks` wird die Länge auf `5pt` zurückgesetzt.

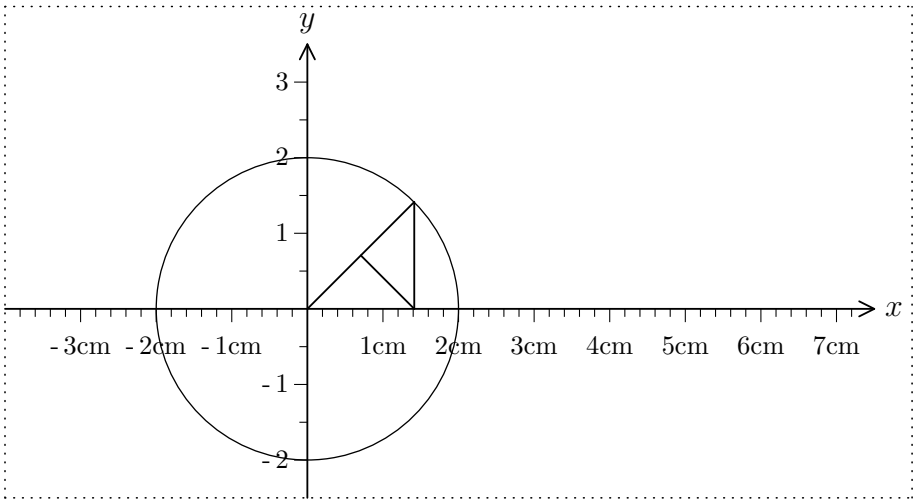
Das Makro `\EinPi` steht für den Wert der Zahl π und `\Euler` für die Eulersche Zahl e. Insgesamt stehen folgende Konstanten zur Verfügung:

`\Wurzelzwei`= 1.4142136 = $\sqrt{2}$,
`\Inversewurzelzwei`= 0.7071067 = $1/\sqrt{2}$
`\Logzwei`= 0.6931471 = $\ln(2)$,
`\Euler`= 2.7182818 = e,
`\InverseEuler`= 0.3678794 = $1/e$,
`\LgEuler`= 0.4342944 = $\log_{10}(e)$,
`\EinPi`= 3.14159 = π,
`\PiHalbe`= 1.570795 = $\pi/2$,
`\ZweiPi`= 6.28318 = $2 \cdot \pi$,
`\PiViertel`= 0.78540 = $\pi/4$.

Man kann diese Konstanten auch in anderen Zeichenbefehlen verwenden. Hier ist ein Beispiel:

```
\InitGraph{12}{6.5}{4}{2.5}{1cm}
...
\UserLine[1](1,1){cm}
\UserLine[3](1,1){cm}
```

```
\BCoordinates(1,5)(1,2)<7.5,3.5>
\TextAt(7.4,0)[r]{$x$}
\TextAt(0,3.4)[t]{$y$}
\CircleAt(0,0)(2)
\LineAt(0,0,\Wurzelzwei,\Wurzelzwei)
\LineTo(\Wurzelzwei,0)
\LineTo(\Inversewurzelzwei,\Inversewurzelzwei)
...
\CloseGraph
```



Kapitel 5

Kurven

§ 1 Standardfunktionen

MGA-TEX kann einige Funktionen plotten. Intern existiert eine Routine, die beliebige parametrisierte ebene Kurven $t \mapsto (x(t), y(t))$ zeichnen kann. Die Standard-Funktionen werden nun als Kurven $t \mapsto (t, f(t))$ für $t_1 \le t \le t_2$ gezeichnet. Ein optionaler Parameter n gibt an, wieviele Streckenstücke zum Approximieren des Graphen benutzt werden. Standard ist $n = 50$, für einen ersten Eindruck reicht meistens $n = 10$.

Folgende Funktionen stehen zur Verfügung.

1. \Parabel$(a, b, c)(t_1, t_2)[n]$ ergibt $f(t) := at^2 + bt + c$.
2. \Kubik$(a, b, c, d)(t_1, t_2)[n]$ ergibt $f(t) := at^3 + bt^2 + ct + d$.
3. \Sinus$(a, b, c, d)(t_1, t_2)[n]$ ergibt $f(t) := a \cdot \sin(bt + c) + d$.
4. \Exponential$(a, b)(t_1, t_2)[n]$ ergibt $f(t) := a \cdot e^{bt}$.
5. \Tangens$(t_1, t_2)[n]$ ergibt $f(t) := \tan(t)$.
   \Cotangens$(t_1, t_2)[n]$ ergibt $f(t) := \cot(t)$.
   Der Benutzer muß bei Tangens und Cotangens selbst darauf achten, daß die Werte innerhalb der Zeichenfläche bleiben!
6. \ArcTangens$(t_1, t_2)[n]$ ergibt $f(t) := \arctan(t)$.
7. \Logarithmus$(t_1, t_2)[n]$ ergibt $f(t) := \ln(t)$.
8. \ArcSinus$(t_1, t_2)[n]$ ergibt $f(t) := \arcsin(t)$.
9. \UpperSemiEllipseAt$(x_0, y_0)(a, b)(t_1, t_2)$ ergibt

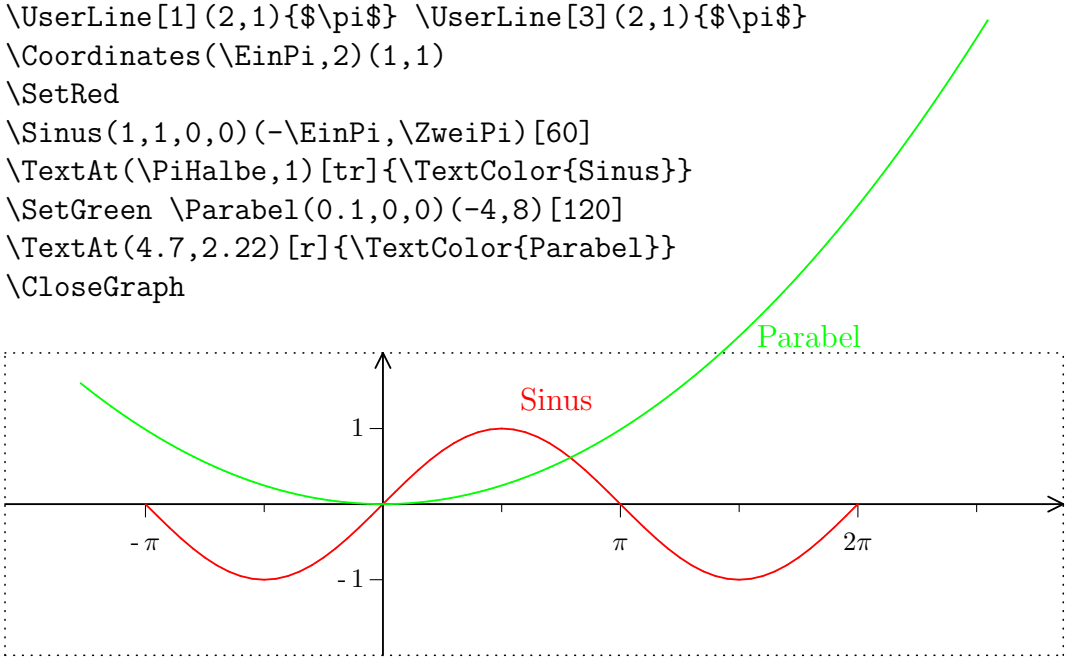
$$u(t) := (x_0 + t,\ y_0 + b \cdot \sqrt{1 - (\frac{t}{a})^2}),\ \text{für}\ -a \le t_1 \le t \le t_2 \le a.$$

10. \LowerSemiEllipseAt$(x_0, y_0)(a, b)(t_1, t_2)$ ergibt

$$u(t) := (x_0 + t,\ y_0 - b \cdot \sqrt{1 - (\frac{t}{a})^2}),\ \text{für}\ -a \le t_1 \le t \le t_2 \le a.$$

Beispiel: (Sinuskurve und Parabel)

```
\InitGraph{14}{4}{5}{2}{1cm}
...
\UserLine[1](2,1){$\pi$} \UserLine[3](2,1){$\pi$}
\Coordinates(\EinPi,2)(1,1)
\SetRed
\Sinus(1,1,0,0)(-\EinPi,\ZweiPi)[60]
\TextAt(\PiHalbe,1)[tr]{\TextColor{Sinus}}
\SetGreen \Parabel(0.1,0,0)(-4,8)[120]
\TextAt(4.7,2.22)[r]{\TextColor{Parabel}}
\CloseGraph
```



§ 2 Bezierkurven und Splines

\Bezier$[n](x_1,y_1)(x_2,y_2)(x_3,y_3)$ zeichnet eine quadratische Bezier-Kurve mit den angegebenen Stützpunkten. Der optionale Parameter n gibt an, aus wievielen Strecken die Kurve zusammengesetzt werden soll. $n = 50$ ist Standard.

\ResBezier$[n](x_1,y_1)(x_2,y_2)(x_3,y_3)(t_1,t_2)$ zeichnet eine quadratische Bezier-Kurve mit den angegebenen Stützpunkten, aber nur für $t_1 \le t \le t_2$. Dabei wird die folgende Parametrisierung benutzt:

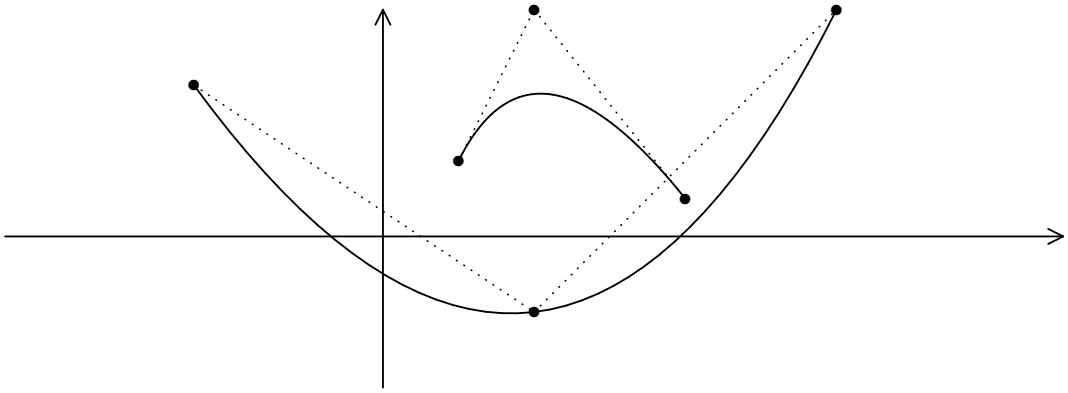
$$B(t) := (1-t)^2 \cdot \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + 2t(1-t) \cdot \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} + t^2 \cdot \begin{pmatrix} x_3 \\ y_3 \end{pmatrix}.$$

\Spline$[n](x_1,y_1)(x_2,y_2)(x_3,y_3)$ zeichnet einen quadratischen Spline durch die drei angegebenen Punkte.

Beispiel: (Bezier-Kurve und Spline)

```
\InitGraph{14}{5}{5}{2}{1cm}
\Axes
\SetDotted
\BigPointAt(1,1)
\LineTo(2,3) \BigPoint
\LineTo(4,0.5) \BigPoint
\BigPointAt(-2.5,2)
\LineTo(2,-1) \BigPoint
```

```
\LineTo(6,3) \BigPoint
\SetNormal
\Bezier(1,1)(2,3)(4,0.5)
\Spline(-2.5,2)(2,-1)(6,3)
\CloseGraph
```



§ 3 Benutzerdefinierte Kurven

\UserCurve$(t_1, t_2)\{Funktionsdefinition\}$ dient dazu, eigene Kurven der Gestalt $t \mapsto (x(t), y(t))$ mit $t_1 \leq t \leq t_2$ zu definieren. Dazu sind genauere Kenntnisse über TEX und MGA-TEX erforderlich.

Hier ist der Versuch einer Gebrauchsanweisung:

Die Funktionsdefinition dient dazu, die Hilfsroutine \CALCPOINTS zu deklarieren, die intern immer benutzt wird, um eine Funktion $t \mapsto (x(t), y(t))$ zu zeichnen. In \CALCPOINTS werden aus einem Parameter t die Werte $x(t)$ und $y(t)$ berechnet. Wie kann man in TEX rechnen?

TEX benutzt sogenannte „dimensionierte Register“ für Rechenfunktionen. Ich nenne sie auch *Variable*. Sie beinhalten Werte wie „1cm“ oder „30pt“. In MGA-TEX wird intern alles in pt umgerechnet. Der Parameter t wird von \CALCPOINTS aus dem dimensionierten Register \TLAUF übernommen. Will man den Wert dieser Variablen ohne die Dimensionierung (**pt**) in eine gewöhnliche TEX-Speicherstelle, etwa mit Namen \tlauf , stellen, so geschieht das mit

```
\Wertvon<\TLAUF>nach<\tlauf>
```

Will man umgekehrt den Wert wieder als dimensionierte Größe in das Register \TLAUF schreiben, so kann dies z.B. folgendermaßen geschehen:

```
\TLAUF=\tlauf\ONEPOINT
```

Dabei ist \ONEPOINT ein Register, das genau den Wert 1pt enthält.

\CALCPOINTS muss nach der Berechnung von $x(t)$ und $y(t)$ diese Werte in die Variablen \XEINS und \YEINS übertragen.

Zur Berechnung selbst können die Rechenbefehle von TEX verwendet werden, und auch eine Reihe zusätzlich von MGA-TEX bereitgestellter Befehle. Als Variablen stehen außer `\TLAUF`, `\XEINS` und `\YEINS` noch zur Verfügung: `\XZWEI`, `\YZWEI`, `\ZEINS`, `\ZZWEI`, `\AEINS`, `\BEINS`, `\AZWEI`, `\BZWEI`, `\XHILF`, `\YHILF`, `\THILF`, `\UEINS`, `\VEINS`, `\UZWEI`, `\VZWEI`.

Zum Rechnen stehen folgende Routinen zur Auswahl:

`\advance` a `by` b liefert die Addition $a := a + b$.

`\Divide<`a`>by<`b`>forming<`c`>` liefert die Division $c := a/b$.

`\Hypotenusevon<`a`>und<`b`>nach<`c`>` liefert die Berechnung $c := \sqrt{a^2 + b^2}$.

`\Kathetevon<`a`>und<`b`>nach<`c`>` liefert die Berechnung $c := \sqrt{a^2 - b^2}$.

`\ARCUSvon<`w`>nach<`x`>` bedeutet: $x := \arc(w)$ (Umrechnung von Grad ins Bogenmaß), dabei muss w eine undimensionierte ganze Zahl sein.
`\DEGREEvon<`x`>nach<`w`>` liefert umgekehrt die Umrechnung vom Bogenmaß in Grad.

`\COSvon<`x`>nach<`y`>` bedeutet: $y := \cos(x)$.

`\SINvon<`x`>nach<`y`>` bedeutet: $y := \sin(x)$.

`\QUADPOL<`a, b, c`>von<`x`>nach<`y`>` bedeutet: $y := ax^2 + bx + c$.

`\KUBPOL<`a, b, c, d`>von<`x`>nach<`y`>` bedeutet: $y := ax^3 + bx^2 + cx + d$.

`\ATANvon<`x`>nach<`y`>` bedeutet: $y := \arctan(x)$.

`\EXPvon<`x`>nach<`y`>` bedeutet: $y := \exp(x)$.

`\LOGNATvon<`x`>nach<`y`>` bedeutet: $y := \ln(x)$.

`\LOGTENvon<`x`>nach<`y`>` bedeutet: $y := \log_{10}(x)$.

Hier ist ein einfaches **Beispiel:**

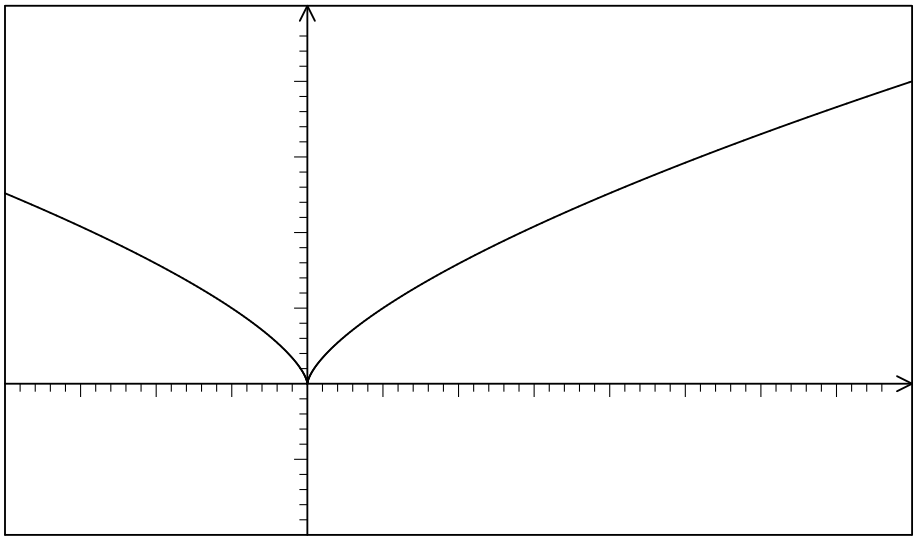
Es soll eine „Neilsche Parabel“ gezeichnet werden: $(x(t), y(t)) = (t^3, t^2)$. Die zugehörige Prozedur kann folgendermaßen aussehen:

$$\begin{aligned} y_1 &:= t \cdot t \\ x_1 &:= y_1 \cdot t \end{aligned}$$

Die Multiplikationen kann man ganz einfach durchführen, indem man eine undimensionierte Zahl vor ein dimensioniertes Register schreibt. Das sieht dann folgendermaßen aus:

```
\InitGraph{12}{7}{4}{2}{1cm}
\DrawBoundary
\StickAxes(1,5)(1,5)
\UserCurve(-1.58740,2)[300]{%
```

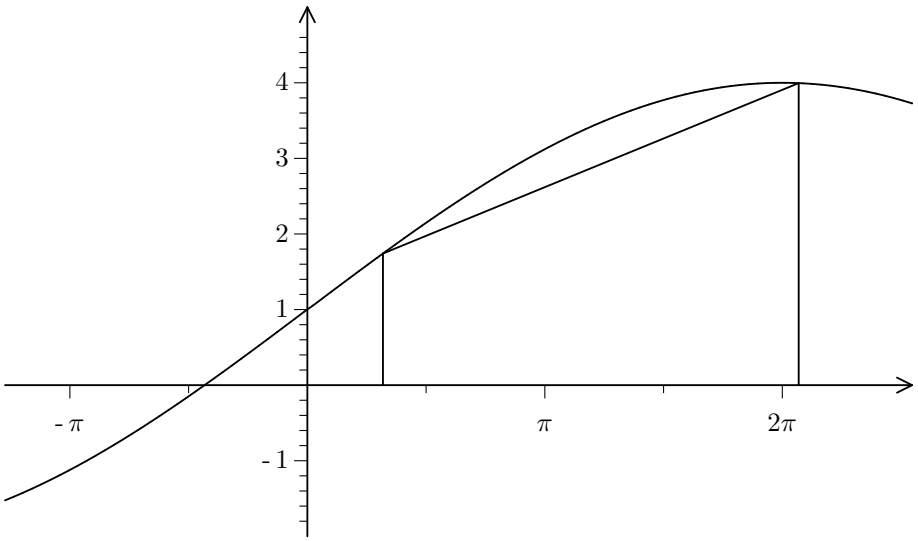
```
   \Wertvon<\TLAUF>nach<\tlauf>
   \YEINS=\tlauf\TLAUF\relax
   \Wertvon<\YEINS>nach<\tlauf>
   \XEINS=\tlauf\TLAUF\relax}
\CloseGraph
```



Zusätzlich gibt es den Befehl \GoToValue$(t)\{Funktionsdefinition\}$.
Er erlaubt es, den Cursor direkt zu der Stelle $(x(t), y(t))$ zu bewegen. Für die Funktionsdefinition gelten die gleichen Regeln wie oben.

Beispiel:

```
\InitGraph{12}{7}{4}{2}{1cm}
\UserLine[1](2,1){$\pi$} \UserLine[3](2,1){$\pi$}
\Coordinates(\EinPi,2)(1,5)
\Sinus(3,0.25,0,1)(-4,8)[100]
\GoToValue(1){%
  \XEINS=\TLAUF
  \SINFKT<3,0.25,0,1>von<\TLAUF>nach<\YEINS>}
\Store(Anfang)
\LineTo(1,0)
\GoToValue(6.5){%
  \XEINS=\TLAUF
  \SINFKT<3,0.25,0,1>von<\TLAUF>nach<\YEINS>}
\Store(Ende)
\LineTo(6.5,0)
\MoveToLoc(Anfang)
\LineToLoc(Ende)
\CloseGraph
```



Dabei wurde die Funktion `\SINFKT<`a, b, c, d`>von<`x`>nach<`y`>` benutzt:

$$y := a \cdot \sin(bx + c) + d.$$

Inzwischen gibt es auch noch die folgenden Funktionen:

`\TANvon<`x`>nach<`y`>maximal<`z`>` liefert: $y := \tan(x)$. Die Variable z hat zur Zeit keine Bedeutung (man kann z.B. `1pt` einsetzen), sie ist reserviert für eine spätere Programmversion.

`\COTvon<`x`>nach<`y`>maximal<`z`>` liefert $y := \cot(x)$. Für z gilt das Gleiche wie beim Tangens.

`\ASINvon<`x`>nach<`y`>` liefert $y := \arcsin(x)$.

`\CARTvon<`w, r`>nach<`x, y`>` berechnet die kartesischen Koordinaten (x, y) aus den Polarkoordinaten (w, r), wobei der Winkel w im Gradmaß angegeben werden muss.

`\POLvon<`x, y`>nach<`w, r`>` berechnet umgekehrt Polarkoordinaten aus kartesischen Koordinaten. Dabei wird intern die folgende Routine benutzt:

`\Direction`(x_1, y_1, x_2, y_2)`To`(n) ordnet dem Vektor von (x_1, y_1) nach (x_2, y_2) eine Zahl n zu, die nach folgendem Schema die Richtung angibt:

```
      6
   2     4
7     0     8
   1     3
      5
```

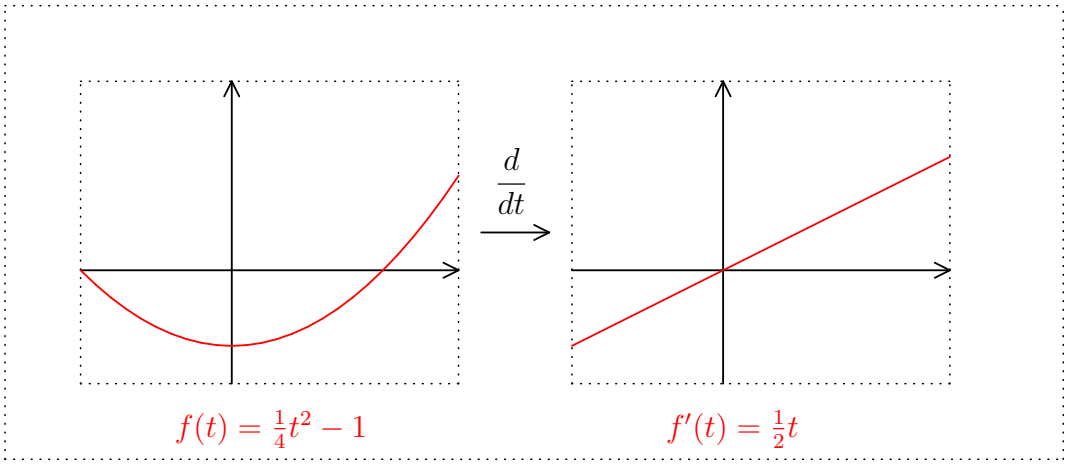
Kapitel 6

Teilbilder

§ 1 Fenster

`\OpenWindowAt`$(x_0, y_0)(b, h)(x_N, y_N)$ eröffnet ein „Fenster“ (d.h. ein Teilbild). Die linke untere Ecke liegt bei (x_0, y_0), die Zahlen b und h geben Breite und Höhe an, x_N, y_N sind die Offset-Werte eines neuen Nullpunktes, relativ zur linken unteren Ecke. Anschließend beziehen sich alle Angaben auf den neuen Nullpunkt.

Mit `\CloseWindow` wird das Fenster wieder geschlossen und man kehrt zu den alten Koordinaten zurück. Hier ist ein Beispiel:




```
\InitGraph{14}{6}{0}{0}{1cm}
...
\OpenWindowAt(1,1)(5,4)(2,1.5)
\SetDotted \DrawBoundary \SetNormal \Axes
\SetRed \Parabel(0.25,0,-1)(-2,3)[25] \ResetColor
\CloseWindow
%
\ArrowAt(6.3,3,7.2,3)
\TextAt(6.7,3)[t]{$\DST \frac{d}{dt}$}
%
\OpenWindowAt(7.5,1)(5,4)(2,1.5)
\SetDotted \DrawBoundary \SetNormal \Axes
\SetRed \LineAt(-2,-1,3,1.5) \ResetColor
\CloseWindow
%
\SetRed
\TextAt(2,0.4)[r]{\TextColor{$f(t)=\frac{1}{4}t^2-1$}}
```

```
\TextAt(8.5,0.4)[r]{\TextColor{$f'(t)=\frac{1}{2}t$}}
\CloseGraph
```

Man beachte, dass Fenster **nicht** geschachtelt werden können!

§ 2 Put und Multiput

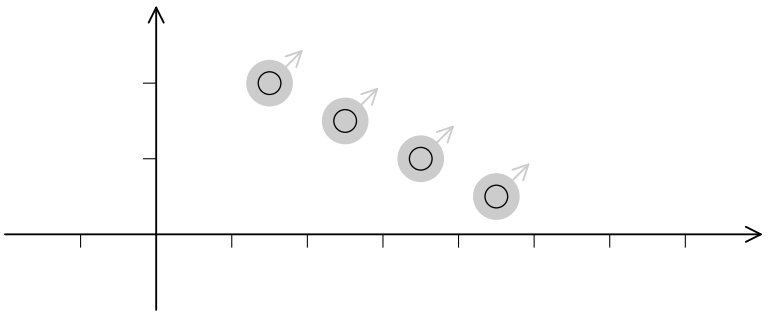
Der Befehl \MultiPut$(x_0, y_0)(\Delta x, \Delta y)\{n\}\{Objekte\}$ funktioniert wie in der LaTeX-Picture-Umgebung. Die angegebenen Objekte werden bei (x_0, y_0), $(x_0+\Delta x, y_0+\Delta y)$ etc. (insgesamt n-mal) gezeichnet. Das macht normalerweise nur bei Objekten Sinn, die nicht bei festen Koordinaten gezeichnet werden. Folgende Befehle können von \MultiPut verarbeitet werden:

\PaintDisk$(r)$, \PaintBox$(b, h)$,
\RelativeLine$(\Delta x, \Delta y)$, \LineDirection$(\alpha, r)$, \LatexLine$\Delta x, \Delta y)(l)$,
\Circle$(r)$, \Ellipse$(a, b)$, \EllipticArc$(a, b)(\alpha_0, \delta)$,
\RelativeArrow$(\Delta x, \Delta y)$, \ArrowDirection$(\alpha, r)$,
\ReverseArrowDirection(α, r), \Text[pos]$\{Text\}$.

Außerdem sind natürlich Befehle für Farbe und Linienstil möglich.

```
\InitGraph{10}{4}{2}{1}{1cm}
\StickAxes(1,1)(1,1)
\MultiPut(1.5,2)(1,-0.5){4}{%
  \SetLightgrey\PaintDisk(0.3)\ArrowDirection(45,0.6)
  \MoveDirection(225,0.6)\ResetColor\Circle(0.15)}
\CloseGraph
```

ergibt folgendes Bild:

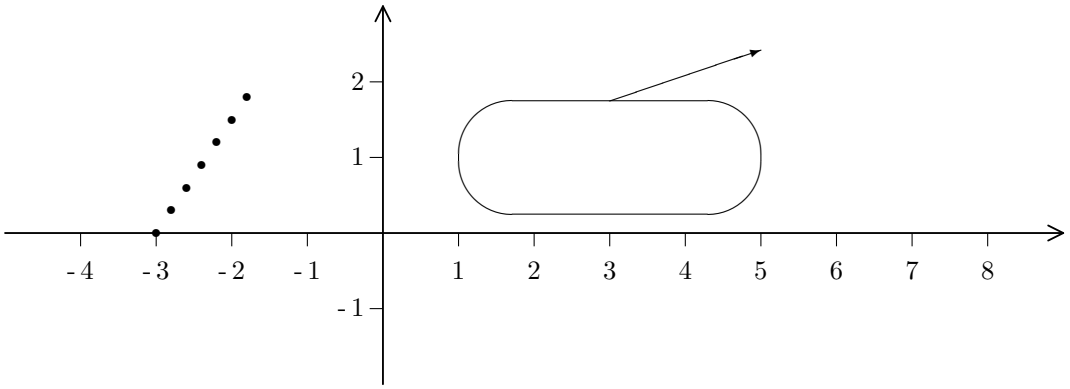


Man beachte, dass einige Befehle (wie \Circle oder \Ellipse) die Cursor-Position nicht verändern, andere aber wohl, wie etwa die \Line- und \Arrow-Befehle.
\LatexCommand{...} ermöglicht es, weiter die Befehle der LaTeX-Picture-Umgebung zu benutzen (z.B. für Ovale). Auch dabei sind alle Angaben in Benutzer-Koordinaten zu verstehen. Allerdings beeinflußt dieser Befehl nicht die Position des Grafik-Cursors.

Unter der Option [ep] sind als Argumente auch Befehle des epic-Paketes zugelassen (z.B. \matrixput oder \jput). Hier soll nicht näher darauf eingegangen werden,

weil ab Version 2.2 unter den Optionen `[ps]` und `[pd]` das epic-Paket nicht mehr geladen wird. **Beispiel:**

```
\InitGraph{14}{5}{5}{2}{1cm}
\StandardCoordinates
\LatexCommand{\put(3,1){\oval(4,1.5)}}
\LatexCommand{\multiput(-3,0)(0.2,0.3){7}{\circle*{0.1}}}
\LatexCommand{\put(3,1.75){\vector(3,1){2}}}
\CloseGraph
```

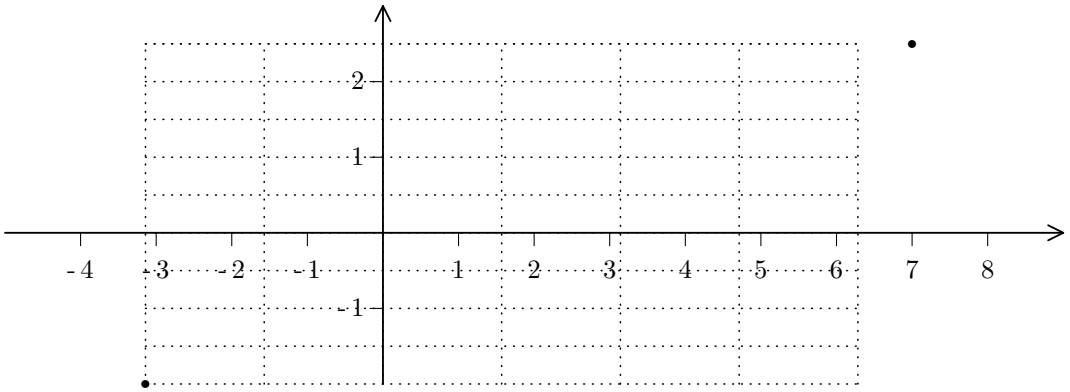


§ 3 Gitter

Mit `\GridAt`$(x_0, y_0)(\Delta x, \Delta y)(x_1, y_1)$ wird ein Gitter mit den Maschenweiten Δx, Δy gezeichnet. Linker unterer Eckpunkt ist (x_0, y_0). Der rechte obere Eckpunkt liegt möglichst nahe bei (x_1, y_1), das Gitter liegt aber auf jeden Fall im Innern des Rechtecks mit der linken unteren Ecke (x_0, y_0) und der rechten oberen Ecke (x_1, y_1).

Beispiel:

```
\InitGraph{14}{5}{5}{2}{1cm}
\StandardCoordinates \SetDotted
\GridAt(-\EinPi,-2)(\PiHalbe,0.5)(7,2.5)
\MedPointAt(-\EinPi,-2) \MedPointAt(7,2.5)
\CloseGraph
```



§ 4 Spezielle Objekte

\RightAngleAt$(x_0, y_0)(d)$ zeichnet bei (x_0, y_0) einen um d Grad gedrehten rechten Winkel (ohne die Schenkel). Es gibt vier Typen:

Der Typ wird mit \SetRightAngleType$\{n\}$ festgelegt, Standard ist $n = 1$ (linkes Beispiel), der Standard wird auch durch \ResetRightAngleType hergestellt. Die weiteren 3 Beispiele zeigen nacheinander die Fälle $n = 2$, $n = 3$ und $n = 4$.

\Angleof$(g)$At$(x_0, y_0)(d)$ zeichnet bei (x_0, y_0) einen Winkelbogen von g Grad, der beim Winkel von d Grad beginnt (ohne die Schenkel).

\NamedAngle$\{Name\}$of(g)At$(x_0, y_0)(d)$ arbeitet wie der vorige Befehl, beschriftet aber den Winkel zusätzlich.

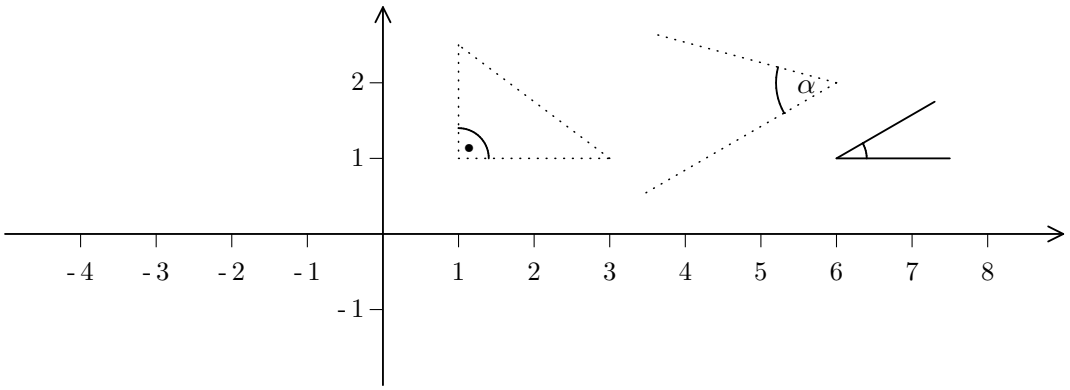
Mit \SetAngleDist$\{d$pt$\}$ bzw. \SetNamedAngleDist$\{d$pt$\}$ kann der Radius von Winkelbögen verändert werden. Mit \ResetAngleDist bzw. \ResetNamedAngleDist können die Standardwerte (4mm bzw. 8mm) wieder hergestellt werden.

Man beachte dabei:

- Bei \Angleof... wird der Abstand des Bogens durch \AngleDist gesteuert.
- Bei \NamedAngle... wird der Abstand des Bogens durch \NamedAngleDist gesteuert.
- Bei \NamedAngle... wird der Abstand der Beschriftung durch \AngleDist gesteuert.

Beispiel:

```
\InitGraph{14}{5}{5}{2}{1cm}
\StandardCoordinates
\SetDotted
\LineAt(1,1,3,1) \LineTo(1,2.5) \LineTo(1,1)
\MoveTo(6,2) \LineDirection(210,3)
\MoveTo(6,2) \LineDirection(165,2.5)
\SetNormal
\RightAngleAt(1,1)(0)
\NamedAngle{$\alpha$}of(45)At(6,2)(165)
\LineAt(6,1,7.5,1)
\MoveTo(6,1) \LineDirection(30,1.5)
\Angleof(30)At(6,1)(0)
\CloseGraph
```



`\PointList`$(x_1, y_1)(x_2, y_2)\ldots(x_n, y_n)$ zeichnet nacheinander alle Punkte der Liste.

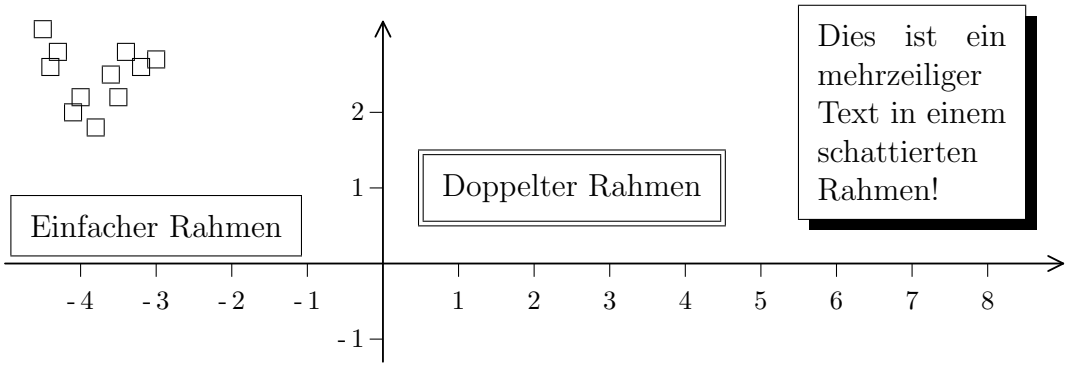
`\FrameAt`$(x_0, y_0)\{Text\}$ positioniert (zentriert) den angegebenen Text mit einem einfachen Rahmen.

`\DFrameAt`$(x_0, y_0)\{Text\}$ tut das gleiche, mit doppeltem Rahmen.

`\SFrameAt`$(x_0, y_0)\{Text\}$ tut das gleiche, mit einfachem Rahmen, der aber mit einem Schatten versehen ist.

Beispiel:

```
\InitGraph{14}{4.5}{5}{1.3}{1cm}
\StandardCoordinates
\SetPointSymbol(6)
\PointList(-4.5,3.1)(-4.3,2.8)(-4.4,2.6)(-4.1,2)(-4,2.2)(-3.8,1.8)%
  (-3.6,2.5)(-3.4,2.8)(-3.2,2.6)(-3.5,2.2)(-3,2.7)
\ResetPointSymbol
\FrameAt(-3,2){Einfacher Rahmen}
\DFrameAt(2.5,1){Doppelter Rahmen}
\SFrameAt(7,2){%
   \begin{minipage}{2.5cm}
     Dies ist ein mehrzeiliger Text in einem
     schattierten Rahmen!
   \end{minipage}}
\CloseGraph
```



Kapitel 7

Schraffuren und Schattierungen

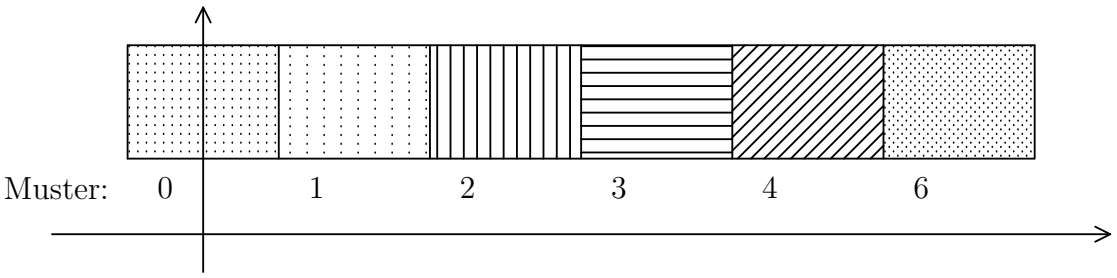
§ 1 Schraffierte Rechtecke und Trapeze

\ShadedRectangle$(x_1, y_1, x_2, y_2)[t]$ gibt ein gefülltes Rechteck (ohne Rand) aus. Der Typ t des Füllmusters ist wie folgt belegt:

Typ	Muster
0	enge Punkte
1	weite Punkte
2	vertikale Striche
3	horizontale Striche
4	aufsteigende Striche
5	aktuelle Farbe (oder hellgrau, wenn keine Farbe vereinbart ist)
6	doppelt (versetzte) enge Punkte

Beispiel 1:

```
\InitGraph{14}{3.5}{2}{0.5}{1cm}
\Axes \Rectangle(-1,1,11,2.5)
\ShadedRectangle(-1,1,1,2.5)[0]
\ShadedRectangle(1,1,3,2.5)[1]
\ShadedRectangle(3,1,5,2.5)[2]
\ShadedRectangle(5,1,7,2.5)[3]
\ShadedRectangle(7,1,9,2.5)[4]
\ShadedRectangle(9,1,11,2.5)[6]
\LineAt(1,1,1,2.5) \LineAt(3,1,3,2.5) \LineAt(5,1,5,2.5)
\LineAt(7,1,7,2.5) \LineAt(9,1,9,2.5)
\CloseGraph
```

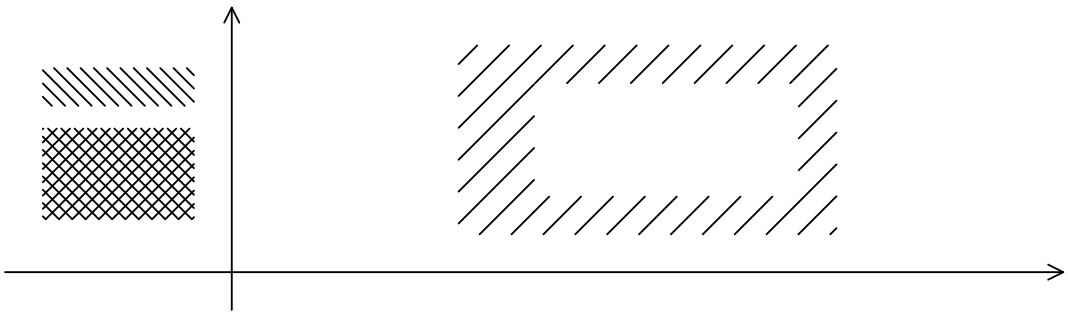


\ShadedRWR$(x_1, y_1, x_2, y_2)(u_1, v_1, u_2, v_2)[t]$ arbeitet wie \ShadedRectangle$(x_1, y_1, x_2, y_2)[t]$, läßt aber ein Rechteck mit den Ecken (u_1, v_1) und (u_2, v_2) frei.

Unter Verwendung einer Spiegelung (näheres zu Transformationen siehe Kapitel 8) kann man mit Muster 4 noch weitere Schraffuren erzeugen.

Beispiel 2:

```
\InitGraph{14}{4}{3}{0.5}{1cm}
\Axes
\begin{Translate}{-2.5}{0.7}
  \ShadedRectangle(0,0,2,1.2)[4]
    \begin{Transform}{1}{0}{0}{-1} % Spiegelung
      \ShadedRectangle(0,-1.2,2,0)[4]
      \ShadedRectangle(0,-2,2,-1.5)[4]
    \end{Transform}
\end{Translate}
\SetShadingDist{12pt} \ShadedRWR(3,0.5,8,3)(4,1,7.5,2.5)[4]
\CloseGraph
```



Achtung!! Mein Drucker ist bei diesen Beispielen bei der epic-Version mit der Meldung „Zu viele Daten“ ausgestiegen. Unter der Option `[ep]` sollte man möglichst auf Schraffuren verzichten oder zumindest einen großen Linienabstand einstellen.

Mit dem Befehl `\SetShadingDist`{*d*pt} kann man bei der aufsteigenden Schraffur den Abstand der Linien steuern. Der Standard (5pt) wird mit `\ResetShadingDist` wieder hergestellt.

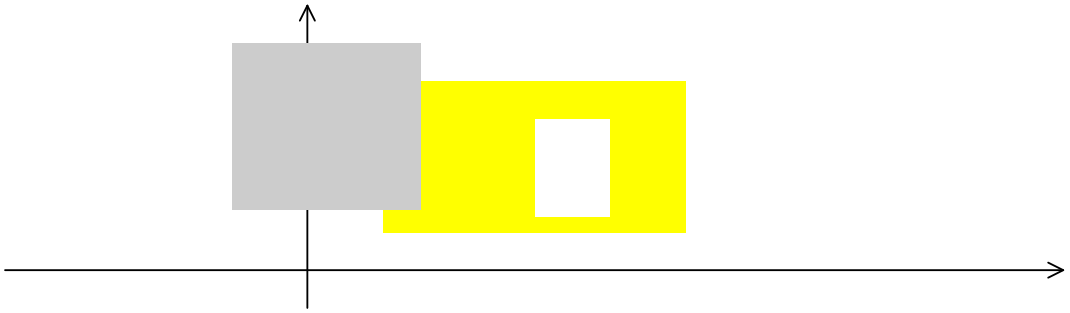
Im Schwarzweiß-Modus steht mit Muster 5 noch Grau zur Verfügung, im Farb-Modus natürlich alle erdenklichen Farben. Das ergibt Effekte, die man genauso mit dem (neueren) Befehl `\PaintBox` erreichen könnte).

Man beachte allerdings, dass farbige Flächen und Linien in ihrem Bereich alles überdecken, was vorher gezeichnet wurde. Die Reihenfolge der Zeichenbefehle ist entscheidend. Beim folgenden Beispiel wäre es daher ratsam gewesen, das Koordinatenkreuz erst am Schluss einzufügen.

Beispiel 3:

```
\InitGraph{14}{4}{4}{0.5}{1cm}
\Axes
\SetYellow
\ShadedRWR(1,0.5,5,2.5)(3,0.7,4,2)[5] % gelb
\ResetColor
```

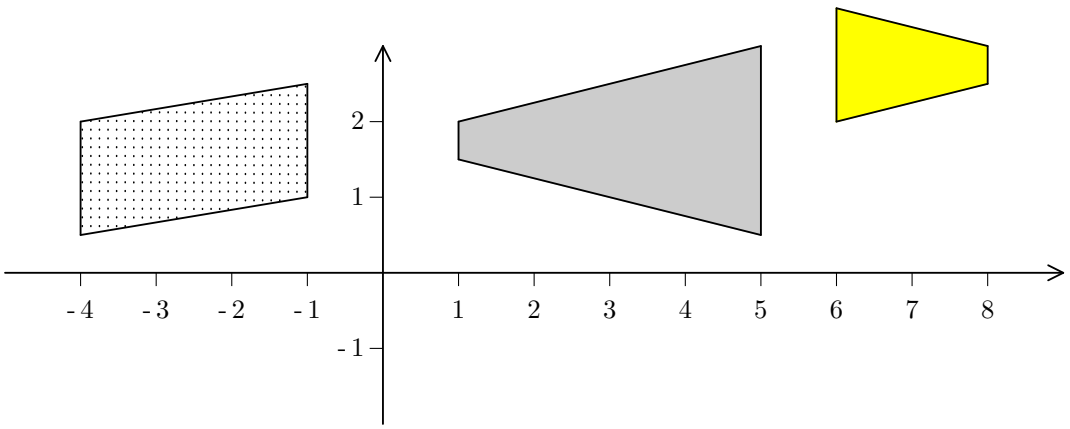
```
\ShadedRectangle(-1,0.8,1.5,3)[5] % grau
\CloseGraph
```



\ShadedDomain$(x_a, y_{a1}, y_{a2})(x_e, y_{e1}, y_{e2})[t]$ füllt ein Trapez-Gebiet zwischen den Strecken von (x_a, y_{a1}) nach (x_e, y_{e1}) und von (x_a, y_{a2}) nach (x_e, y_{e2}). Momentan sind nur die Füll-Typen 0, 1, 2 und (unter den Optionen `[ps]` und `[pd]`) noch 5 (aber nur mit Grau) implementiert. Allerdings erreicht man mit dem (neueren) Befehl \PaintTrapez$(x_a, y_{a1}, y_{a2})(x_e, y_{e1}, y_{e2})$ den gleichen Effekt, mit beliebiger Füllfarbe.

Beispiel:

```
\InitGraph{14}{5}{5}{2}{1cm}
\StandardCoordinates
\ShadedDomain(-4,0.5,2)(-1,1,2.5)[0]
\PolyLine(-4,0.5)(-1,1)(-1,2.5)(-4,2)(-4,0.5)
\ShadedDomain(1,1.5,2)(5,0.5,3)[5]
\PolyLine(1,1.5)(5,0.5)(5,3)(1,2)(1,1.5)
\SetYellow
\PaintTrapez(6,2,3.5)(8,2.5,3)
\ResetColor
\PolyLine(6,2)(8,2.5)(8,3)(6,3.5)(6,2)
\CloseGraph
```



§ 2 Flächen unter Funktionsgraphen

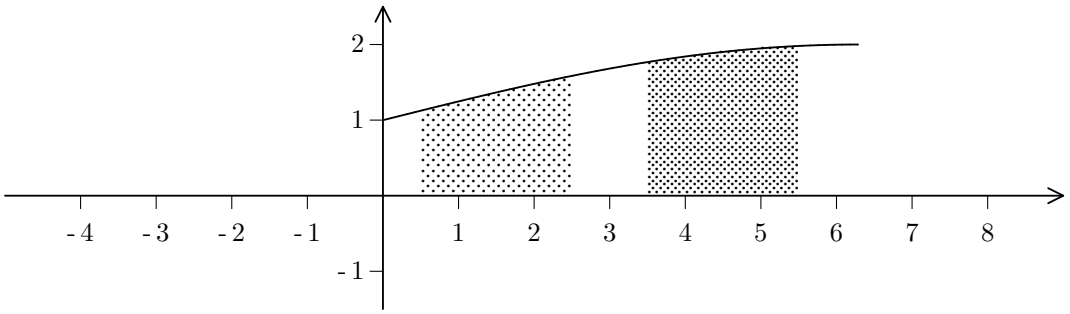
Besonders komfortabel sind die Befehle `\SetShadingUp` und `\SetShadingDown`, mit denen man - beginnend bei der x-Achse - aufwärts bzw. abwärts schattiert, bis zum Graphen der nächsten aufgerufenen Funktion. Mit `\ShadingOff` wird die Prozedur beendet. Es gibt nur ein Muster! So schattiert z.B. die Befehlsfolge

```
\SetShadingUp
\Sinus(1,0.25,0,1)(0.5,2.5)[30]
\ShadingOff
```

unter der Kurve $y = \sin(\frac{1}{4}x) + 1$ zwischen $x_1 = 0.5$ und $x_2 = 2.5$. Der Parameter $n = 30$ (und die Länge des Intervalls $[x_1, x_2]$) beeinflusst die Punktdichte. Der Graph der Funktion selbst wird dabei noch nicht gezeichnet, das muss man extra veranlassen.

Beispiel:

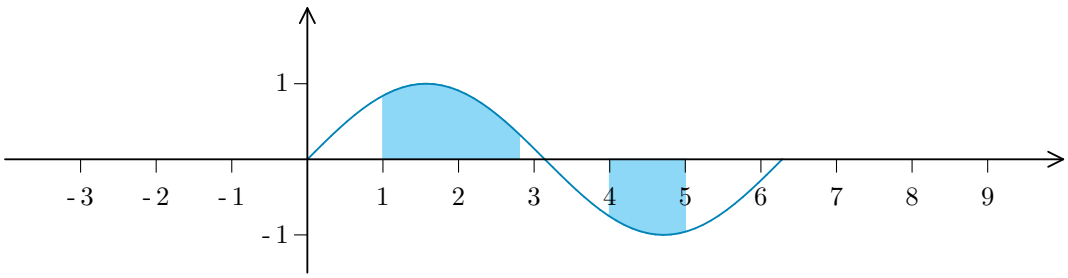
```
\InitGraph{14}{4}{5}{1.5}{1cm}
\StandardCoordinates
\SetShadingUp
\Sinus(1,0.25,0,1)(0.5,2.5)[30]
\Sinus(1,0.25,0,1)(3.5,4.5)[40]
\ShadingOff
\Sinus(1,0.25,0,1)(0,\ZweiPi)[60]
\CloseGraph
```



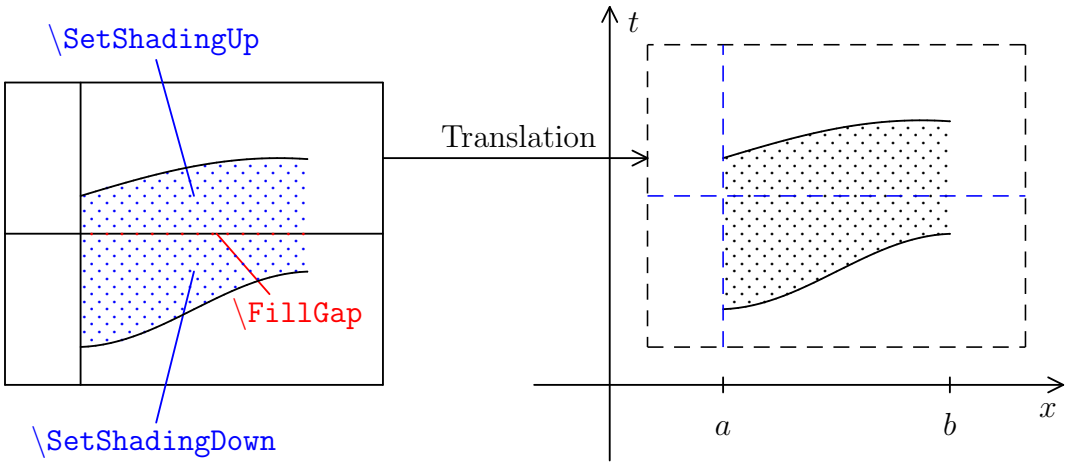
Mit Farbe geht alles etwas komfortabler. Zum Schattieren unterhalb und oberhalb eines Funktionsgraphen gibt es nur das **eine** Kommando `\SetColShading`. Ausgeschaltet wird es durch `\ColShadingOff`.

```
\InitGraph{14}{3}{4}{1.5}{1cm}
\SetCMYKColor(0.4,0,0,0)
\SetColShading
\Sinus(1,1,0,0)(1,2.8)[18]
\Sinus(1,1,0,0)(4,5)[10]
\ColShadingOff
\SetCMYKColor(1,0,0,0.3)
```

```
\Sinus(1,1,0,0)(0,\ZweiPi)[60]
\ResetColor
\StandardCoordinates
\CloseGraph
```



Um zwischen **zwei** Funktionsgraphen zu schattieren, sind ein paar Tricks nötig. Man muss eine horizontale Gerade zwischen den Funktionsgraphen finden. Dann denkt man sich diese Gerade als x-Achse und schattiert oberhalb und unterhalb der Achse. Da zwischen den beiden schattierten Gebieten einige Punkte fehlen, müssen diese durch den Befehl `\FillGap` ergänzt werden. Am Schluss schiebt man alles durch eine Translation in die richtige Position.

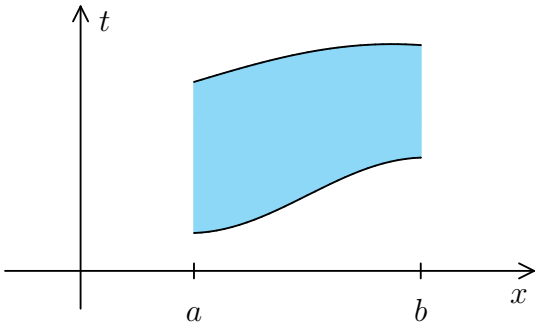


Das Bild auf der rechten Seite (ohne die gestrichelten Linien) erhält man mit

```
\InitGraph{7}{6}{1}{1}{1cm}
\Axes
\TextAt(5.8,0)[b]{$x$} \TextAt(0,4.8)[r]{$t$}
\LineAt(1.5,-0.1,1.5,0.1) \TextAt(1.5,-0.1)[b]{$\mathstrut a$}
\LineAt(4.5,-0.1,4.5,0.1) \TextAt(4.5,-0.1)[b]{$\mathstrut b$}
\begin{Translate}{1.5}{2.5}
\SetShadingUp
\Sinus(0.5,0.6,0,0.5)(0,3)[30]% Schattierung unter oberer Kurve
\FillGapAt(0,0)(0,3)[30]
\SetShadingDown
```

```
\Sinus(0.5,1,-1.5,-1)(0,3)[30]% Schattierung ueber unterer Kurve
\ShadingOff
\end{Translate}
\Sinus(0.5,0.6,-0.9,3)(1.5,4.5)[30]% obere Kurve
\Sinus(0.5,1,-3,1.5)(1.5,4.5)[30]% untere Kurve
\CloseGraph
```

Ähnlich funktioniert das farbige Schattieren zwischen zwei Funktionsgraphen. Hier müssen allerdings – anders als beim Schwarzweiß-Muster – keine Lücken gefüllt werden. Die folgende Zeichnung



erhält man mit

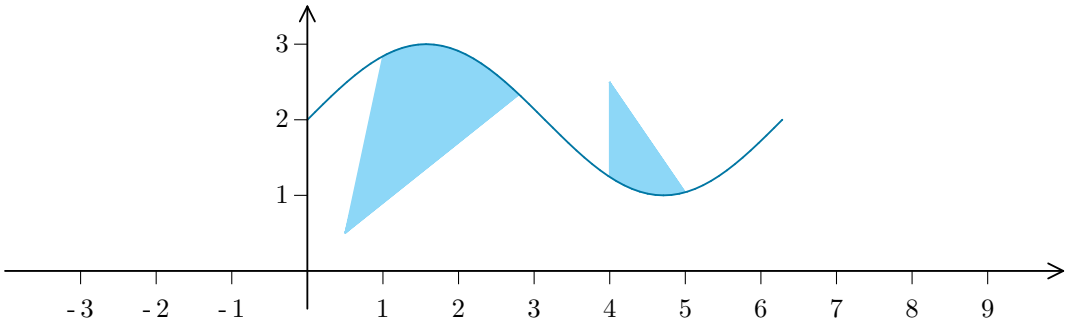
```
\InitGraph{7}{4}{1}{0.5}{1cm}
\Axes
\TextAt(5.8,0)[b]{$x$} \TextAt(0,3.3)[r]{$t$}
\LineAt(1.5,-0.1,1.5,0.1) \TextAt(1.5,-0.1)[b]{$\mathstrut a$}
\LineAt(4.5,-0.1,4.5,0.1) \TextAt(4.5,-0.1)[b]{$\mathstrut b$}
\begin{Translate}{1.5}{2}
\SetCMYKColor(0.4,0,0,0) \SetColShading
\Sinus(0.5,0.6,0,0.5)(0,3)[30]% Schattierung unter oberer Kurve
\Sinus(0.5,1,-1.5,-1)(0,3)[30]% Schattierung ueber unterer Kurve
\ColShadingOff
\ResetColor
\end{Translate}
\Sinus(0.5,0.6,-0.9,2.5)(1.5,4.5)[30]% obere Kurve
\Sinus(0.5,1,-3,1)(1.5,4.5)[30]% untere Kurve
\CloseGraph
```

Eine Besonderheit ist das Zeichnen von Kegeln unter einer Kurve. Zunächst muss mit \SetVertex$(x, y)$ die Kegelspitze festgelegt werden. Dann funktioniert es wie beim Schattieren, aber mit der Befehlsfolge \PaintCone ... \StopCone

```
\InitGraph{14}{4}{4}{0.5}{1cm}
\SetVertex(0.5,0.5)
\SetCMYKColor(0.4,0,0,0)
\PaintCone \Sinus(1,1,0,2)(1,2.8)[18] \StopCone
\SetVertex(4,2.5)
```

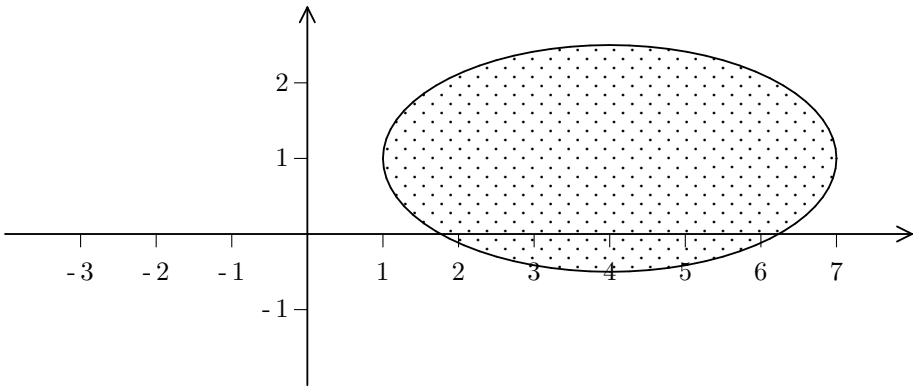
```
\PaintCone \Sinus(1,1,0,2)(4,5)[10] \StopCone
\SetCMYKColor(1,0,0,0.4)
\Sinus(1,1,0,2)(0,\ZweiPi)[60]
\ResetColor
\StandardCoordinates
\CloseGraph
```

liefert



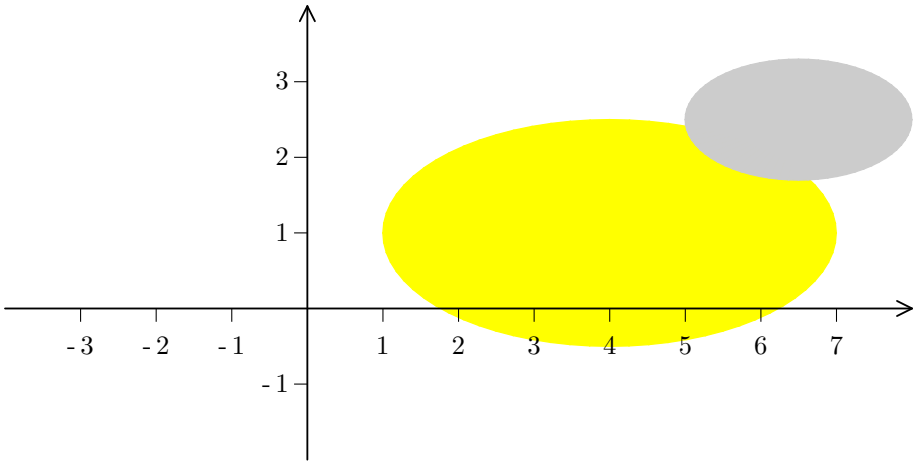
§ 3 Schattierte Ellipsen

```
\InitGraph{12}{7}{4}{2}{1cm}
\StandardCoordinates
\begin{Translate}{4}{1}
\SetShadingUp
\UpperSemiEllipseAt(0,0)(3,1.5)(-3,3)[50]
\FillGapAt(0,0)(-3,3)[50]
\SetShadingDown
\LowerSemiEllipseAt(0,0)(3,1.5)(-3,3)[50]
\ShadingOff
\EllipseAt(0,0)(3,1.5)
\end{Translate}
\CloseGraph
```



Für grau bzw. farbig gefüllte Ellipsen steht der neue Befehl
\PaintEllipseAt$(x_0, y_0)(a, b)$ bereit.

```
\InitGraph{12}{6}{4}{2}{1cm}
\SetYellow
\PaintEllipseAt(4,1)(3,1.5)
\ResetColor
\PaintEllipseAt(6.5,2.5)(1.5,0.8)
\StandardCoordinates
\CloseGraph
```



Kapitel 8

Transformationen

§ 1 Translationen

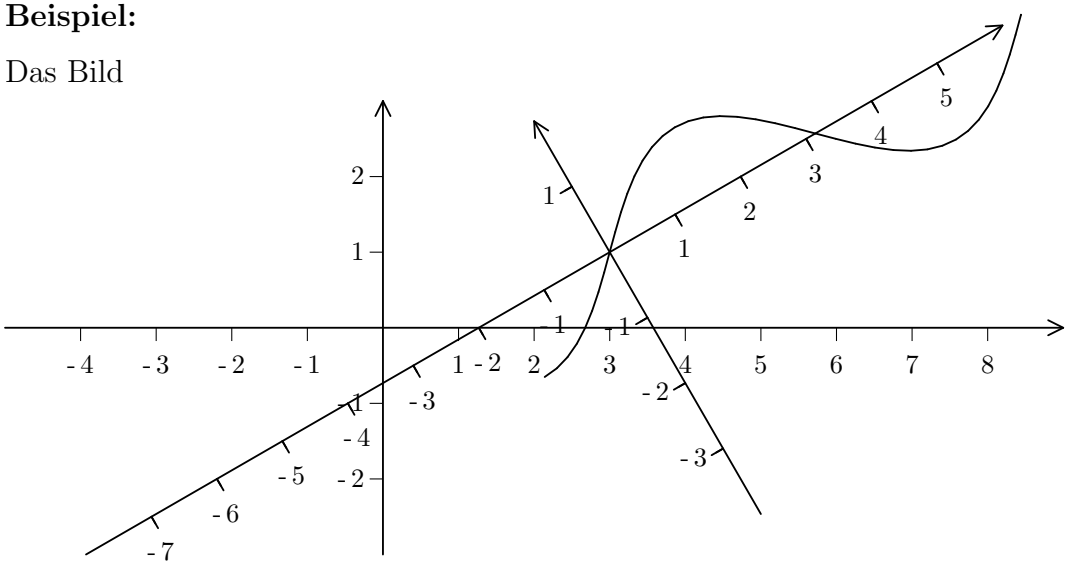
`\begin{Translate}{`Δ_x`}{`Δ_y`}` ... `\end{Translate}` schafft eine Umgebung, innerhalb der alles um (Δ_x, Δ_y) verschoben wird. Die Wirkung ist so ähnlich wie bei dem `\OpenWindow`-Befehl, aber die Seitenabmessungen bleiben erhalten.

§ 2 Drehungen

`\begin{Rotate}{`α`}` ... `\end{Rotate}` schafft eine Umgebung, innerhalb der (fast) alles um den Winkel α (in Grad) um den Nullpunkt gedreht wird. Der Befehl wirkt nicht auf die `Frame`-Befehle und bei `Text`-Befehlen wirkt er auf die Koordinaten, aber nicht auf die Positionierung und den Text selber. Auch bei dem `\PutPath`-Befehl wirkt sich die Transformation nur auf den Referenzpunkt aus.

Beispiel:

Das Bild



erhält man mit Hilfe der Befehle

```
\InitGraph{14}{5}{5}{2}{1cm}
\StandardCoordinates
\begin{Translate}{3}{1}
  \begin{Rotate}{30}
    \StandardCoordinates \Sinus(1,1,0,0)(-\PiHalbe,\ZweiPi)[40]
  \end{Rotate}
\end{Translate}
\CloseGraph
```

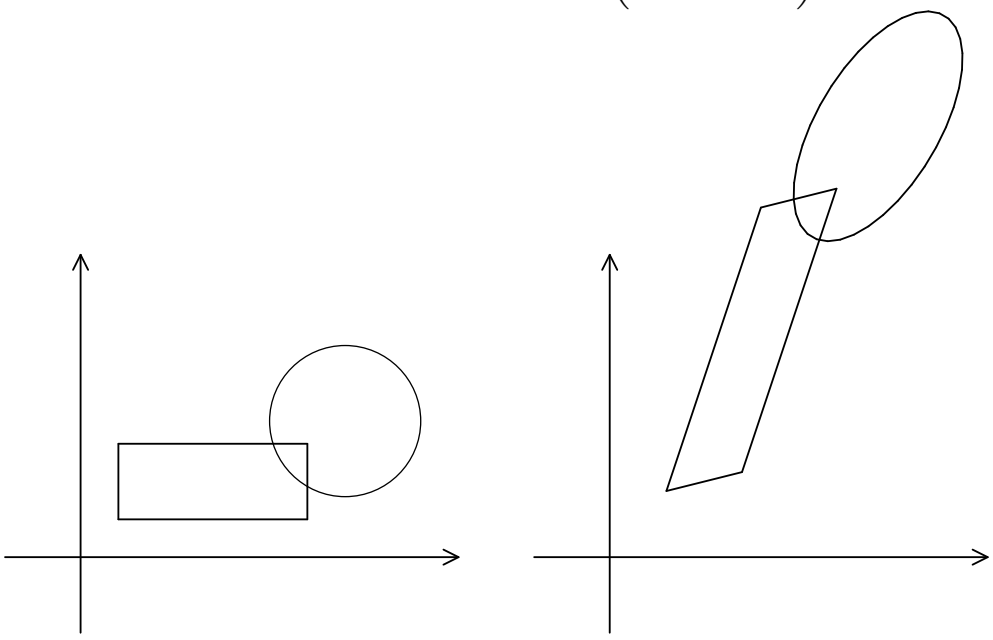
Eine noch allgemeinere Transformation erhält man mit

`\begin{Transform}{`a`}{`b`}{`c`}{`d`}` ... `\end{Transform}`

Es handelt sich um die lineare Transformation

$$\begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix}.$$

Beispiel: Die Transformations-Matrix $A = \begin{pmatrix} 0.5 & 1 \\ 1.5 & 0.25 \end{pmatrix}$ liefert:



```
\InitGraph{14}{8}{0}{0}{1cm}
\OpenWindowAt(0,0)(6,5)(1,1)
\Axes
\Rectangle(0.5,0.5,3,1.5)
\CircleAt(3.5,1.8)(1)
\CloseWindow
%
\OpenWindowAt(7,0)(6,5)(1,1)
\Axes
\begin{Transform}{0.5}{1}{1.5}{0.25}
\Rectangle(0.5,0.5,3,1.5)
\CircleAt(3.5,1.8)(1)
\end{Transform}
\CloseGraph
```

Kapitel 9

Bild-Dateien

§ 1 Vektorgrafiken

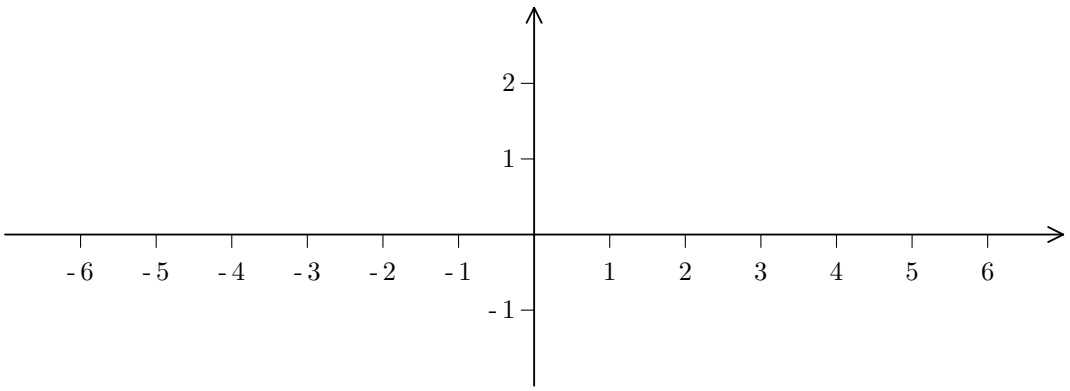
`\LoadCSGAT`$(x_0, y_0)\{1cm\}\{Datei\}$ lädt eine ASCII-Datei, deren Zeilen folgendes festes Format haben müssen:

x	y	l	Δx	Δy

Dabei ist der Buchstabe **l** nur aus (historisch bedingten) internen technischen Gründen erforderlich. Er deutet auf einen `\Line`-Befehl hin. Tatsächlich sorgt jede solche Zeile dafür, dass eine Linie von (x, y) nach $(x + \Delta x, y + \Delta y)$ gezeichnet wird.

Der Vorteil besteht darin, dass man kompliziertere Zeichnungen vorher mit Hilfe eines externen Programms berechnen und in einer Datei (mit der Endung `.csg`) abspeichern kann, wie etwa im Falle der folgenden Fourier-Entwicklung:

```
\InitGraph{14}{5}{5}{2}{1cm}
\StandardCoordinates
\SetRed
\LoadCSGAt(0,0){1cm}{Fourier2.csg}
\CloseGraph
```



Diese Methode ist **rechner-unabhängig**. Die Buchstabenfolge CSG stammt noch aus der Zeit, als das erste Grafik-Paket auf einem Atari entwickelt wurde und sogenannte CS-Special-Befehle für einfache Grafik-Routinen zur Verfügung standen. Für die rechnerunabhängige Version habe ich eine Anleihe bei epic gemacht.

§ 2 Import externer Grafiken

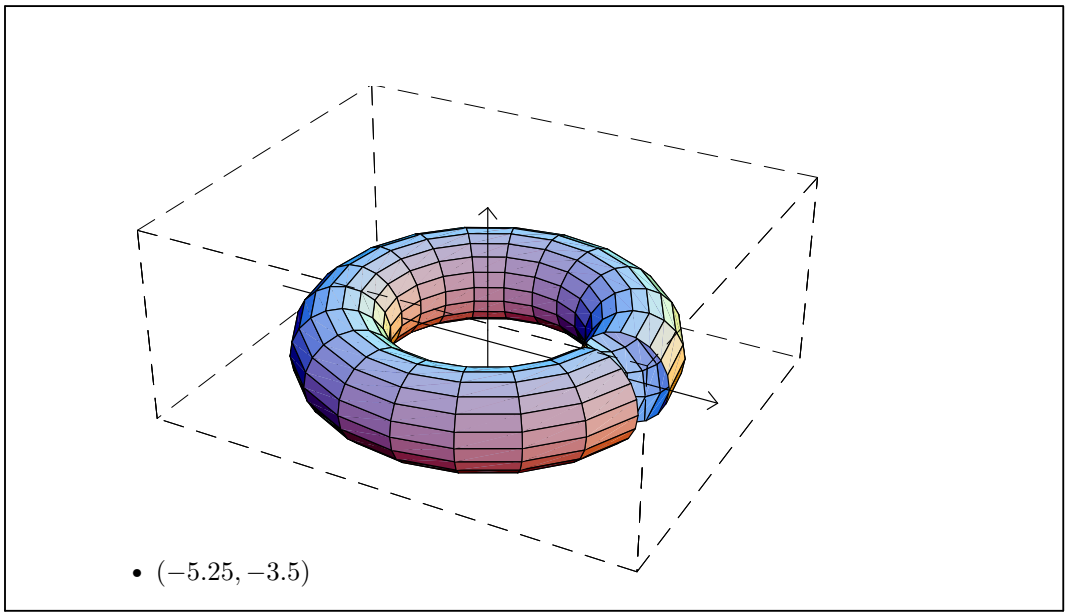
Komplizierter sieht es beim Laden von externen Grafiken aus:

```
\InitGraph{14}{8}{7}{4}{1cm}
\DrawBoundary
\IncludeGraphics[width=9cm](torus3)At(-5.25,-3.5)
\MedPointAt(-5.25,-3.5) \Text[r]{\footnotesize $(-5.25,-3.5)$}
\CloseGraph
```

Intern wird hier der `\includegraphics`-Befehl aus dem graphicx-Paket aufgerufen. In eckigen Klammern sind deshalb alle Optionen aus der 'key val list' des Befehls `\includegraphics` erlaubt. Hier wird mit `width=9cm` die Breite des Bildes festgelegt. Das ist nötig, damit das Erscheinungsbild unter `[ps]` und `[pd]` gleich ist.

Was passiert nun? Unter `[ps]` wird graphicx mit der Option dvips geladen. Deshalb wird nach einem Bild „torus3.eps“ gesucht. Arbeitet man unter `[pd]`, so wird das Bild „torus3.jpg“ erwartet. Die Endung (eps oder jpg) fügt das System selbst ein, aber das Bild sollte mindestens in diesen beiden Versionen vorhanden sein. Für `[pd]` ist auch ein PDF-File möglich, aber es muss wirklich der gleiche Ausschnitt wie bei jpg-Bild sein.

Der Previewer Yap kann die Grafik anzeigen, aber gedruckt werden kann sie nur von GhostView oder Acrobat Reader aus. Auch xdvi in einer nicht allzu alten Version zeigt die POSTSCRIPT-Grafiken an.

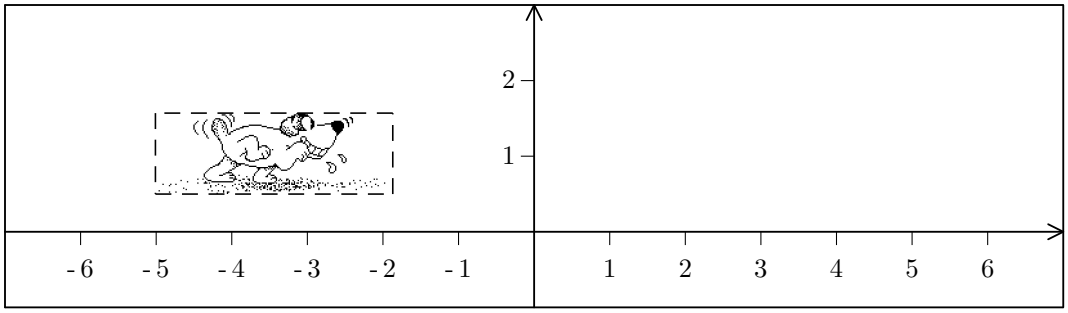


Wenn Sie eine Datei im 'bmp'-Format (etwa mit MS Paint erstellt) einbinden wollen, gibt es folgende Möglichkeit: Wenn Sie mit MikTeX und **nur** mit der Option `[ep]` arbeiten wollen, so geht das wie folgt:

```
\InitGraph{14}{4}{7}{1}{1cm}
\DrawBoundary
\StandardCoordinates
```

```
\LoadImg(3.12cm,1.06cm){Hund}At(-5,0.5)
\SetDashed \Rectangle(-5.01,0.499,-1.87,1.57)
\CloseGraph
```

Es wird dann automatisch die Datei „Hund.bmp“ eingebunden. Allerdings müssen Sie selbst dafür sorgen, dass Ihre Datei die richtige Größe hat, das lässt sich nicht mehr nachträglich steuern. Im Beispiel wird eine Bitmap-Datei mit 736 x 252 Pixel geladen. Bei Previewer und Druckertreiber wird eine Auflösung von 600dpi benutzt. Das führt zu einer Grafik der Größe 3.12 x 1.06 cm, und diese Daten schreiben Sie in den `\LoadImg`-Befehl. Wollen Sie nun das Bild auch unter den Optionen `[ps]` und `[pd]` benutzen, so müssen Sie auch eine .eps- und eine .jpg-Version erstellen. In diesen Fällen wird die Information über Breite und Höhe der Grafik gelesen und die Ausgabe-Grafik dementsprechend nachträglich vom System angepasst.



§ 3 Pfade und Benutzer-Kurven

MGA-TEX bietet keine Möglichkeit, unregelmäßig geformte Flächen mit einer Farbe oder einem Grauton zu füllen. Unter den Optionen `[ps]` und `[pd]` kann man sich aber mit dem `\PutPath`-Befehl behelfen.

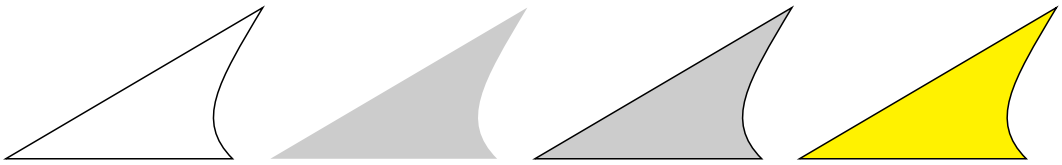
In POSTSCRIPT besteht ein „Pfad“ aus folgenden Bestandteilen:

„`newpath`“ eröffnet einen neuen Pfad.
„x_0 y_0 `moveto`“ setzt den aktuellen Punkt auf die Koordinaten (x_0, y_0).
„x_1 y_1 `lineto`“ fügt eine gerade Linie vom aktuellen Punkt (x_0, y_0) nach (x_1, y_1) an den Pfad an. Der Endpunkt wird der neue aktuelle Punkt.
„x_1 y_1 x_2 y_2 x_3 y_3 `curveto`“ fügt eine kubische Bezierkurve an den aktuellen Pfad an. Als Kontrollpunkte werden der aktuelle Punkt (x_0, y_0) und die Punkte (x_1, y_1), (x_2, y_2) und (x_3, y_3) benutzt, der letzte Punkt wird der neue aktuelle Punkt.
„`closepath`“ schließt den Pfad zu einer geschlossenen Kurve.
„d `setlinewidth stroke`“ zeichnet den Pfad mit der durch d festgelegten Linienstärke.
„d `setgray fill`“ füllt das Innere des Pfades mit dem angegebenen Grauton (man beachte die amerikanische Schreibweise).
„r g b `setrgbcolor fill`“ füllt mit der angegebenen RGB-Farbe.
„`gsave`“ bzw. „`grestore`“ speichert den aktuellen Graphik-Zustand im POSTSCRIPT-

Stack bzw. holt diesen Zustand zurück. Das braucht man z.B., wenn man mit einem Grauton füllt und anschließend die Umrisslinie zeichnen will:

„`gsave 0.8 setgray fill grestore .02 setlinewidth stroke`“

Beispiel:



Diese vier Figuren erzeugt man mit den folgenden Befehlskombinationen

```
newpath 0 0 moveto 3 0 lineto
2.5 0.5 2.8 1 3.4 2 curveto
0 0 lineto closepath .02 setlinewidth stroke
```

—

```
newpath 0 0 moveto 3 0 lineto
2.5 0.5 2.8 1 3.4 2 curveto
0 0 lineto 0.8 setgray fill
```

—

```
newpath 0 0 moveto 3 0 lineto
2.5 0.5 2.8 1 3.4 2 curveto
0 0 lineto closepath
gsave 0.8 setgray fill grestore
.02 setlinewidth stroke
```

—

```
newpath 0 0 moveto 3 0 lineto
2.5 0.5 2.8 1 3.4 2 curveto
0 0 lineto closepath
gsave 1 0.95 0 setrgbcolor fill grestore
.02 setlinewidth stroke
```

Um die POSTSCRIPT-Pfade in der Zeichnung zu positionieren, gibt es den Befehl `\PutPath`(x_0, y_0)\{*Postscript-Befehle*\}\{*PDF-Befehle*\} Dabei wird (x_0, y_0) zum Nullpunkt im PS-Koordinatensystem. Alle weiteren Angaben beziehen sich auf diesen Nullpunkt.

Wie man sieht, muss der Pfad nun auch noch in PDF-Form angegeben werden. Das ist ein bisschen problematisch. Während POSTSCRIPT noch eine halbwegs

anständige Programmiersprache ist, dient PDF auf recht primitive Weise der Seitenbeschreibung. Zum Glück lassen sich die wenigen Befehle, die wir brauchen, ziemlich leicht übertragen:

Vorgang	PS	PDF
Neuen Pfad anfangen	`newpath`	- entfällt -
Bewegung	x y moveto	x y m
Gerade Linie	x y lineto	x y l
Bezierkurve	x_1 y_1 x_2 y_2 x_3 y_3 curveto	x_1 y_1 x_2 y_2 x_3 y_3 c
Pfad schließen	closepath	h (entfällt)
Grauwert wählen	γ setgray	γ g
Farbe wählen	ϱ γ β setrgbcolor	ϱ γ β rg
Liniendicke wählen	δ setlinewidth	δ w
Umriss zeichnen	stroke	s
Fläche füllen	fill	f
Grafikstatus speichern / laden	gsave /grestore	q / Q
Fläche füllen und Umriss zeichnen	fill / stroke	b

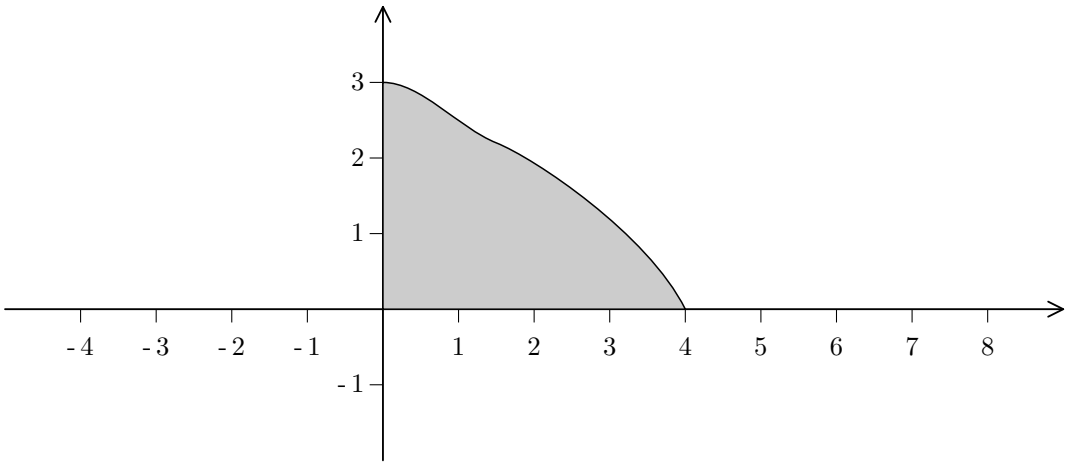
Es folgen jetzt die kompletten Befehle, die zu den obigen vier Figuren führten. Dabei wird jeweils nur der PDF-Teil angegeben, an die Stelle der Pünktchen sind die oben angegebenen POSTSCRIPT-Befehle einzusetzen.

```
\InitGraph{14}{3}{0}{0}{1cm}
\PutPath(0.5,0.5){...}{0 0 m 3 0 l
2.5 0.5 2.8 1 3.4 2 c
0 0 l .02 w s}
\PutPath(4,0.5){...}{0 0 m 3 0 l
2.5 0.5 2.8 1 3.4 2 c
0 0 l 0.8 g f}
\PutPath(7.5,0.5){...}{0 0 m 3 0 l
2.5 0.5 2.8 1 3.4 2 c
0 0 l .02 w 0.8 g b}
\PutPath(11,0.5){...}{0 0 m 3 0 l
2.5 0.5 2.8 1 3.4 2 c
0 0 l .02 w 1 0.95 0 rg b}
\CloseGraph
```

Hier ist noch ein typisches Anwendungsbeispiel:

```
\InitGraph{14}{6}{5}{2}{1cm}
\StandardCoordinates
\PutPS(0,0){newpath 0 0 moveto 0 3 lineto
0.5 3 1 2.4 1.5 2.2 curveto 2 2 3.5 1 4 0 curveto
closepath gsave 0.8 setgray fill grestore
.02 setlinewidth stroke}{0 0 m 0 3 l
0.5 3 1 2.4 1.5 2.2 c 2 2 3.5 1 4 0 c
```

```
0.8 g .02 w b}
\CloseGraph
```



Anhang

§ 1 Turtle-Graphik

Mit der Option `[turt]` lädt man einen kleinen Turtle-Grafik-Modul mit folgenden Befehlen:

1. `\StartTurtleAt`(x_0, y_0) : Die Turtle wird auf (x_0, y_0) gesetzt und schaut nach oben (90° gegen die positive x-Achse).
2. `\PenDown` : Stift senken!
3. `\PenUp` : Stift heben!
4. `\TurnRight`(d) : Um d Grad nach rechts drehen!
5. `\TurnLeft`(d) : Um d Grad nach links drehen!
6. `\Forward`(l) : Um l nach vorne gehen!
7. `\Back`(l) : Um l rückwärts gehen!
8. `\LookDir`(d) : Absoluter Richtungswechsel! $d = 0$ bedeutet: nach rechts.
9. `\GoPosition`(x_0, y_0) : Absoluter Ortswechsel, Richtung nach oben.
10. `\GoHome` : An den Anfangspunkt zurück, Richtung nach oben.
11. `\StopTurtle` : beendet die Turtle-Grafik.
12. `\DoTurtle`$(n)\{Befehle\}$ ermöglicht es, eine Folge von Turtle-Befehlen n-mal hintereinander auszuführen.

Beispiel: (Gleichschenkliges Dreieck, regelmäßiges 6-Eck, Zickzacklinie)

```
\InitGraph{14}{7.5}{5}{3.5}{1cm}
\StandardCoordinates
\SetBlue
\StartTurtleAt(1,-0.5)
\TurnRight(100)
%
% gleichschenkliges Dreieck:
%
\PenDown
\Forward(2)
\TurnLeft(110)
\Forward(3)
```

```
\TurnLeft(140)
\Forward(3)
\PenUp
%
% regelmaessiges 6-Eck:
%
\GoPosition(4,0.5) \BigPoint
\PenDown
\LookDir(0)
\DoTurtle(6){%
  \Forward(2) \TurnLeft(60)}
\PenUp
%
% Zickzacklinie:
%
\GoPosition(-4,-2)
\PenDown
\TurnRight(135)
\DoTurtle(5){%
  \Forward(1) \TurnLeft(90) \Forward(2) \TurnRight(90) \Forward(1)}
\PenUp
\StopTurtle
\CloseGraph
```