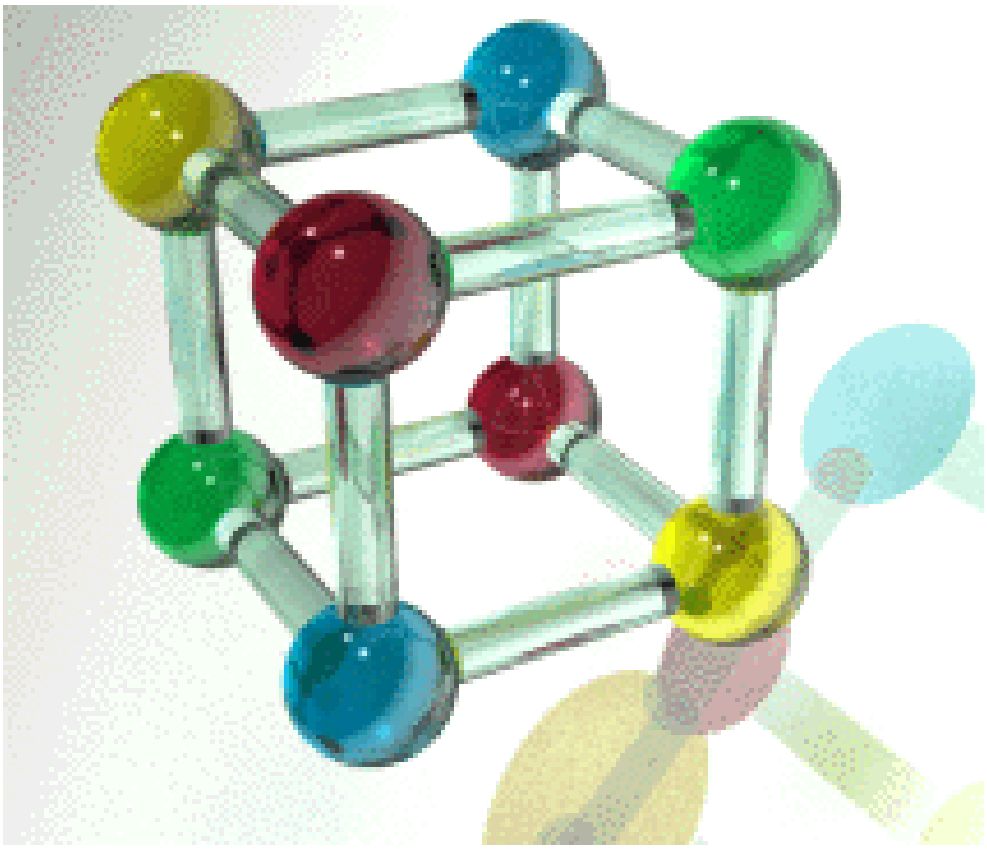
Projektkurs:

Mathematik

mit und am Computer

(unter Verwendung von EXCEL und MuPAD)

Autor: L. Drews



(Q: © SciFace; bearb.: (p) lsp: dre)

Ziele des Kurses:

- Wiederholung mathematischer Kenntnisse
- Üben und Anwenden von Fähigkeiten und Fertigkeiten beim Lösen von mathematischen Aufgaben und Problemen
- Auseinandersetzung mit praktischer Mathematik
- sinnvolle Arbeit mit dem Computer
- Probleme erkennen und Problemlösungen erarbeiten
- Verknüpfung von Fachwissen unter fächerübergreifenden Aspekten
- Arbeiten im Team
- Umgang mit Computern routinieren

Bemerkungen zur Rechtschreibung:
Dieses Skript folgt nicht zwangsläufig der neuen **ODER** alten deutschen Rechtschreibung. Vielmehr wird vom Recht auf künstlerische Freiheit, der Freiheit der Sprache und von der Autokorrektur des Textverarbeitungsprogramms microsoft ® WORD ® Gebrauch gemacht.
Für Hinweise auf echte Fehler ist der Autor immer dankbar.

Danksagung:
Für viele mathematische Tips und Hilfestellungen sowie das Korrekturlesen gilt ein ganz besonderer Dank Frau S. Regenbrecht (Gymnasium Reutershagen (Rostock))

Inhaltsverzeichnis

0. Allgemeine Vorbemerkungen

Wenn man ein Weilchen mit Computerprogrammen - wie MuPAD, MATHEMATICA od. ä. - gearbeitet hat, fragt man sich schnell, wozu man sich eigentlich noch mit Mathematik in der Schule usw. beschäftigen muß? Die Programme können fast alles rechnen, sind genauer und schneller. Zwar braucht man heute noch mathematische Kenntnisse, um das Problem computergerecht einzugeben, dass wird aber in den nächsten Jahren sicher mehr und mehr vom Computer selbst übernommen werden.
Da sind wir bei der Frage: Wozu beschäftigen wir uns überhaupt mit Mathematik in der Schule? Als Argumente kann man anbringen, dass das Rechnen eine Grundfähigkeit für den täglichen Umgang mit anderen Menschen und Waren ist. Die höhere Mathematik fällt wohl weniger in diesen Bereich (wohl eher im Gegenteil). Es ist eher die Entwicklung des abstrakten Denkens, des Umgangs mit Begriffen, Verfahren und Algorithmen, die hier im Mittelpunkt stehen. Nicht zu Vergessen das Entwickeln von Fähigkeiten zum Lösen von Problemen und Aufgaben. Die Anwendung in komplizierten wissenschaftlichen und technischen Problemstellungen spielt in der Schule – welche die Grundlagen legt – leider eine zu geringe Rolle. Dadurch wird der Sinn oft ein wenig verschleiert.
Ich – als Nichtmathematiker - sehe Computer-Mathematik-Programme in der Schule vorrangig als Hilfsmittel zur Kontrolle von handgerechneten Lösungen, als Möglichkeit zum Experimentieren und zur Erweiterung der Sicht auf mathematische Probleme. Da darf auch mal Spaß bei der Beschäftigung mit der Mathematik aufkommen. Für viele Schüler ist oft auch ein veränderter Zugang (z.B. Arbeit mit dem Computer) ein besserer Einstieg zum mathematischen Problem. Später können dann solche Programme viele Routinetätigkeiten übernehmen. Dafür muß der Mensch seine Kontrollfunktionen stärker erfüllen, denn Computer sind eben nur Maschinen. Und eben deshalb muß man schon wissen, was Sache ist.
In den nächsten Jahren wird der Inhalt der Mathematik und die Art und Weise, wie man sich mit ihr beschäftigt, einem ähnlichen Wandel unterliegen, wie die Schul-Mathematik ihn mit der Einführung von Taschenrechnern schon erlebt hat.

1. Tabellenkalkulation

siehe dazu im entsprechenden Skript "Tabellenkalkulation mit EXCEL"

1.1. spezielle Aufgaben zu EXCEL für diesen Kurs

1.1.1. Berechnung und Darstellung von Funktionen

1.1.1.1. Lineare Funktionen

Bauen wir zuerst die Überschriften und Themenbeschreibung unseres Kalkulationsblattes auf. Darunter folgt dann gleich der Eingabe-Bereich. Für eine sinnvolle Berechnung und Darstellung benötigen wir die Funktionsparameter **m** und **n** – also den Anstieg und den Schnittpunkt mit der y-Achse. Für die Berechnung der Funktionswerte und die graphische Darstellung die Kalkulation brauchen wir zudem noch die untere und obere Grenze (Definitionsbereich).
Aus praktischen Erwägungen wollen wir uns auf 20 berechnete Wertepaare beschränken. Damit sollte die Funktion deutlich im gewählten Definitionsbereich dargestellt werden können. Wie Sie sehen werden, kann man aber ohne weiteres noch mehr Zwischenwerte berechnen lassen. Der Eintipp-Aufwand ist eigentlich der Selbe – nur einige Maus-Aktionen werden etwas länger.
Sie sollten es sich zur Gewohnheit werden lassen die einzelnen Bereiche (Eingaben, Konstanten, Berechnungen, Ergebnisse) Ihrer Kalkulation mit unterschiedlichen Farben (z.B.: grün, blau, gelb, orange, ...) zu hinterlegen. So findet man sich auch nach längerer Zeit schnell in der Kalkulation zurecht. Formatierte Überschriften usw. tun dann ihr übriges.

	A	B	C	D	E	F
1	**Lineare Funktionen**					
2						
3	allg. Formel	**y = m x + n**				
4						
5				**Definitionsbereich**		
6	Anstieg (m)	3		untere Grenze	-10	
7	Schnittpunkt (n)	4		obere Grenze	20	
8				Schrittweite		
9						
10	**x**	**y**				
11						

Ergänzen wir nun die eigentlichen Berechnungen.

	A	B	C	D	E	F
1	**Lineare Funktionen**					
2						
3	allg. Formel	**y = m x + n**				
4						
5				**Definitionsbereich**		
6	Anstieg (m)	3		untere Grenze	-10	
7	Schnittpunkt (n)	4		obere Grenze	20	
8				Schrittweite	=(E7-E6)/20	
9						
10	**x**	**y**				
11	=$E$6	=$B$6*A11+$B$7				
12	=A11+$E$8	=$B$6*A12+$B$7				
30						
31						

Die etwas ungewöhnliche Zellen-Adressen mit den Paragraph-Zeichen erklären sich aus der festen (absoluten) Position der Funktionsparameter in der Kalkulations-Tabelle. Wir wollen das immer genau mit der Zelle gerechnet wird.
Ab Zeile 13 können wir die automatische Ausfüll-Funktion von EXCEL nutzen. Dazu zieht man einfach an der rechten, unteren, schwarzen Ecke der markierten Zelle(n). Die Formeln od. ä. werden dann soweit ausgefüllt, wie die Maus gezogen wird. In unserem Beispiel bis zur Zeile 30 – womit wir die 20 Wertepaare erhalten. Für mehr Wertepaare ziehen Sie einfach weiter. Beachten Sie aber, dass auch die Schrittweiten-Berechnung in Zelle E8 an die neue Anzahl angepasst werden muß.
Spaßens halber können Sie ja mal probieren, wie die Kalkulation aussehen würde, wenn Sie die Ausfüllfunktion auf Zell-Adressen ohne das Paragraph-Zeichen anwenden würden.

Was uns nun noch fehlt ist ein passendes Diagramm. Markieren Sie für den Diagramm-Assistenten den Bereich, aus dem ein Diagramm erzeugt werden soll (hier: A10:B30). Nun starten wir den Diagramm-Assistenten z.B. über das "Einfügen"-Menü und "Diagramm ...". Diesen Assistenten können Sie nun solange "vor" und "zurück" gehen und Optionen ändern bis das Diagramm in weiten Zügen Ihren Ansprüchen genügt. Nachfolgend sind nur die Änderungen bzw. notwendigen Auswahlen aufgezählt:

Assistenten-Seite	Auswahl / Option	Bemerkungen
Diagrammart	**Punkt (XY)** Punkte mit Linien	ev. Punkte mit interpolierten Linien
Diagramm-quelldaten		
Diagramm-optionen	**Titel** Diagrammtitel: lineare Funktion Rubrikenachse (Y): y Größenachse (X): x **Gitternetzlinien** alle Haupt- und Nebengitternetze auswählen **Legende** Legende anzeigen: deaktivieren	ev. reichen Hauptgitternetze
Diagramm-platzierung	"als Objekt in" aktivieren	

Nach dem "Fertigstellen" erscheint das Diagramm auf dem Rechenblatt. Sie können es nun durch Anfassen und Ziehen an die gewünschte Position bewegen (ev. vorher die Seite einrichten). Über die schwarzen Anfaß-Ecken lässt sich die Größe des Diagramms anpassen.
Zum Drucken des gesamten Blattes (mit Kalkulation und Diagramm) darf das Diagramm nicht ausgewählt sein. Prüfen Sie unbedingt vorher in der Seitenansicht, ob der Ausdruck Ihren Anforderungen genügt und der Papierbedarf optimal (bzw. minimal) ist.

Aufgaben:

1. *Vollziehen Sie das Beispiel nach!*
2. *Drucken Sie sich das Beispiel als ein Blatt (mit Kalkulation und Diagramm) aus!*
3. *Verändern Sie die Funktionsparameter und den Definitionsbereich auf beliebige Werte!*
4. *Erstellen Sie die graphische Darstellung der Funktion* $y = -2{,}3\,x - 6{,}8$ *in den Grenzen* $-12 \geq x \geq 3$ *!*
5. *Drucken Sie dieses Mal nur das Diagramm aus!*

1.1.1.2. Potenzfunktionen

Das soeben entwickelte Kalkulationsblatt lässt sich auch für andere Funktionstypen anpassen. Wir benötigen lediglich eine Funktion für die Potenzdarstellung. In EXCEL dient dazu die Funktion **POTENZ(*basis* ; *exponent*)**. Basis und Exponent können direkt eingegeben oder berechnet werden oder auch Zellbezüge sein.

1.1.1.3. Expotentialfunktionen

Expotentialfunktionen lassen sich genau wie die Potenzfunktionen über die POTENZ-Funktion berechnen. Benötigt man die echte Expotential-Funktion – also zur Basis e (EULERsche Zahl) – dann hilft uns EXCEL sogar mit einer eigenen Funktion. Mit **EXP(*exponent*)** lässt sich somit e^x berechnen (x sei der *exponent*).

1.1.1.4. Wurzelfunktionen

Microsoft wollte wohl seinen Nutzern nicht zu viel Mathematik zumuten. Bei den Wurzeln werden die Möglichkeiten in EXCEL knapp. Lediglich die Quadrat-Wurzel ist direkt zugänglich. Über **WURZEL(*basis*)** erhält man die übliche $\sqrt{x}$ (x sei die *basis*). Eine andere Erklärung für die mangelnde Behandlung von Wurzeln, könnte natürlich auch als Herausforderung für Mathematiker betrachtet werden. Unter Verwendung der Umschreibung in eine Potenzfunktion erhalten wir:

$$\sqrt[n]{x} = x^{\frac{1}{n}}$$

In EXCEL-Schreibweise also einfach **POTENZ(*basis* ; 1 / *n*)** (n sei der *wurzelgrad*).

1.1.1.5. weitere Funktionen

Hier sei nur kurz auf die vielseitig vorhandenen Winkelfunktionen und die unzähligen Spezial-Funktionen hingewiesen. Im EXCEL-Skript (→ "Tabellenkalkulation mit EXCEL") bzw. in der Hilfe sind diese aufgezählt bzw. ausführlich beschrieben.

Aufgaben:

1. *Erstellen Sie sich ein Tabellenblatt für die Berechnung und Darstellung einer beliebigen Potenzfunktion!*
2. *Prüfen Sie die Kalkulation der Funktionen* $y = 2{,}5\ x^{3{,}4}$ *und* $y = -0{,}5\ x^{-3}$ *mit Ihrem Taschenrechner!*
3. *Erstellen Sie sich ein Tabellenblatt für Expotenztialfunktionen!*
4. *Prüfen Sie die Funktionen* $y = 2{,}5 * 3{,}4^{x}$ *und* $y = -0{,}5 * -3^{x}$ *!*
5. *Erstellen Sie sich ein Tabellenblatt für die Berechnung und Darstellung von Winkelfunktion!*
6. *Prüfen Sie die Kalkulation z.B. der Funktionen* $y = \sin x$ *und* $y = -\cos x$ *mit Hilfe von Darstellungen aus Tafelwerken od. ä.!*

1.2. mathematische Aufgaben mit EXCEL

Die nächsten Abschnitte sollen die Einsicht in die praktische Nutzbarkeit von EXCEL vertiefen. Grundlegende Fähigkeiten in der Handhabung einer Tabellenkalkulation werden vorausgesetzt. Wir nehmen einfach bestimmte mathematische Verfahren oder Probleme und zeigen exemplarisch das Vorgehen. Die Reihenfolge ist eher zufällig und ohne direkten bezug zum Vorgehen im Mathematik-Unterricht.

1.2.1. Lösen von Gleichungssystemen durch Interation

(nach /13/)

Für diese Anwendung von EXCEL wählen wir das Beispiel:

$$y = \frac{10 - 2y}{3} \quad \text{und} \quad y = \frac{x - 2}{4} \quad .$$

Natürlich lässt sich die exakte Lösung des Gleichungssystems auch z.B. über das Gleichsetzungsverfahren erzielen.
Wir wählen alternativ ein Einsetzungsverfahren. Zuerst stellen wir uns die eine Gleichung nach x um und erhalten:

$$x = \frac{10 - 3y}{2} \quad \text{und} \quad y = \frac{x - 2}{4} \quad .$$

Durch gegenseitiges Einsetzen und weiterrechnen mit den Zwischenlösungen wollen wir eine immer bessere Annäherung an das Endergebnis erreichen.

	A	B	C
1			
…			
5			
6	**manuelle Interation**		
7	**x**	**y**	
8		=(A8-2)/4	
9	=(10-3*B8)/2		
10			

Die einzutippenden Formeln sind in der nebenstehenden Abbildung sichtbar.
Die Zelle A8 bleibt leer – was einer 0 entspricht - oder sie wird bei Bedarf mit einem passenden Startwert gefüllt. Als nächstes füllen wir die Zelle B9 automatisch von B8 aus.
In der Tabelle erscheint jetzt für **x** und **y** das erste Mal ein berechnetes Lösungspaar.

(Zum besseren Verständnis der mathematischen Zusammenhänge behalten wir die Anzeige der Formel bei – normalerweise erscheinen dort sofort die berechneten Werte.)

	A	B	C
1			
...			
5			
6	**manuelle Interation**		
7	**x**	**y**	
8		=(A8-2)/4	
9	=(10-3*B8)/2	=(A9-2)/4	
10			
...			
40			
41			

Nach dem Markieren von A9:B9 lassen wir wieder die darunter liegenden Zellen (z.B. bis Zeile 40) ausfüllen. (siehe auch linke Abbildung)
In EXCEL sieht man nun, dass die Zwischenlösungen sich wirklich immer mehr annähern. Irgendwann ist der Unterschied zwischen zwei Zwischenlösungen so klein, dass er praktisch keine Rolle mehr spielt. Wir haben also eine Lösung des Gleichungssystems.

Nun ist dies sicher ein möglicher - aber nicht der effektivste - Weg in EXCEL.
Versuchen wir doch einfach die Formeln mit direktem, gegenseitigen Einsetzen zu verwenden.

	A	B	C	D	E	F
1						
...						
5						
6	**manuelle Interation**			**direkte (automatische) Interation**		
7	**x**	**y**		**x=**	=(10-3*E8)/2	
8		=(A8-2)/4		**y=**	=(E7-2)/4	
9	=(10-3*B8)/2	=(A9-2)/4				
10	=(10-3*B9)/2	=(A10-2)/4				
11	=(10-3*B10)/2	=(A11-2)/4				
12	=(10-3*B11)/2	=(A12-2)/4				
				...		
40	=(10-3*B39)/2	=(A40-2)/4				

Gleich nach dem Eintippen der Formeln bei E7 bzw. E8 gibt EXCEL eine Fehlermeldung aus, die besagt, das die Formeln einen Zirkelbezug besitzen.

Dies bedeutet, sie sind voneinander abhängig.
Die Zelle E7 braucht den Wert aus E8. Der muß aber erst aus E7 berechnet werden usw. usf.

	A	...	D	E	F
1					
...					
5					
6			**direkte (automatische) Interation**		
7			**x=**	=(10-3*E8)/2	
8			**y=**	=(E7-2)/4	
9					

Normalerweise geht so etwas natürlich nicht, da jeder Rechner daran unendlich sitzen würde. Wenn wir die Rechnung per Hand (bzw. wie bei manuell gezeigt) berechnen, hören wir ja auch irgendwann auf. Unsere Grenzen sind einmal die Zahl von Schritten oder die Feststellung, dass die berechneten Werte keine Veränderungen mehr aufweisen. Praktisch müssen wir der Tabellenkalkulation all dies nun ir-

gendwie mitteilen. Die notwendige Veränderung muß bei den "Optionen" im Menü "Extras" durchgeführt werden. Unter dem Reiter "Berechnung" findet sich ein Bereich "Interation" mit einem Optionskästchen. Dies muß aktiviert werden. Daneben befinden sich zwei Eingabefeld. Das erste für die maximale Anzahl von Interationen, die berechnet werden sollen. Im zweiten können wir die Genauigkeit eingeben – also die maximal zu erreichende Differenz (Änderung) zwischen zwei Ergebnissen. Je kleiner diese Zahl, um so genauer ist das Ergebnis. Für normale Aufgaben reichen 100 Interationen und eine maximale Änderung von 0,000001 völlig aus.
Für praktische Anwendungen braucht man den manuellen Weg dann nicht mehr. Auch aufwendigere Gleichungssysteme sind so numerisch lösbar (, wenn sie es auch wirklich sind!!!).

Aufgaben:

1. Wandeln Sie das Schema so ab, dass in den Zeilen 2 und 3 die Gleichungen allgemein (als Variablen (grüne Zellen)) eingegeben werden können!

	A	B	C	D	E	F
1						
2	**Gleichung1**	**x =**	-3/2	**y +**	5	
3	**Gleichung2**	**y =**	1/4	**x +**	-1/2	
4						

...

2. Benutzen Sie das Kalkulationsschema zur Lösung der Gleichungssysteme:

a) $3 = 2y - 2x$; $5y - 9 = 2x$

b) $\frac{y}{2} = -3x - \frac{7}{2}$; $\frac{14}{5} = -4x - 2y$

3. Stellen Sie die Gleichungssysteme graphisch dar! Verwenden Sie das Prinzip der Kalkulationen der Grundfunktionen (→ 1.1.1.) – erweitert um eine zweite Funktion!

1.2.2. Lösen von Gleichungssystemen über Matrizen

(nach /12/)

Gegeben ist zum Beispiel das Gleichungssystem aus 3 Gleichungen:

```
3a + 4b + c =  7
     2b - c =  0
-a -  b     = -3
```

Zuerst erstellen wir ein Tabellenblatt, dass die wesentlichen Gleichungs-Parameter in einer matrixartigen Form darstellt:

	A	B	C	D	E	F	G
1	**Lösung eines linearen Gleichungssystems**						
2							
3	Variablen	**a**	**b**	**c**	Konstanten-matrix		
4	Koeffizienten	3	4	1	7		
5		0	2	-1	0		
6		-1	-1	0	-3		
7	Lösung						
8							

Eine solche universelle Darstellung (nur der wichtigsten Parameter in einer tabellenartigen Struktur) wird häufig in der Mathematik verwendet. Bestimmte Rechnungen (Matrixoperationen) lassen sich dann sehr effektiv durchführen / anwenden.

Nun markiert man den Bereich von B7 bis D7 (B7:D7). Und nun wird die Formel eingetippt. Sie landet automatisch in der Zelle B7. Die Eingabe muß nun nur noch mit [Strg] + [⇧] + [↵] (Steuerung-Umschalt-Enter) abgeschlossen werden.

```
=MTRANS(MMULT(MINV(B4:D6);(E4:E6)))
```

Die drei markierten Zellen werden automatisch mit der Funktion versorgt und zeigen das richtige Ergebnis an. Auf Wunsch lässt sich die Probe machen. In den Zellen von B8 bis D10 werden nun die Lösungen eingesetzt und die Zeilensummen (E7:E10) mit den Werten der Konstanten-Matrix verglichen.

	A	B	C	D	E	F	G
1	**Lösung eines linearen Gleichungssystems**						
2							
3	Variablen	**a**	**b**	**c**	Konstanten-matrix		
4	Koeffizienten	3	4	1	7		
5		0	2	-1	0		
6		-1	-1	0	-3		
7	**Lösung**	**3,67**	**-0,67**	**-1,33**			
8	Probe						
9							

Die Summanden in B8 bis D10 lassen sich am schnellsten berechnen, wenn man in B8 die Formel:

```
=B4*B$7
```

schreibt und dann mit dem automatischen Ausfüllen die restlichen Zellen belegt.
Dazu zieht man (am schwarzen Kästchen rechtes unten) z.B. zuerst von B8 nach D8 und danach bei einem markieren Bereich B8:D8 nach B10:D10. Damit sich in der Rechnung von Zelle B8 nur der erste Faktor an die Verschiebung angepasst wird, muß im zweiten Faktor vor der 7 ein Dollar-Zeichen (für feste Zeile) eingefügt werden.
Die Zeilensummen in E8 bis E10 sind kein Problem. Der abschließende Vergleich in F8 bis F10 wird z.B. über:

```
=WENN(E10=E6;"i.O.";"Fehler")
```

realisiert.

Fertig sieht das Schema dann so aus:

	A	B	C	D	E	F	G
1	**Lösung eines linearen Gleichungssystems**						
2							
3	Variablen	**a**	**b**	**c**	Konstanten-matrix		
4	Koeffizienten	3	4	1	7		
5		0	2	-1	0		
6		-1	-1	0	-3		
7	**Lösung**	**3,67**	**-0,67**	**-1,33**		Vergleich	
8	Probe	11	-2,67	-1,33	7	i.O.	
9		0	-1,33	1,33	0	i.O.	
10		-3,67	0,67	0	-3	i.O.	
11							

Das Beispiel lässt sich leicht auf größere Gleichungssysteme ausdehnen. Auf die Probe kann man auch leicht verzichten.

Aufgaben:

1. Berechnen Sie per Hand / Taschenrechner die Lösung der folgenden Gleichungssysteme:

a) $9 = 3c - \frac{3}{2}a$; $3a + 3b + 3c = 3$; $2b = 2$

b) $3a + 4b + c = 2$; $-(-1 + 2c) = 2a + 2b$; $2b + 2c = 1 - a$

c) $x + y - z = 8$; $x + z - y = 6$; $y + z - x = 4$

d) $5x - y = 33$; $2x + z = 21$; $3y - z = 16$

2. Prüfen Sie die Lösungen mit Hilfe des Kalkulationsschemas!

3. Erstellen Sie eine graphische Darstellung zur Lösung der Gleichungssysteme!

4. Erweitern Sie das Schema soweit, dass Sie das folgende Gleichungssystem lösen lassen können! (ev. lieber neues Schema auf einem neuen Tabellenblatt)

$8x + 7y + 10z = 1780$; $12x + 6y + 4z = 1480$; $9x + 9y + 14z = 2300$;
$5x + 10y + 20z = 2600$; $27x + 18y + 20z = 4520$

Für FREAKS:

5. Lösen Sie das folgende Gleichungssystem mit einer angepassten Kalkulation!

$-9e + 6f + 3g - 5h = 1$
$7e + 4f + g - h = -1$
$e - 5f - 2g + 3h = 0$

6. Lösen Sie das folgende Gleichungssystem mit einer angepassten Kalkulation!

$(4x - 8)(y + 2) = (x - 1)(4y + 3)$
$(3x - 8)(z + 1) = (x - 2)(3z + 1)$
$(2y + 1)(z + 2) = (y + 1)(2z + 3)$

1.3. naturwissenschaftliche und technische Sachverhalte mit EXCEL bearbeiten

EXCEL eignet sich natürlich nicht nur für die trockene Mathematik. Auch viele naturwissenschaftliche und technische Problemstellungen lassen sich damit bearbeiten. Als Beispiele werden der "Biorhythmus" und die Wärmeleitung (*in der nächsten Skript-Version*) herangezogen.

1.3.1. Der menschliche Biorhythmus

(nach /15/ und /16/)

Jeder hat sicher schon mal gemerkt, dass es Tage gibt, an denen einfach gar nichts läuft. Nach einer – meines Wissens – noch nicht wirklich bewiesenen Theorie existieren für bestimmte Verfassungszustände feste Rhythmen. Diese sollen mit der Geburt starten und sich bis in den Tod immer wieder sinusförmig wiederholen.
Um den "angeblichen" Biorhythmus zu testen wollen wir folgendermaßen vorgehen.
Zuerst analysieren wir über mindestens einen Monat unsere eigene Befindlichkeiten. Diese Beobachtungen wollen wir dann mit den Berechnungen vergleichen und so zumindestens grob die Glaubwürdigkeit bewerten.
Die Beobachtung der eigenen Befindlichkeiten kann z.B. in der Form einer Tabelle erfolgen, wobei jeder Befindlichkeit in der Skala [sehr gut, gut, mittelmäßig, schlecht, sehr schlecht] aufgenommen wird.
Nach der Theorie des Biorhythmus soll es Rhythmen für die körperliche, geistige und seelische Verfassung geben. Diese werden von jedem Probanten täglich bewertet und in die Tabelle eingetragen.

	A	B	C	D	E	F	G	
1	**Erfassung des Biorhythmus**							
2								
3	Bewertungsskala		sehr gut	gut	mittelmäßig	schlecht	sehr schlecht	
4			+ +	+	0	-	- -	
5			2 +	+	0	-	2 -	
6								
7	Start-Datum		=HEUTE()					
8								
9	Datum	Körper	Geist	Seele	gesamt	Bemerkungen		
10	=$C$7							
11	=A10+1							
12	=A11+1							
45								

Das Abschätzen und Eintragen sollte möglichst immer zur gleichen Zeit (z.B.: mittags; zur Einnahmezeit von Tabletten / Pille etc.) erfolgen, damit die Werte höchstmöglich vergleichbar sind. Natürlich lässt sich auch genauer bestimmen, was in die einzelnen Bereiche gehören soll.

Bereich	Körper	Geist	Seele
mögliche Beobachtungsbereiche	gesundheitlicher Zustand	Denkfähigkeit, Lernleistungen	Zufriedenheit, Streß, Schlafstörungen
Beispiele für Beobachtungskriterien	sportliche Leistungsfähigkeit, Körpertemperatur (Fieber), ...	Schulleistungen (z.B. Vokabellernen), Fehlerzahl in Matheaufgaben, ...	Schlafdauer, Musikauswahl, ...

Für die körperliche Verfassung soll die Phasenlänge (ein vollständiger Durchlauf) 23 Tage betragen. Bei der geistigen Verfassung sei eine Schwingung nach 33 Tagen und bei der seelischen Verfassung alle 28 Tage durchlaufen. Mit der nachfolgenden Kalkulation und graphischen Darstellung besitzen Sie ein Mittel die Glaubwürdigkeit zu prüfen.
Für die Prüfung könnte man folgendes einfaches Verfahren verwenden. Die beobachteten Werte werden einfach in drei Klassen eingeteilt positiv, unbestimmt und negativ. Genau so wird mit den Berechnungen verfahren. Nun vergleichen wir einfach die Übereinstimmung und die Gegensätze der Klassen positiv und negativ. Insgesamt sollten die Übereinstimmungen die Gegensätze überwiegen.
Etwas genauer ist das folgende Verfahren. Dazu könnte man Differenzen zwischen den Klassen (jetzt kann man alle beobachteten Klassen mit einbeziehen) bilden. Die direkte Gegenüberstellung von berechneter Funktion und den Differenzen (z.B. in einem Diagramm) sollte für den Verlauf der Kurven eine Übereinstimmung ergeben.
Bei Abweichungen wäre als Nächstes die Überprüfung der Phasenlänge interessant. Wenn selbst die nicht mehr nachweisbar ist, dann hilft nur eine langfristige Analyse bei der Bestätigung des Biorhythmus – oder er existiert nur im metaphysischen Bereich.
Zur angemessenen und realistischen Darstellung des Sachverhalts sollte aber darauf hingewiesen werden, dass diverse andere Rhythmen sicher nachgewiesen wurden. So z.B. einen 24-Stunden-Rhythmus auch dann, wenn es keinen äußeren Licht-Dunkel-Wechsel gibt. Exakt ist dieser Rhythmus etwas länger als 24 Stunden. Er wird aber normalerweise durch den Tages- und Nacht-Wechsel auf 24 Stunden runter korrigiert. Wegen diesem Rhythmus werden wir gewöhnlich immer zur gleichen Zeit wach oder müde oder ...
Die Frauen brauchen wir nicht wirklich an den ursprünglich Mond-bezogenen Rhythmus von 28 Tagen erinnern. Und die Liste lässt sich weiter fortsetzen.
Interessant ist sicher eine unvoreingenommene Prüfung von vorher beobachteten Daten und den Berechnungen der Kalkulation. Wenigstens die Grundtendenzen (positiv / negativ) sollte stimmen.

	A	B	C	D	E	F	G	H	I
1	**Biorhythmus**								
2									
3	**Name:**	Karl Muster		**Datum**	**Körper**	**Geist**	**Seele**	**Gesamt**	
4	**geb. am:**	12.04.1980		=GANZZAHL (B5-14)	=SIN(REST(D4-$B$4; 23)*2*PI()/23)	=SIN(REST(D4-$B$4; 33)*2*PI()/33)	=SIN(REST(D4-$B$4; 28)*2*PI()/28)	=(E4+F4 +G4)/3	
5	**für Datum:**	=HEUTE()		=D4+1					
6	**Alter [d]:**	=B5-B4							
32									
33									

notwendige Zellen-Formate:

Zelle	Format
B4, B5	Benuterdefiniert; tt.mm.jjjj
B6	Zahl; 0 Nachkommastellen; ev. 1000er Trennung
D4:H4	fett, zentriert

Spalte	Format
E; F; G	Prozent, 0 Nachkommastellen
A	fett

Zeile	
1	Schriftgröße 14

2. Computer-Mathematik mit MuPAD

2.0. Vorbemerkungen und Programm-Versionen

MuPAD gibt es für die Betriebssysteme WINDOWS und LINUX. Neben lizenzpflichtigen Programmversionen gibt es auch freie Versionen. Für die meisten Fälle und zum Ausprobieren reicht eine freie Version aus. Für Schüler und Studenten gibt es verbilligte Lizenzen. Auf der MuPAD-Seite im Internet (→ www.mupad.de) sind auch Bedingungen für den Bezug einer zeitlich begrenzten Voll-Version für Schüler und Studenten zu finden. Bei bestimmten Gegenleistungen (z.B. Hausarbeit usw.) wird die Lizenz dann u.U. entfristet.

2.1. Grundbegriffe

Sitzung:

Mit dem Start des Programms MuPAD beginnt man eine Sitzung. Das Programm-Ende ist gleichzeitig auch das Ende der Sitzung. Zu einer Zeit können durch mehrfaches Starten von MuPAD mehrere unabhängige Sitzungen abgehalten werden.

Notebook: (auch: Worksheet, Dokument)

Ein Notebook (= dt.: Notizbuch) ist sozusagen eine Rechenseite (Notizzettel) in MuPAD. Es können mehrere Notebooks in einer Programmsitzung verwaltet werden. Ein Umschalten zwischen den Notebooks ist jederzeit über das "Fenster"-Menü möglich.
Alle Rechenschritte werden der Reihe nach in einem Notebook verzeichnet. Die Inhalte (Rechenschritte) eines Notebook können korrigiert und dann wieder schrittweise korrekt berechnet werden. Notebooks lassen sich abspeichern und wieder öffnen.
Jedes Notebook ist vollständig von anderen getrennt. Gemachte Eingaben, veränderte Einstellungen (z.B. DIGITS) gelten immer nur für das aktuelle Notebook.

2.2. Eingaben in MuPAD

Innerhalb eines Notebooks sind Eingaben (Anforderungen / Anfragen an das System) und Ausgaben (Antworten / Ergebnisse) gemischt. Die Eingabezeile ist normal rot und beginnt mit einem Punkt (bei WINDOWS; unter UNIX / LINUX mit Doppelwinkel >>). Eingaben werden mit “ENTER“ abgeschlossen und sofort bearbeitet.

```
aufgabe
```

Die Ergebnisse erscheinen blau bzw. bei Grafiken gemischtfarbig.
Automatisch wird sofort wieder eine Eingaberegion (neuer roter Punkt) erzeugt.

aufgabe

Eine Eingabe kann sich auch über mehrere Zeilen erstrecken bzw. sich aus mehreren Einzelzeilen zusammensetzen. Solche Eingabezeilen werden dann nur mit “ ⇑ “ + “ENTER“ abgeschlossen. Erst nach dem “ENTER“ der letzten Eingabe werden dann alle vorherigen Zeilen mit ausgewertet.

Es ist ebenfalls möglich mehrere Eingaben hintereinander in eine Zeile zu schreiben. Die einzelnen Eingaben werden mit einem Semikolon getrennt. Das letzte Semikolon am Ende der Eingabe kann entfallen.
Soll eine Eingabe ohne Ausgabe (hier z.B. **aufg3**) erfolgen (wie bei “ ⇑ “ + “ENTER“), dann wird als Trennzeichen zwischen den Eingaben ein Doppelpunkt eingegebenen.

```
aufg1; aufg2
```

aufg1

aufg2

```
aufg3: aufg4
```

aufg4

Außer den Arbeitseingaben – die MuPad auswerten soll – gibt es die Möglichkeit zusätzlichen Text mit anzugeben. Man kann sich das wie Kommentare oder Beschreibungen / Erklärungen zu den mathematischen Operationen usw. vorstellen. Mit "Einfügen" "Text oben" (Tastenkombination "Strg" + "F6") bzw. "Text unten" ("F6") werden die entsprechenden Bereiche geöffnet.
In den Textbereichen kann man viele Textverarbeitungsfunktionen zum Formatieren des Textes (Schriftattribute (fett, kursiv, unterstrichen), Schriftart, Schriftgröße und Schriftfarbe) nutzen.

Dies ist reiner Text, z.B. für Erläuterungen und Kommentare. Er wird von MuPAD nicht ausgewertet.

Ein ***Text*** lässt **sich auch** - wie in **Textverarbeitungen** *üblich* - gestalten.

2.2.1. Möglichkeiten zur Erstellung von Dokumenten mit mathematischem Inhalten

Wie oben beschrieben, lassen sich in MuPAD vollständige Dokumente zu mathematischen Sachverhalten erstellen. Durch geeignetes Mischen von Eingaben, Ausgaben und (Erläuterungs-)Texten sind alle Wünsche des Nutzers realisierbar.
Ein anderer Weg ist die Nutzung von MuPAD-Eingaben und / oder –Ausgaben in anderen Programmen. Als leistungsfähige Textverarbeitung bietet sich z.B. microsoft-WORD an. Um Ein- oder Ausgaben von MuPAD in WORD oder anderen Programmen nutzen zu können müssen diese in MuPAD markiert (z.B. durch überzie-

hen mit der Maus oder " ⇑ " und den Pfeil-Tasten) und in die Zwischenablage kopiert werden (z.B. über "Bearbeiten" "Kopieren" oder "Strg" + "Einfg").
Danach wechselt man in das Programm, in das eingefügt werden soll und setzt den Cursor an die gewünschte Einfüge-Position. Nun kann man (z.B. über "Bearbeiten" "Einfügen" oder " ⇑ " + "Einfg") den Inhalt der Zwischenablage einfügen. Auf Wunsch lässt sich der Vorgang (Einfügen) beliebig wiederholen, solange bis die Zwischenablage mit einem anderen Inhalt gefüllt wurde.

2.2.2. Korrektur von Eingaben

Fehlerhafte Eingaben lassen sich wie üblich in der entsprechenden Zeile korrigieren. Nach einem “ENTER“ wird die Zeile erneut bearbeitet. Nachfolgende Zeilen, die irgendwie mit der geänderten Zeile zusammenhängen, müssen dann ebenfalls noch mal mit “ENTER“ zur erneuten Bearbeitung übergeben werden.
Fehlende Zeilen lassen sich auch später noch ergänzen. Dazu setzt man den Cursor an die Position, wo die Zeile hin soll und erstellt sich eine neue (vorstehende) Eingabezeile mit "Einfügen" "Eingabe oben" (bzw. Tastenkombination "Strg" + "F5"). Eine nachfolgende Eingabezeile erstellt man mit der Taste "F5" bzw. "Einfügen" "Eingabe unten".
Zwischengeschobene Eingabezeilen , die nur mit “ ⇑ “ + “ENTER“ bestätigt wurden, sind nicht vollständig übernommen worden. Dazu ist ein abschließendes “ENTER“ für den gesamten Eingabebereich notwendig!

2.2.3. Erzeugen eines neuen Notebooks

Für jede separate Aufgabe sollte man ein extra Notebook verwenden. Über "Datei" "Neues Notebook" bzw. die Tastenkombination "Strg" + "N" lässt sich das schnell erledigen.
Auch in der "Symbolleiste" steht eine passende Schaltfläche ("Neu" / "Erstellt ein neues Dokument.") zur Verfügung. 

2.3. Vorbereitungen

2.3.1. Säubern des Notebooks

Die einfachste Methode um ein sauberes Notebook zu erhalten, ist sich ein neues anzulegen (→ 2.2.3. Erzeugen eines neuen Notebooks).
Sollen dagegen aber alte Variablen und Einstellungen erhalten bleiben, dann kann man die Anzeige mit "Notebook" "Ausgaben löschen" säubern. Dabei werden alle Ausgaben gelöscht. Die Eingabezeilen bleiben mit ihren Übernahmeanweisungen ("ENTER" bzw. " ⇑ " + "ENTER") erhalten.
Einzelne Ausgaben / Eingaben lassen sich wie folgt löschen:

1. *Markieren mit der Maus (z.B. Anklicken bzw. Überstreichen)*
2. *den gesamten Bereich kennzeichnen mit "Notebook" "Bereich auswählen" oder Taste "F7"*
3. *Löschen mit Taste "Entf"*

Dieses Verfahren ist auch dann notwendig, wenn ein geladenes Notebook weiter verwendet werden soll. Zusätzlich muß man dann noch einmal alle Eingaben bestätigen, damit alle Variablen usw. wieder mit den richtigen Werten beladen werden.

Aufgaben:

1. *Erstellen Sie sich ein neues Notebook und speichern Sie dieses unter* "Übung1 Eingaben.mnb" *auf Ihrer Diskette bzw. in Ihrem Home-Verzeichnis ab! Achten Sie zukünftig selbstständig auf regelmäßiges Zwischenspeichern des Notebooks!*
2. *Erzeugen Sie sich vor dem aktuellen Eingabecursor ein Textfeld!*
3. *Geben Sie als Überschrift (fett, unterstrichen) die Zeile "Übung1 Eingaben" an!*
4. *Beschreiben Sie im normalen Text kurz, dass Sie jetzt testen wollen, wie das MuPAD-System auf die Eingabe Ihres Vornamen reagiert. Formulieren Sie noch eine kursiv und grün gestaltete Vermutung dazu!*
5. *In der Eingabezeile tippen Sie nun Ihren Vornamen (Rufnamen) an!*
6. *Erläutern Sie in einem nachfolgen Textfeld, warum Ihrer Meinung nach das MuPAD-System so geantwortet hat!*

2.4. Rechnen mit Zahlen / Taschenrechner-Funktion

Um uns in die Eingabe und einige Prinzipien dieses Computer-Algebra-Systems (Abk.: CAS) einzustimmen, wollen wir es zuerst wie einen Taschenrechner benutzen – wenn auch einen sehr viel leistungsfähigenen.

Beginnen wir mit einfachen Rechnungen. Die Addition zweier Zahlen ist schnell erledigt. Einfach die Aufgabe ohne Gleichheitszeichen eingetippt und "ENTER" gedrückt.

```
1234 + 567
```
1801

Nun wollen wir zwei Zahlen multiplizieren. Beide Zahlen sollen Kommastellen haben.

Wieder die Aufgabe eingetippt und die Eingabe bestätigt erhalten wir → (siehe auch nebenan)
völligen Unsinn?!?

```
12,34 * 5,67
```
12, 170, 67

Natürlich nicht! MuPAD hat – die für uns Deutsche unangenehme – computertypische / englische Zahlendarstellung eingebaut. Dezimalstellen müssen mit Punkt abgetrennt werden. Kommas bedeuten in MuPAD Abtrennungen von Zahlen z.B. wie für Punktkoordinaten (x,y) oder Aufzählungen / Mengen [....].

Also noch mal richtig eingegeben und bestätigt:
Mit diesem Ergebnis können wir wohl zufrieden sein.

```
12.34 * 5.67
```
69.9678

Als nächstes wollen wir Brüche berechnen.

Brüche oder Divisionen werden (- wie bei EXCEL -) linearisiert mit Schrägstrich eingegeben.
Das Ergebnis ist exakt, wenn auch vielleicht nicht unbedingt so erwartet.

```
9/12
```
$\frac{3}{4}$

Exponenten lassen sich über das Sonderzeichen **^** ("Akzent") eingeben. Beachten Sie dabei bitte, dass bei der Eingabe nach dem Drücken der Taste **"^"** zuerst nichts auf dem Bildschirm passiert. Erst nach dem nächsten Zeichen wird das Akzent-Zeichen auch angezeigt.

```
10^3
```
1000

MuPAD ist ein exaktes System. D.h. Brüche usw. werden ohne numerische Nährung bearbeitet. Somit treten keine Rundungsfehler oder andere Unschönheiten auf.
Berechnen Sie doch die folgende Aufgabe auch mit dem Taschenrechner!

$$(1+\frac{\frac{1}{3}}{10^7}-1)\cdot 3\cdot 10^7$$

```
(1+1/3/10^7-1)*10^7*3
```
1

Durch Rundungsprobleme entsteht ein fast genaues – aber unschönes –Ergebnis (z.B.: CASIO fx-85v zeigt: ***0.9999***).
In MuPAD erhalten wir ein exaktes Ergebnis – ohne wenn und aber.
(Der Effekt hängt vom Taschenrechner ab. Er läßt sich aber mit höheren Zehnerpotenzen oder anderen Aufgaben bei fast jedem Taschenrechner erzielen. Moderne CAS-Taschenrechner arbeiten wie MuPAD exakt!)

Oft benötigen wir aber statt einem Bruch die numerische Näherung als Ergebnis. Diese lässt sich in MuPAD über die Funktion ***float*** anfordern.

```
9/12
```
$\frac{3}{4}$

Die Funktion ***float*** liefert immer eine Fließkommazahl (Gleitkommazahl, Dezimalzahl).

```
float(9/12)
```
0.75

Andere Funktionen oder Rechen-Operatoren müssen wir über Funktionsnamen aufrufen.

Die Quadratwurzel wird über den Namen ***sqrt*** benutzt. Und wieder erhalten wir ein exaktes Ergebnis. Mit float könnten wir das Ergebnis dann in "verständlicher" Form anzeigen lassen.

```
sqrt(8)
```
$2 \cdot \sqrt{2}$

Die oft benötigte Expotential-Funktion erhalten wir mit ***exp***. Hinter ***ln*** verbirgt sich der passende natürliche Logarithmus. Den dekadischen Logarithmus (**lg**) muß man über die allgemeine ***log***-Funktion ermitteln. Als erstes Argument erwartet diese Funktion immer die Basis des Logarithmus (hier z.B. 10) und als zweites den zu berechnenden Wert (z.B. 1000).

```
ln(1)
```
0

```
log(10,1000)
```
3

Die Winkelfunktionen ***sin***, ***cos***, ***tan*** und ***cot*** werden wie üblich verwendet. Die Winkel müssen in rad eingegeben werden. Die Umrechnung der Grad-Winkel erfolgt gegebenenfalls über die Multiplikation mit 0.0175 (exakt: π / 180° ≈ 0.01745329252)

```
sin(1.570796327)
```
1.0

```
sin(90*0.01745329252)
```
1.0

Da sind wir auch gleich bei dem berühmt berüchtigten π. MuPAD kennt auch dieses und zwar exakt. Intern wird π durch ***PI*** repräsentiert. Mit ***float(PI)*** bekommen wir den Wert ausgegeben. Aber halt, war π nicht eine unendliche rationale Zahl?
Doch natürlich. Die numerische Näherung in MuPAD ist auf 10 Nachkommastellen (***DIGITS***) eingestellt. Auf Wunsch kann man sich diese aber beliebig genau einstellen. Probieren Sie es ruhig mal mit 1000 Nachkommastellen.
Die EULERsche Zahl **e** (entsp. ***exp(1)***) wird als ***E*** verwendet.
Mit ***delete DIGITS*** lässt sich die Grundeinstellung (10) wieder herstellen.

```
float(PI)
```
3.141592654

```
DIGITS:=1000
float(PI)
```
….
(Die Ausgabe wurde bewusst nicht abgedruckt!)

```
DIGITS:=12:
float(E)
```
2.71828182846

Neben der "normalen" (rationalen) Teilung von Zahlen spielt die Teilung mit und ohne Rest im Bereich der natürlichen Zahlen für viele Aufgaben eine Rolle. Mit ***div*** (diviso) wird der Quotient und mit ***mod*** (modulo) der Rest bestimmt.
Mit dem **%**-Operator greift man auf das so und so vielte, zurückliegende Ergebnis zurück. Um nur das letzte Ergebnis weiter zu benutzen reicht auch die alleinige Angabe von **%** aus.

```
23 div 3
```
7

```
23 mod 3
```
2

```
3*%2+%1
```
23

Den absoluten Betrag einer Zahl stellt die Funktion ***abs*** zur Verfügung. Nun fehlt uns – zu unserem mathematischen Glück – noch die Vorzeichen-Funktion ***sign*** (Signum).

- `sign(-23.4)`

$$-1$$

Rundungen auf die nächste ganze Zahl erledigt ***round***. Mit ***ceil*** wird aufgerundet und mit **floor** immer abgerundet.

- `round(-23.78)`

$$-24$$

Der Funktion ***floor*** ist die Funktion ***trunc*** ebenbürtig. Sie schneidet die Nachkommastellen einfach ab. Die Nachkommastellen selbst kann man über ***frac*** erhalten.

- `trunc(-23.78)`

$$-23$$

Betrachten wir noch einige interessante Funktionen, welche die speziellen Fähigkeiten von MuPAD als Super-Taschenrechner belegen.

Wenn wir z.B. wissen wollen, ob die Zahlen 7552029 und 7552031 Primzahlen sind, dann lassen uns normale Taschenrechner schnell im Stich. In MuPAD existiert die Funktion ***isprime***. Als Ergebnis bekommen wir entweder **true** oder **false** (dt. wahr bzw. falsch) zurück.

- `isprime(7552029)`

false

- `isprime(7552031)`

true

Weitere interessante Funktionen für den Umgang mit Primzahlen sind ***nextprime*** und ***ithprime***. Mit ***nextprime*** kann man die erste - nach einer vorzugebenen Zahl – folgende Primzahl ermitteln lassen. Sucht man die so und so vielte Primzahl, dann ist ***ithprime*** die richtige Funktion.

Genau so interessant ist die Funktion ***ifactor***, die eine Zahl in ihre Primfaktoren zerlegt. Exakt bedeutet dies, dass eine Zahl in ein Produkt aus Primzahlen-Potenzen zerlegt wird.

- `ifactor(12345)`

$$3 \cdot 5 \cdot 823$$

Die Zerlegung bestimmter großer Zahlen in ihre (2 (öffentlicher und geheimer Schlüssel)) Primfaktoren ist ein wichtiges Problem in der Kryptographie (Kryptoanalyse (Codebrechen)). Je größer die Zahlen werden, um so länger dauert die Zerlegung in Primfaktoren. Auch MuPAD hat unter diesem Problem zu leiden.

Ohne wirklichen Zeitaufwand ermittelt man aber in MuPAD den größten gemeinsamen Teiler (**ggT**).

Die Funktion ***igcd*** (integer greatest common divisor) ermittelt den ggT bei Übergabe zweier natürlicher Zahlen.

- `igcd(24,16)`

$$8$$

Bleibt aus dem Bereich der Zahlentheorie noch das kleinste gemeinsame Vielfache (**kgV**). Hier bietet MuPAD die Funktion ***ilcm*** (integer least common multiple) an.

- `ilcm(24,16)`

$$48$$

Aufgaben:

1. Berechnen Sie die folgenden Ausdrücke!

$3*12*1{,}345$; $\frac{7+4*3}{5}$; $\frac{12}{7}+\frac{17}{9}-\frac{8}{5}$; $3a+\frac{a}{2}-5\frac{3}{7}a$

2. Geben Sie zu den nachfolgenden Ausdrücken die numerischen Ergebnisse an!

$\frac{12}{7}+\frac{17}{9}-\frac{8}{5}$; $(8+3)*(\frac{5}{2}-\frac{16}{4})+\frac{2}{3}$; $\frac{\frac{2}{3}*\frac{4}{7}}{\frac{3}{2}+\frac{7}{4}}*\frac{\frac{12}{5}}{\frac{9}{4}+\frac{1}{6}}$

3. Bestimmen Sie, ob die folgenden Zahlen Primzahlen sind!

1089 ; 58451 ; 129341 ; 1000000000 ; 450968948897

4. Zerlegen Sie in die Primfaktoren!

1098 ; 9458234 ; 1000000000 ; 6307385169753

5. Runden Sie die nachfolgen Zahlen auf und ab sowie exakt!

12,45 ; 31,8904 ; 4,5 ; -0,7654 ; -23,45

6. Welches Kommando könnte man sich zusammenstellen, um die Rundung auf 3-Nachkommastellen zu erreichen? Prüfen Sie die Kommandofolge an:

$0{,}28603$; $\frac{2}{3}$; $5\sqrt{7}$

Für FREAKS:

7. Zerlegen Sie die folgenden Zahlen in ihre Primfaktoren! Messen Sie die Zeit vom Bestätigen der Eingabe bis zum Erscheinen der Ausgabe! (Vorher eine Testaufgabe zum Aktivieren des MuPAD-Kerns starten!)

6811297 ; 215936196659 ; 97546105974089317335787229159244024 1 ; 9754610597408931664320987643194635966651426 7

Kurzreferenz: Taschenrechnerfunktionen usw. usf.

! In dieser Referenz werden vor allem häufig benötigte Funktionen usw. usf. sowie der häufig gebrauchte Syntax aufgeführt. Für vollständige Informationen wählen Sie bitte die Hilfe bzw. die Programm-Referenz!

DIGITS		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
DIGITS	liefert die Anzahl der aktuell gültigen Dezimalstellen zurück	DIGITS ist eine vordefinierte Umgebungsvariable und darf nicht für andere Zwecke verwendet werden
DIGITS:=*anzahl*	setzt die Anzahl der Dezimalstellen ab sofort auf den Wert, der durch *anzahl* angegeben wird	Beachten Sie, dass eine geringe Anzahl von Dezimalstellen – vor allem bei längeren Rechnungen – zu beachtlichen Rundungsfehlern führen können!
delete DIGITS	setzt die Anzahl der Dezimalstellen auf den Standardwert 10 zurück	

sqrt		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
sqrt(*ausdruck*)	liefert die Quadratwurzel (square root) von *ausdruck*	*ausdruck* kann eine Zahl, eine Variable oder ein mathematischer Ausdruck sein

float		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
float(*ausdruck*)	liefert die numerische Näherung des Parameters *ausdruck* die Genauigkeit wird durch DIGITS bestimmt	*ausdruck* kann eine Zahl, eine Variable oder ein mathematischer Ausdruck sein

ln		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
ln(*ausdruck*)	berechnet den natürlichen Logarithmus von *ausdruck* $f(x) = \log_e(ausdruck)$	*ausdruck* kann eine Zahl, eine Variable oder ein mathematischer Ausdruck sein

exp		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
exp(*ausdruck*)	bestimmt den Expotential-Funktionswert zu *ausdruck* $f(x) = e^{ausdruck}$	*ausdruck* kann eine Zahl, eine Variable oder ein mathematischer Ausdruck sein

log		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
log(*basis,ausdruck*)	liefert den Logarithmus von *ausdruck* zur *basis* zurück $f(x) = \log_{basis}(ausdruck)$	log(10,*ausdruck*) … liefert also den dekadischen Logarithmus von *ausdruck* log(E,*ausdruck*) entspricht ln(*ausdruck*)

sin		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
sin(*ausdruck*)	gibt den passenden Sinus zum *ausdruck* (in Bogenmaß, [rad] Radiant) zurück	*ausdruck* kann eine Zahl, eine Variable oder ein mathematischer Ausdruck sein

cos		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
cos(*ausdruck*)	gibt den passenden Cosinus zum *ausdruck* (in Bogenmaß, [rad] Radiant) zurück	*ausdruck* kann eine Zahl, eine Variable oder ein mathematischer Ausdruck sein

tan		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
tan(*ausdruck*)	gibt den passenden Tangens zum *ausdruck* (in Bogenmaß, [rad] Radiant) zurück	*ausdruck* kann eine Zahl, eine Variable oder ein mathematischer Ausdruck sein

cot		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
cot(*ausdruck*)	gibt den passenden Cotangens zum *ausdruck* (in Bogenmaß, [rad] Radiant) zurück	*ausdruck* kann eine Zahl, eine Variable oder ein mathematischer Ausdruck sein

div		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
dividend div *divisor*	liefert den ganzzahligen Quotient von *dividend* / *divisor* ohne Beachtung des Rests	

mod		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
dividend mod *divisor*	liefert den ganzzahligen Rest von *dividend* / *divisor* ohne Beachtung des eigentlichen Quotienten	

abs		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
abs(*reele_zahl*)	bestimmt von *reele_zahl* den absoluten Betrag (Abstand von 0 auf der Zahlengerade)	

sign		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
sign(*reele_zahl*)	bestimmt von *reele_zahl* das Vorzeichen / den Richtungsvektor	

round		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
round(*reele_zahl*)	rundet *reele_zahl* exakt auf eine ganze Zahl	• `round(23.4)` 23 • `round(-23.4)` −23

ceil		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
ceil(*reele_zahl*)	rundet *reele_zahl* auf die nächst größere ganze Zahl auf	• `ceil(23.4)` 24 • `ceil(-23.4)` −23

floor		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
floor(*reele_zahl*)	rundet *reele_zahl* auf die nächst kleinere ganze Zahl ab	• `floor(23.4)` 23 • `floor(-23.4)` −24

trunc		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
trunc(*reele_zahl*)	schneidet von *reele_zahl* die Nachkommastellen ab	• `trunc(23.4)` 23 • `trunc(-23.4)` −23

frac		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
frac(*reele_zahl*)	liefert die Differenz von *reele_zahl* zur nächst kleineren ganzen Zahl	• `frac(23.4)` 0.4 • `frac(-23.4)` 0.6

isprime		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
isprime(*zahl*)	gibt einen Wahrheitswert (TRUE ≡ wahr bzw. FALSE ≡ falsch) zurück, je nachdem ob *zahl* eine Primzahl ist oder nicht	für *zahl* kann auch eine Variable stehen

nextprime		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
nextprime(*zahl*)	gibt die nächstfolgende Primzahl nach *zahl* zurück	für *zahl* kann auch eine Variable stehen

ithprime		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
ithprime(*zahl*)	liefert die *zahl*-te Primzahl (aus der Liste aller Primzahl)	für *zahl* kann auch eine Variable stehen

ifactor / factor		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
ifactor(*zahl*)	liefert die Primfaktorenzerlegung der *zahl* zurück	für *zahl* kann auch eine Variable stehen

ic		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
igcd(*zahl1,zahl2*)	(integer greatest common divisor) bestimmt den größten gemeinsamen Teiler (ggT) eines Zahlenpaares *zahl1* und *zahl2*	für *zahl1* und / oder *zahl2* können auch Variablen stehen

ic		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
ilcm(*zahl1,zahl2*)	(integer least common multiple) bestimmt das kleinste gemeinsame Vielfache (kgV) des Zahlenpaares *zahl1*, *zahl2*	für *zahl1* und / oder *zahl2* können auch Variablen stehen

2.5. Rechnen mit komplexen Zahlen

Komplexe Zahlen sind die Lösung für das Problem, das sich aus negativen (reellen) Zahlen (**ℝ**) keine Wurzel berechnen lässt.

Auf das Problem stößt man z.B., wenn die quadratische Funktion $x^2+1=0$ gelöst werden soll. Nach Anwendung der Lösungsformel bleibt $x_{1,2}=\sqrt{-1}$ übrig. Somit wäre die genannte quadratische Gleichung im Bereich der reellen Zahlen nicht lösbar.

Zur Lösung des Problems wird eine imaginäre Zahl **i** mit $i^2=-1$ definiert. Passend dazu ergibt sich, dass $i=\sqrt{-1}$ ist. Der Begriff wurde vom italienischen Mathematiker BOMBELLI im 16. Jahrhundert eingeführt, um zu verdeutlichen, das die Zahl eine Vorstellung sei. Später wurde sie dann auch als komplexe Zahl bezeichnet.

Jede Zahl lässt sich nun bei Bedarf so zerlegen, dass das negative Vorzeichen abgetrennt werden kann und der Rest wie üblich behandelt wird.

$-25=25*-1$

Dadurch lässt sich nun auch bei $x^2=-25$ mindestens eine Lösung für x berechnen:

$x=\sqrt{-25}$ → ? (so geht's erst mal nicht in **ℝ**)

Aber über $x^2=-25=25*-1$ und dann $x=\sqrt{25}*\sqrt{-1}$ ergibt schließlich $x=5i$.

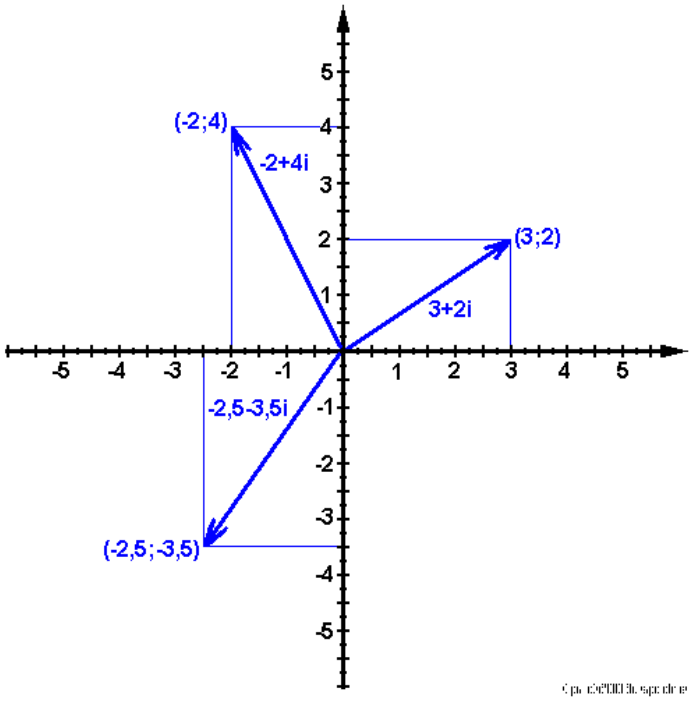
Eine sinnige Herleitung von Real- und Imaginärteil wird in einer zukünftigen Version aufgezeigt.

Allgemein wird eine komplexe Zahlen **z** (aus der Menge der komplexen Zahlen **ℂ**) in der Form $z=x+yi$; x,y $\in$ **ℝ** (algebraisch) dargestellt. **x** stellt den Realteil der Zahl **z** und **y** den Imaginärteil dar. In der Mathematik ist auch eine Darstellung als geordnetes Paar zweier reeller Zahler üblich (Paardarstellung):

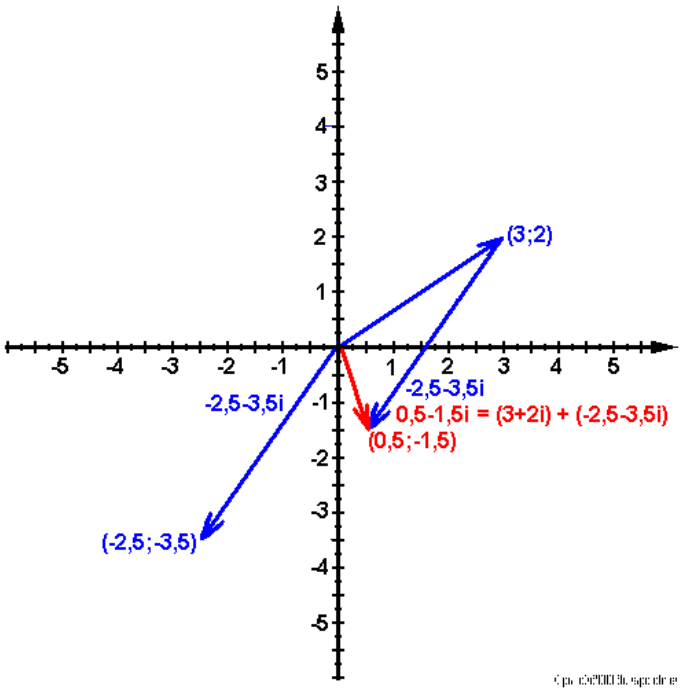
z.B.: $(3;2)\equiv 3+2i$ $(-2{,}5;-3{,}5)\equiv -2{,}5-3{,}5i$

Für die graphische Darstellung der komplexen Zahlen reicht nun die eine Zahlengerade für den Realteil nicht mehr aus.

Wir benötigen eine zweite Dimension für den Imaginärteil. Es handelt sich dann um eine Zahlenebene (GAUSSsche Zahlenebene), wobei jede komplexe Zahl ein Punkt in dieser Ebene darstellt. Die Paardarstellung wird somit zur Punktdarstellung. Real- und Imaginärteil stellen den jeweiligen gerichteten Anteil der Dimensionen dar. Jede komplexe Zahl **z** entspricht damit einem Vektor, der vom Koordinatenursprung zu dem Punkt zeigt, welcher der komplexen Zahl entspricht. Komplexe Zahlen lassen sich also sowohl als Punkt als auch als Vektor darstellen.



Der Betrag des Vektors entspricht dem Betrag der komplexen Zahl.
Die graphische Darstellung der imaginären Zahl als Vektor (Strahl) vom Koordinatenursprung zum Punkt der Addition der Vektoren von Real- und Imaginärteil macht deutlich, dass die komplexe Zahl etwas Neues ist.
Bei der Verknüpfung von komplexen Zahlen werden die Rechenregel wie üblich angewendet. Dabei ist natürlich zu beachten, dass zwischen **y** und **i** ein Punkt (Multiplikationsoperator) steht. Somit müssen Real- und Imaginärteil jeweils für sich betrachtet verrechnet werden.



Zur Veranschaulichung zeigen wir kurz die Addition zweier komplexer Zahlen z_1 und z_2 etwas ausführlicher:

$$z_1 = x_1 + i \cdot y_1$$
$$z_2 = x_2 + i \cdot y_2$$

$$z_1 + z_2 = x_1 + x_2 + i \cdot (y_1 + y_2)$$

Für die Multiplikation gilt entsprechend:

$$z_1 \cdot z_2 = x_1 \cdot x_2 - y_1 \cdot y_2 + i \cdot (x_1 \cdot y_2 + x_2 \cdot y_1)$$

Subtraktion und Division lassen sich leicht daraus ableiten bzw. passend entwickeln. Nur wenige moderne Taschenrechner lassen unkomplizierte Rechnungen mit komplexen Zahlen zu. Dagegen ist MuPAD durch die Definition von **I** als imaginäre Einheit recht einfach für Rechnungen mit komplexen Zahlen zu nutzen.

Das Rechnen mit kompl. Zahlen in MuPAD bleibt einer zukünftigen Version vorbehalten.

Kurzreferenz: Rechnen mit komplexen Zahlen

! In dieser Referenz werden vor allem häufig benötigte Funktionen usw. usf. sowie der häufig gebrauchte Syntax aufgeführt. Für vollständige Informationen wählen Sie bitte die Hilfe bzw. die Programm-Referenz!

I		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
imaginärwert * I	legt den Imaginärteil einer komplexen Zahl auf *imaginärwert* fest	*imaginärwert* $\in$ $\mathbb{R}$ kann praktisch Ganz- (Integer) bzw. Fließkommazahl (Real) sein

Aufgaben:

2.6. Symbole und Bezeichner

In MuPAD können wir – wie in der Algebra üblich – auch mit Buchstaben, Zeichen und anderen Objekten arbeiten. Die Eingabe eines Buchstabens (oder mehrerer z.B. einem Namen) bewirkt die Ausgabe des gleichen Symbols. Als ein solches benutzt MuPAD es nämlich. Als Namen / Symbole sind alle Kombinationen aus Buchstaben, Ziffern und dem Unterstrich _ zugelassen. Sie dürfen nur nicht mit einer Ziffer beginnen.

```
a
```
a

```
dd
```
dd

```
peter
```
peter

Zwischen Groß- und Kleinschreibung wird unterschieden. ***Peter*** und ***peter*** sind also zwei verschiedene Symbole, wie die nebenstehenden Ausgaben von MuPAD belegen.
Im Interesse einer besseren Lesbarkeit sollte man sich auf einen bestimmten Schreibstil für Symbole usw. festlegen.

```
peter + peter
```
$2 \cdot peter$

```
Peter + peter
```
Peter + peter

Wie wir gerade gesehen haben, kann mit solchen Symbolen auch gerechnet werden. Darauf kommen wir aber später ausführlich zurück.

```
mutter + vater + kind
```
mutter + vater + kind

Oft ist einem Symbol ein Wert zugeordnet – wir benutzen das Symbol als Variable (Veränderliche) bzw. als Bezeichner (Name, engl.: identifier) für einen Wert.
In MuPAD erledigt man diese Zuweisung über das Ergibtzeichen (Zuweisungsoperator) ***:=*** . Die Ausgabe der Werte für die Variablen bestätigt die Akzeptanz der Eingabe.

```
a:=23; b:=34;
```
23
34

```
familie:=mutter + vater + kind
```
kind + vater + mutter

Von nun an wird beim Anwenden des Symbols immer dessen Wert benutzt. Auch die Anwendung in Funktionen oder Rechnungen entspricht den üblichen Geflogenheiten.
Der Wert einer Variable kann jederzeit durch Angabe ihres Namens abgefragt werden. Erfolgt die Ausgabe des Namens selbst, dann hat die Variable keinen Wert (entspricht also einem Symbol).

```
a*b
```
782

```
familie
```
kind + vater + mutter

```
b:=float(log(10,a))
```
1.361727836

```
b
```
1.361727836

Bezeichner lassen sich auch Gleichungen, Funktionen usw. zuweisen (→ 2.7.3. Lösen von Gleichungen). Somit ist es uns z.B. möglich, eine eigene Sinus-Funktion zu definieren.
Für Funktionen bietet sich aber auch eine Bezeichnung über die Definition einer Prozedur an. Hierfür wir der Pfeiloperator **->** verwendet. Der Vorteil liegt in der schulmathematiknahen Darstellung und späteren Verwendung der Funktion in der Form **f(x) -> x** (dazu näheres: → 2.7.3. Lösen von Gleichungen).

```
meinsin:=sin
```
sin

```
sin(PI/3)
```
$\frac{\sqrt{3}}{2}$

```
meinsin(PI/3)
```
$\frac{\sqrt{3}}{2}$

Symbole, denen noch kein Wert zugewiesen wurde, bleiben wertlos und werden solange auch als Symbol (→ 2.8. Symbolisches Rechnen) benutzt.

- `a*peter`

  $23 \cdot peter$

Durch die Zuweisung eines neuen Wertes wird der alte Wert überschrieben. Möchte man die Variable völlig wertlos - also wieder zum Symbol – machen, dann bietet sich die Funktion ***delete*** an. Beachten Sie, das ***delete*** nicht etwa die Variable auf **0** oder irgend etwas anderes setzt, dass müssen Sie u.U. selbst erledigen!

- `delete a`
- `a+peter`

  $a + peter$
- `a:=0: a+peter`

  $peter$

Aufgaben:

1. *Bewerten Sie die nachfolgenden Namen (semikolongetrennt), ob sie sich gefahrlos als Bezeichner in MuPAD verwenden lassen! Begründen Sie ihre Meinung! Prüfen Sie die Bezeichner durch Eingabe in MuPAD!*
   x ; 4 ; 3X ; axe ; tan ; Depp ; E ; ein Hund ; _VarX ; _Zwischenwert 3 ; Kennwert§_2

2. *Belegen Sie die nachfolgenden Variablen mit den angegebenen Werten!*

Variable	Wert
Depp	0
T3	1000000
T23	1.23456

Variable	Wert
x	12
y1	12.0
T3	2.51*10^-5

3. *Berechnen Sie die nachfolgenden Verknüpfungen und legen Sie das Ergebnis immer auf die Variable X! Vergewissern Sie sich gegebenenfalls über den aktuellen Wert von X durch Kontrollanzeigen von X!*

Depp + T23
T3 * T3
T23 * x
T23 * y1
T23 / Depp

x * y1
Depp +y1 * x
y1 + x * 0.234 + sin(Depp)
ln(y1) * 0 *T23 + tan(Depp) + 2 * Depp
y1 * T23 + T3 / T23 + Depp *x

2.6.1. Mengen

Mehrere MuPAD-Objekte lassen sich z.B in einer Menge (engl. set) zusammenfassen. Mengen werden durch geschweifte Klammern gekennzeichnet.
Eine leere Menge besteht nur aus den geschweiften Klammern. In Mengen lassen sich beliebige Elemente – auch aus verschiedenen Bereichen – zusammenstellen.
Die Darstellungsreihenfolge der Elemente einer Menge unterscheidet sich meist von der Eingabereihenfolge.

```
{2,4,6,8}
```
$\{2, 4, 6, 8\}$

```
Menge1:={}
```
$\varnothing$

```
Menge2:={3,1,6}
```
$\{1, 3, 6\}$

```
Menge3:={3,4,5,a,sqrt(49),sin(x)}
```
$\{a, \sin(x), 3, 4, 5, 7\}$

MuPAD sortiert die Elemente nach eigenen Kriterien. Prinzipiell spielt die Reihenfolge der Elemente aber in einer Menge keine Rolle. (Ansonsten handelt sich eigentlich um eine Liste → 2.6.2. Listen)

Benötigt man einzelne oder alle Elemente einer Menge, dann kann man mit ***op*** auf die Mengenelemente zugreifen.
Die Anzahl der enthaltenen Elemente einer Menge erhalten wir über ***nops***.
Über das so und so vielte Elemente kann man mit zusätzlicher Angabe der Elementnummer verfügen.

```
op(Menge2)
```
$6, 1, 3$

```
nops(Menge2)
```
3

```
op(Menge3,2..4)
```
$7, a, 5$

Ob es sinnvoll ist, müssen Sie bei der Verwendung entscheiden, da Mengen ja ungeordnet sind und damit der Zugriff so nicht vorgedacht ist.

Wichtiger ist da schon die Frage, ob ein bestimmtes Objekt ein Element der Menge ist. Die Funktion ***contains*** ermöglicht uns diese Prüfung. Als Ergebnis erhalten wir ***true*** (für wahr, ja) oder ***false*** (für falsch, nein)

```
contains(Menge2,sqrt(49))
```
false

```
contains(Menge3,sqrt(49))
```
true

Zwei Mengen lassen sich durch ***union*** zusammenfassen. Dabei sind dann alle doppelten (od. mehrfachen) Elemente nur noch einfach vorhanden – wie wir es bei Mengen erwarten.

```
Menge4:=Menge2 union Menge3
```
$\{a, \sin(x), 1, 3, 4, 5, 6, 7\}$

Weitere Mengenoperationen sind die Schnittmengen- und die Differenzbildung. In MuPAD heißen die Kommandos dafür ***intersect*** und ***minus***.

```
Menge5:=Menge2 intersect Menge3
```
$\{3\}$

```
Menge2 minus Menge3
```
$\{1, 6\}$

2.6.2. Listen

In Listen (engl. lists) werden Elemente in einer bestimmten Reihenfolge zusammengefasst betrachtet. Die Elemente einer Liste können mehrfach auftreten. Listen können geordnet oder ungeordnet sein.
Die Listenelemente werden in MuPAD in eckige Klammern eingeschlossen. Auch hier bedeuten die leeren Klammern eine leeres Sammelobjekt – also eine leere Liste.

```
[2,4,6,4,4,8]
```
$[2, 4, 6, 4, 4, 8]$

```
Liste1:=[]
```
$[]$

```
Liste2:=[3,1,6]
```
$[3, 1, 6]$

```
Liste3:=[3,4,5,a,sqrt(49),sin(x)]
```
$[3, 4, 5, a, 7, \sin(x)]$

Da die Reihenfolge in einer Liste von Bedeutung ist, wird diese auch von MuPAD nicht geändert. Die Anzeige entspricht also der Eingabe.

Eine interessante Möglichkeit - eine Liste zu bilden - ist der Folge-Operator ***$***.

```
Folge:=k*k $ k=1..5
```
$1, 4, 9, 16, 25$

Es wird im Prinzip eine verkürzt geschriebene Zählschleife aufgebaut. (Lesen könnte man den Konstrukt etwa so: Die Folge wird gebildet aus/über: k mal k für k gleich 1 bis 5.)

Wie bei den Mengen können wir auf die Listenelemente über ***op*** zugreifen. Hier macht es natürlich nun auch Sinn, das so und so vielte Element zu benutzen.
Mit ***nops*** läßt sich die Länge der Liste bestimmen, was der Anzahl der Elemente entspricht.

```
op(Liste2)
```
$3, 1, 6$

```
op(Liste3,2..4)
```
$4, 5, a$

```
nops(Liste2); nops(Liste1)
```
3

0

Da reift in uns schnell der Wunsch, ein neues Element an die Liste anzuhängen. Die nötige Funktion heißt ***append***. Bei Bedarf können auch mehrere Elemente mit einem Mal angehängt werden.

```
append(Liste2,5)
```
$[3, 1, 6, 5]$

```
Liste2:=append(Liste2,3,8,1,3)
```
$[3, 1, 6, 3, 8, 1, 3]$

Um ein Element neu zu definieren – also zu ändern-, verwenden wir ***subsop*** unter Angabe des Elementnummer und des neuen Wertes.

```
subsop(Liste2,4=9)
```
$[3, 1, 6, 9, 8, 1, 3]$

Beachten Sie, dass ungewöhnlicherweise die Zuweisung über ein Gleichheitszeichen erfolgt!

Das Aneinanderhängen von Listen kann auch mit ***_concat*** bzw. dem Punkt-Operator erledigt werden.
Beide Funktionen lassen sich um beliebig viele Teillisten erweitern.

```
ListeA:=_concat(Liste2,Liste3)
```
$[3, 1, 6, 3, 8, 1, 3, 3, 4, 5, a, 7, \sin(x)]$

```
ListeB:=Liste2.Liste3
```
$[3, 1, 6, 3, 8, 1, 3, 3, 4, 5, a, 7, \sin(x)]$

Mit der Funktion ***contains*** ermittelt man das erste Auftreten eines Elements in einer Liste. Gibt ***contains*** eine null zurück, ist das gesuchte Element nicht in der Liste enthalten.

```
contains(ListeA,a)
```
$$11$$
```
contains(ListeB,b)
```
$$0$$

Ein Element wird unter Verwendung seiner Position aus der Liste mittels ***delete*** gelöscht.
Alternativ kann die Funktionsform ***_delete*** verwendet werden.

```
delete ListeA[3] :ListeA
```
$$[3, 1, 3, 8, 1, 3, 3, 4, 5, a, 7, \sin(x)]$$
```
_delete(ListeA[3]): ListeA
```
$$[3, 1, 8, 1, 3, 3, 4, 5, a, 7, \sin(x)]$$

Möchte man alle Elemente einer Liste auf eine Funktion etc. anwenden, dann nutzt man am Einfachsten den ***map***-Befehl.

```
ListeX:=[-PI,-0.4,0,2,PI,4.5,2*PI]
```
$$[-\pi, -0.4, 0, 2, \pi, 4.5, 2\cdot\pi]$$
```
map(ListeX,cos)
```
$$[-1, 0.921060994, 1, \cos(2), -1, -0.2107957994, 1]$$

Praktisch ist dies bei Berechnungen usw., wenn die Anzahl der Listenelemente unbekannt oder veränderlich ist (→ 2.10.5.2.1. Schleifen mit feststehender Anzahl der Wiederholungen).

Mit ***sort*** lassen sich Listen sortieren. Zahlen und Zeichenketten werden wie gewohnt behandelt, andere MuPAD-Objekte werden nach internen Kriterien sortiert.

```
ListeA; sort(ListeA);
```
$$[3, 1, 8, 1, 3, 3, 4, 5, a, 7, \sin(x)]$$
$$[a, \sin(x), 1, 1, 3, 3, 3, 4, 5, 7, 8]$$

Aufgaben: (zu Mengen)

1. *Erstellen Sie 3 Mengen! Die erste menge enthält alle ungeraden Zahlen bis 10, die Menge 2 die Ungeraden. In der dritten Menge sollen alle Primzahlen kleiner 10 enthalten sein. Bilden Sie die Schnitt-, die Differenz und die Vereinigungsmenge!*
2. *Prüfen Sie, ob die 3 und die 7 in allen drei Mengen enthalten ist!*

Aufgaben: (zu Listen)

1. *Erstellen Sie über den Folge-Operator je eine Liste der geraden und ungeraden Zahlen bis 25!*
2. *Vereinigen Sie die Listen von 1. zu einer neuen Liste und lassen Sie diese dann sortieren!*
3. *Prüfen Sie an welcher Stelle 13, 17 und 27 in den einzelnen Listen vorkommen!*
4. *Geben Sie das Kommando (nur ein Befehl) zum Löschen der 5 aus der Liste an!*
5. *Lassen Sie sich von der nachfolgenden Liste die Wurzelwerte (mit* map*) ausgeben!*
   Quadrate:=[25, 9, 88679889, 1, 81, 1156]

Kurzreferenz: Symbole, Bezeichner, Mengen und Listen

! In dieser Referenz werden vor allem häufig benötigte Funktionen usw. usf. sowie der häufig gebrauchte Syntax aufgeführt. Für vollständige Informationen wählen Sie bitte die Hilfe bzw. die Programm-Referenz!

_delete		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
_delete(*bezeichner*)	löscht den Wert der Variable *bezeichner* *bezeichner* ist dann wertloses Symbol)	
_delete(*listenelement*)	löscht das angegebene Listenelement *listenelement*; dabei verkürzt sich die Liste	

delete		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
delete *bezeichner*	löscht den Wert der Variable *bezeichner* *bezeichner* ist dann wertloses Symbol)	
delete *listenelement*	löscht das angegebene Listenelement *listenelement*; dabei verkürzt sich die Liste	

op		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
op(*objekt*)	liefert die Elemente des Objektes *objekt* zurück	

nops		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
nops(*objekt*)	liefert die Anzahl der Elemente des Objektes *objekt* zurück	

contains		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
contains(*objekt, element*)	prüft, ob im Objekt *objekt* das Element *element* enthalten ist	bei Mengen wird true od. false; bei Listen die erste Vorkommensposition zurückgegeben (0 bedeutet kein Vorkommen)

union		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
menge1 union *menge2*	bildet die Vereinigungsmenge der Mengen *menge1* und *menge2*	

intersect		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
menge1 intersect *menge2*	bildet die Schnittmenge der Mengen *menge1* und *menge2*	

minus		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
menge1 minus *menge2*	zieht von der Menge *menge1* die in der Menge *menge2* vorkommenden Elemente ab (-bildet die Differenzmenge)	

$		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
$	Folge-Operator	

append		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
append(*objekt1, objekt2*)	hängt das Objekt *objekt2* an das Objekt *objekt1* an	

_concat		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
_concat(*objekt1, objekt2*)	hängt das Objekt *objekt2* an das Objekt *objekt1* an	

subsop		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
subsop(*objekt, element = neuer_wert*)	definiert das Element *element* in dem Objekt *objekt* neu durch Belegung mit dem neuen Wert *neuer_wert*	

map		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
map(*liste, funktion*)	bewirkt die Anwendung der Elemente der Liste *liste* auf die angegebene Funktion *funktion*	

sort		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
sort(*liste*)	sortiert die Liste *liste* aufsteigend	Zahlen und Zeichenketten werden wie gewohnt behandelt, andere MuPAD-Objekte werden nach internen Kriterien sortiert

2.6.3. Symbole und Bezeichner für FREAKS

Beim Versuch einem vordefinierten Namen (Systemnamen, Funktionsnamen usw.) einen Wert zuzuweisen, erlebt man eine kleine Überraschung. Das System gibt eine Fehlermeldung zurück, dass der Bezeichner geschützt (protected) ist.

```
exp:=3
Error: Identifier 'exp' is protected [_assign]
```

Mittels der Funktion ***protect*** können wir unseren eigenen Bezeichnern einen solchen Schutz zukommen lassen und die letzte gültige Schutzebene (Level) anzeigen. Je nach Ebene - erhält man später beim Versuch, der Variable einen Wert zuzuweisen - Warnhinweise bzw. Fehlermeldungen.
Über ***unprotect*** lässt sich der Schutz - ! auch von Systemnamen – aufheben. Dieses Vorgehen ist aber nicht empfehlenswert, da viele Funktionen auf wieder andere zurückgreifen. Die Wirkungen von Wertzuweisungen auf Systemfunktionen sind nicht absehbar und können zum Absturz / Ende der aktuellen Sitzung führen.

```
x:=2
    2
protect(x)
    ProtectLevelNone
protect(x)
    ProtectLevelWarning
protect(x)
    ProtectLevelWarning
protect(x,ProtectLevelError)
    ProtectLevelWarning
protect(x)
    ProtectLevelError
unprotect(x)
    ProtectLevelWarning
```

Benötigt man ein unbenutztes Symbol, bei dem der Name nicht so relevant ist, dann kann man das System mit der Funktion ***genident*** (generate identifier) dazu bewegen, einen neuen, freien Bezeichner zu erstellen. Diese werden beginnend mit X – gefolgt von einer fortlaufenden Ziffer – erstellt.

```
a:=2; x:=0; X1:=3;
    2
    0
    3
genident()
    X2
```

Aufgaben:

1. Welche Schutzebenen haben die verwendeten Variablen nach den angegebenen Eingaben? Erstellen zuerst eine Tabelle mit Voraussagen für die Schutzebene und die erwartete Anzeige (auch Fehler)! Prüfen Sie mit MuPAD und vergleichen Sie Ihre Voraussagen mit den Anzeigen von MuPAD!

```
x
protect(x)
protect(x)
protect(x)
ebene:=protect(x)
protect(ebene)
x:=protect(ebene,ebene)
y:=protect(ebene,ebene)
```

2. Setzen Sie für die angegebenen Variablen die geforderten Schutzebenen!

Variable / Symbol	Ziel-Schutzebene
x	ProtectLevelNone
y	kein Schutz
ebene	ProtectLevelNone
schutz	sicherste Schutzebene
z	ProtectLevelError
y	ProtectLevelWarning

3. Geben Sie zu den Variablen auch die gültigen Kontrollabfragen der Schutzebenen an ohne dabei letztendlich die Schutzeben zu ändern!

Kurzreferenz: Symbole und Bezeichner

! In dieser Referenz werden vor allem häufig benötigte Funktionen usw. usf. sowie der häufig gebrauchte Syntax aufgeführt. Für vollständige Informationen wählen Sie bitte die Hilfe bzw. die Programm-Referenz!

protect		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
protect(*bezeichner*)	schützt die Variable *bezeichner* in der Schutzebene **ProtectLevelWarning** und gibt die letzte gültige Schutz zurück	• `protect(x)` ProtectLevelError • `protect(x)` ProtectLevelWarning
protect(*bezeichner, schutzebene*)	schützt die Variable *bezeichner* in der Schutzebene *schutzebene*	**Schutzebenen:** **ProtectLevelNone** … kein Schutz **ProtectLevelWarning** … Überschreiben löst Warnung aus (Standard) **ProtectLevelError** … absoluter Schreibschutz

unprotect		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
unprotect(*bezeichner*)	setzt die Schutzebene der Variable *bezeichner* auf **ProtectLevelNone;** nimmt also jeden Schutz zurück	

genident		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
genident()	erstellt einen neuen, freien Bezeichner in der Form X??, wobei ?? eine fortlaufende Nummer ist	
genident(*zeichenkette*)	erstellt einen neuen, freien Bezeichner in der Form *zeichenkette*??, wobei ?? eine fortlaufende Nummer ist	• `genident ("ArbVar")` ArbVar1

2.7. Lösen von Gleichungen

Gleichungen bzw. Gleichungssysteme zu lösen, ist sicher einer der wichtigsten Aufgaben der Standardmathematik. MuPAD stellt für diese Aufgabe mit **solve** (to solve: lösen) ein sehr effektives Mittel zur Verfügung. Diese MuPAD-Funktion wollen wir hier ausführlicher betrachten. Es sei noch mal darauf hingewiesen, dass MuPAD nur das Ergebnis liefert. Der Weg muß selbst nachvollzogen werden. Dies gilt besonders für Mathematik-Lernende. Bedenken Sie immer, dass man auch in einer computerlosen Situation (z.B. Prüfung) die Aufgabe lösen können muß. Der Computer (und MuPAD) sollte aber zur Kontrolle, für Routinearbeiten und als Arbeitswerkzeug das Mittel der Wahl sein.

Bevor wir mit dem Besprechen der eigentlichen **solve**-Funktion beginnen sei auf eine interessante Möglichkeit von MuPAD hingewiesen. Man kann in MuPAD eine Gleichung und sogar ganze Gleichungssysteme einer Variable (Bezeichner; → 2.6. Symbole und Bezeichner) zuweisen.

- `Gl:=0=-x^2+2*x+2`

$$0 = 2 \cdot x - x^2 + 2$$

- `Gl`

$$0 = 2 \cdot x - x^2 + 2$$

Damit spart an sich u.U. bei mehreren Rechnungen viel Schreibarbeit. Die Übersichtlichkeit und die Fehleranfälligkeit gestalten sich zumeist auch günstiger.

Mit der hinter dem Bezeichner ***Gl*** verborgenen Gleichung kann die Lösungsfunktion ***solve*** nun gefüttert werden. Die geschweifte Klammer umschließt die gesamte Lösungsmenge und die eckigen Klammern die einzelnen Lösungen (- die Elemente der Lösungsmenge).

- `solve(Gl)`

$$\{[x = \sqrt{3} + 1], [x = 1 - \sqrt{3}]\}$$

Wieder erhalten wir ein exaktes Ergebnis, welches ev. mit ***float*** in reelle Zahlen umgerechnet werden muß – z.B. für eine Darstellung in einem Diagramm.

- `float(solve(Gl))`

```
{[x=2.732050808],
[x=-0.7320508076]}
```

Mehrere Gleichungen werden als Gleichungssystem in geschweiften Klammern zusammengefasst.

- `Gln:={r=x-s*y,s=x+y}`

$$\{s = x + y,\ r = x - s \cdot y\}$$

Bei Gleichungen mit mehreren Unbekannten kann man auch angeben, für welche Variablen(-gruppen) man eine Lösung sucht. Der "normale" Aufruf von ***solve*** geht davon aus, dass man für die herausgehobenen Variablen eine Lösung sucht.

- `solve(Gln)`

$$\{[r = x - x \cdot y - y^2,\ s = x + y]\}$$

Ansonsten übergibt man die gewünschten Variablen(-gruppen) einfach als zweiten Parameter an die Funktion ***solve***. Auch hier müssen mehrere Bezeichner in geschweifte Klammern zusammengefasst werden. In unserem Beispiel ist der Aufruf von ***solve(Gln)*** identisch zu ***solve(Gln,{r,s})***.

- `solve(Gln,{r,s})`

$$\{[r = x - x \cdot y - y^2,\ s = x + y]\}$$

- `solve(Gln,{x,y})`

$$\left\{\left[x = \frac{r + s^2}{s + 1},\ y = \frac{s - r}{s + 1}\right]\right\}$$

Wie gewohnt lässt sich aber auch alles über Variablen regeln.

```
ges:={r,x}:solve(Gln,ges)
```

$$\{[r = s - y - s \cdot y,\ x = s - y]\}$$

Die Lösungen von ***solve*** lassen sich auch wieder auf einen Bezeichner zuweisen. Damit spart man sich wiederum Tipparbeit bei längeren Rechenwegen bzw. größeren Lösungsmengen.

```
erg:=solve(Gln,ges)
```

$$\{[r = s - y - s \cdot y,\ x = s - y]\}$$

```
erg
```

$$\{[r = s - y - s \cdot y,\ x = s - y]\}$$

Will man nun einen bestimmten Wert für eine Variable in die Gleichung einsetzen, dann soll dies nicht unbedingt die originale Gleichung verändern. Zum Einsetzen von Werten für bestimmte Variablen gibt es die Funktion ***subs*** (substitute: substituieren, ersetzen). Für die Dauer des Aufrufs der neuen Funktion werden die ausgewählten Bezeichner (oder Werte) mit dem neuen Inhalt belegt.

```
subs(Gln,x=0)
```

$$\{s = y,\ r = -s \cdot y\}$$

```
subs(Gln,x=1,y=2)
```

$$\{s = 3,\ r = 1 - 2 \cdot s\}$$

```
Gln
```

$$\{s = x + y,\ r = x - s \cdot y\}$$

Bei Bedarf kann man diese veränderte Gleichung etc. auch wieder einer Variablen übergeben.

Hier noch mal die wichtigsten Funktionen in einem einfachen Beispiel zusammengefasst.

Die Gleichung **y = 3 x - 4** wird auf den Bezeichner **Gl** zugewiesen. In der zweiten Zeile wird durch Einsetzen von **0** für **y** die Nullstellengleichung gebildet und diese der Variable **NStGl** übergeben. Der zweite Befehl fordert die Lösung dieser Nullstellengleichung an.

```
Gl:=y=3*x-4
```

$$y = 3 \cdot x - 4$$

```
NStGl:=subs(Gl,y=0);solve(NStGl)
```

$$0 = 3 \cdot x - 4$$

$$\left\{\left[x = \frac{4}{3}\right]\right\}$$

2.7.1. Lösen von Gleichungen für FREAKS

Die Funktion ***subs*** führt vorrangig die gewünschte Ersetzung durch. Resultierende Gleichung usw. werden normalerweise vereinfacht. Manchmal muß man, um eine / die endgültige Form oder Lösung zu erhalten, auf eventuell benutzte (Zwischen-)Ergebnis-Bezeichner zurückgreifen oder die Auswertung mit ***eval*** (evaluate: evaluieren, bewerten, beurteilen) vervollständigen.

Im nebenstehenden Beispiel ersetzt die Funktion ***subs*** in ***ln(a)*** die Variable ***a*** durch ***1***. Das Ergebnis befriedigt aber nicht wirklich. Eine Prüfung mit ***eval*** bestätigt, dass der Ausdruck noch zu vereinfachen geht. Zur Erinnerung, das Prozentzeichen **%** steht als Variable für das Ergebnis des vorhergehenden MuPAD-Kommandos.

```
subs(ln(a),a=1)
```

$$\ln(1)$$

```
eval(%)
```

$$0$$

$$0$$

Mit einem vollständigen Aufruf von ***eval*** kann eine Auswertung jederzeit erfolgen.

```
eval(subs(ln(a),a=1))
```

$$0$$

Für viele Funktionen sind keine Lösungen im Bereich der rationalen Zahlen vorhanden. Es existieren dann aber vielleicht Lösungen bei den komplexen Zahlen.

```
solve(x^2+3,x)
```

$$\{(-i)\cdot\sqrt{3}, i\cdot\sqrt{3}\}$$

Wenn diese z.B. von der Lösung (im normalen Schulmathematik-Bereich) ausgeschlossen werden sollen, dann muß man beim Funktionsaufruf den Lösungsraum als ***Domain*** (domain: Bereich, Gebiet, Raum) mit angegeben.

Weitere Möglichkeiten zum Einschränken des Zahlenraum sind ***Dom::Rational*** und ***Dom::Integer*** (integer: ganze Zahlen).

```
solve(x^2+3,x,Domain=Dom::Real)
```

$$\varnothing$$

Ist eine Gleichung für einen ganzen Zahlenraum erfüllt, dann werden auch diese Mengen als Lösung angeboten. Die ganzen Zahlen besitzen das Zeichen $\mathbb{Z}$ (bzw. Z_), die rationalen Zahlen $\mathbb{Q}$ (Q_) und für die reellen Zahlen wird $\mathbb{R}$ (R_) verwendet.

```
solve(cos(x)=sin(x+PI/2),x)
```

$$\mathbb{C}$$

Letztendlich werden die mit komplexen Zahlen mit $\mathbb{C}$ (C_) abgekürzt.

Eine numerische Lösung einer Gleichung kann auf zwei verschiedenen – aber völlig äquivalenten – Aufrufen erzielt werden. Einmal existiert eine spezielle Form der Lösungsfunktion **numeric::solve**. Zum anderen kann man mit hold die symbolische Verarbeitung einschränken. Der vollständige Aufruf lautet **float(hold(solve)(...))**.

```
numeric::solve(exp(x)=sin(x),
x=-10..10)
```

$$\{-9.424858654\}$$

```
float(hold(solve)(exp(x)=sin(x),
x=-10..10))
```

$$\{-9.424858654\}$$

Bei numerischen Lösungen sollte man die Angabe eines Wertebereiches in Betracht ziehen, da immer nur eine (die letzte) Lösung gesucht / berechnet / zurückgegeben wird. Im Beispiel ist dies durch ***x=-10..10*** gemacht worden.

Die universale Lösungsfunktion ***solve*** ist mehr ein Verteiler in spezielle Gleichungslösungs-Funktionen. **solve** ermittelt die Art der zu lösenden Gleichung und gibt dann die Aufgabe an die spezielle Lösungsfunktion weiter. Mittels des Aufrufs ***?solvers*** erhält man eine Hilfeseite mit Informationen zu den speziellen integrierten Lösungsfunktionen.

Kurzreferenz: Lösen von Gleichungen etc.

! In dieser Referenz werden vor allem häufig benötigte Funktionen usw. usf. sowie der häufig gebrauchte Syntax aufgeführt. Für vollständige Informationen wählen Sie bitte die Hilfe bzw. die Programm-Referenz!

solve		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
solve(*gleichungen*)	löst die Gleichung / das Gleichungssystem (/ Gleichungsmenge) *gleichungen*	• `solve(x^2+1/2*x=0)` $\left\{[x=0],\ \left[x=-\frac{1}{2}\right]\right\}$
solve(*gleichungen, unbekannte*)	löst die Gleichung / das Gleichungssystem (/ Gleichungsmenge) *gleichungen* hinsichtlich der bei *unbekannte* angegebenen Menge von Variablen	• `solve(x^2+1/2*x=0,x)` $\left\{-\frac{1}{2},\ 0\right\}$
solve(*gleichungen, unbekannte, zahlenbereich*)	löst die Gleichung / das Gleichungssystem (/ Gleichungsmenge) *gleichungen* hinsichtlich der bei *unbekannte* angegebenen Menge von Variablen, zulässige Lösungen stammen aus dem definierten *zahlenbereich*	• `solve(x^2+1/2*x=0,x, Domain=Dom::Integer)` $\{0\}$

numeric::solve		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
numeric::solve(*gleichungen, [unbekannte], [zahlenbereich]*)	löst die Gleichung (→ solve) auf numerischen Weg, zulässige Lösungen stammen aus dem definierten *zahlenbereich*	völlig äquivalent ist: float(hold(solve)(*gleichungen, [unbekante], [zahlenbereich]*))

subs		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
subs(*objekt, bezeichner = neuer_wert*)	ersetzt im MuPAD-Objekt *objekt* (z.B. eine Gleichung) die Variable *bezeichner* durch den Wert von *neuer_wert*	

eval		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
eval(*objekt*)	beurteilt das Objekt objekt rekursiv - durch Einsetzen von Variablen und dem nachfolgen wiederholten evaluieren	

Domain		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
Domain = Dom::*zahlentyp*	begrenzt die Bearbeitung auf den Zahlenraum *zahlentyp*	*zahlentyp*: Integer $\mathbb{N}$ Real $\mathbb{R}$ Rational $\mathbb{Q}$ $\mathbb{C}$ *(Standard)*

Kommando-Folge für ein unkompliziertes Lösen von Gleichungen / Gleichungssystemen:

für eine Gleichung mit einer Unbestimmten:

- `Gleichung:={`***Gleichung1***`}`
- `Variable:={`***Variable1***`}`
- `Loesungen:=solve(Gleichung,Variable)`

für beliebig viele Gleichungen mit entsprechend vielen Unbestimmten:

- `Gleichungen:={`***Gleichung1,Gleichung2,...***`}`
- `Variablen:={`***Variable1,Variable2,...***`}`
- `Loesungen:=solve(Gleichungen,Variablen)`

Aufgaben:

1. Lösen Sie die folgenden linearen Gleichungen! (Prüfen Sie durch schriftliches Rechnen!)

a) $0 = 4(x-2)+6$ b) $0 = 2(-x+3)-(-x-1)$

c) $4 = 3-\{-2a+[a-(a-3)\cdot 2]-4a\}+1$

2. Lösen Sie die folgenden Gleichungen!

a) $0 = x^2-4$ b) $0 = (x-3)^2\cdot(x^2-4x+2)$

für FREAKS: c) $0 = 3x^3-2x^2+4x-1$

3. Lösen Sie die folgenden Gleichungssysteme!

a) $2x-3y+2=0$
$4x+y-4=0$

b) $a-3b+2c=4$
$2a-2b+c=-1$
$1\frac{1}{2}a+3b-2c=4$

c) $1\frac{1}{4}a-\frac{2}{3}b+\frac{1}{7}c=-\frac{1}{3}$
$4a-b+3c=-2$
$0{,}5a+0{,}2b+0{,}3c=0{,}7$

2.8. Funktionen

Beginnen wir wieder mit einer kurzen Einführung / Wiederholung zum Thema. Damit soll dann auch das verwendete Begriffssystem beschrieben werden. Wir wollen hier den Begriff der Funktion auf zwei verschiedene Weisen einführen. Dadurch soll deutlich werden, dass ein Fachgebiete eigentlich immer auf mindestens zwei verschiedenen Wegen zugänglich ist. (Dies schon deshalb notwendig, da man den einen Weg zum Herleiten und den zweiten zum Beweisen benötigt.)
Damit bietet sich auch für Nichtfanatiker (der Mathematik) ein nachvollziehbarer Zugang zum Verständnis der Funktionen an.
Unser erster Zugang soll den schulmathematischen Weg beschreiten. Diesen Weg werden Sie jederzeit mit den Unterlagen aus dem Mathematik-Unterricht oder passenden Büchern abgleichen können. Im zweiten Herleitungsansatz wollen wir von systemtheoretischen Sachverhalten ausgehen, die mehr dem Geist diese Skripts / Skript-System ("Mathematik mit dem Computer"; → "Spiele, Modellierung und Simulation") und meinem Verständnis der Mathematik als Hilfsmittel für andere Wissenschaften entsprechen.

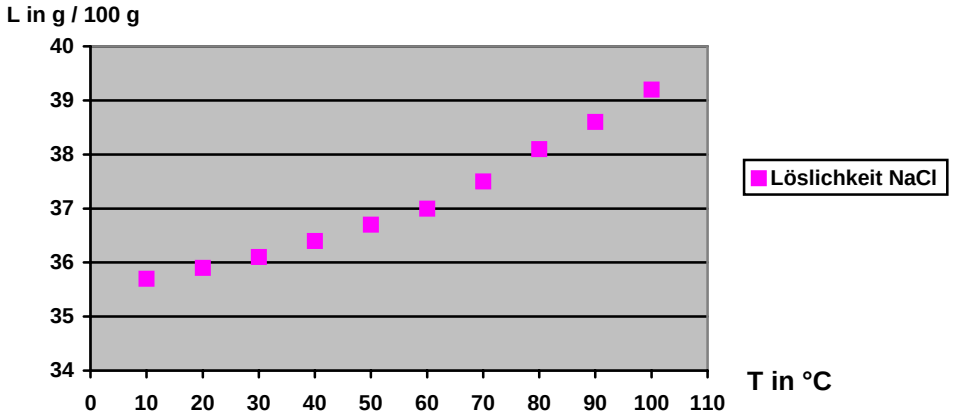
2.8.1. Schulmathemathische Herleitung des Funktions-Begriffs

Gehen wir von einem praktischen Beispiel aus. Bei einem Versuch die Löslichkeit von Kochsalz in Wasser in Abhängigkeit von der Temperatur **T** (des Lösungsmittels (wird im Thermostaten konstant gehalten)) zu ermitteln, wurden die nachfolgenden Messwerte erzielt.

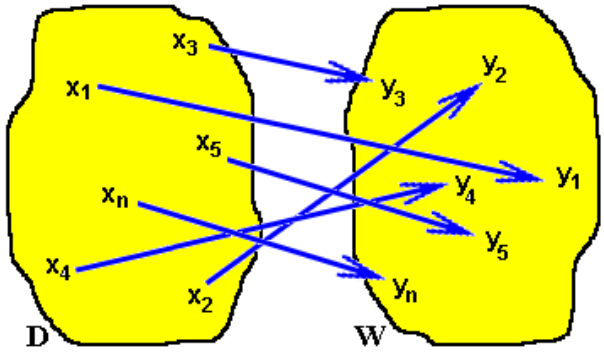
Temperatur in °C	**gelöste Masse in g / 100 g [H_2O]**
10	35,7
20	35,9
30	36,1
40	36,4
50	36,7
60	37,0
70	37,5
80	38,1
90	38,6
100	39,2

Jeder bestimmten Temperatur **T** ist also eine bestimmte Menge / Masse gelöstes Salz $\mathbf{m_{NaCL}}$ zugeordnet. d.h. die Zuordnung (Relation) $\mathbf{T} \mapsto \mathbf{m_{NaCl}}$ ist eindeutig (in eine Richtung gültig). Ein Gegenversuch bestätigt auch schnell, dass die umgedrehte Zuordnung nicht so existiert $\mathbf{m_{NaCl}} \not\mapsto \mathbf{T}$. Vielleicht ergeben sich auch unterschiedliche Temperaturen, wenn verschiedene Mengen Salz gelöst werden, aber niemals die in unsere gegebenen Messreihe aufgetretenen Werte.

Die tabellarischen Werte können natürlich auch graphisch dargestellt werden. Da nur Einzelwerte vorliegen, lässt der Gesamtzusammenhang zwischen beiden Größen nur erahnen und eventuell als "vermutlich" dazuskizzieren. (Der exakte Zusammenhang lässt sich u.U. mit Hilfe der Regression bestimmen → 2.7.5.)

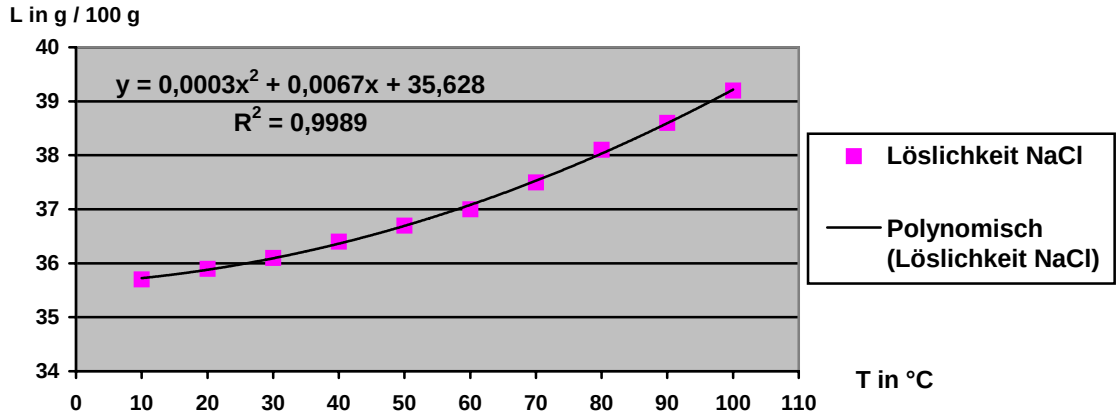


Eine eindeutige Relation (Zuordnung) zwischen mindestens zwei Größen (z.B.: **x** und **y**) wird in der Mathematik als Funktion bezeichnet (empfohlene Zeichen: Kleinbuchstaben, z.B.: **f**). Für unser Beispiel anders formuliert, ist die gelöste Menge Kochsalz eine Funktion der Temperatur $\mathbf{f_L: T \mapsto m_{NaCl}}$ (sprich: aus der Temperatur **folgt** die gelöste Masse Salz). Umgangssprachlich würden wir sagen: Die Löslichkeit / lösbare Menge des Salzes ist von der Temperatur des Lösungsmittels abhängig.
Die Temperatur ist in unserem Beispiel die vorgegebene Größe, die lösbare Menge Salz die resultierende Größe. Die vorgegebene Größe wird in der Mathematik Argument der Funktion genannt. Die resultierende Größe nennt man den Funktionswert.
Den Argumenten aus dem Definitionsbereich **D** sind bestimmte Funktionswerte aus dem Wertebereich **W** zugeordnet. Definitionsbereich und Wertebereich sind also Mengen.
Für unser Beispiel nutzten wir nur einige ausgewählte Elemente aus beiden Bereiche. Andere Temperaturen (beliebige Zwischenwerte) bringen auch andere Löslichkeiten.
Die Beziehung zwischen den beiden Mengen lässt sich in Form eines Pfeildiagramms / Mengenbeziehungs-Diagramm darstellen. Zur Veranschaulichung wählen wir hier nur eine verallgemeinerte Form.
Die Pfeile kennzeichnen die Beziehungen der Werte des betrachteten Zusammenhangs zueinander.



Der Definitionsbereich ist oft begrenzt – so auch in unserem Beispiel. Die Löslichkeit des Kochsalzes unter 0°C bzw. über 100°C zu bestimmen, stößt bei Wasser auf natürliche Grenzen. Der Definitionsbereich unseres Beispiel lässt sich also so umschreiben: $\mathbf{0 \le D \le 100}$; $\mathbf{D \in \mathbb{R}}$.
Auch der Wertebereich ist zu mindestens nach unten hin objektiv begrenzt, da es negative Mengen Salz nicht gibt: $\mathbf{0 \le W \le \infty}$; $\mathbf{W \in \mathbb{R}}$.
Unsere verallgemeinerte Funktion **f** sieht im Augenblick so aus:
$\mathbf{f: x \mapsto y}$ (sprich: aus x **folgt** y).
Nun ist es uns praktisch aber nicht möglich alle Wertepaare (x,y) aus der Tabelle zu ermitteln. Wir brauchen eine allgemeine Bildungsvorschrift mittels der wir aus jedem beliebigen Wert **x** des Definitionsbereichs **D** einen zugeordneten Wert **y** des Wertebereichs **W** ableiten / berechnen können. Da **y** als resultierende Größe genau nach dieser Vorschrift gebildet werden soll und von den Werten von **x** abhängt, schreiben

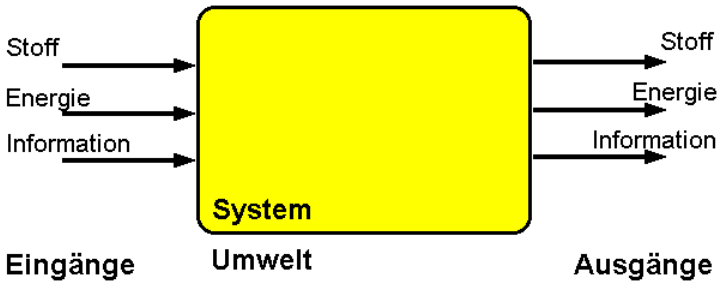
wir: **f: x ⟼ f(x)** und damit auch (wegen der Gleichsetzung) **y = f(x)** . Die Funktion **f** ist genau die gesuchte Bildungsvorschrift zur Berechnung neuer Funktions-Werte.
Eine Funktion können wir also auch als gezielte / gerichtete Abbildung (Umwandlung) von Werten aus dem Definitionsbereich (Argumente) in Werte des Wertebereichs (Funktionswerte) verstehen.
Wie schon die Werte aus der Tabelle, können wir auch die Funktion graphisch darstellen. Da wir jetzt alle Zwischenwerte berechnen können, erhalten wir einen vollständigen Graph der Funktion für den gewählten Ausschnitt (x-Achse) aus dem Definitionsbereich.



Analysis ist der Bereich der Mathematik, der sich mit solchen Funktionen beschäftigt, bei denen sowohl Definitionsbereich als auch Wertebereich im Bereich / Zahlenraum der reellen Zahlen liegen.

2.8.2. Herleitung des Funktions-Begriffs aus systemtheoretischen Ansätzen

Mehr oder weniger komplexe (komplizierte) Sachverhalte bezeichnet man allgemein als System. Systeme können z.B. ein Staat sein, aber auch eine Zelle, ein Atom oder eine chemische Reaktion. Systeme sind von der Umwelt umgeben, mit der sie über Eingänge (Inputs, Zugänge) und Ausgänge (Outputs, Abgänge) in Verbindung treten.



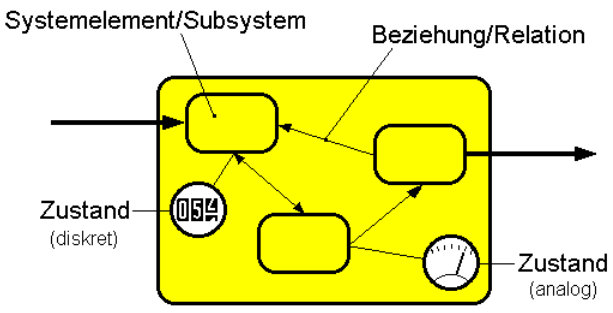
Die Ausgänge werden neben den Eingängen auch von den inneren Situationen eines Systems bestimmt. Man nennt sie allgemein Zustände. Praktisch entsprechen die Ausgänge den Zuständen, denn sie stellen die erfassbaren Größen des Systems dar. Gleichzeitig wirken viele Zustände aber eventuell auch wieder auf das System zurück – sie stellen möglicherweise also auch innere Eingangsgrößen dar. Zwischen Umwelt und System können Materie, Energie und Information ausgetauscht werden.

(Moderne Forscher (MATURANA, VARELA) gehen davon aus, dass eigentlich nur Materie und Energie ausgetauscht werden. Diese können dann im System selbst oder in einem anderen System etwas bedeuten - sie stellen dann erst Informationen dar.)

Im Inneren kann ein System wiederum aus untergeordneten Systemen (Subsysteme, Untersysteme) bestehen. Die innere Strukturierung kann u.U. beliebig fortgesetzt werden.

System sind durch messbare Zustände charakterisiert. Die Zustände machen Aussagen über das System selbst und über die Situation des Systems.

In der Praxis unterscheiden wir diskrete und analoge Zustände.



Die Eingänge bestimmen entscheidend die Reaktion (Zustände und Ausgänge) des Systems. Die Eingangsgrößen sind vom System unabhängig und vorgegeben. Deshalb werden die Eingangsgrößen (Eingangsvariablen) als unabhängige (vorgegebene) Größen (Variablen) bezeichnet. Die Zustände und Ausgangsgrößen hängen vom System selbst ab. Wir bezeichnen sie deshalb als abhängige Größen (Variablen).

Die Eingangsgrößen $\mathbf{x_E}$ bewirken ev. Veränderungen der Systemzustände und der Ausgänge $\mathbf{y_A}$. In deterministischen (logischen, zusammenhängenden, nicht chaotischen) Systemen – (und nur diese betrachten wir hier) - sind bestimmten Eingangsgrößen $\mathbf{x_E}$ immer bestimmte Ausgangsgrößen $\mathbf{y_A}$ zugeordnet. Der Übergang von einem bestimmten Eingang zu einem bestimmten Ausgang $\mathbf{x_E} \rightarrow \mathbf{y_A}$ wird von der Funktion **f** (Aufgabe, Zweck) des Systems bestimmt: $x_E \xrightarrow{f} y_A$

Betrachten wir ein System mit einer Ausgangsgröße, welches nur von einer (inneren oder äußeren) Eingangsgröße abhängig sein soll. Jeder Wert dieser Eingangsgröße erzeugt immer den gleichen Wert der Ausgangsgröße. Die Menge aller Werte der Eingangsgröße bestimmen den Bereich für den unser System definiert / bestimmt ist – entspricht also dem Definitionsbereich. Die Werte der Ausgangsgröße bilden die Menge des Wertebereichs – also den Wertebereich.

Im Zuge der Betrachtung von untergeordneten Systemen kann man die Funktion **f** auch als Verknüpfung der Unterfunktionen $\mathbf{f_x}$ betrachten.

$f = f_1 \circ f_2 \circ f_3 \circ \ldots \circ f_n$ $\quad\quad$ $\circ$ .. Zeichen für Verknüpfung / Komposition

2.8.3. Definition und Verwendung von Funktionen

Funktionen können in MuPAD z.B. durch Zuweisungen definiert werden. Dies sichert vor allem die Verwendbarkeit im gesamten Notebook. Dadurch spart man viel Tipparbeit.
Auf die Funktion kann man über den Bezeichner zugreifen. Statt dem Bezeichner verarbeitet MUPAD den Wert des Bezeichners – also die angegebene Funktion.
Über subs lassen sich Variablen durch Werte austauschen und so die Funktionswerte bestimmen.

- `f:=2*x+5`

$$2 \cdot x + 5$$

- `subs(f,x=21)`

$$47$$

- `g:=y=2*x+5`

$$y = 2 \cdot x + 5$$

- `subs(g,x=21)`

$$y = 47$$

Ein anderer Weg ist die Verwendung des Pfeiloperators. Dieser dient zur Definition von MuPAD-Prozeduren. Solche Prozeduren können weit komplizierter sein, als Gleichungen usw. Sie stellen aber immer eine mathematische Funktion im Sinne der gerichteten Umwandlung von Werten des Definitionsbereichs in Werte des Wertebereichs (→ 2.8.1. Schulmathemathische Herleitung des Funktions-Begriffs). Man nennt dies auch eine mathematische Abbildung.

- `h:=x->2*x+5`

$$x \to 2 \cdot x + 5$$

- `h(21)`

$$47$$

Die definierten Funktion f, g und h sind mathematisch gleichbedeutend, nur die Verwendung in MuPAD unterscheidet sich.
Die Verwendung des Pfeiloperators hat den Vorteil, dass die Funktionen, wie üblich dargestellt und verwendet werden können.

Auch Funktionen mit mehreren Abhängigen lassen sich unkompliziert definieren und verwenden.
Die Abhängigen müssen in der Pfeil-Darstellung in Klammern notiert werden.
Auch im nebenstehenden Beispiel sind r, s und t mathematisch gleichbedeutende Funktionen in verschiedenen Schreibweisen und Verwendungen.

- `r:=x^2+3-y^2/5`

$$x^2 - \frac{y^2}{5} + 3$$

- `subs(r,x=3,y=4)`

$$\frac{44}{5}$$

- `s:=z=x^2+3-y^2/5`

$$z = x^2 - \frac{y^2}{5} + 3$$

- `subs(s,x=3,y=4)`

$$z = \frac{44}{5}$$

- `t:=(x,y)->x^2+3-y^2/5`

$$(x,\ y) \to x^2 + 3 - \frac{y^2}{5}$$

- `t(3,4)`

$$\frac{44}{5}$$

1. Definieren Sie sich die folgenden Funktionen mit dem Pfeil-Operator und bestimmen Sie die Funktionswerte!

a) $x^3 - x^2$ b) $2x^2 + 7x - 14$ c) $(x-3)^2 \cdot (x^2 - 4x + 2)$

x	**-2,5**	**0**	**0,000336**	**1**	$\sqrt{3}$
$f_a(x)$					
$f_b(x)$					
$f_c(x)$					

2. Definieren Sie sich die folgenden Funktionen über die Bezeichner GlA, GlB und GlC und bestimmen Sie jeweils alle Funktionswerte für die drei Gleichungen

a) $x^2 - 6$ b) $x^3 + x^2$ c) $(x-3)^2 + (x^2 - 4x + 2)$

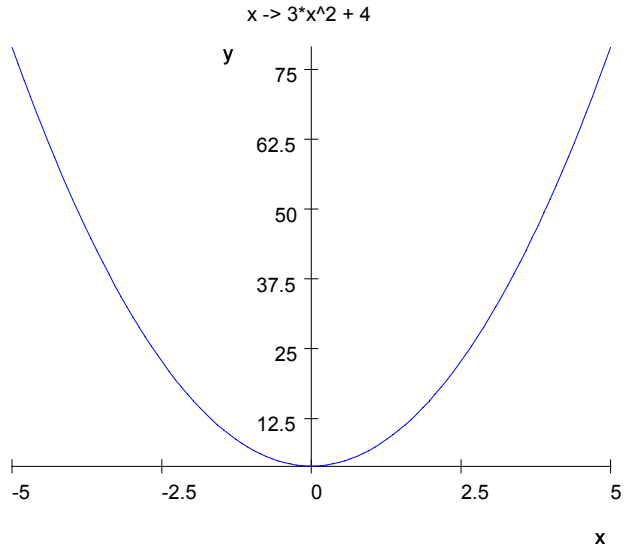
x	**-2,5**	**0**	**0,000336**	**1**	$\sqrt{3}$
$f_a(x)$					
$f_b(x)$					
$f_c(x)$					

2.8.3. Zeichnen von Funktionen

Die graphische Darstellung einer Funktion sagt uns gewöhnlich viel mehr als tausend Zahlen und Werte.
In MuPAD sind sehr leistungsfähige Funktionsplotter (Funktionszeichner) integriert.

Mit der Funktion **plotfunc2D** lassen sich in MuPAD solche Funktionen darstellen, die über ein Argument verfügen.
Der einfachste Aufruf von ***plotfunc2d*** erfolgt nur mit Angabe der gewünschten Funktion als Parameter. Die Auswahl der Grenzen für das Koordinatensystem übernimmt MuPAD dann eigenständig.

- `plotfunc2d(3*x^2+4)`

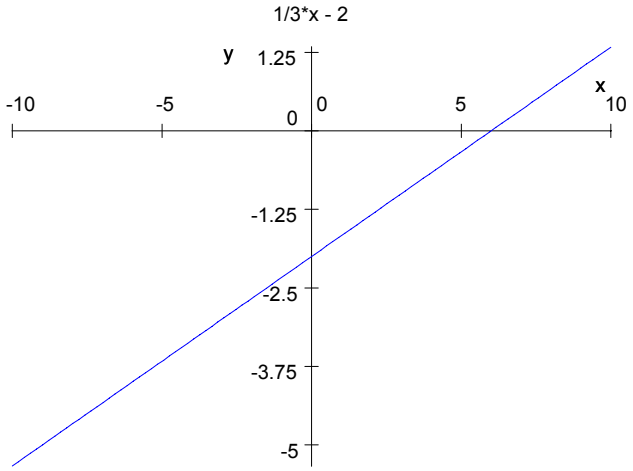


Durch Angabe eines Bereiches für die Argumente bzw. die Funktionswerte lässt sich ein beliebiger Ausschnitt aus der Funktion darstellen. Ber Bezeichner muß natürlich mit der Variable der Originalfunktion übereinstimmen.
Als Argument kann ***plotfunc2d*** die Funktion selbst oder ein Bezeichner – hinter dem sich eine Funktion verbirgt – übergegeben werden.

- `g:=x/3-2`

$$\frac{x}{3} - 2$$

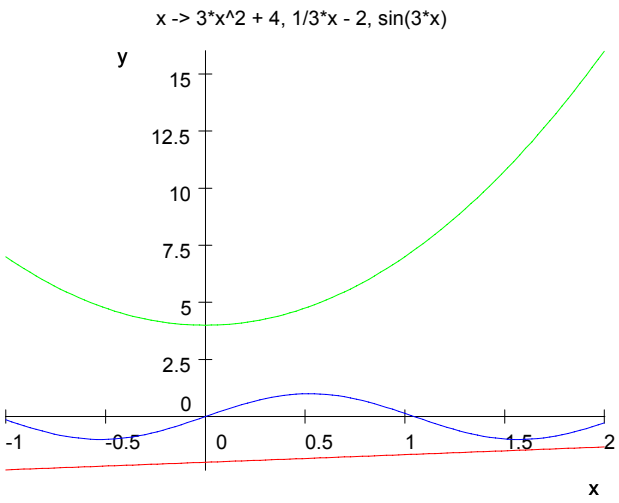
- `plotfunc2d(g,x=-10..10)`



Auch mehrer Funktionen lassen sich in einem Diagramm zusammen darstellen. Die Einschränkungen für Definitions- und Wertebereich gelten für die gesamte Darstellung und muß eindeutig sein.

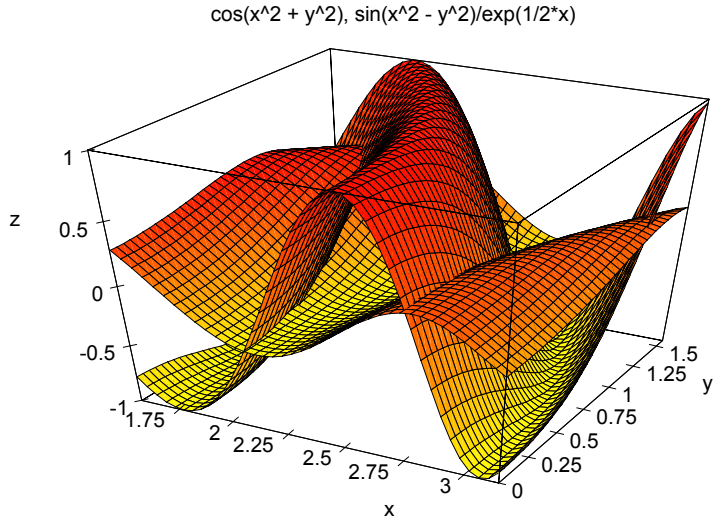
Alle Funktionen müssen weiterhin alle die gleiche Variable als Argument haben.
Für professionellere Graphiken mit Titel, Hintergrund, Gitterlinien, Achsenskalierung usw. gibt es Szeneoptionen (→ Hilfe).

- `plotfunc2d(f,x=-1..2,g,sin(3*x))`



Funktionen, die zwei Argumente (Veränderliche) beinhalten müssen mit ***plotfunc3d*** gezeichnet werden. Wieder lassen sich als Argumente außer der / den Funktion(en) auch die Darstellungsbereiche angeben. Die Verwendung von Bezeichnern innerhalb des Funktionsaufrufs erhöht u.U. die Übersichtlichkeit.
Mit ***Grid=[40,40]*** wird die Anzahl der Stützpunkte festgelegt.

- `f1:=cos(x^2+y^2): f2:=sin(x^2-y^2)/exp(x/2):`
- `DefB:=x=PI/2..PI: WrtB:=y=0..PI/2:`
- `Aufl:=Grid=[40,40]:`
- `plotfunc3d(f1,f2,DefB,WrtB,Aufl)`



Standard ist 20 in jede Argument-Dimension. Das Ergebnis kann dann – je nach Gestaltung der Funktion - recht eckig aussehen.

Aufgaben

1. Erstellen Sie von den nachfolgenden Funktionen jeweils eine graphische Darstellung!

a) $y = 4(x-2)+6$ b) $y = 2(-x+3)-(-x-1)$

c) $y = 3-4-\{-2a+[a-(a-3)\cdot 2]-4a\}+1$

2. Erstellen Sie von den nachfolgenden Funktionen jeweils eine graphische Darstellung! Untersuchen Sie auffällige Regionen durch Eingrenzen der Darstellungsbereiche! Achten Sie auf Unregelmäßigkeiten und Fehler in den Graphen!

a) $f(x) = \dfrac{x+58x^2-3}{15x-\dfrac{1}{2}}$ b) $f(x) = \dfrac{0{,}3x^5-2x^2+7x}{x^4-3x^3}$

2.8.3.1. Zeichnen von Funktionen über graphische Objekte

Komplizierter wird die Sache, wenn man z.B. verschieden dimensionale Funktionen zeichnen, weitere graphische Objekte zum Funktionsdiagramm hinzugeben oder einfach nur die Farben verändern möchte. In so einem Fall verwendet man statt ***plotfunc2d*** bzw. ***plotfunc3d*** die elementareren ***plot::Function2d*** bzw. ***plot::Function3d***. Diese Funktionen geben die Graphen nicht direkt aus, sondern erzeugen "nur" Zeichnungs-Objekte. Sie werden dann später mit der ***plot***-Funktion einzeln oder zusammen dargestellt.

Häufig wünscht man die Darstellungen von Punkten und Punktlisten (Messwerten) . Später soll dann noch die resultierende Funktion mit eingezeichnete werden.
Hierzu gibt es die Möglichkeit, die einzelnen Plots über ***plot::Pointlist*** und ***plot::Function2d*** zu erstellen und dann gemeinsam in einem Plot auszugeben.

- `punkte:=[2,4],[3,9],[4,16],[5,25],[6,36]`

  [2, 4], [3, 9], [4, 16], [5, 25], [6, 36]
- `funktion:=x^2`

  x^2
- `PktPlot:=plot::Pointlist(punkte)`

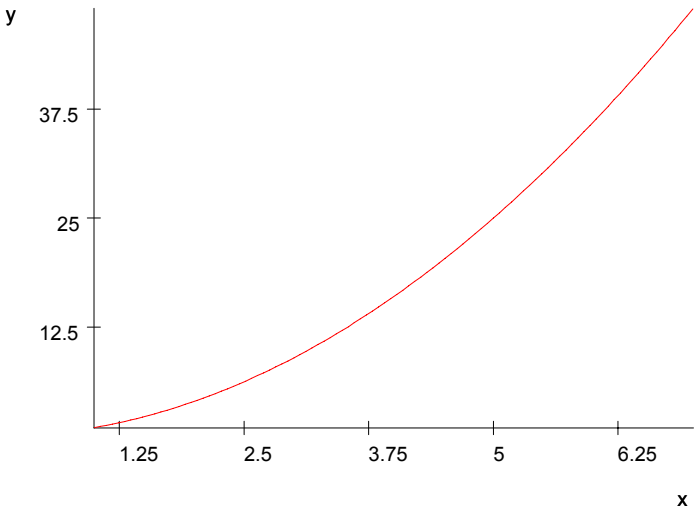
  plot::Pointlist()
- `FktPlot:=plot::Function2d(funktion,x=1..7)`

  plot::Function2d(x^2, $x = 1..7$)
- `plot(FktPlot,PktPlot)`

plot::Pointlist erwartet als Argument mindestens eine Punktliste. Bei ***plot::Function2d*** gibt man die Funktion und den gewünschten Wertebereich an.

Die eigentliche Darstellung erstellt dann die Funktion plot. Ihr werden alle vorher erstellten Einzel-Plots übergeben.
Durch Manipulation der Einzel-Plots lässt sich später z.B. eine bestimmte Farbe einstellen und auch mit der gemeinsamen Darstellung und verschiedenen Optionen für die Einzel-Plots experimentieren.

- `plot(FktPlot,PktPlot)`



In einem weiteren Beispiel sollen etwas kompliziertere Funktionen einzeln und zusammen in einem Graph dargestellt werden.
Nach der Definition von gemeinsamen Werte- und Definitionsbereichen (hier: DefB und WerteB) werden die einzelnen Graphik-Objekte (Plots) als Graph1 und Graph2 erzeugt.

- `DefB:=0..PI: WerteB:=0..PI:`
- `Graph1:=plot::Function3d(sin(x^2+y^2), x=DefB,y=WerteB)`

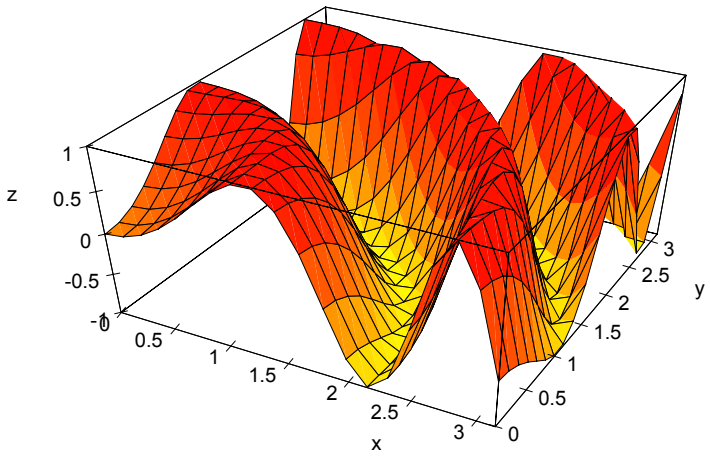
$$\text{plot::Function3d}(\sin(x^2+y^2),\ x=0..\pi,\ y=0..\pi)$$

- `Graph2:=plot::Function3d(cos(x^2+y^2), x=DefB,y=WerteB)`

$$\text{plot::Function3d}(\cos(x^2+y^2),\ x=0..\pi,\ y=0..\pi)$$

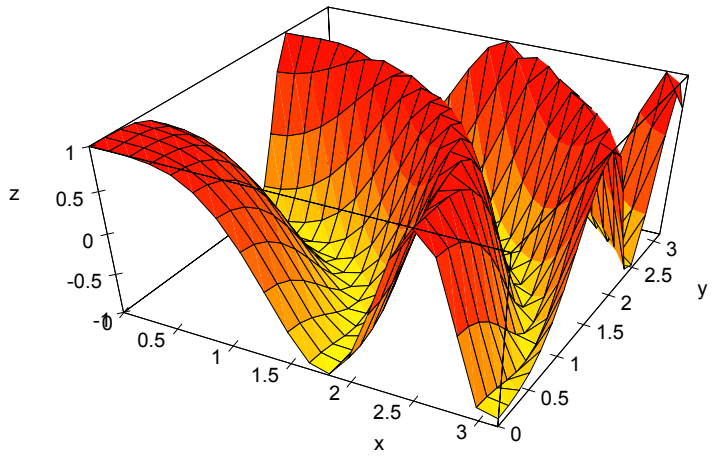
Zum Schluß realisieren wir die Ausgabe der Einzel-Plots über die Darstellungsfunktion ***plot***.
Zuerst nur die erste Funktion (Graph1).

- `plot(Graph1)`



Jetzt als einzelne Funktionsdarstellung das zweite Graphik-Objekt (Graph2).

- `plot(Graph2)`

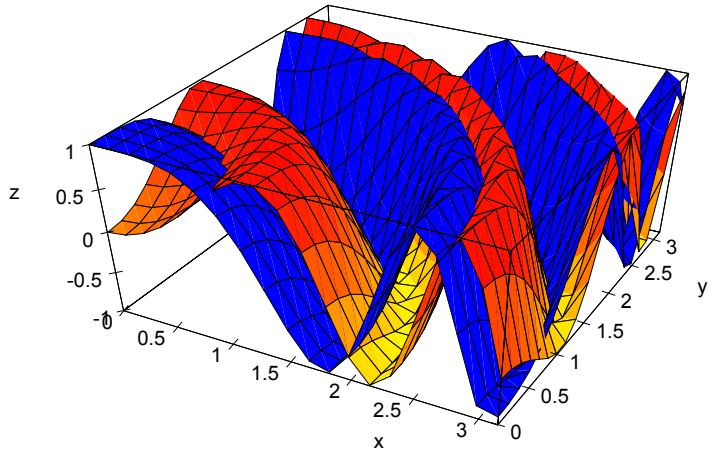


Zur besseren Unterscheidung der Graphen soll der Graph2 blau eingefärbt und dann beide Graphen gemeinsam dargestellt werden.

- `Graph2::Color:=RGB::Blue`

  [0.0, 0.0, 1.0]

- `plot(Graph1,Graph2)`



Nicht auszudenken, wie viel Tipparbeit die direkten Aufrufe von ***plotfunc3d*** (ohne Variablen usw.) bedeutet hätten. Der resultierende Mehrzeiler für die gemeinsame Graphik wäre wohl auch kaum noch übersichtlich bzw. gar nicht machbar.
Fehler lassen sich in indirekten Plots und den damit zusammenhängenden Funktionen wesentlich besser finden und korrigieren.
Durch einen Doppelklick auf die Graphik-Fläche wechselt man in den interaktiven Modus zur Bearbeitung einer Graphik. Nun lassen sich Perspektive etc. zur Echtzeit ausprobieren.

Aufgaben:

1. *Erstellen Sie eine Ansicht der folgenden Funktion! Beschreiben Sie das entstandene Objekt!*

$$f(x,y) = \sqrt{16 - x^2 - y^2}$$

2. *Lassen Sie sich den Graphen der folgenden Funktion zeichnen! Untersuchen Sie auffällige Regionen durch Eingrenzen der Darstellungsbereiche!*

$$f(x,y) = \frac{7x - 3y + 3}{3x^2 + y - 2}$$

3. *Erstellen Sie von der Funktion* $y = 3x^2 - 2x$ *eine graphische Darstellung über die Plot-Funktion für den Definitionsbereich* $-3 \le x \le 4$ *und den Wertebereich* $-2 \le y \le 5$*!*

4. *Färben Sie die Funktion grün und den Hintergrund gelb ein!*

5. *Kombinieren Sie die erste Darstellung (von Aufg. 3) mit der Funktion* $y = 3x + 4$ *und den zusätzlichen Punkten* [1,1]; [2,2]; [5,5]; [-5,5] *!*

6 *Probieren Sie die folgenden Befehlszeilen aus! Was wird dargestellt?*

```
orbit:=plot::HOrbital(3,2,0,Grid=[30,30],Title="H-Orbital n=3  l=2  n=1")
plot(orbit,Axes=None)
```

Verändern Sie die Parameter für n, l und n! Lassen Sic sich das Objekt dann jeweils neu anzeigen!

Kurzreferenz: Zeichnen von Funktionen

! In dieser Referenz werden vor allem häufig benötigte Funktionen usw. usf. sowie der häufig gebrauchte Syntax aufgeführt. Für vollständige Informationen wählen Sie bitte die Hilfe bzw. die Programm-Referenz!

plotfunc2d		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
plotfunc2d(*funktionen*)	zeichnet die Funktion(en) *funktionen* in ein geeignetes, zweidimensionales Koordinatensystem	
plotfunc2d(*funktionen, bereiche*)	zeichnet die Funktion(en) *funktionen* in ein durch die Achsenbereiche *bereiche* eingeschränktes, zweidimensionales Koordinatensystem	
plotfunc2d(*optionen, funktionen, bereiche*)	zeichnet die Funktion(en) *funktionen* in ein durch die Achsenbereiche *bereiche* eingeschränktes, zweidimensionales Koordinatensystem; zusätzlich werden die angegebenen Szeneoptionen *optionen* ausgewertet	

plotfunc3d		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
plotfunc3d(*funktionen*)	zeichnet die Funktion(en) *funktionen* in ein geeignetes, dreidimensionales Koordinatensystem	
plotfunc3d(*funktionen, bereiche*)	zeichnet die Funktion(en) *funktionen* in ein durch die Achsenbereiche *bereiche* eingeschränktes, dreidimensionales Koordinatensystem	
plotfunc3d(*optionen, funktionen, bereiche*)	zeichnet die Funktion(en) *funktionen* in ein durch die Achsenbereiche *bereiche* eingeschränktes, dreidimensionales Koordinatensystem; zusätzlich werden die angegebenen Szeneoptionen *optionen* ausgewertet	

plot::Point		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
plot::Point(*punkt*)	zeichnet den Punkt *punkt* in ein zweidimensionales Plot-Objekt	
plot::Point(*punkt, zeichenoptionen*)	zeichnet den Punkt *punkt* mit den angegebenen Optionen *zeichenoptionen*	

plot::Pointlist		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
plot::Pointlist(*punktliste*)	zeichnet die Punktliste *punktliste* in ein zweidimensionales Plot-Objekt	
plot::Pointlist(*punktliste, zeichenoptionen*)	zeichnet die Punktliste *punktliste* mit den angegebenen Optionen *zeichenoptionen*	

plot::Point		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
plot::Point(*punkt*)	zeichnet den Punkt *punkt* in ein zweidimensionales Plot-Objekt	
plot::Point(*punkt, zeichenoptionen*)	zeichnet den Punkt *punkt* mit den angegebenen Optionen *zeichenoptionen*	

plot::Function2d		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
plot::Function2d(*funktionen*)	zeichnet die Funktion(en) *funktionen* in ein zweidimensionales Plot-Objekt	
plot::Function2d(*funktionen, bereiche*)	zeichnet die Funktion(en) *funktionen* in ein durch die Achsenbereiche *bereiche* eingeschränktes, zweidimensionales Plot-Objekt	
plot::Function2d(*funktionen, bereiche, optionen*)	zeichnet die Funktion(en) *funktionen* in ein durch die Achsenbereiche *bereiche* eingeschränktes, zweidimensionales Plot-Objekt; zusätzlich werden die angegebenen Szeneoptionen *optionen* ausgewertet	

plot::Function3d		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
plot::Function3d(*funktionen*)	zeichnet die Funktion(en) *funktionen* in ein dreidimensionales Plot-Objekt	
plot::Function3d(*funktionen, bereiche*)	zeichnet die Funktion(en) *funktionen* in ein durch die Achsenbereiche *bereiche* eingeschränktes, dreidimensionales Plot-Objekt	
plot::Function3d(*funktionen, bereiche, optionen*)	zeichnet die Funktion(en) *funktionen* in ein durch die Achsenbereiche *bereiche* eingeschränktes, dreidimensionales Plot-Objekt; zusätzlich werden die angegebenen Szeneoptionen *optionen* ausgewertet	

plot		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
plot(*plot_objekte*)	zeichnet die Funktion(en) / Plots aus *plot_objekte*	

weitere interessante Zeichen-Funktionen: (Informationen über die Hilfe)

plot::data	*erstellen einer Daten-Darstellung, eines Histogramms*
plot::Curve2d	*erstellen einer zweidimensionalen Kurve / Funktion*
plot::Curve3d	*erstellen einer dreidimensionalen Kurve / Funktion*
plot::Ellipse2d	*erstellen einer zweidimensionalen Ellipsen- / Kreis-Darstellung / Funktion*
plot::contour	*erstellen eine Konturen-Plots (Höhenlinien-Plot)*
plot::matrixplot	*erstellen einer dreidimensionalen Darstellung aus einer Matrix*
plot::piechart2d	*erstellen eines Kreisdiagramms*
plot::piechart3d	*erstellen eines Tortendiagramms*
plot::Polygon	*erstellen eines Polygons (geschlossenen Linien-Objekts)*
plot::Rectangle	*erstellen eines Rechtecks*
plot::Scene	*erstellen einer graphischen Szene aus Graphikobjekten mit Angabe von gemeinsamen Szene-Optionen*
plot::vector	*Darstellung eines Vektors*
plot::vectorfield	*erstellen einer Darstellung von Vektoren in einem zweidimensionalen Feld (z.B. Strömungsdarstellung)*
plot::Turtle	*erstellen einer Turtle-Grafik (Schildkröten-Grafik)*

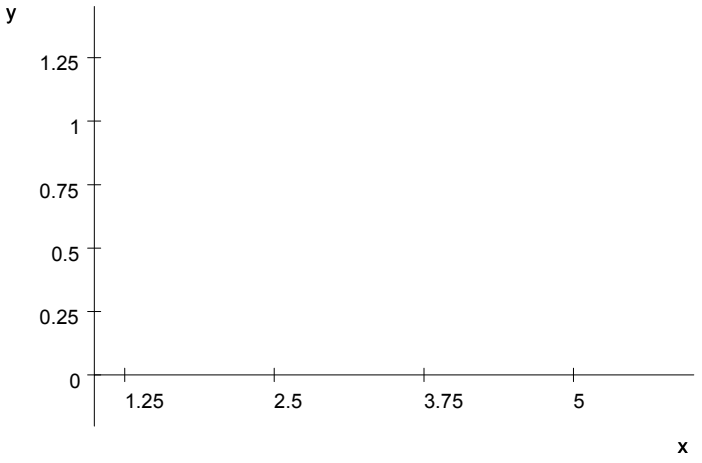
2.8.4. Exkurs: Wie kommt man von experimentellen Werten zu Funktionen?

Aus punktuellen Messwerten möchte man gern für spätere Anwendungen eine allgemeingültige Funktion ableiten. Bei linearen Zusammenhängen reicht oft das Einzeichnen in ein Diagramm. Die Messwerte werden zu einer Linie verbunden und offensichtlich fehlerhafte Werte einfach rausgelassen. Als Ergebnis erhält man einen (ungefähr stimmenden) graphischen Zusammenhang. Dann schnell noch die Ordinate des Schnittpunkts mit der y-Achse **n** und den Anstieg **m** berechnen oder ablesen – und fertig ist die Funktion vom Typ: **y = m x + n** .

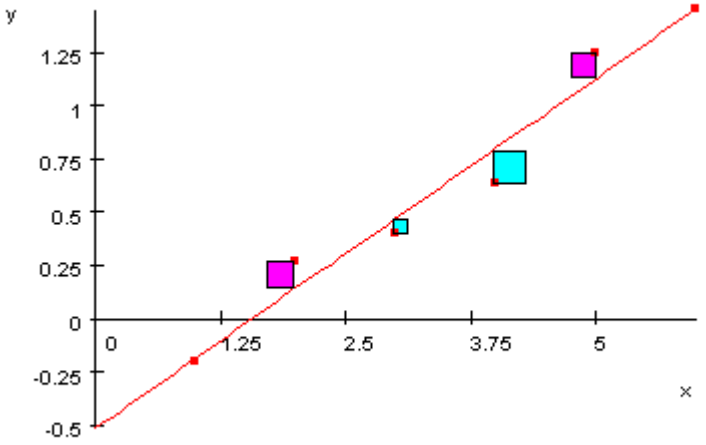
Was für lineare Zusammenhänge noch recht leicht sein kann, wird bei anders gearteten funktionellen Beziehungen schnell zum Problem.

Des weiteren haben experimentell ermittelte Zusammenhänge einige Nachteile. Sie stellen meist nur Einzelwerte aus einem großen Geltungsbereich (Definitionsbereich) dar und sie sind mit Fehlern behaftet.

Wenn die Werte von der "resultierenden" Geraden abweichen – die Messwerte streuen - dann hilft nur noch ein mathematisches Verfahren, dass einen Ausgleich zwischen den abweichenden Werten berechnet und somit eine Ausgleichgerade bestimmt.



Das Verfahren heißt Regression und gehört eigentlich zur Statistik (Wir arbeiten ja auch mit einer Stichprobe an Messwerten!). Die Regression minimiert die Quadrate der Abweichungen zur gedachten Ausgleichsgerade. Dabei werden auch **m** und **n** berechnet.



Der letztendlich berechnete Korrelationskoeffizient **r** sagt sowohl etwas über die Richtung des Zusammenhangs (Monotonie) als auch über die Qualität der Ausgleichsgerade aus.

(Je kleiner die Abweichungen (Quadrate), umso genauer / besser liegt die Ausgleichsgerade.)

Der Korrelationskoeffizient **r** kann einen Wert zwischen -1 und 1 einnehmen. Das Vorzeichen besagt, ob die Funktion monoton steigen (positives Vorzeichen) bzw.

monoton fallend (negatives Vorzeichen) ist. Ein Betrag des Korrelationsfaktor von **1** besagt 100%igen linearen Zusammenhang – d.h. alle Werte liegen direkt auf der Ausgleichgerade (Es gibt also gar keine Abweichungen.) Dagegen besagt ein Wert von **0**, dass eigentlich keine sinnvolle Ausgleichgerade existiert. Ein (linearer) Zusammenhang zwischen den beiden Größen existiert wohl nicht. Meist werden die Werte für **r** irgendwo dazwischen liegen. Mittels Testverfahren lässt sich dann testen, ob ein gesicherter Zusammenhang (in Abhängigkeit von der Anzahl der vorliegenden Meßwerte) vorliegt (→ chi-Quadrat-Test).
Wir wollen hier keine Herleitung od. ä. vornehmen – wir wollen das Verfahren einfach nur nutzen.
Zur Berechnung der Regressionsgeraden benötigen wir zuerst den (arithmetischen) Mittelwert der Stichproben-Reihen:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} \quad \text{und} \quad \bar{y} = \frac{\sum_{i=1}^{n} y_i}{n} \quad .$$

Mit deren Hilfe kann dann der Anstieg **m** (oft auch mit **b** bezeichnet) berechnet werden:

$$m = \frac{\frac{1}{n}\sum_{i=1}^{n} x_i y_i - \overline{\bar{x}\bar{y}}}{\frac{1}{n}\sum_{i=1}^{n} x_i^2 \bar{x}^{-2}}$$

Die Ordinate des Schnittpunkts mit der y-Achse n ergibt sich dann über: $n = \bar{y} - m\bar{x}$
Zur "Gegenprobe" wird der entgegengesetzte Zusammenhang zwischen **x** und **y** benutzt. Auch hierfür berechnet sich die Regressionsgerade wie oben besprochen – nur mit getauschten **x** und **y**. Zur Unterscheidung der Anstiege und Schnittpunkte mit der y-Achse sei die "Gegen"-Funktion so definiert: **x = s y + t**

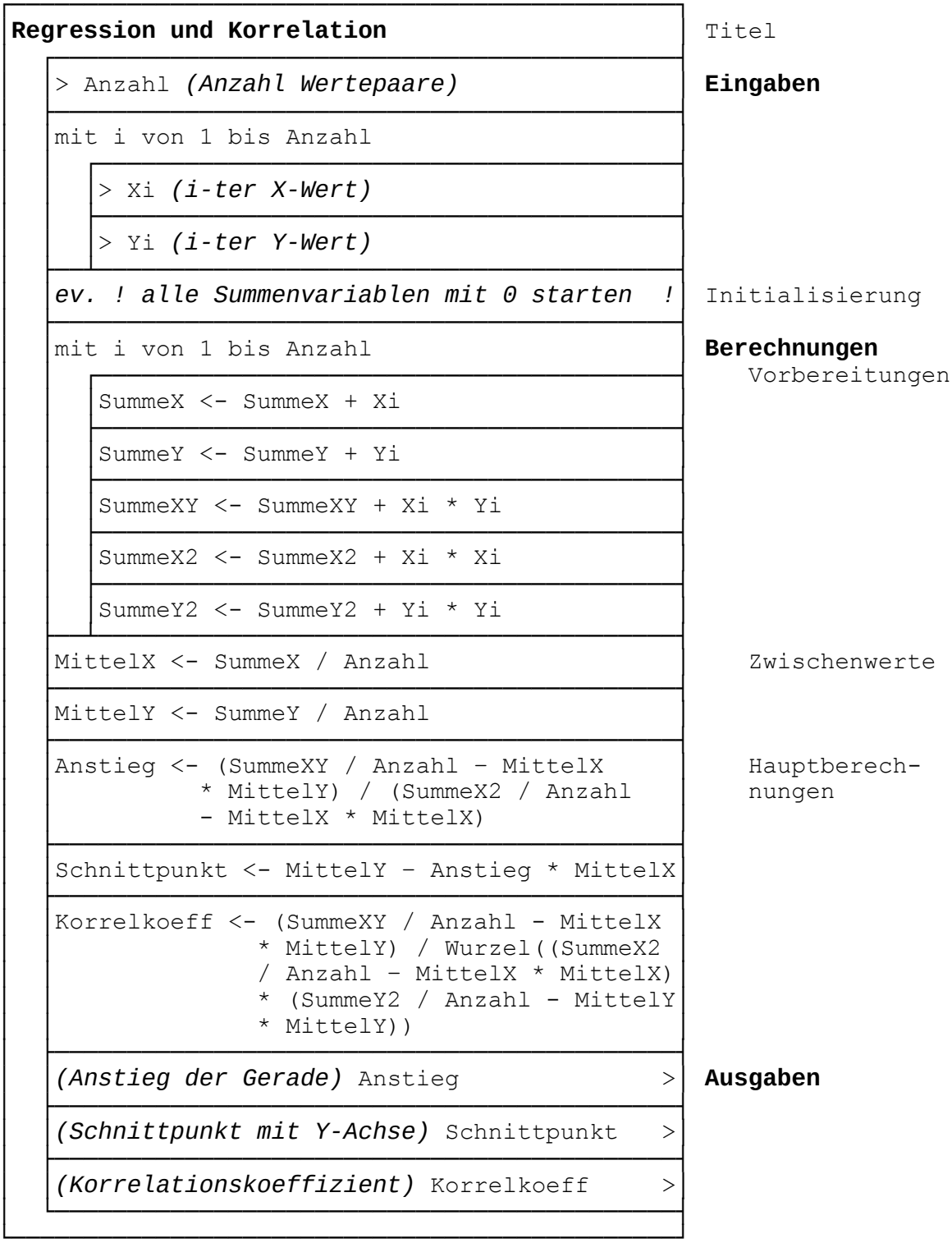
$$s = \frac{\frac{1}{n}\sum_{i=1}^{n} x_i y_i - \overline{\bar{x}\bar{y}}}{\frac{1}{n}\sum_{i=1}^{n} y_i^2 \bar{y}^{-2}} \quad \text{und} \quad t = \bar{x} - s\bar{y}$$

Beide Regressionsgeraden sind normalerweise nicht deckungsgleich, so dass sie um den Schnittpunkt **S** $(\bar{x};\bar{y})$ eine Schere bilden. Das Maß für die Öffnung der Schere ist der erwähnte Korrelationskoeffizient **r**.

$$r = \frac{\frac{1}{n}\sum_{i=1}^{n} x_i y_i - \overline{\bar{x}\bar{y}}}{\sqrt{\left(\frac{1}{n}\sum_{i=1}^{n} x_i^2 - \bar{x}^{-2}\right)\left(\frac{1}{n}\sum_{i=1}^{n} y_i^2 - \bar{y}^{-2}\right)}}$$

Da der Rechenaufwand doch recht hoch ist, bietet sich die Verarbeitung der Daten mit einem Computer-Programm an. Zuerst zeigen wir Struktogramm. Dieses ist so gestaltet, dass eine Umsetzung in eine imperative (befehlende, prozedurale) Programmiersprache leicht möglich sein sollte.

Struktogramm



Am Beispiel PASCAL wird nun das allgemeine Struktogramm in ein Programm umgesetzt. Für PASCAL gibt es freie Versionen im Internet, so dass ein Nachvollziehen kein Problem sein dürfte.
Auch anderer imperative Programmiersprachen eigenen sich zur direkten Umsetzung des Struktogramms. Natürlich läßt sich das Programme beliebig verbessern und erweitern.

PASCAL-Programm

```
program regression;

const maxanzahl=100;

var x,y: array[1..maxanzahl]of real;
    anzahl,k: word;
    summeX,summeY,mittelX,mittelY,summeX2,summeY2,summeXY: real;
    anstiegMX,schnittNY,korrelkoeffR: real;

begin

  repeat
    write('Anzahl der Wertepaare (max. ',maxanzahl,'): ');
    readln(anzahl);
  until anzahl<=maxanzahl;
  for k:=1 to anzahl do
    begin
      write(k,'. X-Wert: ');readln(x[k]);
      write(k,'. Y-Wert: ');readln(y[k]);
    end;

  summeX:=0;
  summeY:=0;
  summeX2:=0;
  summeY2:=0;
  summeXY:=0;
  for k:=1 to anzahl do
    begin
      summeX:=summeX+x[k];
      summeY:=summeY+y[k];
      summeX2:=summeX2+x[k]*x[k];
      summeY2:=summeY2+y[k]*y[k];
      summeXY:=summeXY+x[k]*y[k];
    end;
  mittelX:=summeX/anzahl;
  mittelY:=summeY/anzahl;
  anstiegMX:=(summeXY/anzahl-mittelX*mittelY) →
    /(summeX2/anzahl-mittelX*mittelX);
  schnittNY:=mittelY-anstiegMX*mittelX;
  korrelkoeffR:=sqrt((summeX2/anzahl-mittelX*mittelX) →
    *(summeY2/anzahl-mittelY*mittelY));
  korrelkoeffR:=(summeXY/anzahl-mittelX*mittelY)/korrelkoeffR;

  writeln('f(x):  Y  =  ',anstiegMX,'  *  X  +  ',schnittNY);
  writeln('Korellationskoeffizient: ',korrelkoeffR);

  readln;
end.
```

→ bedeutet, das diese Zeile mit der nachfolgenden aufgefüllt werden muß
kursive Zeilen können auch weggelassen werden

Ein weitere Umsetzung des Regressionsverfahrens zeigt das folgende BASIC-Programm. Der Algorithmus ist etwas anders und auch undurchsichtiger. Dafür ist das Programm schneller eingetippt und mit einer zusätzlichen Berechnung für den t-Test versehen.

BASIC-Programm
(etwas anderer Berechnungsalgorithmus (etwas undurchsichtiger, dafür aber weniger Tipp-Aufwand))

```
CLS
PRINT "Regression und Korrelation"
PRINT "============================================================="
PRINT
PRINT "Eingaben:"
PRINT "---------"
INPUT "Anzahl der Wertepaare: "; n
DIM x(n), y(n)
FOR k = 1 TO n
  PRINT k; ". Wertepaar:"
  INPUT "   X= "; x(k)
  INPUT "   Y= "; y(k)
NEXT k
PRINT "Berechnungen:"
PRINT "-------------"
FOR k = 1 TO n
  summeX = summeX + x(k)
  summeY = summeY + y(k)
  summeX2 = summeX2 + x(k) * x(k)
  summeY2 = summeY2 + y(k) * y(k)
  summeXY = summeXY + x(k) * y(k)
  PRINT "*";
NEXT k
mittelX = summeX / n
mittelY = summeY / n
hilfXX = summeX2 - summeX * mittelX
hilfYY = summeY2 - summeY * mittelY
hilfXY = summeXY - summeX * mittelY
anstiegM = hilfXY / hilfXX
schnittN = mittelY - anstiegM * mittelX
korrelkoeffR = hilfXY / SQR(hilfXX * hilfYY)
Ttest = anstiegM / SQR((hilfYY - b * hilfXY) / (n - 2) * hilfXX)
PRINT "|"
PRINT "Ergebnisse:"
PRINT "-----------"
PRINT "Mittel der X-Werte      : "; mittelX
PRINT "Mittel der Y-Werte      : "; mittelY
PRINT "Anstieg der Geraden     : "; anstiegM
PRINT "Schnittpunkt mit Y-Achse: "; schnittN
PRINT "Korrelationskoeffizient : "; korrelkoeffR
PRINT "Wert für t-Test         : "; Ttest
```

Als nächstes Problem kann auf uns zukommen, dass der gesuchte Zusammenhang gar nicht linear ist. Durch einige kleine Tricks kann man aber viele Grundformen von Funktionen in eine lineare Form überführen. Die dann über die Regression berechneten Geradengleichungen müssen dann später wieder auf die Originalfunktion zurück transformiert werden.
Nehmen wir zuerst den Funktionstyp $\mathbf{y = a\ x^{b}}$. Durch eine Logarithmierung erhalten wir $\mathbf{lg(y) = b\ lg(x) + lg(a)}$, also genau wieder unsere lineare Grundfunktion. Wir brauchen für unser oben besprochenes Verfahren einfach nur die Logarithmen von **x** und **y** benutzen. Zum Schluß reicht die Anwendung von **m** als **b** und die Berechnung von **a** über die Potenzierung: $\mathbf{a = 10^{n}}$.
Das gleiche Prinzip können wir auch bei dem Funktionstyp $\mathbf{y = a\ b^{x}}$ nutzen. Hier lautet die logarithmierte Form: $\mathbf{lg(y) = lg(b)\ x + lg(a)}$. In der Regression brauchen wir nun die logarithmierten Werte von **y** und die Originalwerte von **x**. Die Faktoren **a** und **b** ergeben sich wie folgt: $\mathbf{a = 10^{n}}$ und $\mathbf{b = 10^{m}}$.
Bleibt von den Grundfunktionstypen noch die Logarithmusfunktion selbst:

y = a lg(x) + b . Es müssen diesmal die logarithmierten Werte von **x** und die Originalwerte von **y** in die Regression eingehen und dann über **a = m** und **b = n** der Faktor und der Korrekturwert für die Gerade übernommen werden.
Im Zweifelsfall muß man alle Grundfunktionen berechnen und dann die mit dem besten Korrelationskoeffizienten auswählen.
Für die FREAKS sei noch darauf hingewiesen, dass noch einige zusätzliche Grundfunktionen über das Benutzen der Reziproken von **x** und / oder **y** zugänglich sind.
Alle Funktionen mit mehreren Potenzen (Polynome) oder mit Winkelfunktionen müssen über andere Verfahren (Polynomzerlegung, FOURIER-Analyse, ...) gelöst werden. In der Praxis (von einfachen Schul- und Laborexperimenten) haben sich die besprochenen Grundformen als völlig hinreichend erwiesen. Erst bei stark ungünstigen Korrelationskoeffizienten sollte man die Anzahl der Stichprobe erhöhen (mehr Messwerte) bzw. nach anderen Verfahren Ausschau halten.

Auch in MuPAD ist die Regression integriert. Leider fehlt wohl eine Funktion zum Berechnen des passenden Korrelationskoeffizienten.
Die Funktion ***stats::linReg*** ermittelt aus einer Liste von Wertepaaren bzw. Listen von zwei Größen die Parameter ***[n,m]*** für die lineare Funktion **y = n + m x** .

```
stats::linReg([[1,1],[2,2],[3,3]])
```

$$[0, 1]$$

```
X:=[0,2,8,-2,-5]:
Y:=[3,7,19,-1,-7]:
stats::linReg(X,Y)
```

$$[3, 2]$$

Kurzreferenz: Regression

! In dieser Referenz werden vor allem häufig benötigte Funktionen usw. usf. sowie der häufig gebrauchte Syntax aufgeführt. Für vollständige Informationen wählen Sie bitte die Hilfe bzw. die Programm-Referenz!

stats::linReg		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
stats::linReg(*punktliste*)	berechnet aus den Werten in *punktliste* die Parameter **[n,m]** für die lineare Funktion: y = n + m x	
stats::linReg(*x_wertliste*, *y_wertliste*)	berechnet aus den beiden Wertelisten *x_wertliste* und *y_wertliste* die Parameter **[n,m]** für die lineare Funktion: y = n + m x	

Aufgaben:

1. *Berechnen Sie die Regressionsgerade für die folgenden Messwerte (als Paare)!*
   Experiment: Temperaturabhängigkeit eines OHMschen Widerstands

Temperatur in °C	-10	-5	+15	+20	+22	+55
Widerstand in Ω	894	893	973	1000	1022	1142

2. *Erstellen Sie eine graphische Darstellung der gefundenen Funktion!*
3. *Berechnen Sie die Regressionsgerade für die folgenden Messwerte (als Größenliste)!*
   Statistische Analyse: Zusammenhang von Körpergröße und Körpermasse bei Jugendlichen

Körpergröße in m	1,52	1,60	1,66	1,70	1,74	1,78	1,8	1,84
Körpermasse in kg	50	55,9	60,6	63,9	66,8	69,8	71,3	74,7

4. *Erstellen Sie über Plots eine gemeinsame graphische Darstellung der Messpunkte und des funktionellen Zusammenhangs!*

2.9. Symbolisches Rechnen

Bei der Besprechung von Bezeichner sind wir schon darauf eingegangen, dass MuPAD mit diesen Variablen auch direkt rechnen kann. Die Bezeichner werden als Symbole betrachtet. Das Hantieren mit solchen Symbolen stellt einen wichtigen Bereich in der Mathematik dar. Eigentlich kann man so weit gehen und behaupten: "Die ganze Mathematik ist ein Jonglieren von Symbolen nach bestimmten Regeln." Die Eins (1) ist im Prinzip auch nur ein Symbol für ein definiertes Etwas. Durch Aufstellen von Regel, wie z.B.: 1+1 = 2 wird die elementare – und mit noch umfassenderen und komplizierteren Regeln - auch die höhere Mathematik möglich.
Das Umstellen von Termen, das Differenzieren oder das Integrieren sind typische Beispiele für das Rechnen mit Symbolen.

2.9.1. Umstellen und Vereinfachen von Termen

Beim Umstellen von Termen können wir auf die Funktion ***solve*** (→ 2.7.3. Lösen von Gleichungen) zurückgreifen.
Das Beispiel:

$\frac{r}{t} = e*a$ nach **r** umgestellt ist also:

$r = a*e*t$

- `f:=r/t=e*a`

$$\frac{r}{t} = a \cdot e$$

- `solve(f,r)`

$$\{a \cdot e \cdot t\}$$

Interessant ist die Ausgabe von solve, wenn mathematische Problemstellen auftauchen (Division durch Null usw.). Oft sind solche Informationen gar nicht gefragt, so dass man sich dann einfach "allgemeingültige" Lösung heraussuchen muß. Im Beispiel:

$t = \frac{r}{a*e}$

- `solve(f,t)`

$$\begin{cases} \varnothing & \text{if } r \neq 0 \wedge (a = 0 \vee e = 0) \\ \left\{\frac{r}{a \cdot e}\right\} & \text{if } a \neq 0 \wedge e \neq 0 \wedge r \neq 0 \\ \mathbb{C} & \text{if } a = 0 \wedge e = 0 \wedge r = 0 \\ \mathbb{C} & \text{if } a = 0 \wedge r = 0 \wedge e \neq 0 \\ \mathbb{C} & \text{if } e = 0 \wedge r = 0 \wedge a \neq 0 \\ \varnothing & \text{if } r = 0 \wedge a \neq 0 \wedge e \neq 0 \end{cases}$$

Vorgegebenen oder berechneten Ausdrücke werden oft viel zu kompliziert dargestellt. Es ist dann angebracht, das Ergebnis usw. an ***normal*** zur Vereinfachung zu übergeben. Die Funktion fasst rationale Ausdrücke zusammen, indem sie auf einen gemeinsamen Nenner gebracht werden.

- `c:=1+(x^2-1)/(x+1)`

$$\frac{x^2 - 1}{x + 1} + 1$$

- `normal(c)`

$$x$$

Das kann auch probeweise passieren. Ist das Ergebnis schon recht "normal", dann wird der Ausdruck unverändert zurückgegeben.
Den z.B. hinter einem Bezeichner steckenden Ausdruck kann man - "normalisiert" - wieder dem Bezeichner zuweisen.

- `c:=normal(c)`

$$x$$

Das ist ja auch kein Problem, da beide ja (mathematisch) gleich sind. Wir ersetzen lediglich den kompliziert formulierten durch den einfacheren.

Um einen rationalen Ausdruck in einfache Terme mit einfachen Nennern zu zerlegen, benutzt man ***partfrac*** (eng.: partial fraction => dt.: Partialbruch). Damit steht uns gewissermaßen eine Umkehrung von ***normal*** zur Verfügung.

- `f:=(x^2+5*x-9)/(x^2-x-6)`

$$\frac{5 \cdot x + x^2 - 9}{x^2 - x - 6}$$

- `g:=partfrac(f,x)`

$$\frac{3}{x+2} + \frac{3}{x-3} + 1$$

- `normal(g)`

$$\frac{5 \cdot x + x^2 - 9}{x^2 - x - 6}$$

2.9.2. Grenzwerte von Funktionen

Grenzwerte von Funktionen lassen sich mit ***limit*** bestimmen. Neben Funktion erwartet ***limit*** auch die Angabe einer Grenze. Mit ***infinity*** ist unendlich (∞) gemeint. Entsprechend gilt für -∞ (minus unendlich) ***-infinity***.

- `limit(tan(x)/x,x=0)`

$$1$$

- `limit(x^2,x=infinity)`

$$\infty$$

Stellen wir beim Prüfen einer Funktion auf Stetigkeit mit ***discont*** fest, dass Polstellen vorhanden sind, dann können wir auch an diesen Stellen die Grenzwerte ermitteln. Zu beachten ist dabei natürlich, dass eine Annäherung an die Polstelle von links bzw. rechts möglich ist und zumeist auch verschiedene Ergebnisse bringt (Wechsel von $+\infty$ nach $-\infty$ bzw. umgekehrt).

- `a:=sin(x)/(x-2)`

$$\frac{\sin(x)}{x-2}$$

- `discont(a,x)`

$$\{2\}$$

- `limit(a,x=2,Left)`

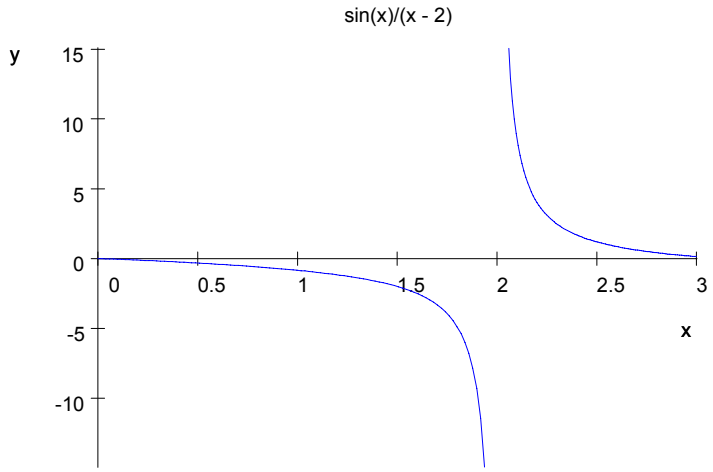
$$-\infty$$

- `limit(a,x=2,Right)`

$$\infty$$

Zur "Kontrolle" noch schnell die graphische der kritischen Stelle.

- `plotfunc2d(a,x=0..3)`



2.9.3. Differentiation von Funktionen

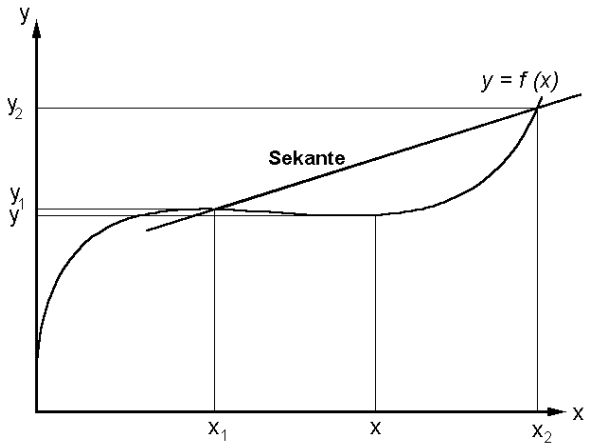
Die Grundlagen für die Differentialrechnung wurden in der 2. Hälfte des 17. Jahrhunderts unabhängig voneinander durch Isaac NEWTON (1643 - 1727) und Gottfried Wilhelm LEIBNITZ entwickelt. Während NEWTON an der Klärung von Funktionen zu Bewegungsabläufen (z.B. Momentangeschwindigkeit in gleichmäßig beschleunigten Bewegungen) interessiert war, forschte LEIBNITZ an der Berechnung von Tangenten an bestimmten Funktionsgraphen.

Mathematisch steckt das gleiche Problem dahinter: Das Verhalten einer Kurve / Funktion in einem bestimmten Punkt (der Kurve / Funktion).

Mit den üblichen mathematischen Mitteln läßt das Verhalten einer Funktion über den Differenzenquotienten:

$$\frac{\Delta y}{\Delta x} = \frac{y_2 - y_1}{x_2 - x_1}$$

berechnen. Graphisch entspricht dies einer Sekante.



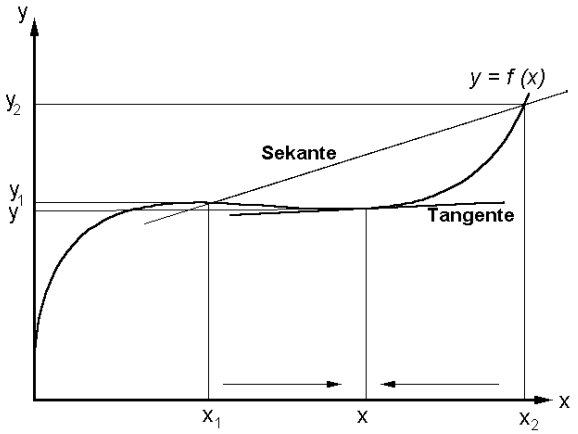
Wie wir gleich sehen werden, ist dies sehr ungenau, da der genaue Verlauf der Kurve / Funktion unberücksichtigt bleibt. Sowohl die mittelwertähnliche Sekante durch (x1,y1) und (x2,y2) entspricht nicht einer Tangente, wenn sie durch den Punkt (x,y) gehen würde (durch Parallelverschiebung). Auch die seitlichen Sekanten (x1,y1)(x,y) und (x,y)(x2,y2) erfüllen unsere Anforderung nach nur einem Schnittpunkt mit der Funktion.

Das Ergebnis wird aber umso besser, je kleiner man den Abstand der x-Werte (Δx) von x wählt.

Das genaueste Ergebnis erhält man, wenn es gelungen ist, die Differenz der x-Werte gegen 0 zu schrauben. Wir betrachten sozusagen einen Grenzwert – den Differenzialquotient:

$$\frac{dy}{dx} \quad \text{oder} \quad \frac{df(x)}{dx}$$

Die Tangente zeigt in unserem Beispiel doch einen deutliche veränderten Anstieg (eine starke Lageabweichung) zur (Mittelwerts-)Sekante.



Mit Kenntnis der Tangente lassen sich nun genaue Aussagen über das Verhalten der Funktion im betrachteten Punkt machen.

Im gezeigten Beispiel besaß die Funktion erst einmal in einem Punkt eine Tangente – sie war in diesem Punkt differenzierbar. Ist die Funktion im gesamten betrachteten Intervall differenzierbar, dann erhält man eine neue abgeleitete Funktion (sozusagen für alle "Tangenten"):

$$f'(x) \quad \text{oder} \quad y'$$

Ableitungen von mathematischen Ausdrücken sind praktische Hilfsmittel für Kurvendiskussion oder Extremwertaufgaben. Aus menschlicher Sicht scheinen sie schwierig.

Für Computer stellen sie kaum ein Problem dar. In einigen Programmiersprachen (PROLOG, LISP) lassen sich einfache symbolische Differenzierer mit wenigen Befehlszeilen realisieren. (BASIC und PASCAL bräuchten dafür etliche Seiten! Dort ist schon die Erkennung der Eingabefunktion und ihrer Bestandteile ein sehr aufwändiger Programmteil.)

PROLOG-Quelltext	**Kommentare / Erläuterungen**

`op(10,yfx,^).`
`d(X,X,1):=!.`
`d(C,X,0):- atomic(C), C\=X, !.`
`d(C,X,0):- number(C).`
`d(B^E,X,E*DX*B^(C - 1):- atomic(E),`
`E\=X, !, d(B,X,DX).`
 	
`d(- Y,X,- DX):- d(Y,X,DX).`
 	
`d(sin(X),X,cos(X)).`
`d(cos(X),X,0-sin(X)).`
`d(exp(X),X,exp(X)).`
`d(log(X),X,1/X).`
	Daten-Basis
	- Einführung eines Potenz-Symbol (**^**)
	
	- Konstantenregel
	
	- Potenzregel
	bei fehlendem Potenz-Symbol (**^**):
	`d(pot(X,N),X,N*pot(X,N1)):- N>1,`
	`N1 is N - 1.`
	- Negation
	ev.: `d(~Y,X,~DX):- d(Y,X,DX).`
	
	bzw. bei Symbol **`ln`** statt **`log`** für natürlichen Logarithmus:
	`d(ln(X),X,1/X).`

(In PROLOG muß die Groß- und Kleinschreibung beachtet werden! Variablen werden mit Großbuchstaben gekennzeichnet. Ev. kann bei verschiedenen PROLOG-Versionen zu Problemen mit dem Präfix-Minus und mit der Potenz-Realisierung kommen! Obiges Programm wurde für die nachfolgende Benutzung in der Datei `DIFF.PRO` gespeichert.)

Das war's! PROLOG liefert damit bei einfachen Funktionen schon mathematisch exakte Ergebnisse.

```
?- consult('diff.pro').
Consulting diff.pro
Yes
?- d(sin(x),x,Erg).
Erg = cos(x)
No
?- d(pot(x,5),x,Erg).
Erg = 5 * pot(x,4)
No
```

Ausgenommen sind lediglich verkettete Funktionen.

```
?- d(sin(x)+pot(x,5),x,Erg).
No
```

(Zur Kennzeichnung von Eingaben und Ausgaben verwende ich die bekannten Farben, obwohl sie im verwendeten PROLOG-System nicht so angezeigt werden!)

Um auch verkettete Funktionen zu bearbeiten sind Änderungen an einigen Grundfunktionen nötig. Diese habe ich oben nicht mit hineingenommen, damit die Einfachheit der Prinzipien und Umsetzung der Differentiation deutlich wird.

geänderter PROLOG-Quelltext	Kommentare / Erläuterungen
dPsinPY),X,cosPY)*DY)Q- dPY,X,DY). dPcosPY),X,-sinPY)*DY)Q- dPY,X,DY). d(pot(A,N),X,N*pot(A,N1)*DA):- N>1, N1 is N - 1, d(A,X,DA). d(F+G,X,DF+DG):- d(F,X,DF), d(G,X,DG). d(F-G,X,DF-DG):- d(F,X,DF), d(G,X,DG). d(F*G,X,DF*G+F*DG):- d(F,X,DF), d(G,X,DG). d(F/G,X,(DF*G-F*DG)/(G*G)):- d(F,X,DF), d(G,X,DG). d(X+C,X,DX):- atomic(C), C\=X, d(X,X,DX), !. d(C+X,X,DX):- atomic(C), C\=X, d(X,X,DX), !. d(X-C,X,DX):- atomic(C), C\=X, d(X,X,DX), !. d(C-X,X,DX):- atomic(C), C\=X, d(X,X,DX), !. …	***Regel-Basis*** - Summenregel - Produktregel - Quotientenregel - weitere Regeln für den Umgang mit Konstanten und Rechenoperationen

Leider sehen die Ableitungen bei komplexeren Funktionen recht unschön (aufgebläht) aus. Zusammenfassungen und die Eliminierung überflüssiger Elemente bedürfen eines zusätzlichen Programmieraufwand oder der Handarbeit auf dem Rechenblatt.

```
?- reconsult('diff.pro').
Reconsulting diff.pro
Yes
?- d(sin(x)+pot(x,5),x,Erg).
Erg = cos(x) * 1 + 5 * pot(x,4)
No
?- d(sin(x+pot(x,5)),x,Erg).
Erg = cos(x + pot(x,5)) * (1 + 5 * pot(x,4))
No
```

erweiterter PROLOG-Quelltext	Kommentare / Erläuterungen

`vereinfadd(0+X,X).`
`vereinfadd(X+0,X).`
`vereinfsub(X - 0,X).`
`vereinfsub(0 - X,-X).`
`vereinfmult(X*1,X).`
`vereinfmult(1*X,X).`
`vereinfmult(0*X,0).`
`vereinfmult(X*0,0).`
`vereinfdiv(X/1,X).`
`vereinfpotPpotPX,1),X).`
	Vereinfachungsregeln für Grundoperationen

`vereinfachen(A,A):- atomic(A), !.`
`vereinfachen(X,Z):- vereinfadd(X,Z),!.`
`vereinfachen(X,Z):- vereinfsub(X,Z),!.`
`vereinfachen(X,Z):- vereinfmult(X,Z),!.`
`vereinfachen(X,Z):- vereinfdiv(X,Z),!.`
`vereinfachen(X,Z):- vereinfpot(X,Z),!.`
`vereinfachen(X+Y,X1+Y1):- vereinfachen(X,X1),`
`vereinfachen(Y,Y1).`
`vereinfachen(X*Y,X1*Y1):- vereinfachen(X,X1),`
`vereinfachen(Y,Y1).`
`vereinfachen(X - Y,X1 - Y1):- vereinfachen(X,X1),`
`vereinfachen(Y,Y1).`
`vereinfachen(X/Y,X1/Y1):- vereinfachen(X,X1),`
`vereinfachen(Y,Y1).`
`vereinfachen(pot(X,Y),pot(X1,Y1)):-`
`vereinfachen(X,X1), vereinfachen(Y,Y1).`
	Zusammenführen und Anwenden der Vereinfachungsregeln

`diff(Y,X,Z):- d(Y,X,X1), vereinfachen(X1,Z).`	das neue Aufruf-Prädikat

Nun liefert uns PROLOG schon akzeptable Ergebnisse.

```
?- reconsult('diff.pro').
Reconsulting diff.pro
Yes
?- d(sin(x)+pot(x,5),x,Erg).
Erg = cos(x) + 5 * pot(x,4)
No
```

In der Literatur /19/ bzw. im Internet /z.B.: 18/ lassen sich noch weitaus effektivere Vereinfachungsalgorithmen und komplexere Symbolische Differenzierer finden. Dabei verliert sich aber auch die Verständlichkeit für Ungeübte und nicht Nicht-PROLOGer.
Bei der abschließenden Betrachtung der Materie kommt man leicht zu dem ketzerischen Schluß, das so einfach, wie das Ganze ist, der normale Mensch (Nicht-Mathe-Freak) den Wald vor lauter Bäumen nicht sieht. (Und mir scheint auch der Schluß nicht weit, das Mathematiker eigentlich nur (einfach gestrickte, aber geschickte) Symbolschuppser sind.)

MuPAD ist durch seine exakte Daten-Verarbeitung ein sehr leistungsfähiger Differenzierer.
Die Funktion ***diff*** bildet genau die erste Ableitung ihres Arguments. Natürlich muß mit angegebenen werden, nach welcher Unbestimmten abgeleitet werden soll. Ansonsten bekommt man die Nullte Ableitung – also die Original-Funktion selbst – zurück.

- `t:=x^2+3`

$$x^2 + 3$$

- `diff(t,x)`

$$2 \cdot x$$

- `diff(t)`

$$x^2 + 3$$

Durch mehrfachen (geschachtelten) Aufruf von ***diff*** lassen sich auch weitere Ableitungen errechnen.
Alternativ kann man die Ableitungsstufe hinter dem Folgeoperator ***$*** angeben. Somit entspricht dann ***diff(t,x$3)*** dem geschachtelten Aufruf ***diff(diff(diff(t,x),x),x)***

- `diff(t,x$1)`

$$2 \cdot x$$

- `diff(t,x$4)`

$$0$$

Der Differentialoperator ***D*** findet immer dann Anwendung, wenn der Parameter für das Differenzieren als (variablenlose) Funktion vorliegt. Alternativ lässt sich dann auch der Strich-Operator **'** (Ableitungsstrich) benutzen. Die Schreib- und Benutzungsweise entspricht dann der schulmathematischen Tradition. Sie bringt besonders beim Berechnen von Funktionswerten Vorteile.

- `D(sin*cos)`

$$\cos \cdot \cos - \sin^2$$

- `D(tan)`

$$\tan^2 + 1$$

Der Bezeichner ***id*** im Ergebnis steht für den eigentlich erwarteten Identifizierer (die Variable) – also das Argument der zu integrierenden Funktion.

- `D(ln)`

$$\frac{1}{\mathrm{id}}$$

Zum Vergleich die gleiche Funktion mit abzuleitender Variable:
id entspricht also in unserem Beispiel der Variable x.

- `diff(ln(x),x)`

$$\frac{1}{x}$$

Aufgaben:

Lösen Sie die Aufgaben immer erst auf dem Papier! Nutzen Sie dann MuPAD zur Kontrolle!

1. Stellen Sie die nachfolgen Terme nach allen darin enthaltenen Variablen um!

a) $0 = x^2 - 7$

b) $3x = 4y - 2x + y$

c) $8a + 3b = 2c - 5a + b$

2. Untersuchen Sie die nachfolgenden Funktionen auf Diskontinuitäten (Polstellen, Definitionslücken, ...)! Bestimmen Sie dann die Grenzwerte im Unendlichen bzw. an den Polstellen!

a) $f(x) = x^3 - 2x^2 + 3x$

b) $f(x) = \frac{3 - x^3}{x^2}$

c) $f(c) = \frac{2}{3c^3}$

d) $g(x) = \frac{x^2 - 2}{1 + 3x^3}$

3. Bilden Sie die ersten drei Ableitungen der nachfolgenden Funktionen! Stellen Sie immer alle drei Ableitungen und die Original-Funktion in jeweils einer Ansicht graphisch dar!

a) $f(x) = 5x^4 - x^3 + 2x^2 - 3$

b) $f(x) = \frac{3x^2}{4} + \frac{4x^3}{x}$

c) $f(c) = 3c^3 + c^2$

d) $f(s) = \frac{2s^2 - 4}{3 - s}$

4. Berechnen Sie mit MuPAD!

a) 5. Ableitung von: $f(x) = \frac{1}{3}x^8 - 2x^7 + 4x^4 - x^3 + 2x^2 + x - 4$

b) 9. Ableitung von: $h(a) = 0{,}42x^{21} - 1{,}3x^{13} + 4{,}6x^5 - x^2 + 82{,}4$

c) 5. Ableitung von: b)

5. Differenzieren Sie die nachfolgenden Funktionen!

a) $\sin \cdot \tan$

b) $\sin \cdot \cos \cdot \frac{\tan}{\cos}$

c) $\ln \cdot \frac{\cos}{\sin}$

2.9.4. Integration von Funktionen

Beim Integrieren sind wir auf der Suche nach der Stammfunktion F einer gegebenen Funktion f. Dabei ist die Funktion f die Ableitung der Stammfunktion F. Wie wir schon gesehen haben, geht beim Ableiten (Differenzieren) immer ein kleiner Teil Funktion "verloren". Daraus ergibt sich, dass viele Funktionen die Stammfunktion gewesen sein könnten. Die Menge aller Stammfunktionen einer Funktion f wird unbestimmtes Integral genannt. Durch Differenzieren kann jederzeit die Integration geprüft werden.
Auf die Integrationsregeln wollen wir hier nicht eingehen, da sie in jedem guten Schul-Tafelwerk (bis zum Abitur) aufgezeigt werden.
Beim bestimmten Integral betrachten wir zuerst einmal nur einen bestimmten Abschnitt – ein Intervall - der Funktion f. Die Grenzen a und b sind Minimum und Maximum des betrachteten (lückenlosen) Definitionsbereichs der Funktion.
Das bestimmte Integral ist eine Zahl, die den Flächeninhalt der Figur angibt, die durch die x-Achse, die Intervallgrenzen (Geraden x=a und x=b) und dem Graphen der Funktion eingeschlossen wird.

Die Integration einer Funktion übernimmt ***int***. Dabei ist sowohl die unbestimmte als auch die bestimmte Integration möglich.
Der Funktion ***int*** muß als erster Parameter die Integrandenfunktion (der Integrand) übergeben werden. Für die unbestimmte Integration reicht als zweiter Parameter die Übergabe der Variable, nach der integriert werden soll (Integrationsvariable).

- `a:=sin(x)`

$$\sin(x)$$

- `int(a,x)`

$$-\cos(x)$$

Bei der bestimmten Integration muß dann auch noch der Definitions- / Integrationsbereich (Grenzen) angegeben werden.

- `int(a,x=-PI..0)`

$$-2$$

Die gleiche Fläche erhalten wir durch Berechnung über den Hauptsatz der Differenzial- und Integralrechnung:

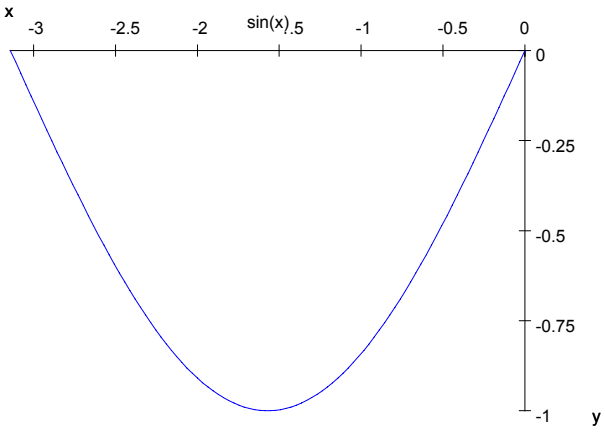
$$\int_a^b f(x)dx = F(b) - F(a) \qquad \text{für: } f = F'$$

- `-cos(0)-(-cos(-PI))`

$$-2$$

In unserem Fall negativ, weil die Fläche unter der x-Achse liegt.
Die graphische Darstellung der Funktion soll hier zur Veranschaulichung dienen.

- `plotfunc2d(sin(x),x=-PI..0)`



Ist der Definitionsbereich der Funktion nicht mit dem Integrationsgrenzen vereinbar, dann ergibt sich eine passende Fehlermeldung.
Die Funktion ***int*** liefert in solchen Fällen immer sich selbst zurück. In Windows-Systemen ist die Ausgabe entsprechend anspruchsvoll. Sie sollte aber nicht darüber hinwegtäuschen, das MuPAD nichts (neues) berechnet hat.

- `a:=int(exp(-z*x^2),x=0..infinity)`

```
Warning: Found potential discontinuities of the antiderivative.
Try option 'Continuous' or use properties (?assume). [intlib::antiderivative]
```

$$\int_0^{\infty} e^{-x^2 \cdot z}\, dx$$

Es bleibt nun z.B. die Möglichkeit einfach die Grenzen passend abzustecken oder aber mit ***assume*** eine saubere Definition vorzugeben.
Jetzt wird die undefinierte Stelle z=0 weggelassen und schon klappt es mit dem Integrieren.

- `assume(z>0)`

$$> 0$$

- `b:=int(exp(-z*x^2),x=0..infinity)`

$$\frac{\sqrt{\pi}}{2 \cdot \sqrt{z}}$$

Bei Integrationen – besonders bei expotentiellen Funktionen – tritt u.U. die Fehlerfunktion ***erf*** :

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}$$

auf. Unter Nutzung von float erhalten wir dann trotzdem ein brauchbares Ergebnis.
Alternativ bleibt eine numerische Integration über ***numeric::int***.

- `int(exp(-x^2),x=-1..3)`

$$\frac{\sqrt{\pi} \cdot \mathrm{erf}(1)}{2} + \frac{\sqrt{\pi} \cdot \mathrm{erf}(3)}{2}$$

- `float(%)`

$$1.633031481$$

- `numeric::int(exp(-x^2),x=-1..3)`

$$1.633031481$$

Aufgaben:

Lösen Sie die Aufgaben immer erst auf dem Papier! Nutzen Sie dann MuPAD zur Kontrolle!

1. Integrieren Sie die nachfolgenden Ausdrücke!

a) $f(x) = 3x^5 - 2x^2 + \frac{1}{3}x$

b) $f(x) = 23x^3 - 4$

c) $f(x) = 2x^{-3} + 3x^2 - 2$

2. Berechnen Sie Integrale für die nachfolgenden Ausdrücke!

a) $f(x) = 3x^5 - 2x^2 + \frac{1}{3}x$

b) $f(a) = 23a^3 - 4$

c) $f(x) = 2x^{-3} + 3x^2 - 2$

Kurzreferenz: Symbolische Rechnen

! In dieser Referenz werden vor allem häufig benötigte Funktionen usw. usf. sowie der häufig gebrauchte Syntax aufgeführt. Für vollständige Informationen wählen Sie bitte die Hilfe bzw. die Programm-Referenz!

normal		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
normal(*ausdruck*)	liefert die "Normalform" des rationellen Ausdrucks *ausdruck* zurück (Summen von rationellen Audrücken erhalten einen gemeinsamen Nenner)	Zähler und Nenner werden dabei expandiert (vergrößert)

partfrac		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
partfrac(*ausdruck*)	liefert eine Summe rationeller Terme mit einfachen Nennern vom Ausdruck *ausdruck* zurück (Partialbruchzerlegung)	Zählergrad wird kleiner als Nennergrad

factor		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
factor(*ausdruck*)	liefert die Faktorisierung des rationellen bzw. polynomialen Ausdrucks *ausdruck* zurück (Summen von rationellen Audrücken erhalten gemeinsamen Nenner, danach werden Zähler und Nenner faktorisiert)	Zähler und Nenner werden dabei faktorisiert (verkleinert)

limit		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
limit(*funktion, grenze*)	berechnet den Grenzwert der Funktion *funktion* an der angegebenen Stelle oder Grenze *grenze*	unendlich wird durch ***infinity*** deklariert
limit(*funktion, grenze, annäherung*)	berechnet den Grenzwert der Funktion *funktion* an der angegebenen Stelle *grenze* bei Annährung aus der Richtung *annäherung* (***Left*** (links) od. ***Right*** (rechts))	

discont		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
discont(*funktion, bereich*)	berechnet Diskontinuitäten der Funktion *funktion* in den Grenzen von *bereich*	

diff		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
diff(*funktion*, *variable*)	liefert die Ableitung (1. Ableitung) der Funktion funktion nach der Variable variable zurück	
diff(*funktion*, *variable*, *variable*)	liefert die nächst höherer Ableitung (2. Ableitung) der Funktion funktion nach der Variable variable zurück	lässt sich für beliebig hohe Ableitungen erweitern
diff(*funktion*, *variable* $ *grad*)	liefert die Ableitung vom Grad *grad* (*grad*-sche Ableitung) der Funktion funktion nach der Variable variable zurück	Vereinfachung der vorherigen Variante

$		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
ausdruck $ *anzahl*	der Ausdruck *ausdruck* wird in einer Folge *anzahl*-oft wiederholt	Funktion ist weit mächtiger! Hier wird nur die für ***diff*** notwendige Form aufgezeigt!

D		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
D(*funktion*)	bewirkt als Differentialoperator die Ableitung, der bei *funktion* angegebenen (variablenfreien) Funktion	

int		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
int(*funktion, variable*)	ermittelt das unbestimmte Integral der Funktion *funktion* nach der Variable *variable*	Ergebnis ist eine spezielle Stammfunktion, zur Verallgemeinerung fehlt die additive Konstante
int(*funktion, variable, bereich*)	ermittelt das bestimmte Integral (Integralwert) der Funktion *funktion* nach der Variable *variable* in den Grenzen, die durch *bereich* angegeben wurden	bei Problemen die Funktion ***numeric::int*** verwenden

numeric::int		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
numeric::int(*parameter*)	berechnet die Integration über numerische Verfahren; die Funktionsparameter parameter entsprechen den möglichen Aufrufen von ***int***	

2.9.5. Symbolisches Rechnen für Fortgeschrittene und FREAKS

Durch ***combine*** lassen sich Ausdrücke mit passenden Funktionen zusammenfassen. Zuerst sind dies einmal die Potenzgesetze, die angewendet werden. Meist wird das Zusammenfassen schon automatisch erledigt. Man kann mit zusätzlichen Optionen das Zusammenfassen auf weitere Funktionsgruppen erweitern. Im nebenstehenden Beispiel wurde eine Zusammenfassung bezüglich der Logarithmen (mit: ***ln***) angefordert. Weitere Zusammenfassungsoptionen sind ***exp***, ***sincos***, ***arctan*** und ***sinhcosh***.

- `h:=ln(x)+ln(2)+ln(3/2)`

$$\ln(x)+\ln(2)+\ln\left(\frac{3}{2}\right)$$

- `combine(h)`

$$\ln(x)+\ln(2)+\ln\left(\frac{3}{2}\right)$$

- `combine(h,ln)`

$$\ln(3\cdot x)$$

Die Funktion ***expand*** bewirkt hinsichtlich der Potenzgesetze die (fast vollständige) Umkehrung von ***combine***.
Mit ***factor*** lassen sich Ausdrücke in ihre Faktoren zerlegen. Somit stellt ***factor*** eine "Umkehrung" von ***expand*** dar.

- `z:=(x+y)^3*(x-y)`

$$(x+y)^3\cdot(x-y)$$

- `s:=expand(z)`

$$x^4-y^4-2\cdot x\cdot y^3+2\cdot x^3\cdot y$$

- `factor(s)`

$$(x-y)\cdot(x+y)^3$$

Eine weitere Möglichkeit zum Vereinfachen von Ausdrücken ist ***simplify***. Auch bei ***simplify*** kann man zusätzliche Steueroptionen für das Vereinfachen angeben. In MuPAD sind hierfür z.B. ***exp***, ***ln***, ***sin*** und ***cos*** möglich. Für Ausdrücke mit Wurzeln steht alternativ die Funktion ***radsimp*** zur Verfügung. Da man meistens nicht absehen kann, ob ein Ausdruck nach dem Bearbeiten mit den Vereinfachungsfunktionen übersichtlicher geworden ist, hilft nur probieren.

- `ff:=(x^2-1)/(x-1)`

$$\frac{x^2-1}{x-1}$$

- `simplify(ff)`

$$x+1$$

- `w:=sqrt(3*sqrt(3+2*sqrt(5-12*sqrt(3-2*sqrt(2))))+14)`

$$\sqrt{3\cdot\sqrt{2\cdot\sqrt{5-12\cdot\sqrt{3-2\cdot\sqrt{2}}}+3}+14}$$

- `simplify(w)`

$$\sqrt{3\cdot\sqrt{2\cdot\sqrt{17-12\cdot\sqrt{2}}+3}+14}$$

- `radsimp(w)`

$$\sqrt{2}+3$$

Aufgabe für Super-FREAKS:

Zeigen Sie, dass aus dem als w *definierten Wurzelausdruck durch Vereinfachung* $\sqrt{2}+3$ *folgt!*

Kurzreferenz: Symbolische Rechnen (für FREAKS)

! In dieser Referenz werden vor allem häufig benötigte Funktionen usw. usf. sowie der häufig gebrauchte Syntax aufgeführt. Für vollständige Informationen wählen Sie bitte die Hilfe bzw. die Programm-Referenz!

combine		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
combine(*ausdruck*)	ersetzt – wenn möglich – Produkte aus Potenzen durch einzelne Potenzausdrücke	

expand		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
expand(*ausdruck*)	wandelt Produkte von Summen in Summen von Produkten um (Realisierung des Distributivgesetzes)	"Umkehrung" von ***factor***

simplify		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
simplify(*ausdruck*)	Vereinfachung von *ausdruck* unter Verwendung von Termersetzungsregeln	

radsimp		
Syntax	**Beschreibung**	**Beispiele / Bemerkungen**
radsimp(*ausdruck*)	Vereinfachung von Wurzeln (Radikalen) in *ausdruck*	

2.10. Programmierung

Neben dem interaktiven Arbeiten mit MuPAD kann man die Arbeitschritte auch automatisieren. Im Allgemeinen nennt man so ein Vorgehen Programmieren. Einem System wird mitgeteilt, wie und wann es in welcher Form auf bestimmte Sachverhalte und Situationen (z.B. Eingaben) zu reagieren hat (Ausgaben). Auf gleiche Eingaben erwarten wir zumeist auch gleiche Ausgaben. Die Schrittfolge zum Erreichen eines solchen Ziels nennt man auch Algorithmus.
Der Programmierer muß sich beim Programmieren immer darüber im Klaren sein, dass ein System alle Schritte irgendwie gesagt bekommen muß (zumindestens bei imperativen Programmiersprachen (z.B. BASIC, PASCAL)). Jede Aufforderung und jeder Verarbeitungsschritt muß - genau so wie die Ausgabe - exakt spezifiziert werden. Je nach verwendeter Programmiersprache übernimmt diese bestimmte Teilaufgaben ev. (halb-)automatisch.
Logische Programmiersprachen (z.B. PROLOG) benötigen nur ein Regelwerk und eine passende Datenbasis. Den konkreten Lösungsweg ermitteln sie dann eigenständig.
Die in MuPAD integrierte Programmiersprache ist an den interaktiven Kommandosatz angelehnt und durch Anweisungen zur Eingabe und Ausgabe erweitert. Zusätzlich beinhaltet die Programmiersprache noch Elemente zur Steuerung des Programmablaufs (Schleifen (Wiederholungen), Bedingungen (Verzweigungen)).
Die Programmierung in MuPAD erfolgt imperativ. D.h. bei der Programmierung müssen wir Schritt für Schritt den Algorithmus umsetzen und auch möglichst alle Eventualitäten (Sonderfälle, ...) mit berücksichtigen. Alle Programme sind Prozeduren bzw. beim Aufruf innerhalb eines Notebooks Funktionen. In MuPAD wird nicht so genau zwischen Funktionen und Prozeduren unterschieden. Beide Begriffe werden mehr oder weniger synonym verwendet. Wir nennen solche Programmierung auch prozedural bzw. funktional.

2.10.1. Grundaufbau eines MuPAD-Programms

Jedes MuPAD-Programm muß einem bestimmten Muster entsprechen, damit der MuPAD-Kern es richtig verarbeiten kann.
Das nebenstehende Programm ***nichts*** macht genau das - was sein Name suggeriert – nämlich nichts.
Hinter einem neuen Namen für unsere neue MuPAD-Funktion teilen wir dem System mit, dass jetzt eine Prozedur folgt.

```
• nichts:=proc()
    begin
    end_proc
```

Die abzuarbeitenden Schritte innerhalb der Prozedur (Funktion) werden zwischen ***begin*** und ***end_proc*** notiert. Es gehört zum guten Programmierstil, zusammengehörende, untergeordnete Befehle immer einzurücken.
(Zur Erinnerung: Einen Zeilenwechsel innerhalb einer Eingaberegion ohne Auswertung durch den MuPAD-Kern erreicht man durch " ⇑ " + "ENTER".)
Nach ***end_proc*** beenden wir die Programmeingabe mit "ENTER". Das System teilt uns nun mit, dass eine Funktion / Prozedur ***nichts*** existiert.

```
proc nichts() ... end
```

Alternativ erscheinen Fehlermeldungen, wenn das Programm syntaktisch (von der Schreibweise her) falsch ist.
Nun können wir unser Programm sofort testen. Da es einer Funktion entspricht, müssen wir die Klammern mitschreiben. Die Funktionsparameter bleiben leer.

```
• nichts()
  Nil
```

Das Ergebnis der Programmabarbeitung ist ***Nil*** (sprich: niel). ***Nil*** bedeutet so viel wie "nirgendwo" und meint, dass die Funktion nirgendwohin zeigt. (Nil = engl. Abk.: Not in list = dt.: Nicht in der Liste)

Natürlich kann man eine Prozedur auch ohne Einrückungen und / oder nur in eine Zeile schreiben.

```
auch_nichts:=proc() begin; end_proc;
```
```
proc auch_nichts() ... end
```

Unter dieser Schreibweise leidet aber die Übersichtlichkeit. Gewöhnen Sie sich von Anfang an einen bestimmten Stil an. Spätestens bei größeren Programmen und der Fehlersuche darin, werden Sie meinen Rat verstehen.

2.10.2. Speichern und Aufrufen von Programmen

Wie wir gerade schon gezeigt haben, wird ein Programm (eine Prozedur) über ihren Namen mit den notwendigen Parametern in Klammern aufgerufen.

```
nichts()
```
```
Nil
```

Dieses Vorgehen ist aber mit einem großen Nachteil verbunden. Das Programm muß sich in dem benutzten Notebook befinden. Zum Anderen besteht die Gefahr, dass ein fein ausgetüfteltes Programm versehentlich oder absichtlich etwas verändert wird. Dadurch kann die Funktionsweise völlig vom ehemaligen Ziel abweichen.

Eine Lösung ist das gesonderte Abspeichern eines Notebooks mit dem Programm ohne irgendwelche Ein- und Ausgaben. Der zweite Weg nutzt die Möglichkeit separate MuPAD-Programme zu schreiben und abzuspeichern.

In MuPAD öffnen wir uns dazu über das "Datei"-Menü eine "Neue Programmdatei". Wir befinden uns dann in einer leistungsfähigen Editor-Umgebung mit sogenannten Syntax-Highlighting (farbliche Hervorhebung bestimmter Programm-Elemente).

```
nichts:=proc()
  begin
  end_proc:
```

Im Folgenden gehe ich davon aus, dass unsere geschriebenen Programme in der Datei "*Programmsammlung1.mu*" gespeichert werden.

Sie können auch jeden anderen Editor verwenden. Die Dateien müssen dann als Textdateien mit der Endung **.mu** (Dateityp) abgespeichert werden.

Solche abgespeicherten MuPAD-Programme werden dann vor dem Benutzen im Arbeits-Notebook mit ***read*** eingelesen / geladen / geöffnet. Achten Sie dabei auf die ungewöhnliche Schreibweise der Pfade mit einem normalen Schrägstrich **/** (engl.: Slash) – kein Backslash!

(Praktisch entspricht das einem Einschieben des Programmtextes an der Ladestelle.)

```
read("A:/Programmsammlung1.mu"):
```

Nun können wir unsere Prozedur ***nichts*** benutzen, als wenn wir sie in diesem Notebook geschrieben hätten.

Zum Ändern der Prozedur müssten wir in die Programmdatei wechseln.

```
nichts()
```
```
Nil
```

Dadurch ist das Programm zu mindestens vor dem versehentlichen Ändern geschützt.

Geänderte Programmversionen müssen vor der Nutzung wieder neu geladen werden. Ansonsten befindet sich immer noch die zuletzt geladene Version im MuPAD-Speicher.

2.10.3. Ausgaben

2.10.3.1. einfache Ausgaben

Nun wollen wir uns ansehen, wie man innerhalb von Programmen Ausgaben auf dem Bildschirm erzeugt.
Als Beispiel verwenden wir das – in der Welt der Programmierer wohl bekannte – Anfänger-Programm "Hallo Welt". Dieses soll nichts weiter machen, als den Gruß "Hallo Welt" auf dem Bildschirm zu erzeugen.

Wir definieren uns also eine Prozedur mit einem treffenden Namen (hier: ***gruss***) und dem eigentlichen Inhalt – einem ***print***-Befehl.

```
gruss:=proc()
  begin
    print("Hallo Welt!");
  end_proc:
```

Die grüne Hinterlegung (Markierung) soll die Besonderheiten oder Änderungen in den Programmen besonders deutlich machen.

Der Aufruf der Funktion (Prozedur) ***gruss*** bewirkt die gewünschte Ausgabe im aktuellen Notebook.

```
• gruss()
                "Hallo Welt!"
```

Wen die Anführungsstriche stören, der kann diese durch eine Variante (Option) des ***print***-Befehls verschwinden lassen.

```
gruss2:=proc()
  begin
    print(Unquoted,"Hallo Welt!");
  end_proc:
```

Der Aufruf der geänderten Funktion ***gruss2*** bringt die gewünschte Ausgabe.

```
• gruss2()
                Hallo Welt!
```

2.10.3.2. Ausgabe von Funktionswerten (Rückgabewerte von Funktionen und Prozeduren)

Die meisten üblichen MuPAD-Funktionen liefern ein (oder mehrere) Ergebnis(e) zurück. So etwas nennt man Rückgabewert(e) der Funktion.

Solche Rückgabewerte werden mit der Anweisung ***return*** zugeordnet. In unserem Fall soll die Prozedur gruss3 den "Hallo Welt!"-Gruß als Rückgabewert übergeben.

```
gruss3:=proc()
  begin
    return("Hallo Welt!");
  end_proc:
```

Das klappt auch unkompliziert.
Nun können wir auch mit dem ***%***-Operator auf das Ergebnis unserer Funktion ***gruss3*** zurückgreifen.
Das funktioniert bei den anderen ***gruss***-Prozeduren nicht, weil diese keinen Rückgabewert haben. Sie erzeugten die Ausgabe über eine interne ***print***-Anweisung.

```
• gruss3()
    "Hallo Welt!"
• %
    "Hallo Welt!"
• %1
    "Hallo Welt!"
```

2.10.3.3. formatierte Ausgaben

Durch Aneinanderreihung mit Komma-Trennung oder über differenzierte ***print***-Anweisungen lassen sich auch Variablenwerte ausgeben bzw. zurückgeben.

Die Definition von zwei prozedurinternen (lokalen) Variablen – hier a und b - ist guter Programmierstil.

```
ausgabe_AB:=proc()
  local a,b;
  begin
    a:=3.456;
    b:=sqrt(sin(a)*cos(c));
    print("Die Variable a hat den Wert: ",a);
    print("Die Variable b hat den Wert: ",b);
    a:=a*2;
    print("a hat nun den Wert: ",a);
    return(a,b);
  end_proc:
```

Damit werden später Rückwirkungen auf das rufende Programm vermieden, denn auch dort könnte zufällig die gleiche Variable gewählt werden.

Der Aufruf der Prozedur ***ausgabe_AB*** bringt auch schon fast ein perfektes Ergebnis.

```
• ausgabe_AB()
        Die Variable a hat den Wert: , 3.456
                                                    1/2
Die Variable b hat den Wert: , (-0.309252924 cos(c))
          a hat nun den Wert: , 6.912
```

$$6.912,\ \sqrt{-0.309252924\cdot\cos(c)}$$

Störend erscheinen jetzt nur noch die Kommata in den ***print***-Ausgaben. Sie lassen sich auf zwei unterschiedliche Weisen vermeiden, die aber jeweils Vor- und Nachteile haben.

Über Zeichenkettenmanipulationen lässt sich die Ausgabe zu einer Zeichenkette zusammenstellen. Dadurch entfallen ungewünschte Zeichen zur Abtrennung der Einzel-Ausgaben. Mit dem Punkt-Operator **.** werden dazu Zeichenketten aneinander gehängt.

```
ausgabe_AB2:=proc()
  local a,b;
  begin
    a:=3.456;
    b:=sqrt(sin(a)*cos(c));
    print(Unquoted,"Die Variable a hat den Wert: ".expr2text(a));
    print(Unquoted,"Die Variable b hat den Wert: ".expr2text(b));
    a:=a*2;
    print(Unquoted,"a hat nun den Wert: ".expr2text(a));
    return(a,b);
  end_proc:
```

Nachteilig ist der höhere Aufwand durch Verwendung von Umwandlungsfunktionen. Alle Zahlen bzw. Formeln müssen mit ***expr2text*** in Zeichenketten umgewandelt werden. Komplizierte Formeln lassen sich damit leider auch nicht mehr wie gewohnt ausgeben.

```
• ausgabe_AB2()
            Die Variable a hat den Wert: 3.456
Die Variable b hat den Wert: (-0.309252924*cos(c))^(1/2)
              a hat nun den Wert: 6.912
```

$$6.912,\ \sqrt{-0.309252924\cdot\cos(c)}$$

2.10.3.3. formatierte Ausgaben (II) – Bildschirm- und Datei-Ausgaben

Etwas einfacher in der Programmierung ist die Verwendung des Befehls ***fprint***. Bei ihm kann man zudem noch angeben, ob man eine Angabe auf dem Bildschirm (0) oder in eine Textdatei (1) wünscht.

```
ausgabe_AB3:=proc()
  local a,b;
  begin
    a:=3.456;
    b:=sqrt(sin(a)*cos(c));
    fprint(Unquoted,0,"Die Variable a hat den Wert: ",a);
    fprint(Unquoted,0,"Die Variable b hat den Wert: ",b);
    a:=a*2;
    fprint(Unquoted,0,"a hat nun den Wert: ",a);
    return(a,b);
  end_proc:
```

Die Darstellung komplizierter Formeln ist ebenfalls nicht mehr in der gewohnten Form möglich. Alle Ausgaben erfolgen einzeilig als ASCII-Zeichen (sozusagen im Schreibmaschinen-(Zeichen)satz).

```
• ausgabe_AB3()
Die Variable a hat den Wert: 3.456
Die Variable b hat den Wert: (-0.309252924*cos(c))^(1/2)
a hat nun den Wert: 6.912
```

$$6.912,\ \sqrt{-0.309252924\cdot\cos(c)}$$

2.10.4. Eingaben

Bevor wir uns in die Programmierarbeit stürzen, sollen die drei – prinzipiell unterschiedlichen – Eingabe-Varianten vorgestellt werden. Die erste Variante entspricht dem Gegenstück eines ***print***-Befehls. Am Bildschirm soll der Nutzer irgendwelche Angaben machen, die dann im weiteren Verlauf des Programms verarbeitet werden. Solche Eingaben nennen wir interaktiv. Es erscheint auf dem Bildschirm eine Eingabeaufforderung (Prompt) (bzw. ein Eingabe-Fenster (unter WINDOWS)) und der Nutzer muß darauf reagieren.
Viel häufiger treten aber direkte Eingaben (eigentlich Übergaben) für die einzelnen Prozeduren bzw. Funktionen auf. Aus diesen Eingaben berechnet die Prozedur dann den Funktionswert und gibt ihn dann zurück. Diese Eingaben werden in den Klammern hinter dem Funktionsnamen übergeben. Wir nennen sie allgemein Funktions- bzw. Prozedur-Parameter. Die Übergabe und Übernahme von Parametern ist ein etwas komplizierterer Vorgang, den wir entsprechend umfangreich erläutern wollen.
Als letztes bleibt die Möglichkeit Eingaben über Dateien zu realisieren. Dies wird zumeist zur Eingabe von Daten – z.B. aus anderen Programmen – genutzt.

2.10.4.1. interaktive Eingaben

Die einfachste Form (für den Programmierer) ist der einfache ***input***-Befehl. Als einziger und notwendiger Parameter muß eine Variable angegeben werden, auf welche die Eingabe gespeichert werden soll.

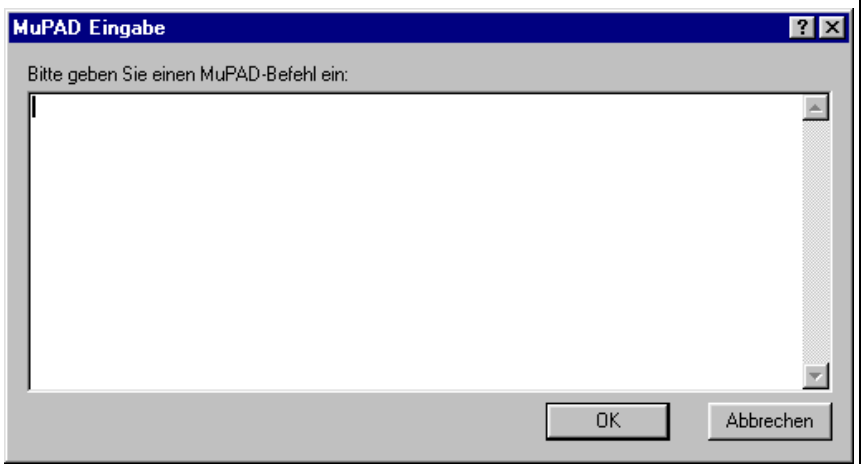
```
eingabe:=proc()
  local x;
  begin
    x:=input();
    fprint(Unquoted,0,"x hat in der Prozedur den Wert: ",x);
  end_proc:
```

Beachten Sie bitte unbedingt den veränderten Syntax der ***input***-Anweisung in WINDOWS-Systemen. Hier muß – entgegen aller Hilfen, Tutorien etc. – die Variable nicht in Klammern folgen, sondern vor der ***input***-Anweisung mit dem Ergibt-Symbol (**:=**) stehen!

Anweisung lt. Hilfen u. Tutorium	**Anweisung in WINDOWS** **(MuPAD pro V. 2.5)**	
`input(x);`	`x:=input();`	

Der Aufruf der ***eingabe***-Prozedur verläuft völlig gewohnt. An der Stelle – wo die Eingabe (***input***) erfolgen soll, erzeugt MuPAD ein Eingabe-Fenster.
In dieses können wir nun jeden beliebigen MuPAD-Ausdruck hinein schreiben. Der wird nach "OK" der Variable ***x*** (ungeprüft) zugeordnet.

```
• eingabe()
```



Unsere interne Ausgabe zeigt dann auch den eingegebenen Wert.

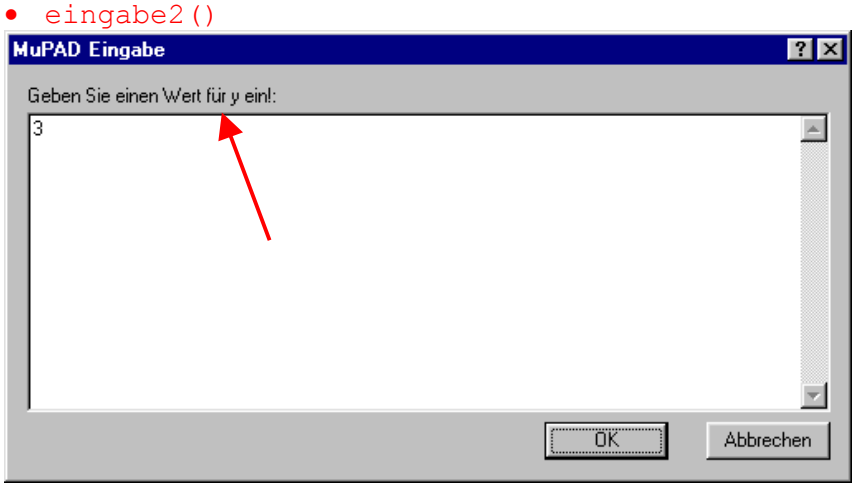
```
• eingabe()
    x hat in der Prozedur den Wert: 3
```

Die Eingabe lässt sich dahingehend verfeinern, dass dem Nutzer mitgeteilt werden kann, was er jetzt eingeben soll. Dies gehört zu einem vernünftigen Programmierstil. Selbst die Programmierer wissen nach Monaten meist nicht mehr, was in welcher Reihenfolge eingegeben werden musste.

```
eingabe2:=proc()
  local y;
  begin
    y:=input("Geben Sie einen Wert für y ein!: ");
    fprint(Unquoted,0,"y hat in der Prozedur den Wert: ",y);
  end_proc:
```

Das erscheinende Eingabe-Fenster trägt nun die gewünschte Überschrift. Dem Nutzer wird bewusst, er muß jetzt etwas eingeben, was beim Programm der Variable y zugeordnet wird.

```
• eingabe2()
```




```
    y hat in der Prozedur den Wert: 3
```

Die Variablenbezeichnung im Programm muß nicht mit der Ausgabebezeichnung für den Nutzer übereinstimmen. Eine sinnige Eingabe für einen Mittelwert könnte dann z.B. so aussehen:

```
mw:=input("Geben Sie den Mittelwert ein!: ");
```

Natürlich können die Eingaben auch an das rufende Programm über ***return*** zurückgegeben werden. Normalerweise werden die Eingaben aber in der Prozedur selbst verarbeitet. Dazu mehr unter → 2.9.5. Verarbeitung .

2.10.4.2. Parameter für Funktionen und Prozeduren

Die meisten Prozeduren / Funktionen sollen einen oder mehrere Werte verarbeiten. Diese Werte werden der Funktion in den Klammern nach dem Funktions-Namen mitgeteilt.

Im Beispiel wird der Funktion ***sin*** der Wert $x = \frac{\pi}{4}$ mitgeteilt.

Die Funktion liefert dann den zugehörigen Funktionswert $\frac{\sqrt{2}}{2}$ als Rückgabewert zurück.

- `x:=1/4*PI`

  $\frac{\pi}{4}$
- `sin(x)`

  $\frac{\sqrt{2}}{2}$

Bei geeigneten Funktionen kann die Liste der Parameter auch wesentlich länger sein. Das nächste Beispiel zeigt dies an Hand der Hintereinanderhänge-Funktion ***_concat*** für Zeichenketten.

- `Wort3:="Satz"; Wort2:="ist"; Wort1:="Dies"; Wort4:="ein"`

  "Satz"

  "ist"

  "Dies"

  "ein"
- `LeerZeichen:=" "; SatzZeichen:="."`

  ""

  "."
- `_concat(Wort1,LeerZeichen,Wort2,LeerZeichen,Wort4, LeerZeichen,Wort3,SatzZeichen)`

  "Dies ist ein Satz."

Wenn wir nun selbst Werte in Prozeduren verarbeiten wollen, dann müssen wir die Übergabe und Übernahme dieser Werte organisieren. Hierbei ist sehr gewissenhaftes, umsichtiges und vorausschauendes Arbeiten angebracht.
Hier soll das Ganze soweit vereinfacht werden, dass wir nicht hinter alle ablaufenden Vorgänge schauen wollen. Wir verwenden eine – nicht weiter kommentierte – aber recht sichere Vorgehensweise. Interessierte FREAKS verweisen wir auf entsprechende Literatur (z.B. zur Programmiersprache PASCAL) bzw. einen später erscheinenden Exkurs in diesem Skript.

Als Vorlage benutzen wir eine Funktion mit zwei Parametern (Eingaben, Übergabewerte). Die Funktion soll ***vorlage*** heißen und nur die übernommenen Werte ***p1*** und ***p2*** (Parameter1, Parameter2) wieder zurückgeben. Die Zuweisung auf die lokalen Variablen ***i1*** und ***i2*** ist soll die – sonst übliche Verarbeitung verdeutlichen. Am Schluß werden die "verarbeiteten Werte" zurückgegeben.

```
vorlage:=proc(p1,p2)
  local i1,i2;
  begin
    i1:=p1;
    i2:=p2;

    return(i1,i2);
  end_proc:
```

Nun wollen wir uns den Aufruf und die benutzen Variablen ansehen.
Die Variablen ***a*** und ***b*** werden mit Zahlen belegt. Sie später sollen an die Funktion ***vorlage*** übergeben werden. Alle anderen – im Beispiel benutzten - Variablen bleiben erst einmal leer.

```
• a:=0;b:=3;c;p1;p2;i1;i2
    0
    3
    c
    p1
    p2
    i1
    i2
```

Der Aufruf der Funktion ***vorlage*** ist ganz unspektakulär – sie liefert die erwarteten Werte zurück.

```
• c:=vorlage(a,b)
    0, 3
```

Diese werden noch zusätzlich auf die Variable ***c*** geschrieben. Die zusätzliche Abfrage von ***c*** zeigt ja auch die entsprechende Belegung (s.a. 3. Ausgabezeile).
Interessant ist nun die Belegung der anderen Variablen. Diese scheinen immer noch leer zu sein – so wie vor dem Aufruf von ***vorlage***.
Und so ist es auch. Die Variablen, die in der Funktion ***vorlage*** benutzt werden – sind lokal – also nur innerhalb der Lebens- und Arbeitsdauer der Funktion ***vorlage*** gültig. Die vor dem Funktionsaufruf benutzten Variablen ***p1***, ***p2***, ***i1*** und ***i2*** haben mit den internen Variablen nichts zu tun.

```
• a;b;c;p1;p2;i1;i2
    0
    3
    0, 3
    p1
    p2
    i1
    i2
```

Sie haben nur zufällig den gleichen Namen – was aber nicht von Bedeutung ist.
Wenn man die Prinzipien der Vorlage:

1. *Parameter auf interne, lokale Variablen umlegen,* (muß nicht unbedingt sein, aber man behält die originalen Parameter)
2. *nur mit internen Variablen (oder den Übergabe-Parametern) weiterarbeiten,*
3. *berechnete Werte über die return-Anweisung zurückgeben,*

beachtet, dann läuft man nicht Gefahr mit irrtümlich oder zufällig geänderten Variablen weiterzuarbeiten. Solche Problem nennt man Seiteneffekte und entstehen zumeist durch gleichnamige (- nicht lokalisierte -) Variablen.

Aufgaben:

1. *Ändern Sie die Prozedur* vorlage *so ab, dass intern eine Anzeige aller - intern und extern verwendeten - Variablen erfolgt! Testen Sie mit verschiedenen externen Variablenwerten!*
2. *Erstellen und testen Sie eine Vorlage für einen und für drei Übergabeparameter. (Die Anzahl der Rückgabewerte soll gleich der Anzahl der Parameter sein!)*
3. *Schreiben und testen Sie eine Funktion* sum3*, die drei Übergabeparameter summiert und als eine Summe zurückgibt!*

2.10.5. Verarbeitung

Die Verarbeitung schließt jedwege Veränderung der Daten (Eingaben) ein. Sie können hier prinzipiell jede MuPAD-Funktion benutzen. Da wir logischerweise nicht noch mal alle Funktionen besprechen wollen, verweisen wir auf die vorherigen Kapitel, das Tutorium, die Hilfe oder jede andere Literatur.
Hier wollen wir uns um solche Verarbeitungshilfen kümmern, die z.B. dazu dienen, um Spezialfälle usw. zu behandeln. Auch die Organisation von automatisierten Wiederholungen wollen wir darlegen.

2.10.5.1. Verzweigungen / Tests / Entscheidungen / Bedingte Programmverläufe

Im Verlauf wohl jeder größeren Prozedur tritt das Problem auf, dass wir den einen oder anderen Sonderfall beachten müssen. Oder es sind spezielle Ausgaben (Hinweise) erwünscht, wenn bestimmte Funktionswerte auftauchen.
Man führt also zu bestimmten Zeiten einen Test durch, ob der eine oder andere Werte diesen oder jenen Wert hat. Je nach eingetretenem Fall soll das Programm dann anders reagieren. So etwas nennt man Verzweigung.
Verzweigungen bestehen in MuPAD immer aus der einleitenden Anweisung ***if***. Nun folgt eine oder mehrere Bedingungen - der eigentliche Test. Der Test muß für MuPAD entweder wahr oder falsch ausfallen. Ist der Test wahr, wird hinter der ***then***–Anweisung weitergemacht. Für den alternativen Fall (Test also fasch) wird hinter der ***else***–Anweisung fortgesetzt. Der vorherige ***then***–Teil wir übersprungen. Eine Verzweigungsstruktur endet mit der ***end_if***–Anweisung. Der ***else***-Teil kann auch entfallen. Geht in so einem Fall der Test negativ (falsch) aus, dann wird hinter der ***end_if***-Anweisung mit dem "normalen" Programm weitergemacht.
(Es gibt auch Mehrfachverzweigungen über ***elif***-Anweisungen. Dazu mehr weiter hinten (→ 2.10.5.1.1. Mehrfachverzweigungen) und in der Hilfe oder im Tutorium.)
Für ein erstes Beispiel soll zu einer Zahl angegeben werden, ob sie gerade oder ungerade ist. Wir verwenden die Division mit Rest (Modulo) für diesen Zweck.

```
test_gerade_ungerade:=proc(zahl)
  begin
    if zahl mod 2=0
      then
        fprint(Unquoted,0,"Die Zahl ",zahl," ist gerade.");
      else
        fprint(Unquoted,0,"Die Zahl ",zahl," ist ungerade.");
    end_if;
  end_proc:
```

Der Test in einem MuPAD-Notebook läuft wie erwartet. Der Funktion kann der Parameter direkt oder indirekt über eine Variable übergeben werden. Selbst Rechnungen usw. lassen sich einsetzen. Wichtig ist nur, dass der Parameter eine auswertbare Zahl ist.

```
• test_gerade_ungerade(2)
Die Zahl 2 ist gerade.
• zz:=-3
                    -3
• test_gerade_ungerade(zz)
Die Zahl -3 ist ungerade.
```

Bei der Eingabe einer unpassenden Zahl gibt es eine folgerichtige – aber für den Nicht-Programmierer – sprich Nur-Nutzer der Funktion – unverständliche Fehlermeldung. Woher soll er wissen, dass wir die Modulo-Operation verwendet haben.

```
• test_gerade_ungerade(2.5)
Error: Illegal operand [_mod];
during evaluation of
'test_gerade_ungerade'
```

Um solche Probleme sauber abzufangen und passende Fehlerausgaben zu erzeugen, braucht es mehr als eine Verzweigungsstruktur.
Innerhalb des ***then***- bzw. ***else***-Teils dürfen beliebig viele Anweisungen stehen. Natürlich können dies auch wieder Verzweigungen sein. Verzweigungen müssen aber immer sauber geschachtelt sein.

```
if … then
  …
  if … then
  end_if;
  …
  if … then
    …
  else
    …
  end_if;
  …
else
  if … then
    if … then
      …
    end_if;
  else
    …
  end_if;
end_if;
```

In den nebenstehenden Grobgerüsten sind zusammengehörende Verzweigungs-Anweisungen gleichartig eingefärbt.
Die Punkte kennzeichnen die Bereiche, in denen die Arbeitsbefehle für den entsprechenden Zweig stehen.
Wir verwenden zwei verschiedene, aber häufig verwendete Einrück-Muster. Welches Sie verwenden, überlassen wir Ihrem persönlichen Geschmack. Beide Grobgerüste sind semantisch und syntaktisch gleichwertig.
Das Linke ist kompakter. Dafür lässt sich im rechten Schema die Struktur sehr genau erkennen. In MuPAD ist diese Schreibweise nicht unbedingt notwendig, da es entgegen anderen Sprachen (z.B.: PASCAL, C, ...) eindeutige ***end***-Befehle gibt.

```
if …
  then
    …
    if …
      then
    end_if;
    …
    if …
      then
        …
      else
        …
    end_if;
    …
  else
    if …
      then
        if …
          then
            …
          end_if;
      else
        …
    end_if;
  end_if;
```

Soll nun unsere ***test_gerade_ungerade***-Funktion eventuelle Problemfälle abfangen, dann bedarf es der vorherigen Prüfung der übergebenen Zahl.

```
test_gerade_ungerade2:=proc(zahl)
  begin
    if domtype(zahl)=DOM_INT
      then
        if zahl mod 2=0
          then
            fprint(Unquoted,0,"Die Zahl ",zahl," ist gerade.");
          else
            fprint(Unquoted,0,"Die Zahl ",zahl," ist ungerade.");
        end_if;
      else
        fprint(Unquoted,0,zahl," ist keine Ganze Zahl.");
      end_if;
  end_proc:
```

Nun fällt auch der Test unserer erweiterten Funktion test_gerade_ungerade2 wie erwartet aus.
Wie Sie gesehen haben, ist es sehr wichtig, sein Programmierergebnis sehr gründlich zu testen.
Verwenden Sie die Testwerte aus dem Bereich, den Sie später nutzen wollen. Weiterhin sollten Extreme und ungewöhnliche Werte nicht fehlen, damit man die Grenzen der Gültigkeit für die Funktion ermitteln kann.

```
• test_gerade_ungerade2(2)
Die Zahl 2 ist gerade.
• zz:=-3
    -3
• test_gerade_ungerade2(zz)
Die Zahl -3 ist ungerade.
• test_gerade_ungerade2(2.5)
2.5 ist keine Ganze Zahl.
• test_gerade_ungerade2(w)
w ist keine Ganze Zahl.
```

Weiterhin sollte man sich eine bestimmte Form der Dokumentation (und auch der Kommentierung) der Programme zu eigen werden lassen. Nach ein paar Monaten oder vielen neuen Programmen kann man sich meist nicht mehr an alle Einzelheiten erinnern. Da sind kleine Hilfen mehr als erwünscht. Auch eine spätere Korrektur oder Ergänzung der Programme ist dann leichter möglich.
In der nachfolgenden Version von ***test_gerade_ungerade2*** wurden diverse Kommentare in die Befehlszeilen eingearbeitet. Als einfache Dokumentation dienen die Kommentarzeilen unter der Funktionsdefinition.

```
test_gerade_ungerade2:=proc(zahl)
// Funktion testet über die Division mit Rest (Modulo), ob der
// Übergabe-Parameter zahl eine gerade bzw. ungerade Zahl ist
// zahl wir vorher auf Zugehörigkeit zum Zahlenbereich der Ganzen
// Zahlen überprüft
// programmiert: L. Drews    am: 01.05.2004
// getestet: L. Drews    am: 01.05.2004    Status: i.O.
  begin
    if domtype(zahl)=DOM_INT  //Test, ob Ganze Zahl
      then  //ist Ganze Zahl
        if zahl mod 2=0  //Test, ob gerade Zahl über Division
                         //mit Rest
          then  //ist gerade Zahl
            fprint(Unquoted,0,"Die Zahl ",zahl," ist gerade.");
          else  //keine gerade Zahl --> ungerade Zahl
            fprint(Unquoted,0,"Die Zahl ",zahl," ist ungerade.");
        end_if;
      else  //keine Ganze Zahl --> Fehleranzeige
        fprint(Unquoted,0,zahl," ist keine Ganze Zahl.");
      end_if;
  end_proc:
```

Ganz so ausführlich muß eine Kommentierung nicht erfolgen. Aber die ungewöhnlichen oder komplizierten Stellen sollte man in keinem Fall unkommentiert lassen.

Im nächsten Bespiel wollen wir eine Rabatt-Funktion schreiben. Diese soll abhängig vom (Waren-)Wert die nebenstehenden Rabatt-Stufen verwirklichen und den Rabatt zurückliefern.

Stufe	Warenwert	Rabatt in %
1	>= 100; <500	1
2	>= 500; <1000	2
3	>= 1000; <5000	3
4	>= 5000; <10000	5
5	>=10000	10

2.10.5.1.1. Mehrfachverzweigungen

Prinzipiell lässt sich die Aufgabe mit den oben besprochenen ***if-then-(else-)end_if***-Strukturen lösen. Schöner geht es mit einer Variante der Verzweigung – der Mehrfachauswahl. Programmierer kennen z.B. die ***case***-Anweisung in PASCAL.
In MuPAD übernimmt die Mehrfachauswahl die Hilfsanweisung ***elif*** innerhalb von ***if-then-(else-)end_if***-Strukturen.

```
rabatt:=proc(wert)
  local rabattwert;
  begin
    if wert<100  //unter 100 gibt es kein Rabatt
      then rabattwert:=0;  //kein Rabatt
      elif wert<500  //1. Rabattstufe (obere Grenze)
        then rabattwert:=wert*0.01;  // 1%
      elif wert<1000  //2. Rabattstufe (obere Grenze)
        then rabattwert:=wert*0.02;  // 2%
      elif wert<5000  //3. Rabattstufe (obere Grenze)
        then rabattwert:=wert*0.03;  // 3%
      elif wert<10000  //4. Rabattstufe (obere Grenze)
        then rabattwert:=wert*0.05;  // 5%
      //>=10000
      else rabattwert:=wert*0.1; // 10% Rabatt
    end_if;
    return(rabattwert);
  end_proc:
```

Der Test mit verschiedenen Zahlen im Gültigkeitsbereich ergibt die richtigen Funktionswerte.

- `rabatt(2)`

  0
- `rabatt(100400)`

  10040.0
- `rabatt(300)`

  3.0

Aufgaben:

1. Prüfen Sie die folgenden Verzweigungen auf ihre syntaktische und semantische Korrektheit!

`if x>100`
`  then x:=x-100;`
`  else x:=2*x;`
	`if x<100 then if x<10 then x:=x+10 else x:=  end_if; else x:=x-100; end_if; if (x>=10) and (x<=100) then ergebnis:="i.O." end_if;`

`if farbe="rot"`
`  then Text:="Warten!";`
`if farbe="gelb";`
`  then Text:="Achtung!";`
`if Farbe:="gruen"`
`  else Text:="Fahren!";`
	`if x>100`
	`  then x:=x-100;`
	`  else`
	`if x<10`
	`  then x:=x+100; end_if;`
	`end_if;`

2. Erstellen Sie ein Programm, dass prüft, ob eine Zahl durch eine andere geteilt werden kann!
3. Was passiert, wenn die rabatt-Funktion einen negativen Wert übergeben bekommt! Prüfen Sie den Durchgang durch die Funktion zuerst theoretisch und prüfen Sie dann in einem MuPAD-Notebook ihre Voraussagen!
4. Schreiben Sie das rabatt-Programm nur mit if-then-else-end_if-Strukturen! (elif ist nicht zugelassen)

Aufgabe für FREAKS:

Schreiben Sie das Programm zur Teilbarkeit so um, dass intern eine Anzeige des Quotienten erfolgt, wenn dies in einem dritten Parameter so gewünscht wird!

2.10.5.2. Schleifen / Wiederholungen

In der Praxis kommen viele Sachverhalte vor, bei denen bestimmte Arbeitsschritte mehrfach wiederholt werden müssen. Oft macht man deshalb bestimmte (kleine) Wiederholungen, damit man ein (großes) Problem nicht mit einmal lösen muß. Ein gutes Beispiel ist eine Treppe. Da wir nur mit größerer Mühe von einer Etage in die nächste klettern könnten, haben clevere Vorfahren die Treppe erfunden. Mit kleinen – gut zu kontrollierenden Schritten – schaffen wir letztendlich den großen Höhenunterschied.
Man unterscheidet drei Arten von Wiederholungen:

1. **Schleifen mit feststehender Anzahl von Durchläufen und konstanten Schrittgrößen *(Zählschleife)*** *(u.U. dürfen diese Schleifen vorher verlassen werden)*
   Beispiele:
   - in einer Lotto-Simulation sollen sechs plus eine Zahl (- also sieben Zahlen -) aus 49 gezogen werden (ohne Dopplung)
   - genau vier Klausurnoten sollen zu einer Gesamtnote zusammengefasst werden (50%-Anteil für Jahresnote)
   - ...

2. **Schleifen, bei denen nach jedem Durchlauf geprüft wird, ob noch (mindestens) eine weitere Wiederholung notwendig / möglich ist *(Schleife mit nachlaufender Prüfung; fußbedingte Schleife)***
   Beispiele:
   - beim Eröffnungswürfeln für "Mensch-Ärgere-Dich-Nicht": War es eine Sechs oder wurde schon 3x gewürfelt?
   - ...

3. **Schleifen, bei denen vor einem Durchlauf geprüft wird, ob noch (mindestens) eine Wiederholung notwendig / möglich ist *(Schleife mit vorlaufender Prüfung; kopfbedingte Schleife)***
   Beispiele:
   - Kontrolle von Eingaben und gegebenenfalls Wiederholung der Eingabe
   - ...

2. und 3. gehören zur gemeinsamen Gruppe der bedingten Schleifen. Schleifen bestehen allgemein aus einem Schleifenkopf (Kontrollbereich) und dem Schleifenkörper (Arbeitsbereich).
Für jede Art Schleife gibt es eigenständige – z.T. aber ähnliche – Programmierbefehle und –strukturen. Grundsätzlich gilt, dass Schleifen ineinander geschachtelt werden dürfen. Sie dürfen sich aber nicht überlappen. Jede Art Schleife läßt sich in der Praxis durch jede andere ersetzen. Oft steigt dann aber der Programmieraufwand um zusätzliche Tests usw.
Ebenfalls als Schleife könnte man die Rekursion einordnen. Rekursionen sind Programmstrukturen, die sich wieder selbst aufrufen bis ein bestimmtes Ziel erreicht ist. Man versucht dabei ein großes Problem auf ein gleichartiges - aber kleineres – Problem zu reduzieren. Graphisches Äquivalent ist aber eher eine Spirale als eine Schleife. Da Rekursionen in der Programmierung (besonders bei mathematischen Problemen) eine große Rolle spielen, besprechen wir sie am Ende dieses Abschnittes.

2.10.5.2.1. Schleifen mit feststehender Anzahl der Wiederholungen

Greifen wir das Beispiel der Notenberechnung auf. Laut gesetzlichen Bestimmungen sollen in einem bestimmten Kurs genau vier Klausuren geschrieben werden. Diese werden zu einer Note zusammengefasst und ergeben eine Hälfte der Jahresnote. Die zu schreibende Funktion soll diese Noten in einem Feld (Tabelle, Vektor, Array) übergeben bekommen und den Durchschnitt sowie die Gesamtnote (gerundet) zurückliefern.

Der einfachste Algorithmus ist wohl die Addition der Einzelnoten und dann die Mittelwertbildung über die Division durch die Anzahl. Die Gesamtnote wird dann einfach durch Rundung erzeugt.

MuPAD bietet für feststehende Schleifen den Konstrukt ***for-from-to-do-end_for*** an.

Hinter ***for*** wird eine sogenannte Laufvariable erwartet. Diese wird innerhalb der Schleifendurchläufe automatisch erhöht und vor dem nachfolgenden Durchlauf geprüft, ob schon der letzte Durchlauf / das Ende der Schleife erreicht ist. (also ist die ***for***-Schleife eigentlich auch nur eine Schleife mit nachlaufender Prüfung)

Typischerweise verwendet man passende Namen oder i, j, k, m, ... für die Laufvariablen. Manche Programmierer bevorzugen Doppelbuchstaben (ii, jj, kk, mm, ...), damit Verwechslungen mit anderen Variablen möglichst ausgeschlossen werden.

Mit ***from*** und ***to*** werden die Grenzen für die Laufvariable (untere und obere) festgelegt. Nach ***do*** folgen dann die Anweisungen, die jeweils wiederholt werden sollen.

Wird einmal eine Laufvariable benötigt, die von einem größeren Wert zu einem kleineren laufen soll, dann läßt sich dies durch Einbau von ***downto*** statt ***to*** realisieren.

Ein weiterer Sonderfall ist eine Schleife, in der nur bestimmte Elemente einer Menge benutzt werden soll, deren Elemente vielleicht keinem funktionellem Zusammenhang genügen. Hier verwendet man das Konstrukt ***for-in-do-end_for*** . Hinter ***in*** folgt eine Menge oder Liste mit Elementen für die Schleifendurchläufe erledigt werden sollen.

Am Schluß muß immer ***end_for*** als Schleifenende angehängt werden.

```
klausur_Gesamtnote:=proc(klausuren)
  local summe,notenNr,durchschnitt,gesamtnote;
  begin
    summe:=0;
    for notenNr from 1 to 4 do
      summe:=summe+klausuren[notenNr];
    end_for;
    durchschnitt:=summe/4;
    gesamtnote:=round(durchschnitt);
    return(float(durchschnitt),gesamtnote);
  end_proc:
```

Der Test unserer Funktion liefert mathematisch exakte Ergebnisse. Egal, ob wir das Notenfeld über eine Variable oder direkt übergeben.

- `Klausuren:=[3,4,2,2]`

  $[3, 4, 2, 2]$

- `klausur_Gesamtnote(Klausuren)`

  $2.75, 3$

Unter pädagogischen Gesichtspunkten ließe sich die Funktion sicher noch verbessern.

- `Klausuren:=[5,5,2,2]`

  [5, 5, 2, 2]
- `klausur_Gesamtnote(Klausuren)`

  3.5, 4
- `klausur_Gesamtnote([2,3,3,1])`

  2.25, 2

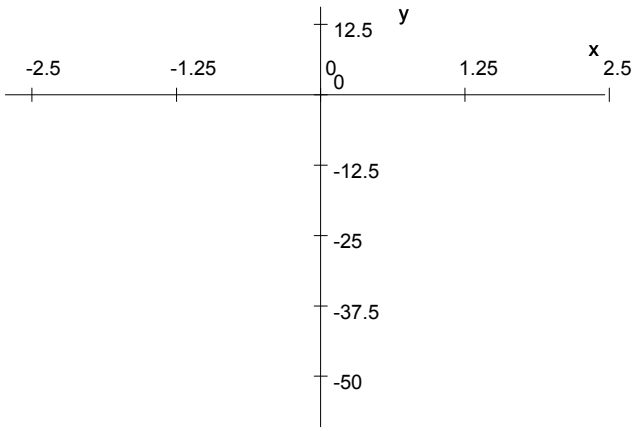
Im nächsten Beispiel wollen wir für eine beliebige – zu übergebende – Funktion eine graphische Darstellung erzeugen. Jeweils 20 Stützpunkte sollen in einem – ebenfalls zu übergebenden - Definitionsbereich (untere und obere (nicht eingeschlossene) Grenze) gezeichnet werden. Hierbei wollen wir auf eine weitere Möglichkeit der Schleifenschreibung eingehen. Mit dem $ lassen sich feste Schleifen deutlich schneller schreiben. Darunter leidet aber die Lesbarkeit.

```
zeichne_Funktion20Pkt:=proc(funktion,untereGrenze,obereGrenze)
  local schritte, schrittweite, x_wert, grafik, zaehler;
  begin
    schritte:=20;   //Stützpunktezahl
    schrittweite:=(obereGrenze-untereGrenze)/(schritte+2);
    grafik:=plot::Pointlist(
                [(untereGrenze+zaehler*schrittweite),  // X
                 (subs(funktion,x=untereGrenze+zaehler
                   *schrittweite))]                    // Y
                 //Schleifen-/Folgen-Anweisung
                 $ zaehler=1..schritte);
    plot(grafik);
  end_proc:
```

Ein erster Test bringt ein positives Ergebnis. Stößt man aber auf nicht definierte Punkte, dann schmeißt uns die Funktion aus der Prozedur und es kommt eine Fehlermeldung. Schöner wäre es, wenn bei undefinierten Punkten der konkrete Punkt einfach ausgelassen würde.

- `f:=2*x^3-3*x^2+4; zeichne_Funktion20Pkt(f,-3,3)`

  $2 \cdot x^3 - 3 \cdot x^2 + 4$



2.10.5.2.2. Schleifen mit vorlaufender Prüfung der Wiederholungsbedingung

Bei diesen Schleifen wird vor einem - auch vor dem ersten – Durchlauf geprüft, ob die Bedingungen erfüllt sind. Ist dies nicht der Fall wird hinter der Schleife fortgesetzt. Der Schleifenkörper wird also nur ausgeführt, wenn die Bedingungen dafür erfüllt sind.

In MuPAD findet sich der ***while-do-end_while***-Befehlskonstrukt für solche Schleifen.

Zwischen ***while*** und ***do*** erwartet MuPAD eine oder mehrere Bedingungen (Tests), die vor dem Schleifendurchlauf geprüft werden soll(en).

Als einfaches Beispiel wählen wir den EUKLIDischen Algorithmus zur Bestimmung des größten gemeinsamen Teilers (ggT) zweier natürlicher Zahlen.

```
ggT:=proc(a,b)
  local rest;
  begin
    while b<>0 do
      rest:=a mod b;
      a:=b;
      b:=rest;
    end_while;
    return(a);
  end_proc:
```

Der Test verläuft wie erwartet. Die Analyse des zurückgelieferten Ergebnisses obliegt aber dem umgebenden Programm bzw. dem Nutzer. Das Ergebnis 1 bedeutet ja schließlich, dass die Zahlen teilerfremd sind.

```
• ggT(24,16)
                    8
• ggT(17,24)
                    1
```

Vorteilhaft sind die ***while***-Schleifen immer dann, wenn mehrere Bedingungen bei Schleifendurchläufe beachtet werden müssen.

2.10.5.2.3. Schleifen mit nachlaufender Prüfung der Wiederholungsbedingung

Im Unterschied zur kopfbedingten Schleife wird bei Schleifen mit nachlaufender Prüfung die Bedingung erst nach dem Schleifendurchlauf geprüft. Nur wenn die Bedingung(en) erfüllt ist / sind, wird mit einem weiteren Durchlauf fortgesetzt. Bei Nichterfüllung geht es gleich hinter der Schleife weiter im Programm. Bei dieser Schleife wird der Schleifenkörper also mindestens einmal durchlaufen!

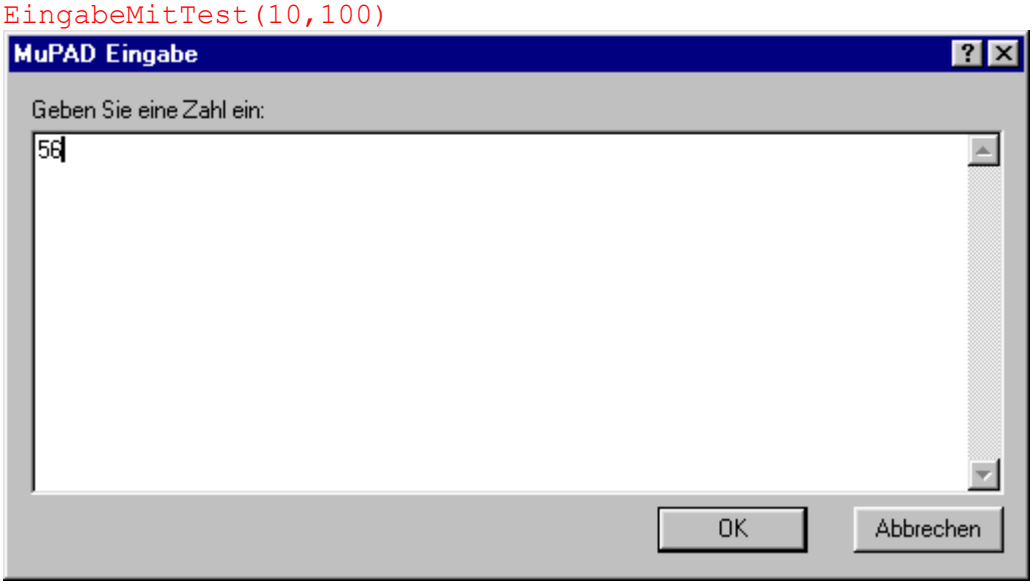
Die zugehörige Struktur ist ***repeat-until-end_repeat***. Die Bedingungen sind zwischen ***until*** und ***end_repeat*** zu platzieren.

Sehr praktisch sind ***repeat***-Schleifen bei der Absicherung von Eingaben. Da man als Programmierer nicht weiss, wie viele Versuche ein beliebiger Nutzer braucht, ist eine Zählschleife ungeeignet. Aber auch den eigentlichen Wert kennt man erst nach der Eingabe. Somit geht auch eine ***while***-Schleife nicht für diesen Zweck.

Im Anwendungsbeispiel soll der Nutzer eine Zahl zwischen einer unteren und einer oberen Grenze (z.B. 10 und 100) eingeben (die Grenzen gehören mit zum Gültigkeitsbereich). Bei falscher Eingabe soll ein erneuter Versuch gestartet werden.

```
EingabeMitTest:=proc(untereGrenze,obereGrenze)
  local Eingabe;
  begin
    repeat
      Eingabe:=input("Geben Sie eine Zahl ein: ");
    until Eingabe>=untereGrenze
          and Eingabe<=obereGrenze end_repeat;
    return(Eingabe);
  end_proc:
```

```
EingabeMitTest(10,100)
```




```
                    56
```

Das Programm verhält sich im Test so penetrant – wie erwartet. Solange keine geeignete Zahl eingegeben wurde, wird das Eingabefenster wiederholt.
Leider konnte die Aufforderung in der ***input***-Anweisung nicht mit den Grenzen versehen werden. Jeder Versuch mit einer zusammengesetzten Zeichenkette endete in der Standardaufforderung.

2.10.5.2.4. Schleifen für FREAKS

Wenn Sie ordentliche Programmierung lernen wollen oder beherrschen, dann überspringen Sie diesen Abschnitt ganz schnell. Nichts ist so verpönt, wie Sprünge (GO-TO's) in Algorithmen. Im Prinzip sind die Anweisungen ***next*** und ***break*** Äquivalente für Sprünge in oder aus Schleifen. Da es auch ohne sie geht, sollte man auf diese Befehle verzichten. Ich empfehle sie nur dann zu benutzen, wenn der Quelltext ohne sie wesentlich komplizierter und unübersichtlicher (schlechter lesbar) wird.
Aber nun zu den Anweisungen. Mit ***next*** erzwingt man einen Abbruch der Anweisungsfolge im aktuellen Schleifendurchlauf. Es wird der nächste Schleifendurchlauf initiert. Dagegen bricht ***break*** die Schleife vollständig ab. Es wird hinter der Schleife mit dem Algorithmus fortgesetzt.

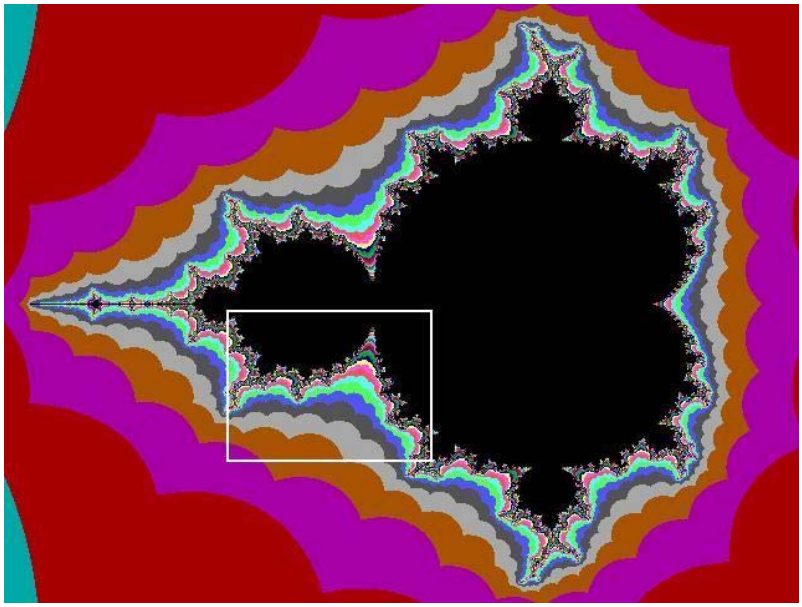
Aufgaben:

. Ändern Sie die Funktion ggT so ab, dass die Zwischenschritte und das Ergebnis in der Funktion angezeigt werden! (Teilerfremdheit soll als Text angezeigt werden!)

2.10.5.2.5. Rekursion

Vielleicht erinnern Sie sich noch an die Besprechung der Interation (→ 1.2.1. Lösen von Gleichungssystemen durch Interation) oder Sie haben schon mal computergenerierte Bilder oder Muster – sogenannte Fraktale – gesehen. Da sind wir direkt im Bereich der Rekursion.

Bei der Interation wurde immer die gleiche Berechnung wiederholt, bis eine der Grenzen (der Unterschied in den Ergebnissen sehr klein oder der Anzahl der Wiederholungen oder ein bestimmter Wert) erreicht war.

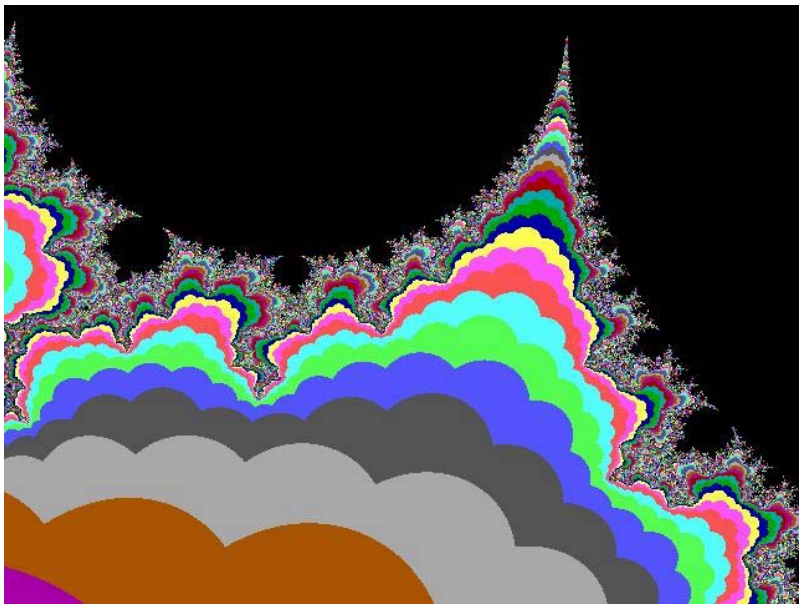


erstellt mit /B/

(der vergrößerte und neu berechnete Bereich für die nächste Abbildung ist eingerahmt)

Bei vielen Fraktalen berechnet man eine zweidimensionale Funktion und wiederholt den Funktionsaufruf mit den gerade berechneten Werten. Am Ende wird dann geprüft, wo sich das Ergebnis hinbewegt. Das Ergebnis oder die Tendenz werden dann graphisch dargestellt und bilden die berühmten Muster (z.B.: JULIA-Menge, MANDELBROT-Apfelbäumchen, ...)

Besonders interessant ist die Selbstähnlichkeit der Fraktale.



An den Grenzbereichen kann man die Wertebereiche immer weiter verkleinern und erhält immer wieder noch kleinere und feinere Musterwiederholungen. (siehe Abbildungen).

In den meisten Schriften zur rekursiven Programmierung wird das Beispiel Fakultät gestresst. Dies lassen wir uns für die Übungsaufgaben.

Wir verwenden als Beispiel die FIBONACCI-Zahlenreihe. Der Legende nach soll FIBONACCHI (Leonardo von Pisa od. auch Filius BONACCI, Ende 12. – Anfang 13. Jhd.) nach einer Zahlenreihe gesucht haben, welche die Populationsentwicklung von einer sich selbst überlassenen Kaninchen-Population beschreibt. (Das Modell wurde von FIBONACCI im Buch "Liber Abaci" (1202) vorgestellt.).

Das Modell läßt sich wie folgt beschreiben:

- gestartet wird mit einem Paar
- jedes Paar ist nach 2 Monaten (Zeittakten, Rechentakten) fortpflanzungsfähig
- jedes Paar bringt von da an jeden Monat (Zeittakt, Rechentakt) ein neues Paar in die Population ein
- alle Paare leben unendlich

Das erste Glied der FIBONACCI-Reihe ist die 0 – also F(1) = 0. Das zweite Glied ist ebenfalls definiert mit F(2) = 1. Alle nachfolgenden Glieder ergeben sich aus der Summe der letzten beiden Vorgänger. Also ist F(3) = F(1) + F(2) = 0 + 1 = 1.
Durch Weiterrechnen erhält man folgende Reihe:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

Praktisch findet man die entstehende Folge bei der Teilung der Nautilus-Muschel, der Schuppenstellung am Kiefern-Zapfen bis hin zur Kreuzschraffur der Sonnenblume /15/. Heute sind sogar Anwendungen in der Analyse von Aktienkursen bekannt.
Jedes neue Glied errechnet sich also: F(n) = F (n-1) + F(n-2)
Wir haben also eine Funktion, zu deren Berechnung der Aufruf von sich selbst notwendig ist. In unserem Beispiel ist dies sogar zweifach notwendig. Hier wird schon sichtbar, das die Rekursion eine große Gefahr birgt – die Funktion könnte sich immer wieder und immer wieder aufrufen. Es muß also eine Grenzpunkt her. Dies könnte z.B. die Anzahl der Wiederholungen sein. Oder man verwendet einen Vergleich zwischen den letzen berechneten Ergebnissen. Ist der Unterschied nicht mehr relevant (- also unbedeutend oder vernachlässigbar), dann wird der Eigenaufruf abgebrochen. In unserem Fall ist der Abbruch der nächstkleineren ***FIBONACCI***-Funktion dann gegeben, wenn die vordefinierten Glieder 1 und / oder 2 erreicht werden.
Die rekursiv programmierte Funktion ***FIBONACCI*** im folgenden Bildschirmausdruck ist so zu verstehen.
Zuerst wird geprüft, ob wir eine der zwei vordefinierten Reihenglieder geliefert werden soll, wenn ja, dann wird sie zurückgegeben und fertig. Wenn aber ein größeres Glied gesucht wird, dann wird die Bildungsformel einer FIBONACCI-Zahl als Summe der beiden Vorgänger der Reihe angewendet. Diese werden durch Aufruf genau unserer Funktion ***FIBONACCI*** ermittelt. Dies kann sich bei höheren Gliedern also sehr oft wiederholen. Der Rechenzeit-Bedarf von MuPAD belegt dies dann auch.
Somit entsteht die Funktions-Struktur:
FIBONACCI(1) → 0
FIBONACCI(2) → 1
FIBONACCI(n) → FIBONACCI(n-1) + FIBONACCI(n-2) ; für ***n= 3, 4, 5, …***

```
FIBONACCI:=proc(ElementNummer)
  local FibonacciZahl;
  begin
    if ElementNummer=1
      then FibonacciZahl:=0;
      else
        if ElementNummer=2
          then FibonacciZahl:=1;
          else FibonacciZahl:=FIBONACCI(ElementNummer-1)
                              +FIBONACCI(ElementNummer-2);
     end_if;
    end_if;
    return(FibonacciZahl);
  end_proc:
```

Beim Ausprobieren unserer Funktion werden wir mit recht schnellen Berechnungen belohnt.

```
FIBONACCI(1);FIBONACCI(2)
```

```
0

1
```

Auch höhere Glieder werden exakt berechnet – brauchen aber auch deutlich mehr Zeit.

```
FIBONACCI(4);FIBONACCI(7)
```

```
2

8
```

In MuPAD ist eine Funktion für die FIBONACCI-Zahlen schon eingebaut (***numlib::fibonacci***). Wir können diese gut zur Prüfung unserer eigenen Funktion benutzen.
Und erleben eine böse Überraschung. Was bis eben so schön stimmte, hält scheinbar der Prüfung im allmächtigen MuPAD nicht stand.

```
glied:=17; FIBONACCI(glied);
numlib::fibonacci(glied);
```

```
17

987

1597
```

Der Blick in die Hilfe zu ***numlib::fibonacchi*** löst den Wiederspruch auf. In MuPAD ist das erste Glied der Reihe (- was wohl historisch auch richtiger ist -) das Nullte (unserer Reihe). Wir brauchen also nur den Aufruf korrigieren – und schon ist alles in bester Ordnung.

```
glied:=17; FIBONACCI(glied);
numlib::fibonacci(glied-1);
```

```
17

987

987
```

Aufgaben

1. *Sichern Sie die FIBONACCI-Funktion so ab, dass nur positive Zahlen ausgewertet werden!*
2. *Erstellen Sie eine rekursive Funktion* Fakultaet(n) *und prüfen Sie deren Berechnungen mit der systemeigenen Fakultät (***fact(***n***)****)!*
3. *Schreiben Sie eine interative Funktion* Fakultaet(n) *und prüfen Sie deren Berechnungen mit der systemeigenen Fakultät (***fact(***n***)****)!*

2.11. Anwendung: Kurvendiskussion

Kurvendiskussion – das ist langes Rechnen und immer die Ungewissheit: *"Hab' ich mich vielleicht schon bei der 1. Ableitung vertan?"* Zur Kontrolle von Rechnungen ist MuPAD unser Wahl-Mittel. Kombiniert mit der Programmierbarkeit besitzen wir ein sehr hilfreiches Werkzeug zur routinemäßigen Prüfung unserer Rechenkünste.
Als Algorithmus orientieren wir uns an der Schrittfolge aus "Summa 3" und "TCP Mathematik GK", wobei auch Informationen, Hilfen und Tips aus anderen Quellen mit eingearbeitet wurden, um einen möglichst universellen Zugang zu erhalten.
Als Beispiel verwenden wir begleitend die Funktion:

$$y = f(x) = x^4 - 3x^2 + 1$$

0 – Definition der Funktion und Vorbereitung des Notebooks

Dieser Schritt ist für die Arbeit mit MuPAD zu empfehlen, um bei der Verwendung der Funktion den Tipp-Aufwand zu reduzieren und die Werte von verwendeten Variablen zu löschen.

- `delete x`
- `f:=x^4-3*x^2+1`

$$x^4 - 3 \cdot x^2 + 1$$

Die folgenden Schritte lassen sich später fast immer ohne Änderungen verwenden, da sie auf die vordefinierte Funktion ***f*** zurückgreifen.

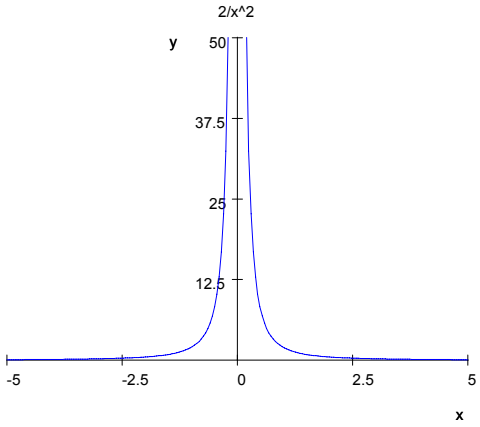
I - Bestimmung des Definitionsbereichs D

In diesem Teilbereich der Kurvendiskussion müssen wir uns noch stark auf unsere Mathematikerfahrungen stützen. Grundsätzlich kann der Definitionsbereich schon durch die Definition der Funktion selbst gegeben sein. Ganzrationale Funktionen – wie unsere Beispielfunktion haben als Definitionsbereich den Zahlenkörper der rationalen Zahlen $\mathbb{R}$.

Unendlichkeitsstelle	Die Funktion f hat an der Stelle x_U eine ***Unendlichkeitsstelle*** $\underset{Def}{=}$ (1) x_U gehört nicht zum Definitionsbereich von f (2) $\frac{1}{f}$ ist in einer (punktierten) Umgebung von x_U definiert und stetig (3) es ist $\lim\limits_{x \dashrightarrow x_U} \frac{1}{f(x)} = 0$ Für die Unendlichkeitsstellen rationaler Funktionen gilt bei der normierten Darstellung $f(x) = \frac{u(x)}{v(x)}$: $v(x) = 0$ und $u(x) \neq 0$. Sie heißen Pole, und die Gerade $x = x_P$ wird Polasymptote genannt

aus /9/

Besonderheiten sind z.B. bei Funktionen zu erwarten, die aus Quotienten bestehen, bei denen der Nenner den Wert 0 annehmen kann. In diesem Fall wäre die Funktion zu mindestens an dieser Stelle nicht definiert. Anders ausgedrückt wächst bzw. fällt der Funktionswert der Funktion ins Unendliche, wenn es eine Nullstelle der Nennerfunktion gibt, die aber nicht auch gleichzeitig Nullstelle der Zählerfunktion ist.



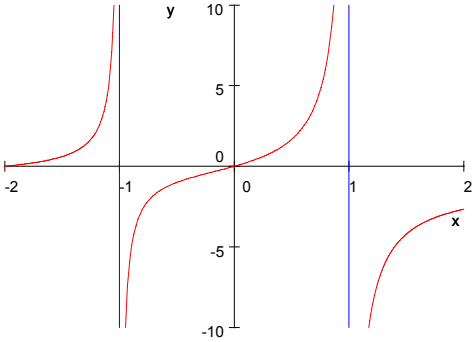
Eine solche Stelle wird Polstelle genannt. Die Funktion nähert sich beidseitig dieser Parallelen zur y-Achse an.
Nähert sich ein Funktionsgraph entweder von einer Seite einer Gerade an oder schneidet er diese dauernd, dann spricht man bei dieser Geraden von einer Asymptote. Man unterscheidet waagerechte, senkrechte (Pol(ar)asymptoten) und schräge Asymptoten. Für gebrochenrationale Funktionen verlaufen die Senkrechten durch Polstellen und die Waagerechten sind jeweils durch die Grenzwerte bestimmt. Schräge Asymptoten entsprechen dem linearen Anteil der durch Polynomdivision zerlegten Funktion.

Einige einfache Beispiele sollen diese Sonderfälle verdeutlichen:

- Pol(ar)asymptoten

  gegeben sei: $f(x) = \frac{x^2 + 2x}{1 - x^2}$

  Bei $x = 1$ und $x = -1$ finden wir Polarasymptoten, da bei $x = 1$ der Nenner 0 ist und damit die Funktion nicht definiert ist (Division durch 0). Zum Anderen ist $x = -1$ sowohl Nenner als auch Zähler gleich 0, so dass wir ein - schon oben besprochenes - Loch vorliegen haben.

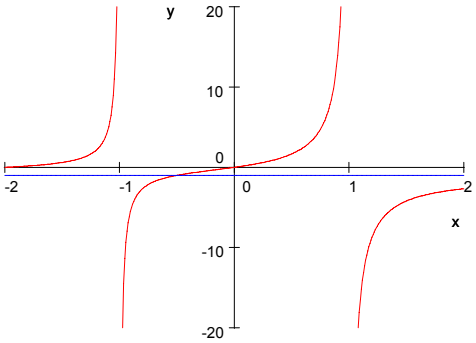


- (waagerechte) Asymptoten

  gegeben sei: $f(x) = \frac{x^2(1 + \frac{2}{x})}{x^2(\frac{1}{x^2} - 1)}$

  Diese Funktion zeigt bei $y = -1$ eine waagerechte Asymptote.

  (natürlich hat diese Funktion auch noch zwei Polarasymptoten bei $x = -1$ und $x = 1$)



- (schräge) Asymptoten

  gegeben sei: $f(x)=\frac{x^3+2x}{x^2+1}$

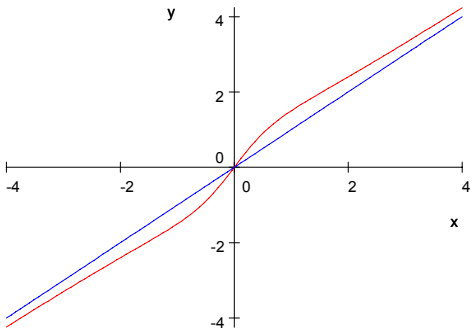
  Führen wir eine Polynomdivision

  $$(x^3+2x):(x^2+1)=x+\frac{x}{x^2+1}$$

  $$\frac{-(x^3+x)}{x}$$

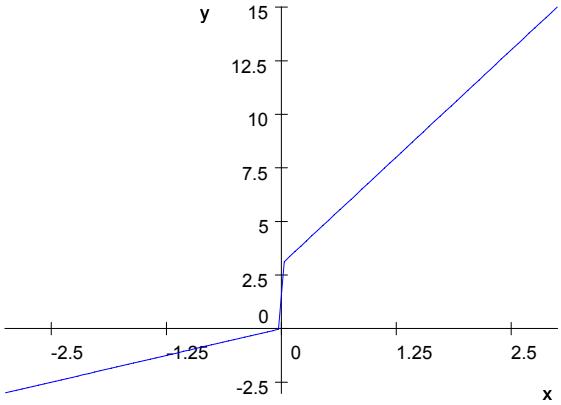
  durch, dann erhalten wir wegen der Vernachlässigung des 2. Gliedes der Summe $(\frac{x}{x^2+1})$ (dieses wird schließlich bei steigendem x immer kleiner) die Beziehung $y=x$

  Die resultierende Asymptote ist also die Gerade $y=x$



Besondere Definitionslücken sind z.B. auch sogenannte hebbare Löcher. Dies sind Stellen in gebrochenrationalen Funktionen, bei denen sowohl Zähler als auch Nenner an der vermeintlichen Stelle den Wert 0 annehmen können.
Besonders bei abschnittsweise definierten Funktionen, wie z.B.:

$$f(x)=\begin{cases} x & \text{für } x<0 \\ 4x+3 & \text{für } x>0 \end{cases}$$



die offensichtlich an der Stelle $x=0$ eine Definitionslücke besitzt. So eine Lücke wird auch als Sprung bezeichnet.
MuPAD wäre natürlich ein schlechtes Mathematik-Programm, wenn es uns nicht auch bei der Suche nach Unstetigkeiten im Funktionsverlauf helfen würde.

Die Funktion ***discont*** (eng.: discontinuities) prüft den Funktionsverlauf.
Unsere Funktion hat keine Unstetigkeitsstellen und ist damit stetig.

- `discont(f,x)`

  $\varnothing$

Anders dagegen z.B. die Funktion:

$f(x)=\frac{x^4-3x^2}{x^2-4}$.

Hier zeigt uns ***discont*** die zu erwartende "Problemstellen" bei $x^2-4=0$ an.

- `a:=(x^4-3*x^2)/(x^2-4)`

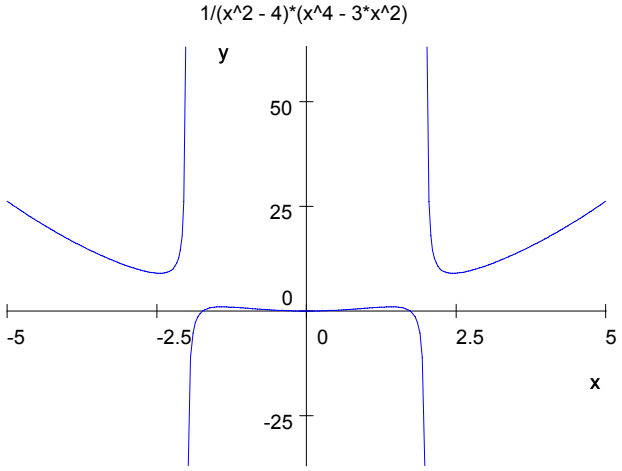
  $$\frac{x^4-3\cdot x^2}{x^2-4}$$

- `discont(a,x)`

  $\{-2, 2\}$

Eine kleine Prüfung an Hand der graphischen Darstellung zeigt dann auch die Polarasymptoten bei -2 und 2.

- `plotfunc2d(a)`



II – Untersuchung der Symmetrieeigenschaften

Für ganzrationale Funktionen können wir schon an dieser Stelle Aussagen wagen, wenn alle Exponenten nur geradzahlig sind. Dann ist der Graph dieser Funktion achsensymmetrisch zur y-Achse (Ordinate, Größenachse). Unser Beispiel erfüllt diese Bedingung und ist somit als y-achsensymmetrisch einzuordnen.

Durch Einsetzen der negierten Variable lässt sich die Achsensymmetrie auch in MuPAD testen. Dies wird durch Substitution mittels ***subs*** und nachfolgenden Test über ***is*** erreicht.

- `g:=subs(f,x=-x)`

$$x^4 - 3 \cdot x^2 + 1$$

- `is(f=g)`

true

Ein anderer Fall ist z.B. gegeben, wenn alle Exponenten ungerade sind und das Absolutglied den Wert 0 annimmt. In diesem Fall ist die Funktion zentralsymetrisch zum Koordinatenursprung (0;0).

Im Zweifelsfall kann man über zufällige – möglichst weit gestreute – Variablenwerte (x-Werte) testen, ob $|f(x)| = |f(-x)|$ ist und damit überhaupt eine Symmetrie vorliegt. Für die Achsensymmetrie hinsichtlich der Ordinate muss $f(x) = f(-x)$ sein.

Später lässt sich auch noch die Symmetrie hinsichtlich der bestimmten Wendepunkte (→ VIII – Bestimmung der Wendestellen / Wendepunkte) genauer untersuchen.

III - Verhalten am Rande des Definitionsbereichs

Ab hier kann uns MuPAD direkt unterstützen. Es stellt sich die Frage nach dem Verhalten der Funktion im Unendlichen sowie an Sonderstellen, die uns bei den Betrachtungen zu I (→ I - Bestimmung des Definitionsbereichs D) aufgefallen sind..

Grenzwert einer Funktion	Die Funktion f hat an der Stelle x_0 den ***Grenzwert*** g, also $\lim_{x-->x_0} f(x) = g$: $\underset{Def}{=}$ (1) f ist in einer Umgebung von x_0 (ev. unter Ausschluß der Stelle x_0) definiert (2) für jede gegen x_0 konvergierende Folge (x_n) $(x_n \neq x_0$ für alle $x_0)$ deren Glieder dieser Umgebung angehören, konvergiert die Folge der zugehörigen Funktionswerte $[f(x_n)]$ gegen g

aus /9/

Über die Grenzwert-Funktion ***limit*** (→ 2.9.2. Grenzwerte von Funktionen) können die Untersuchungen geführt werden. Statt ***infinity*** (für unendlich,) lassen sich auch die Definitionslücken etc. eingeben, um das Verhalten an diesen Stellen zu erforschen.

```
limit(f,x=infinity)
```

$$\infty$$

```
limit(f,x=-infinity)
```

$$\infty$$

IIII – Schnittpunkte mit der y-Achse (f(x)-Achse, Ordinate, Größenachse)

Schnittpunkt mit der f(x)-Achse	Für die Funktion f wird der Punkt / werden die Punkte gesucht, an den gilt: P(0;f(0))

Dieser Teil ist mathematisch wohl eher unkompliziert. Wir suchen die Schnittpunkte mit der y-Achse – also die Punkte, wo x = 0 ist. Exakt formuliert: die Ordinate des Schnittpunktes mit der y-Achse.
Dies lässt sich leicht ausrechnen, indem die x einfach durch Null ersetzt werden. Meist fällt schon durch Multiplikationen mit Null eine Vielzahl von Funktionselementen aus den Betrachtungen heraus.
Wir geben es als Substitution – unter Zuhilfenahme von ***subs*** - in MuPAD ein und lassen das System rechnen.

```
x0:=subs(f,x=0)
```

$$1$$

V – Bestimmung der Nullstellen

Nullstelle / Schnittpunkt mit der x-Achse	Die Funktion f besitzt eine Nullstelle, wenn ein Element x des Definitionsbereichs existiert, an dem gilt: $\underset{Def}{=}$ f(x)=0

Die Berechnung der Nullstellen ist u.U. schon wieder etwas aufwendiger. Da y nun Null sein muß, lösen wir die Gleichung entsprechend mit ***solve***.

- `solve(f)`

$$\left\{\left[x=-\frac{\sqrt{5}}{2}-\frac{1}{2}\right],\left[x=\frac{1}{2}-\frac{\sqrt{5}}{2}\right],\left[x=\frac{\sqrt{5}}{2}-\frac{1}{2}\right],\left[x=\frac{\sqrt{5}}{2}+\frac{1}{2}\right]\right\}$$

Die angezeigten Nullstellen sind exakt, aber unschön. Um sie später in der graphischen Darstellung zu verwenden, ergänzen wir den Funktionsaufruf um ***float***. Damit erhalten wir die numerischen Werte.

- `float(solve(f))`

$$\{[x=-0.6180339887],[x=0.6180339887],[x=-1.618033989],[x=1.618033989]\}$$

VI – Bildung der Ableitungen

Differenzierbarkeit	Die Funktion f ist an der Stelle x_0 ***differenzierbar***: $\underset{Def}{=}$ (1) f ist in einer Umgebung von x_0 definiert (2) der Grenzwert (Differentialquotient) $\lim_{h-->0}\frac{f(x_0+h)-f(x_0)}{h}=f'(x_0)$ existiert

aus /9/

Die Bildung der Ableitungen ist in MuPAD durch die eingebaute Funktion ***diff*** ein Kinderspiel (→ 2.9.3. Differentiation von Funktionen). Ob dabei die nachfolgenden Ableitungen über die Schachtelung (wie bei f2) oder durch Nutzung der vorherigen Ableitung (wie ab f3) ermittelt wird, hat praktisch keine Bedeutung. Die Ableitungen speichern wir für spätere Verwendungen (→ VII – Extremstellen, VIII – Wendepunkte, VIIII – Wendetangenten) in passende Variablen.

- `f1:=diff(f,x)`

$$4\cdot x^3-6\cdot x$$

- `f2:=diff(diff(f,x),x)`

$$12\cdot x^2-6$$

- `f3:=diff(f2,x)`

$$24\cdot x$$

- `f4:=diff(f3,x)`

$$24$$

- `f5:=diff(f4,x)`

$$0$$

VII – Bestimmung der (lokalen) Extremstellen

Von einer lokalen (od. relativen) Extremstelle spricht man, wenn zwei Monotoniebögen ohne Knick aneinander stoßen. Die Tangente in diesem Punkt verläuft also parallel zur x-Achse.

Lokales Extremum	Die Funktion f hat an der Stelle x_0 ein lokales ***Maximum***: $\underset{Def}{=}$ es gibt ein $\varepsilon > 0$ derart, dass für jedes x mit $x \neq x_0$ und $x_0 - \varepsilon < x < x_0 + \varepsilon$ gilt $f(x) < f(x_0)$ Die Funktion f hat an der Stelle x_0 ein lokales ***Minimum***: $\underset{Def}{=}$ es gibt ein $\varepsilon > 0$ derart, dass für jedes x mit $x \neq x_0$ und $x_0 - \varepsilon < x < x_0 + \varepsilon$ gilt $f(x) > f(x_0)$

aus /9/

Die notwendige Bedingung für die Ermittlung der Extremstellen ist das Vorhandensein von Null-Stellen in der ersten Ableitung.
Wir bilden oder verwenden die 1. Ableitung (über ***diff*** ; → VI – Bildung der Ableitungen) und lösen diese dann entsprechend (mit ***solve***).

Zur besseren weiteren Verwendung lassen wir uns noch zusätzlich das numerische Äquivalent (mit ***float***) anzeigen.

- `ExtremSt:=solve(f1=0,x)`

$$\left\{0,\ -\frac{\sqrt{6}}{2},\ \frac{\sqrt{6}}{2}\right\}$$

- `float(ExtremSt)`

$$\{-1.224744871,\ 0.0,\ 1.224744871\}$$

Nun müssen wir noch über die hinreichende Bedingung ermitteln, ob es sich um ein Minimum oder ein Maximum handelt.

Wir brauchen nun die 2. Ableitung und setzen (mit ***subs***) die ermittelten Nullstellen ein.

- `ArtExtremSt:=subs(f2,x=ExtremSt)`

$$\{-6,\ 12\}$$

Sehr unschön ist die nicht eindeutige Zuordbarkeit der berechneten Werte zu den Null-Stellen. Also greifen wir lieber einzeln auf die Elemente der Ergebnisliste (hier: ExtremSt) zu. Dies ermöglicht uns ***op***. Mittels ***subs*** setzen wir die Werte in die 2. Ableitung (→ VI) ein. In jeder Zeile lassen wir uns dann das genutzte Element und den zugehörige Funktionswert in der 2. Ableitung anzeigen.

Da die ersten beiden Werte größer als Null sind folgt daraus, das sich bei $\frac{\sqrt{6}}{2}$ und $-\frac{\sqrt{6}}{2}$ jeweils ein Minimum befindet. Beim dritten Wert folgt entsprechend ein Maximum für die Extremstelle am Definitionswert 0.

- `op(ExtremSt,1); subs(f2,x=op(ExtremSt,1))`

$$\frac{\sqrt{6}}{2}$$

$$12$$

- `op(ExtremSt,2); subs(f2,x=op(ExtremSt,2))`

$$-\frac{\sqrt{6}}{2}$$

$$12$$

Bei einem berechneten Funktionswert gleich Null bleibt die Frage bezüglich einer Extremstelle offen.

```
op(ExtremSt,3); subs(f2,x=op(ExtremSt,3))
```

$$0$$

$$-6$$

Mit Hilfe von links- und rechtsseitigen Testwerten können aber auch – sozusagen numerisch - Aussagen zur Art eines Extremas gemacht werden.

VIII – Bestimmung der Wendestellen / Wendepunkte

Wendepunkte finden wir an Stellen, wo ein Konvexbogen auf einen Konkavbogen (ohne Knick) stößt oder umgekehrt – sich also die Krümmung ändert. Ist die Tangente im Wendepunkt eine Parallele zur x-Achse handelt es sich um speziell einen Terassenpunkt (auch Stufenpunkt, Horizontalwendepunkt).

Wendepunkt	Die in einer Umgebung von x_0 differenzierbare Funktion f hat an der Stelle x_0 einen ***Wendepunkt*** $\underset{Def}{=}$ f' hat in x_0 ein lokales Extremum

aus /9/

Ähnlich dem vorherigen Vorgehen bestimmen wir nun die Wendepunkte über die Null-Stellen der 2. Ableitung.

Mittels ***solve*** und den schon gebildeten Ableitungen (→ VI – Bildung der Ableitungen) erhalten wir zwei mögliche Wendestellen.

```
WendeSt:=solve(f2=0,x)
```

$$\left\{-\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}\right\}$$

Die Prüfung erfolgt diesmal über die 3. Ableitung, wobei der Funktionswert dann ungleich Null sein muß.
Beide Werte sind in unserem Beispiel ungleich. Somit besitzt die untersuchte Funktion zwei Wendestellen.

```
op(WendeSt,1); subs(f3,x=op(WendeSt,1))
```

$$\frac{\sqrt{2}}{2}$$

$$12\cdot\sqrt{2}$$

```
op(WendeSt,2); subs(f3,x=op(WendeSt,2))
```

$$-\frac{\sqrt{2}}{2}$$

$$-12\cdot\sqrt{2}$$

Die exakten Koordinaten der Wendepunkte ergeben sich aus den in die untersuchte Funktion (über ***subs***) eingesetzte Null-Stellen der 3. Ableitung:

Die Wendepunkte liegen also bei $(12\sqrt{2}\,;-\frac{1}{4})$ und $(-12\sqrt{2}\,;-\frac{1}{4})$

Bei Bedarf lassen sich die numerischen Werte wieder mit ***float*** erzeugen.

```
subs(f,x=op(WendeSt,1));
```

$$-\frac{1}{4}$$

```
subs(f,x=op(WendeSt,2));
```

$$-\frac{1}{4}$$

VIIII – Bestimmung der Wendetangenten

Zur Ermittlung der Wendetangenten setzen wir (mittels ***subs***) die Wendepunkte in die 1. Ableitung ein. Da wir zwei Wendestellen haben, müssen auch zwei Wendetangenten berechnet werden.

```
m1WT:=subs(f1,x=op(WendeSt,1));
float(m1WT)
```

$$-2\cdot\sqrt{2}$$

$$-2.828427125$$

Bezugnehmend auf die entsprechenden Wendepunkte ergeben sich die zwei nebenstehenden Anstiege (exakter und numerischer Wert ausgegeben!)

```
m2WT:=subs(f1,x=op(WendeSt,1));
float(m2WT)
```

$$2\cdot\sqrt{2}$$

$$2.828427125$$

Über das Einsetzen der ermittelten Anstiege und die Koordinaten der Punkte in die Normalform der Gleichung $y = mx + n$ erhalten wir die Wendetangenten-Gleichungen:

$y = -2{,}82843x + n$

und

$y = +2{,}82843x + n$

Dabei ermitteln wir n über:

$n = y_w - mx_w$

also für unser Beispiel über:

$n = y_w + 2{,}82843x_w$

```
n1WT:=yw1-m1WT*op(WendeSt,1);
float(n1WT)
```

$$\frac{7}{4}$$

$$1.75$$

```
n2WT:=yw2-m2WT*op(WendeSt,2);
float(n2WT)
```

$$\frac{7}{4}$$

$$1.75$$

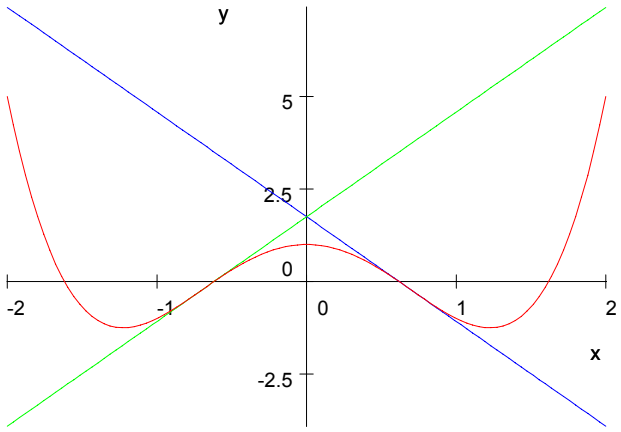
Die Wendetangenten sind also:

$y = -2{,}82843 + 1{,}75$

bzw.: $n = y_w - 2{,}82843x_w$

und: $y = +2{,}82843x + 1{,}75$

Ein Zwischendarstellung der untersuchten Funktion und der beiden Wendetangenten bestätigt unsere Rechnungen.



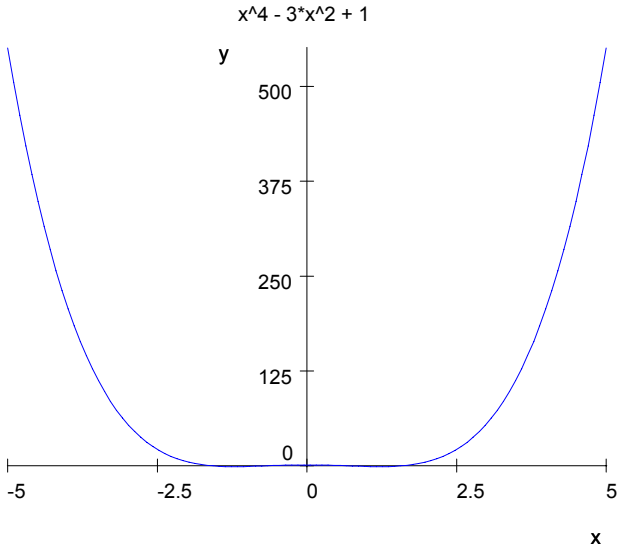
X – Darstellung der Funktion / Schaubild

Nun zum angenehmen Teil. Die Darstellung einer zweidimensionalen Funktion ist über ***plotfunc2d*** kein Problem. Als Parameter reichen die Funktion und ein Definitionsbereich.

Den Definitionsbereich kann man zuerst recht großzügig wählen. So erhält man einen groben Überblick über die Funktion. Gute Wertebereichen liegen z.B. zwischen:

- -1000 .. 1000
- -100 .. 100
- -10 .. 10
- -5 .. 5
- -3 .. 3
- -1 .. 1
- -0.1 .. 0.1

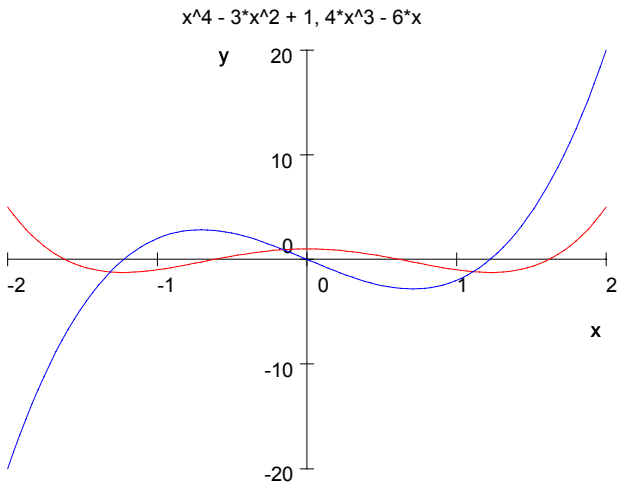
- `plotfunc2d(f,x=-5..5)`



Dabei besteht natürlich die Gefahr, dass wichtige Details nicht ausreichend sichtbar werden. Das Anpassen des Bereichs geht schnell vonstatten, so dass wir einfach ein wenig experimentieren können. Gerade, wenn sich zwischen zwei aufeinander folgenden Bereichen die Form des Graphen völlig verändert ist, dann liegt der optimale Bereich meist genau dazwischen.

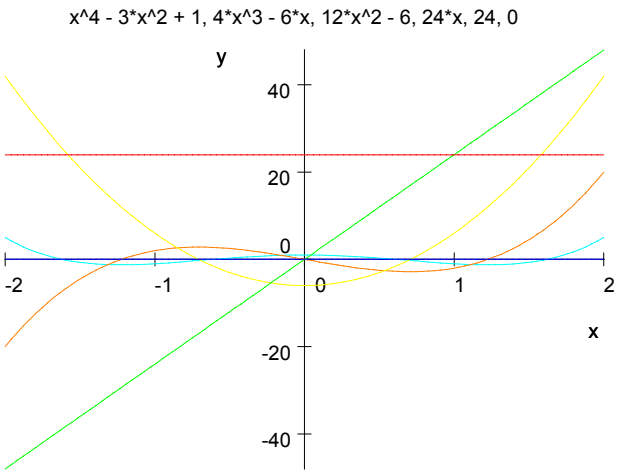
Auf diese Art und Weise können wir auch weitere Funktionen ergänzen.
Da bieten sich natürlich die Ableitungen an. So erhält man einen schönen Blick dafür, wie Extremstellen usw. mit den Ableitungen korrespondieren.

- `plotfunc2d(f,f1,x=-2..2)`



Durch den Einbau aller relevanten Funktionen in den Graphik-Aufruf, erhält man eine befriedigende Übersicht. Schade, dass die Funktionen nicht farblich beschriftet sind, dann wäre eine Zuordnung zu den Graphen kein Problem mehr.
Die letzte Funktion in der Liste erhält die Farbe rot. Die zweitletzte wird blau gezeichnet.

```
plotfunc2d(f,f1,f2,f3,f4,f5,x=-2..2)
```



Es folgen gelb und grün. Bei noch mehr Funktionen geht es wieder mit rot von vorne los.
Für normale Fälle sollte diese Darstellung eigentlich reichen.
Nachteil des obigen Vorgehens sind immer der Tipp-Aufwand für den gesamten Funktionsaufruf und die eingeschränkten Variationsmöglichkeiten. Bei komplizierten Aufrufen und speziellen Darstellungswünschen artet der Aufwand schnelll aus. Zum Anderen ist es schwierig, in so komplexen Aufrufen später Fehler zu finden.

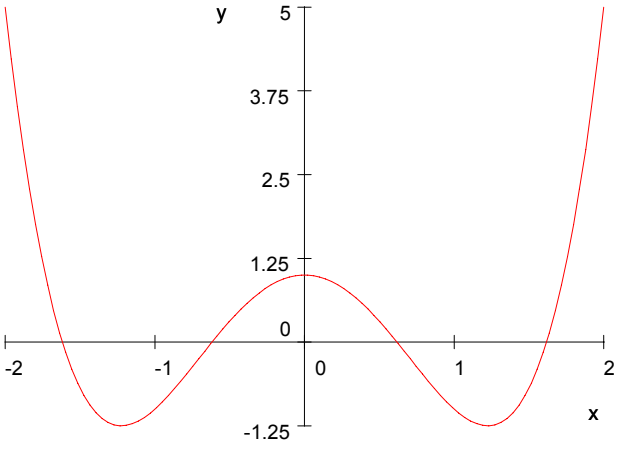
Zum effektiveren Arbeiten definieren wir uns einzelne Plots für die einzelnen Objekte und lassen uns diese dann zusammen oder teilweise zeichnen. So erhalten wir eine flexible Möglichkeit später einzelnen Parameter (Farbe etc.) unkompliziert zu ändern und die Darstellungen auszuprobieren.

Mit Plots lassen sich auch verschiedenartige graphische Objekte (Punkte, Funktionen, Graphikprimitive, ...) kombinieren. Damit die einzelnen Plots für die reine zeichenfunktion ***plot*** nutzbar sind, ordnen wir sie Variablen (hier g, g1, g2, ...) zu. Beginnen wir mit der Stammfunktion als Plot g1. Die probeweise Darstellung ist auch schon befriedigend.

- `g:=plot::Function2d(f,x=-2..2)`

  $\text{plot::Function2d}\left(-3\cdot x^2+x^4+1,\ x=-2..2\right)$

- `plot(g1)`



Nun können wir alle anderen interessanten Ableitungen in Plots einbringen.

Das Ergebnis kann sich als Gesamtobjekt zwar sehen lassen. Einen Schönheits- oder Übersichtlichkeitspreis können wir damit aber nicht gewinnen.
Es ist nun notwendig - durch geschicktes Anpassen einzelner Plots - die Darstellung ansprechender gestalten.
Aus diesem ersten Plot ersehen wir leicht, dass die lineare und die quadratische Funktion (entsprechen 4. und 3. Ableitung) den Wertebereich bestimmen. Da aber bei größeren Werten gar keine interessanten Informationen (über den Kurvenverlauf) liegen, begrenzen wir bei den betreffenden Plots einfach auch den y-Bereich.

- `g1:=plot::Function2d(f1,x=-2..2)`

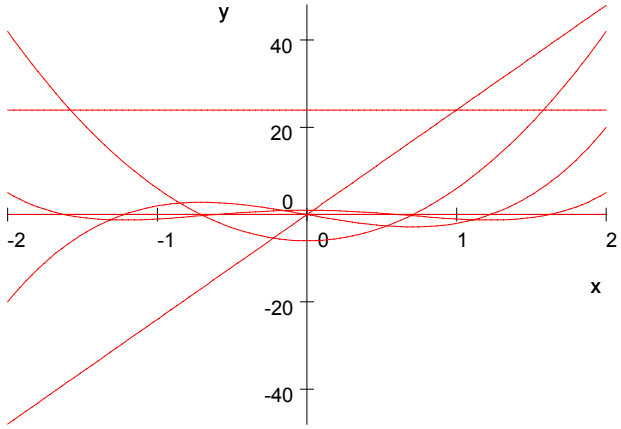
  plot::Function2d$(-6\cdot x+4\cdot x^3,\ x=-2..2)$
- `g2:=plot::Function2d(f2,x=-2..2)`

  plot::Function2d$(12\cdot x^2-6,\ x=-2..2)$
- `g3:=plot::Function2d(f3,x=-2..2)`

  plot::Function2d$(24\cdot x,\ x=-2..2)$
- `g4:=plot::Function2d(f4,x=-2..2)`

  plot::Function2d$(24,\ x=-2..2)$
- `g5:=plot::Function2d(f5,x=-2..2)`

  plot::Function2d$(0,\ x=-2..2)$
- `plot(g,g1,g2,g3,g4,g5)`



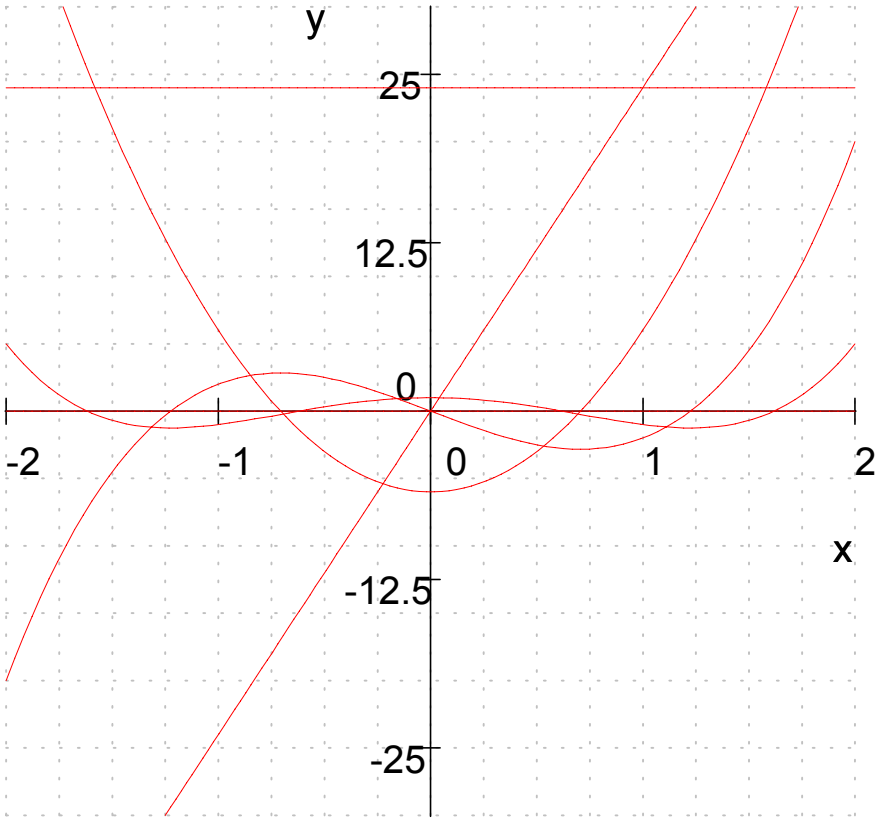
Zusätzlich lassen wir uns noch ein paar Gitterlinien mit einzeichnen, um das Auffinden von irgendwelchen Punkten etc. zu vereinfachen..

- `g2:=plot::Function2d(f2,x=-2..2,y=-30..30)`

  plot::Function2d$(12\cdot x^2-6,\ x=-2..2)$
- `g3:=plot::Function2d(f3,x=-2..2,y=-30..30)`

  plot::Function2d$(24\cdot x,\ x=-2..2)$
- `plot(g,g1,g2,g3,g4,g5,GridLines=[10,10])`

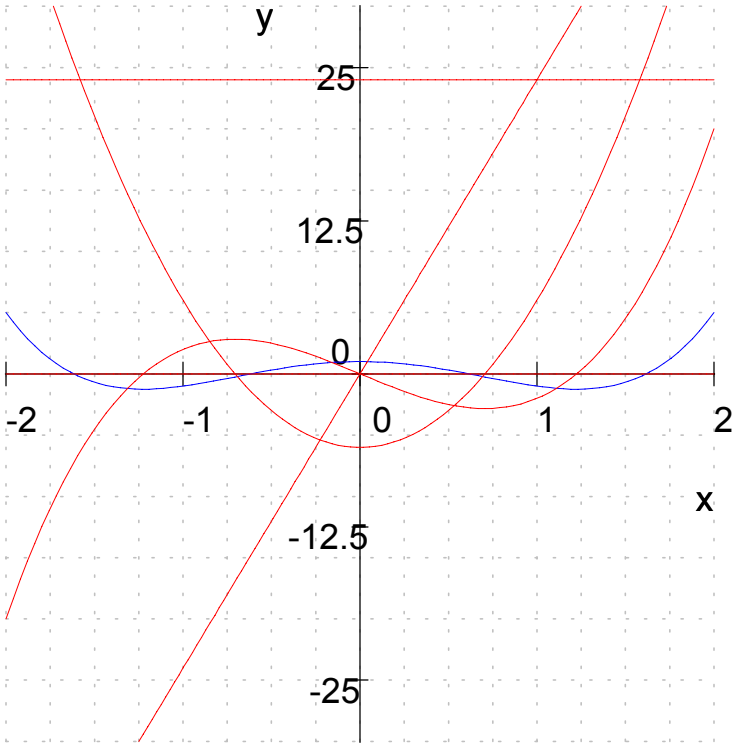
Der Ergebnis-Plot läßt sich auch noch (durch Anklicken und Ziehen an den schwarzen Kästchen) vergrößern.



Wenn auch noch Farbe ins Spiel gebracht und die Funktionen beschriftet werden sollen, dann lassen sich solche Wünsche auch mit einbauen.
Durch beliebiges Kombinieren der Plots können eigentlich alle gewünschte Sachverhalten und Funktionskennzeichen darstellen werden.
Die ganz Eifrigen können sich dann auch noch Textfelder mit Erklärungen in das Notebook hinzufügen.

- `g:=plot::Function2d(f,x=-2..2,LineWidth=2, Color=[Flat,RGB::Blue])`

  $\text{plot::Function2d}\left(-3\cdot x^2+x^4+1,\ x=-2..2\right)$



Der Aufwand für die Grafik-Erstellung wächst so aber auch schnell an. Ob so viel Arbeit notwendig ist, muß dann jeder für sich und den Verwendungszweck entscheiden.
Dafür ist das Ergebnis fast perfekt – der Mathelehrer kann's bestimmt auch nicht besser.

Aufgaben:

1. Führen Sie mit folgenden Funktionen eine umfassende Kurvendiskussion durch!

a) $f(x)=x^4-x^2$

b) $f(x)=\dfrac{2x-1}{x^2}$

c) $g(x)=3x^4-2x^2-4$

d) $g(x)=x^3+2x^2-x+1$

2. Berechnen Sie für die folgenden Funktionen die wichtigsten charakteristischen Punkte! Stellen Sie den Graph dar!

a) $f(x)=-0{,}25x^3+2x$

b) $g(x)=\dfrac{3x^2-8x}{(x-2)^2}$

2.10.1. Wiederverwendung des Notebooks

Für die spätere Verwendung / Besichtigung können wir das Notebook mit der Kurvendiskussion abspeichern. Aber jedes Mal alle Befehle erneut eintippen ist doch etwas zu aufwendig (- für faule Computernutzer). Eine gute Möglichkeit zur Arbeitserleichterung ist die Wiederverwendung eines Notebooks. Dazu speichern wir den getesteten Notebook in eine extra Datei. Als Name könnte man z.B. "Vorlage für ..." verwenden.
Nun löscht man alle Ausgabenbereiche über das Menü "Notebook" "Ausgaben löschen". Die Nachfrage beantworten wir mit "OK". Nun sieht man nur noch die roten Eingabebereiche. Dieses Notebook wird nochmals abgespeichert und ab jetzt nur noch als Vorlage benutzt. Jedes Mal, wenn man das Notebook für eine neue Untersuchung gebraucht, speichert man es sofort nach dem öffnen unter einem neuen Namen ab. So bleibt die Vorlage sauber erhalten.
In dem neu abgespeicherten Notebook für die aktuelle Arbeitsaufgabe können nun beliebige Änderungen eingetragen werden.
In der oben gemachten Funktionsuntersuchung braucht man im Allgemeinen nur die Variable ***f*** für die zu untersuchende Funktion neu definieren. Damit ist auch der richtige Zeitpunkt gekommen, das Vorlagen-Notbook als ein spezielles Arbeits-Notebook (z.B. "Kurvendiskussion xyz" abzuspeichern. (Die Vorlagendatei bleibt dann ungeändert erhalten.)
Dann werden Zeile für Zeile alle Eingabebereiche noch einmal mit ENTER bestätigt und fertig ist die neue Kurvendiskussion. Beachten Sie aber, dass u.U. auch mal mehr Extremstellen etc. auftauchen können. Dann müssen ev. zusätzliche Zeilen dazugeschrieben werden. Also verstehen muß man die Befehle trotzdem, sonst geht's mit der Nutzung schnell mal daneben.

2.10.2. Programmierte Kurvendiskussion

Die wichtigsten Schritte der Kurvendiskussion - zusammengestellt als Algorithmus - ergibt den Text für das nachfolgende Programm-Notebook: Das Programm stammt aus /10/ und betrachtet nur die wichtigsten und allgemeinsten Elemente der Kurvendiskussion. Es lassen sich noch diverse Erweiterung und Verbesserungen einbauen. Mit den Kenntnissen aus dem Abschnitt → 2.10. Programmierung sind der eigenen Kreativität keine Grenzen gesetzt.

Programm	Kommentare
01 Kurvendiskussion:=**proc**(f)	
02 **begin**	
03 Nullstellen:=**float**(**solve**(f(x)=0,x));	
04 **print**("Nullstellen= ",Nullstellen);	
05 Ableitung1:=f'(x);	
06 **print**("f'(x)= ",Ableitung1);	
07 NullstAbl1:=**float**(**solve**(f'(x)=0,x));	
08 **print**("Nullstellen d. 1. Ableitung= ", NullstAbl1);	
09 **print**("f(",NullstAbl1,")= ", **map**(NullstAbl1,f));	
10 Ableitung2:=f''(x);	
11 **print**("f''(x)= ",Ableitung2);	
12 HKExtrema:=**map**(NullstAbl1,f'');	
13 **print**("f''(",NullstAbl1,")= ", HKExtrema);	
14 NullstAbl2:=**float**(**solve**(f''(x)=0,x));	
15 **print**("Nullstellen d. 2. Ableitung= ", NullstAbl2);	
16 Ableitung3:=f'''(x);	
17 **print**("f'''(x)= ",Ableitung3);	
18 **print**("f'''(",NullstAbl2,")= ", **map**(NullstAbl2,f'''));	
19 **plotfunc2d**(f(x),f''(x),f'''(x));	
20 **end_proc**:	

2.12. Anwendung der Kurvendiskussion: Extremwert-Aufgaben

Dieser Bereich von mathematischen Aufgaben wird auch häufig als lineare Optimierung oder Extremwert-Probleme bezeichnet. Er hat einen sehr hohen Praxisbezug und ist deshalb ein gutes Beispiel für eine sinnvolle Beschäftigung mit Mathematik. Ziel der linearen Optimierung ist das Finden von besonders geeigneten Lösungen für eine Aufgabe. Mal soll möglichst viel Material gespart werden und das zu bauende Teil soll besonders groß sein. Das nächste Mal sucht man nach einer Möglichkeit den Verlust so gering wie möglich zu halten usw. usf.
Typische Beispiele für Extremwertaufgaben:

- Körper und Flächenprobleme
- Transportprobleme
- Lageprobleme / Anordnungsprobleme / Entfernungsprobleme

Immer sind mindestens zwei (scheinbar) widersprüchliche / entgegen gesetzte Sachverhalte so zu vereinen, dass eine optimale (besonders günstige) Lösung heraus kommt.
Prinzipiell lassen sich solche Aufgaben graphisch lösen, wenn man die Variablen und die zugehörigen Funktionen kennt. Leider sind die graphischen Lösungen oft sehr ungenau (z.B. weil sich z.B. die Funktionen sehr flach schneiden). Deshalb gibt man einer exakten Lösung durch die Differentialrechnung den Vorzug.
Hier erhält die vielleicht etwas abgesetzt erscheinende Kurvendiskussion ihre praktische Berechtigung.

Schrittfolge zur Lösung von Extremwertaufgaben (mit Hilfe der Differentialrechnung)

nach /7/ und /8/

I. Analyse der Aufgabe / Festlegung der Variablen
II. Aufstellen der Funktionen / Beziehungen zwischen den Variablen
a) Aufstellen der Hauptbeziehung (Extremalbeziehung, Zielfunktion)
b) Aufstellen der Nebenbeziehung / Nebenbedingungen / Hilfsbeziehungen
III. Einsetzen der Nebenbeziehung in die Hauptbeziehung (➔ erweiterte Zielfunktion)
! Reduktion der Variablenzahl in der erweiterten Zielfunktion auf 1
IIII. Festlegung des Definitionsbereiches
V. Bildung der 1. und 2. Ableitungen
VI. Bestimmung der Extremwerte innerhalb des Definitionsbereiches
a) Bestimmung lokaler Extremwerte
b) Untersuchung der Randfälle
VII. Berechnung der anderen Variablen an den Extremstellen / Interpretation der berechneten Werte

Ich ergänze immer gerne noch zwei Schritte:

VIII. Darstellung der erweiterten Zielfunktion und der Ableitungen / Graphische Prüfung
VIIII. Probe(n)

Desweiteren kann man – gerade weil es in MuPAD kein Problem ist – eine erste graphische Darstellung der erweiterten Zielfunktion schon gleich mit Kenntnis der Selbigen erzeugen (z.B. als *IIII.a) Skizze der erweiterten Zielfunktion*). Das gibt bei den nachfolgenden Rechnungen hier und dort das Gefühl von Sicherheit.
Die Probleme in diesen Aufgaben sind eher die Bildungen der mathematischen Modell aus den sächlichen Gegebenheiten. Die reine Mathematik ist eigentlich mehr Beschäftigungstherapie (vorausgesetzt man beherrscht Termumstellung und Differentiation).

I. Analyse der Aufgabe / Festlegung der Variablen

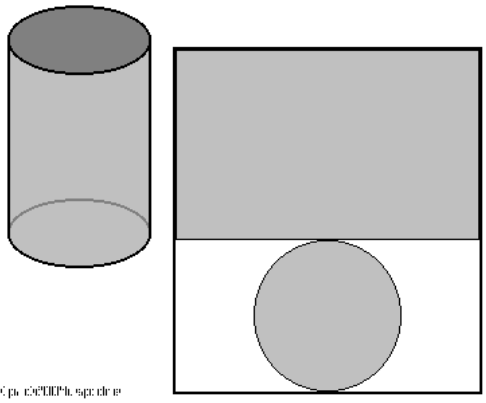
Die Analyse der Aufgabenstellung bleibt vorrangig unserem Gehirn vorbehalten. Zur Lösung solcher komplexen Aufgaben empfehle ich immer, möglichst alle Informationen aus der Aufgabe erst einmal in übliche Symbole (Volumen = V, Fläche = A, ...) umzusetzen. Dann hat man es später mit den Formeln aus Tafelwerken usw. einfacher.
Als Beispielaufgabe wählen ich folgendes einfaches Extremwert-Problem:
Unter möglichst geringem Blechverbrauch soll ein oben offener, zylindrischer Eimer gebaut werden, in den genau 5 l passen (randvoll). (Die Kanten werden verschweißt.) Welche Höhe und welchen Durchmesser hat der optimale Eimer.

Die Analyse bringt die folgenden Informationen mit passenden Formelzeichen:
geg.: V = 5 l (Zylinder) *ges.: h, d (Zylinder)*
Ziel: A_{ges} → Min

Da wir Längen suchen, wandeln wir das Volumenmaß Liter gleich in die äquivalente Einheit dm³ um.
geg.: V = 5 l = 5 dm³ (Zylinder) *ges.: h, d (Zylinder)*
Ziel: A_{ges} → Min (Zylinder, Blech)

II. Aufstellen der Funktionen / Beziehungen zwischen den Variablen

Mit Hilfe einer kleinen Übersichtsskizze kann man sich den Sachverhalt noch etwas klarer machen.



Somit stellen wir fest, dass die Oberfläche des Eimers A_{ges} (- der bestimmt den Blechverbrauch -) einmal aus dem Mantel A_M und einmal aus der Grundfläche A_0 besteht (A_{ges} = A_M + A_0). Die typische Oberflächenformel $A_{ges} = 2 * \pi * r^2 + 2\pi * r * h$ für einen (geschlossenen) Zylinder muß also um eine Grundfläche reduziert werden oder die Summe aus beiden Teilflächen gebildet werden.
Damit ergibt sich: $A_{ges} = \pi * r^2 + 2\pi * r * h$.

II.a) Aufstellen der Hauptbeziehung (Extremalbeziehung, Zielfunktion)

Die Zielfunktion ist in Abhängigkeit von der Aufgabenstellung eine Flächenformel ($A_{ges} = \pi * r^2 + 2\pi * r * h$). Die Angaben zum Volumen ($V = 5l$) sollen später in die Nebenbedingungen (Nebenfunktionen) eingehen.
Die derzeitige Hauptfunktion ist für unser Verfahren noch nicht nutzbar, da sie noch von 2 Variablen abhängt: $A_{ges} = f(r,h)$.

II.b) Aufstellen der Nebenbeziehung / Nebenbedingungen / Hilfsbeziehungen

Wir müssen uns nun Formeln suchen, die zum einen unsere Bedingung vom gewünschten Volumen mit einbezieht und zum anderen eine der Variablen unnötig macht.
Beim Heraussuchen und dem Notieren von Formeln aus Tafelwerken etc. sollte man nicht zu sparsam sein. Lieber eine Formel zu viel – die man später nicht braucht – als gar kein Lösungsansatz finden. Als günstig hat sich auch erwiesen, das Ziel der Suche zu notieren, z.B.:

$A_{ges} = f(r,h)$ → V, ??? → $A_{ges} = f(h)$ ODER $A_{ges} = f(r)$

(**Gesprochen:** Ausgehend von der Zylinderfläche, die eine Funktion von Radius und Höhe ist, wird zu mindestens unter Verwendung des gegebenen Volumens entweder eine Funktion für die Fläche gesucht, die nur von der Höhe ODER nur vom Radius abhängig ist.)

Beginnen wir bei die Suche mit dem Offensichtlichen – der Volumenformel für einen Zylinder: $V = \pi * r^2 * h$. Da das Volumen fest mit 5 Litern vorgeben ist, können wir die Volumenformel nach einer Variable umstellen und dann in die Zielfunktion einsetzen.

$5dm^3 = \pi * r^2 * h$

Wir wählen sinnigerweise die Höhe - sie kommt in der Zielfunktion nur einmal vor.

$$h = \frac{5dm^3}{\pi * r^2}$$

III. Einsetzen der Nebenbeziehung in die Hauptbeziehung (➔ erweiterte Zielfunktion)

Durch Einsetzen der eben aufgestellten Nebenbeziehung in die Hauptbeziehung erhalten wir:

$$A_{ges} = \pi * r^2 + 2\pi * r * \frac{5dm^3}{\pi * r^2}$$

Nun kürzen wir und fassen passende Faktoren zusammen.

$$A_{ges} = \pi * r^2 + \frac{10dm^3}{\pi * r}$$

Damit sind wir schon am Ziel. Die gefundene Funktion ist nur noch von einer Variable – dem Radius r – abhängig. Unsere gesuchte und nutzbare (erweiterte) Zielfunktion lautet somit:

$$A_{ges} = \pi * r^2 + \frac{10}{\pi * r} \quad [dm^3]$$

Für das reine Rechnen wurde noch die Einheit dm³ heraus genommen. Am Ende dürfen wir aber nicht vergessen, dass die Ergebnisse dann in dm herauskommen.

Diese Schritte lassen sich weitgehend auch mit MuPAD erledigen. Doppelt hält erfahrungsgemäß besser und wir haben gleich eine Kontrolle.

Mit ***delete*** werden die undefinierten Variablen gelöscht. Der Aufruf von ***assume*** ist für ein vereinfachtes Arbeiten angebracht. Es wird für die Variablen r und h angenommen (engl.: assume), das sie nur größer als 0 (bzw. positiv und ungleich 0) sein dürfen.

Dann definieren wir die Ziel- und die Nebenfunktion. Zum Einsetzen der Nebenfunktion lösen wir diese (mittels ***solve***) nach h auf. Dabei wird auch gleich noch das gegebene Volumen eingesetzt (über ***subs***). Das Umstellungsergebnis legen wir auf die Variable NB1, um sie im nächsten Schritt in die Zielfunktion einzusetzen.

- `delete r,h;`
- `assume(r>0); assume(h>0)`
- `Ages:=PI*r^2+2*PI*r*h`

$$2 \cdot h \cdot r \cdot \pi + r^2 \cdot \pi$$

- `V:=r^2*PI*h`

$$h \cdot r^2 \cdot \pi$$

Variante:

- `NB1:=solve(5=V,h)`

$$\left\{ \frac{5}{r^2 \cdot \pi} \right\}$$

- `erwZFkt:=subs(Ages,h=NB1[1])`

$$\frac{10}{r} + r^2 \cdot \pi$$

Natürlich hätten wir die erweiterte Zielfunktion auch direkt eingeben und der Variable erwZFkt zuweisen können. Die ersten vier Eingaben (von delete bis V:=...) sind für das weitere Arbeiten natürlich trotdem vorher einzugeben.

!!! Alternative:

- `erwZFkt:=2*PI*r-10/r^2`

$$2 \cdot r \cdot \pi - \frac{10}{r^2}$$

IIII. Festlegung des Definitionsbereiches

Auf dem Papier würden wir nun die Grenzen des Definitionsbereichs festlegen. In MuPAD haben wir dies schon mit ***assume*** getan, da sonst schon recht komplexe – aber eben exakte – Zwischenergebnisse und Termumformungen aufgetreten wären.

Der Radius r des Zylinders muß logischerweise größer Null sein. Negative Radien sind nicht möglich und auch ein Radius mit einer Null-Ausdehnung ergibt keinen sinnvollen dreidimensionalen Körper.

Für MuPAD sollte man also bei Bedarf den obigen Algorithmus dahingehend ändern, dass die Festlegung des Definitionsbereichs schon bei der Analyse der Aufgabenstellung und der Definition von Variablen (→ I.) erfolgt.

V. Bildung der 1. und 2. Ableitungen

Hier ist MuPAD nicht zu schlagen. Die differenzierten Funktionen (Ableitungen) liegen ratz batz vor.

- `erwZFkt1:=diff(erwZFkt,r)`

$$2 \cdot r \cdot \pi - \frac{10}{r^2}$$

- `erwZFkt2:=diff(erwZFkt1,r)`

$$2 \cdot \pi + \frac{20}{r^3}$$

VI. Bestimmung der Extremwerte innerhalb des Definitionsbereiches

Der absolute Extremwert (absolutes Minimum; absolutes Maximum) ist der kleinste bzw. größte Funktionswert der (erweiterten) Zielfunktion im Definitionsbereich. (Ist er nicht über Ableitungen zugänglich, dann liegt er an den Grenzen des Definitionsbereichs.

Die Suche erfolgt also zweigeteilt. Einmal müssen wir innerhalb des Kurvenverlaufs nach Extremwerten suchen (→ VI.a). Zum Zweiten ist abzuprüfen, ob sich vielleicht absolute Extremwerte an den Rändern des Definitionsbereichs befinden (→ VI.b).

VI.a) Bestimmung lokaler Extremwerte

Zuerst suchen wir über die Nullstellen der ersten Ableitung (hier NsterwZFkt1) nach lokalen Extremstellen (unter Verwendung von ***solve***). Die zu lösende Funktion (1. Ableitung gleich null gesetzt) wird oft auch als "Schlüsselgleichung" bezeichnet.

Die angebotene Nullstelle muß nun über die 2. Ableitung geprüft werden. Das Einsetzen über ***subs*** bringt bei ExtremSt ein positives Ergebnis, was für uns bedeutet: "Hier liegt ein Minimum." und so was haben wir ja gesucht.

- `NSterwZFkt1:=solve(erwZFkt1=0,r)`

$$\left\{\sqrt[3]{\frac{5}{\pi}}\right\}$$

- `float(NSterwZFkt1)`

$$\{1.167544325\}$$

- `ExtremSt:=subs(erwZFkt2,r=NSterwZFkt1[1])`

$$6 \cdot \pi$$

VI.b) Untersuchung der Randfälle

Sichern wir unser Ergebnis bei den lokalen Extremwerten damit ab, dass wir den Kurvenverlauf und ev. die Funktionswerte an den Grenzen mit unserem Extremwert vergleichen.

Über die Grenzwert-Untersuchengen mittels ***limit*** bekommen wir zu mindestens raus, dass im Bereich des größer werdenden Radius auch der Kurvenverlauf (- die Oberfläche -) ins Unendliche läuft / steigt.

- `limit(erwZFkt,r=infinity)`

$$\infty$$

- `limit(erwZFkt,r=0,Right)`

$$\infty$$

Da bei einem Radius von Null die Funktion undefiniert ist, erkunden wir die Tendenz durch Abfrage von zwei relativ dichten Werten. Und wir sehen, dass mit kleiner werdenden Radius der Wert immer mehr steigt. Also quasi auch ins unendliche steigt.

- `limit(erwZFkt,r=0.001)`

  10000.0

- `limit(erwZFkt,r=0.000001)`

  10000000.0

An den Grenzen sind also keine Minima zu finden, was unser lokales Extrema zum absoluten macht.

VII. Berechnung der anderen Variablen an den Extremstellen / Interpretation der berechneten Werte

Der "minimale" Radius ist nun Ausgangspunkt, um die restlichen benötigten (und interessanten) Werte zu berechnen.

Zuerst setzen wir (mittels ***subs***) den "minimierten" Radius (entspricht der geprüften Nullstelle in der 1. Ableitung) in die Volumen-Gleichung des Zylinders ein. Wie wir sehen, entsteht so eine Gleichung, die leicht nach der Höhe h umgestellt und gelöst (mit solve) werden kann. Den unhandlichen Ergebnisausdruck formen wir mit float in eine reele Zahl um.

- `EimerVolumen:=subs(V=5,r=NSterwZFkt1[1])`

  $$h \cdot \pi \cdot \left(\frac{5}{\pi}\right)^{\frac{2}{3}} = 5$$

- `EimerHoehe:=solve(EimerVolumen,h)`

  $$\left\{\frac{5}{\pi \cdot \left(\frac{5}{\pi}\right)^{\frac{2}{3}}}\right\}$$

- `float(EimerHoehe)`

  {1.167544325}

Aus informativen Gründen können jetzt noch die benötigte Blech-Fläche und das Volumen des Eimers berechnet werden. (Dazu mehr bei den Proben.)

Den Blechbedarf berechnen wir durch Einsetzen der gerade bestimmten Höhe und des optimalen Radius in die Flächen-Formel des Eimers.
Das Volumen wird äquivalent über die Volumen-Formel nachgerechnet. Der Wert 5,0 sollte uns nicht wirklich überraschen.

- `Fl:=subs(Ages,r=NSterwZFkt1[1],h=EimerHoehe)`

  $$\pi \cdot \left(\frac{5}{\pi}\right)^{\frac{2}{3}} + \left\{\frac{5}{\pi \cdot \left(\frac{5}{\pi}\right)^{\frac{2}{3}}}\right\} \cdot \left(2 \cdot \pi \cdot \sqrt[3]{\frac{5}{\pi}}\right)$$

- `float(Fl)`

  {12.84747798}

- `Vol:=subs(V,r=NSterwZFkt1[1],h=EimerHoehe)`

  $$\left\{\frac{5}{\pi \cdot \left(\frac{5}{\pi}\right)^{\frac{2}{3}}}\right\} \cdot \left(\pi \cdot \left(\frac{5}{\pi}\right)^{\frac{2}{3}}\right)$$

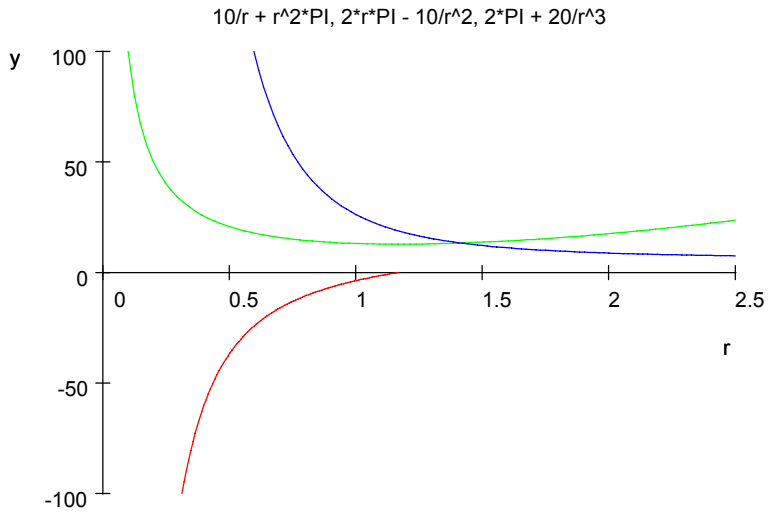
- `float(Vol)`

  {5.0}

VIII. Darstellung der erweiterten Zielfunktion und der Ableitungen / Graphische Prüfung

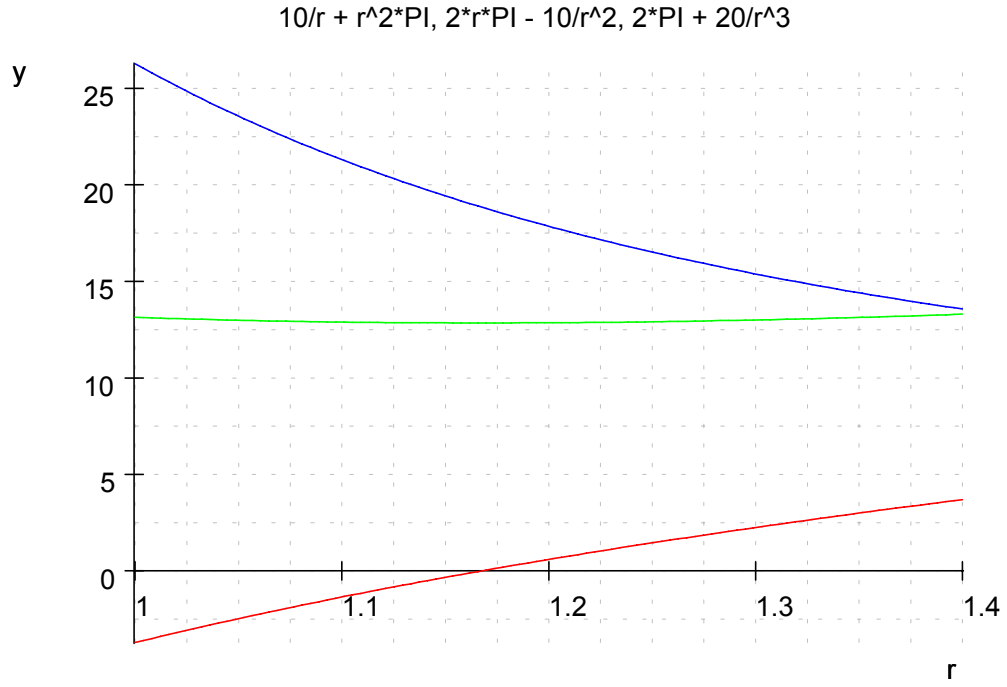
Mit dem einfachen Graphikaufruf ***plotfunc2d*** erzeugen wir uns die Graphen der erweiterten Zielfunktion und deren ersten beiden Ableitungen. Die Grenzen für den Radius r kann man natürlich erst größer wählen und sich dann nach und nach an einen geeigneten Bereich herantasten.

- `plotfunc2d(erwZFkt,erwZFkt1,erwZFkt2,r=0..2.5)`



Für die genaue Prüfung der Lösung kann dann auch noch die entscheidende Stelle um den Extremwert herausgezeichnet werden.

- `plotfunc2d(erwZFkt,erwZFkt1,erwZFkt2,r=1.0..1.4,GridLines=[10,10])`



Hier sehen wir deutlich, dass eine graphische Lösung schwierig wäre, da die Hauptfunktion (erwZFkt, grün) an der interessanten Stelle sehr flach verläuft.

VIIII. Probe(n)

Zur Prüfung unserer Ergebnisse für Radius und Höhe des Zylinders (Eimers) setzen wir diese in die Volumen-Formeln (Nebenbedingung) ein. Das Ergebnis sollte zumindestens mit dem vorgegebenen Eimer-Volumen übereinstimmen – was es auch tut.

```
probe:=subs(V,r=NSterwZFkt1[1],h=EimerHoehe)
```

$$\left\{\frac{5}{\pi\cdot\left(\frac{5}{\pi}\right)^{\frac{2}{3}}}\right\}\cdot\left(\pi\cdot\left(\frac{5}{\pi}\right)^{\frac{2}{3}}\right)$$

```
float(probe)
```

$$\{5.0\}$$

```
Fl:=subs(Ages,r=NSterwZFkt1[1],h=EimerHoehe)
```

$$\pi\cdot\left(\frac{5}{\pi}\right)^{\frac{2}{3}}+\left\{\frac{5}{\pi\cdot\left(\frac{5}{\pi}\right)^{\frac{2}{3}}}\right\}\cdot\left(2\cdot\pi\cdot\sqrt[3]{\frac{5}{\pi}}\right)$$

Die Berechnung der benötigten Blechfläche (ohne Verschnitt) – hier Fl - ist hier eher pro forma, da wir über keine Vergleichswerte verfügen.

```
float(Fl)
```

$$\{12.84747798\}$$

Diese ließen sich aber z.B. in einer Tabelle für leicht kleinere und größere Werte des Radius die passende Höhe, und daraus dann die resultierende Oberfläche berechnen. Sie sollten in jedem Fall größer sein, als der Wert beim optimalen Radius.

Außer dem geänderten Radius (hier rx) müssen wir natürlich auch die neue passende Höhe bestimmen. Als Änderung addieren ich hier 0,01 dm. Nun werden zuerst der neue Radius, die geänderte Volumen-Formel und die passende Höhe ausgegeben. Zum Schluß lassen wir uns noch die Oberfläche berechnen.

```
rx:=NSterwZFkt1[1]+0.01;
Vx:=subs(V=5,r=rx);
hx:=solve(Vx,h);
float(subs(Ages,r=rx,h=hx))
```

$$\sqrt[3]{\frac{5}{\pi}}+0.01$$

$$h\cdot\pi\cdot\left(\sqrt[3]{\frac{5}{\pi}}+0.01\right)^2=5$$

$$\left\{\frac{5}{\pi\cdot\left(\sqrt[3]{\frac{5}{\pi}}+0.01\right)^2}\right\}$$

$$\{12.84841512\}$$

Diese ist erwartungsgemäß größer als die Optimierte (12,84747798 dm²).
Wenn man auf die Zwischenausgaben verzichten möchte, braucht man statt der Semikola nur Doppelpunkte setzen.

In diesem – so geänderten – Aufruf wird nun die Fläche für einen um 0,05 dm kleineren Radius berechnet.

```
rx:=NSterwZFkt1[1]-0.05:
Vx:=subs(V=5,r=rx):
hx:=solve(Vx,h):
float(subs(Ages,r=rx,h=hx))
```

$$\{12.87174271\}$$

Auch diese Fläche ist wiederum größer. Damit haben wir die Extremwertaufgabe gelöst und können einen Antwortsatz od.ä. zurückliefern.
Z.B.:
Um einen minimalen Blechverbrauch beim Bau des 5-Liter-Eimers zu erzielen, muss der Radius rund 1,1675 dm (= 116,75 mm) und die Höhe ebenfalls rund 1,1675 dm (= 116,75 mm) betragen.

Aufgaben:

1. *Prüfen Sie durch Verändern der Eingaben, ob sich auch für andere Volumen (z.B.: 1l, 10l, 50l) immer gleiche Radien und Höhen als optimale Maße des Eimers ergeben!*
2. *Mit der Post dürfen bei mit einem bestimmten Tarif quaderförmige Pakete ins Ausland verschickt werden, wenn Länge, Höhe und Breite zusammen 9 dm nicht überschreiten. Welche Kantenmaße muß man wählen, wenn ein möglichst großes Volumen gewünscht wird und sich Länge zu Breite wie $\sqrt{2}$: 1 verhalten sollen?*
3. *Von einer Versorgungsleitung liegen die zwei Orte A-Dorf und B-Dorf jeweils 10 und 12 km entfernt (kürzester Abstand). Wo muß eine gemeinsame Abzweigungstelle für beide Orte auf der Leitung installiert werden, wenn die Orte 23 km von einander liegen? Wie lang wären die seperaten Versorgungsabzweigungsleitungen zu den Orten? (Für FREAKS: Wie könnte man einen doppelt so großen Verbrauch in A-Dorf mit in die Optimierung einbeziehen?)*
4. *Aus 6 Stangen mit einer Länge von 4 m soll ein Zeltgerüst aufgestellt werden, das die Form einer regelmäßigen Pyramide hat. Wie hoch muß das Gerüst sein, damit das Zeltvolumen ein Maximum annimmt?*
5. *Ein Rechteck mit einem Umfang von 60 cm rotiert um eine seiner Mittelachsen, so dass ein Zylinder entsteht. Welche Maße muß das Rechteck haben, damit das Volumen maximal wird?*
6. *Eine Firma, die das Monopol für Xentatane besitzt, möchte den Preis und die Stückzahl so kalkulieren, dass der Gewinn möglichst groß wird. Bekannt sind folgende anfallende Kosten:*
    - *Fixkosten 40000 Euro*
    - *variable Kosten, die für jedes Xentatan neu anfallen*
        - *9 Euro Herstellungskosten*
        - *25% des Verkaufspreis als Provision für den Verkäufer*
        - *10% des Verkaufspreis als Honorar für den Entwickler*

    Die Firma schätzt, dass sie ab ca 50 Euro ihr Produkt nicht mehr verkaufen kann (Prohibitivpreis) und die Stückzahl (maximal absetzbare Stückzahl), die theoretisch bei einem Preis von 0 Euro abgesetzt werden kann, bei ca 50000 Xentatanen liegt. Ermitteln Sie den Preis und die Stückzahl für einen optimalen Gewinn!

3. Literatur / Quellen

/1/ CREUTZIG, C.; GERHARD, J.; OEVEL, W.; WEHMEIER, S.:
Das MuPAD Tutorium: plattformunabhängige Einführung ab Version 2.0.-2. Aufl.-Berlin; Heidelberg; New-York; Barcelona; Hongkong; London; Mailand; Paris; Tokio: Springer, 2002
ISBN 3-540-43573-5

/2/ GEHRS, Kai; POSTEL, Frank:
Ein Streifzug durch die Physik mit MuPAD: Mathematik 1 x anders: Materialien und Werkzeuge für computerunterstütztes Lernen, Band 1.-Paderborn: SciFace Software GmbH & Co. KG, 2001.
ISBN 3-933764-02-5

/3/ DELL´AERE, Alessandro:
MuPAD – Eine praktische Einführung: Mathematik 1 x anders: Materialien und Werkzeuge für computerunterstütztes Lernen, Band 2.-Paderborn: SciFace Software GmbH & Co. KG, 2001.
ISBN 3-933764-03-3

/4/ Tabellenbuch Chemie.-Leipzig: Dt. Verl f. Grundstoffind.-1980.-8. überarb. Aufl.

/5/ FISCHER, Peter:
BASIC für Anfänger.-1. Aufl.-Berlin: Verl. Die Wirtschaft, 1987
ISBN 3-349-00149-1

/6/ KOCH, Steffen:
Anleitung zum Lösen mathematischer Aufgaben – aus dem Bereich des Mathematikunterrichts an Fachschulen, Volkshochschulen und erweiterten Oberschulen.-5. Aufl.-Leipzig: Fachbuchverl., 1985

/7/ WEBER, Karlheinz; ZILLMER, Wolfgang:
Mathematik – Analysis, Analytische Geometrie, Stochastik: Sekundarstufe II – Grundkurs mit Erweiterungen und Vertiefungen.- 2. Aufl.-Berlin: paetec Gesellschaft für Bildung u. Technik mbH, 1996
Reihe: TCP – Theoria Cum Praxi
ISBN 3-89517-208-1

/8/ MENSE, Johannes, ROHDE, Heinrich, GÖRRIES, Gerd:
Summa 3 – Mathematik (Analysis) für die Sekundarstufe II insbesondere für Fachoberschulen und berufliche Gymnasien.- 1. Aufl.-Stuttgart: E. Klett Verl. f. Wissen u. Bildung, 1991
ISBN 3-12-802110-4

/9/ Tabellen und Formeln – Mathematik, Physik, Chemie.-8. Aufl.-Berlin: Volk u. Wissen Verl., 1980

/10/ SARNOW, Karl:
Mathe-Ass – MuPAD 2.0.-In: Linux-Magazin 06/02 S.90-95

/11/ Internetseiten: www.mupad.de und www.sciface.com

/12/ MOHR, Paul:
Excel: Gleichungssysteme.-IN: PC-Magazin September 1997.-S. 165 f.

/13/ BROSIUS, Gerhard:
Excel 5.0 Professionell : Tabellenkalkulation mit Windows.-Bonn; Paris; Reading, Mass. [u.a.]: Addison-Wesley, 1994 (Edition Software-Klassiker)
ISBN 3-89319-703-6

/14/ SIMON, Hans; STAHL, Kurt:
Nachschlagebücher für Grundlagenfächer – MATHEMATIK.-Leipzig: Fachbuchverl, 1976.-12. Aufl.

/15/ KAPLAN, Robert:
Die Geschichte der Null.-München: Piper Verl., 2004.-3. Aufl.
ISBN 3-492-23918-8

/16/ http://www.mathe.tu-freiberg.de/~hebisch/cafe/fibonacci.html
(genutzt am: 30.08.2004)

/17/ GÖHNER, Hartmut; HAFENBRAK, Bernd:
Arbeitsbuch PROLOG.-Bonn: Dümmler Verl., 1995.-Reihe: Bausteine Informatik
ISBN 3-427-46863-1

/18/ http://www.cs.albany.edu/~mosh

/19/ LÄMMEL, Uwe (Dr. Ing.):
Die logische Programmiersprache PROLOG.-Skript zu Vorlesungen.-Universität Rostock.-1993, 1995

Abbildungen, Skizzen und Schemata entstammen den folgenden Sammlungen bzw. Quellen:

/A/ Software: SciFace Software GmbH: MuPAD Version 2.5 © 2002

oder sind geistiges Eigentum des lern-soft-projekt; (c,p) 1996 – 2004 lsp: dre