

Vorwort

Seit 1995 benutze ich LINUX (damals SUSE 3.X) und in der Folge andere freie Software bzw. sogar Open Source Software. Immer wieder stieß ich in meinem Unterricht auf ein gewisses Unverständnis (wieso sollte man freie Software verwenden, wenn ich Ähnliches auch für Geld bekomme) - sowohl von Seiten meiner SchülerInnen als auch Eltern.

An Hand dieser gesammelten Beispiele will ich zeigen, dass sich mit freier Software durchaus "richtige Mathematik" bzw. Physik betreiben lassen.

Außer die "digitale Kompetenz" (wie es heute so schön heißt) zu fördern, wollte ich auch die Lücke in der mathematischen Literatur zwischen Sekundarstufe II und Universitätsniveau - die meiner Meinung nach herrscht - etwas ausfüllen.

Verwendet wurde *GeogebraDesktop4/5*, *wxMaxima*, *Gnuplot*, *Inkscape*, *LibreOffice-Calc* und *GNU-Octave*. Auch ein bisschen *Python* (Kapitel Gradientenabstieg) wurde verwendet.

Zielgruppe sind Lehrende bzw. Lernende mit dem mathematischen Background der Sekundarstufe II (manchmal kann es auch etwas mehr sein!)

Manche Beispiele sind einfach und im Mathematikunterricht direkt zu gebrauchen - andere wiederum recht ambitioniert und erfordern eine gewisse Einarbeitungszeit.

Ich lege Wert darauf, dass die Programme nicht nur benutzt werden, sondern erklärt wird, wie sie "funktionieren".

Damit reiht sich dieses Werk ein in die Reihe

"Learning *Geogebra* und *wxMaxima* (and other free software) by Example".

An sich sind die meisten Kapitel unabhängig voneinander durchzuarbeiten - das bedingt, dass manche Dinge im Buch mehrfach erklärt werden und sich so eine gewisse Redundanz ergibt (z.B. "Einstein'sche Summationskonvention", Drehmatrizen, usw.). Dies sorgt natürlich für eine gewisse "Dicke" des Buches, aber der Leser braucht nicht "hin- und herspringen".

Viel Spaß beim Durcharbeiten

Inhaltsverzeichnis

1	Rigips Überdeckung	11
1.1	Problemdarstellung	11
1.2	Lösung durch Konstruktion	12
1.3	Lösung durch Rechnung mit <i>wxMaxima</i>	15
1.4	Lösung mit <i>GNU-Octave</i>	16
2	Die harmonische Reihe	19
2.1	Eine unwahrscheinliche Geschichte:	19
2.2	Konvergenz erwingen durch Ausdünnen	23
2.3	Die alternierende harmonische Reihe	24
2.3.1	Experimentalmathematik	24
2.3.2	Ist 0.6931... ein (möglicher) Grenzwert?	27
2.3.3	Anderer Grenzwert gefällig?	28
2.3.4	Der Riemann'sche Umordnungssatz	29
2.4	Umordnungssatz von Gilula	32
2.4.1	Was macht ϕ ?	32
2.4.2	Spezieller Grenzwert $\log(2)$ (bzw. $\ln 2$)	35
2.4.3	Dichtheit der Grenzwerte	35
2.5	Der Kochsalzkristall	38
2.5.1	2 dimensionale Näherung	38
2.5.2	3 dimensionale Näherung	44
2.6	ANHANG	53
3	Zykloiden	55
3.1	Gleiten auf einem Kreis	55
3.2	Kreis rollt auf Kreis - außen: Epizykloide	56
3.2.1	Spezielle Epizykloiden	58
3.3	Kreis rollt auf Kreis - innen: Hypozykloide	59
3.3.1	Spezielle Hypozykloiden	60
3.4	ANHANG: Rotationsmatrizen	61

4	Rotationen - Basics	63
4.1	Matrixmultiplikation	63
4.1.1	Skalare Produkte	63
4.1.2	Linearkombinationen	63
4.2	Rotation um eine beliebige Achse durch den Ursprung	65
4.3	Rotation um eine beliebige Achse außerhalb des Ursprungs(1)	70
4.4	Rotation um eine Achse(2) - Eulerwinkel	72
4.5	Rotation um eine beliebige Achse(3) - Rodrigues	76
4.6	Rotation um eine Achse(4) - Quaternionen	79
4.6.1	Quaternionen in <i>wxMaxima</i>	80
4.6.2	Quaternionen in <i>GNU-Octave</i>	86
4.7	ANHANG	88
4.8	Installation des Quaternion-Pakets	91
4.8.1	GNU-Octave	91
5	Rotation einer Schachtel	93
5.1	Transformation ins Augensystem	93
5.2	Projektion auf das Zeichenblatt	96
5.3	Sichtbarkeit	97
5.4	Berechnung der Transformationsmatrix	98
5.5	Implementation in <i>Geogebra</i>	100
5.6	ANHANG: Aktive und passive lineare Transformationen	105
5.6.1	Lineare Transformationen in verschiedenen Basisdarstellungen	106
5.6.2	Unser eigenes <i>Geogebra</i> -Werkzeug	108
6	Rotationskörper mit <i>Geogebra</i>	109
7	Moivre-Laplace und Stetigkeitskorrektur	113
7.1	Moivre-Laplace Theorem	113
7.2	GNU-Plot	113
7.3	Binomial- und Normalverteilung mit <i>wxMaxima</i>	114
8	Lagrange Calculus	121
8.1	Der Grundgedanke	121
8.2	Ein Beispiel	122
9	Satz von Weierstrass	125
9.1	Funktionsfolgen	125
9.2	Die sup-Norm	133
9.3	Binomialverteilung	133
9.4	Binomialverteilung einer Zufallsvariablen X	135
9.5	Approximationssatz von Weierstrass	137

10 Räuber-Beute Modell	147
10.1 “Brute force”Attacke	147
10.2 Lösung mit Differenzgleichungen	150
10.3 Stationäre Lösung und Richtungsfeld	152
10.3.1 Richtungsfeld in <i>Geogebra</i>	155
10.3.2 Stützpunkte der Phasenraumkurve	159
11 Coriolis-Kraft	161
11.1 Winddrift	161
11.2 Projektilbahn auf rotierender Scheibe	162
11.3 Näherung für die Erde an den Polen	165
11.4 Einfluss der geografischen Breite	167
11.5 Duell auf einer Scheibe	170
11.5.1 Simulation in <i>Geogebra</i>	170
11.5.2 Berechnung in <i>wxMaxima</i>	173
11.6 Duell mit Vorhaltewinkel	174
11.6.1 “Brute force” mit <i>wxMaxima</i>	176
11.6.2 Analytischer Ansatz	181
11.6.3 Fiktive Kräfte im rotierenden Bezugssystem	186
11.6.4 Foucault’sches Pendel	193
11.7 Rollende Kugel auf einer Scheibe	198
12 Sonnenscheindauer	203
12.1 Vereinfachungen	203
12.2 Terminologie und Rechnung	204
12.3 Anhang	207
13 Zeitgleichung	209
13.1 Sterntag vs. Sonnentag	211
13.2 6 wichtige Punkte in der Erdumlaufbahn	212
13.2.1 Erstellung des Arbeitsblattes	213
13.3 Die “mittlere Sonne” (mean sun)	214
13.3.1 Nachbetrachtung zur “mittleren” Sonne	216
13.4 Die fiktive Sonne	217
13.4.1 Berechnung der Tilt-Korrektur	222
13.4.2 Berechnung mit numerischer Integration	225
13.4.3 Berechnung mit Fourierreihe	227
13.4.4 Berechnung mit Potenzreihe	230
13.4.5 Simulation mit <i>wxMaxima</i>	233
13.5 Position der wahren Sonne $\psi(t)$ - Keplerkorrektur	237
13.5.1 Berechnung mit <i>wxMaxima</i>	239
13.5.2 Simulation mit <i>Geogebra</i>	241
13.6 Kepler- und Schiefstellungskorrektur	242
13.6.1 Mit <i>Geogebra</i> als Simulation	242

13.6.2	Zeitgleichung mit <i>wxMaxima</i>	244
13.6.3	Addition der Fehler - eine Analyse	246
13.6.4	Mit <i>Geogebra</i> als Rechnung mit Datum	248
13.7	Analemma	252
13.7.1	Positionsbestimmung eines Himmelsobjekts auf der Erde	252
13.7.2	Analemma	256
13.8	Lichter Tag - "Wiederaufnahme des Verfahrens"	258
13.9	Sonnenauf- bzw. Sonnenuntergang	260
13.10	ANHANG1: Datenlisten von <i>wxMaxima</i> nach <i>Geogebra</i>	261
13.10.1	Mit dem Clipboard und einem Editor	261
13.10.2	Export von <i>wxMaxima</i>	261
13.10.3	Import in <i>Geogebra</i>	263
13.11	ANHANG2: Integrale von Sinus-Potenzen	264
14	Erdbahngleichung	265
14.1	Qualitative Überlegungen	265
14.2	Numerisch: Runge-Kutta	267
14.3	Numerisch: Integrieren mit Taylor-Näherung	268
14.4	Versuchs doch mal mit Reihen!	270
14.5	Alternative Reihenformel nach Stenlund	276
14.6	Näherungsterm für $\psi(t)$	279
14.7	Alternative Herleitung von $\psi(t)$	281
14.8	Historische Variante von Kepler	283
14.9	Implemtation in <i>Geogebra</i>	288
14.9.1	Die Konstruktion	289
14.10	ANHANG1: Keplergleichung mit Integralrechnung	291
14.10.1	Ellipsenabschnitt	291
14.10.2	Keplerproblem	291
14.11	ANHANG2: Das 2. Gesetz Kepler's - "der Flächensatz"	293
14.12	ANHANG3: $(1+x)^{-2}$ als Reihe für $0 \leq x \ll 1$	294
14.13	ANHANG4: $(1-x)^{\frac{3}{2}}$ als Reihe für $0 \leq x \ll 1$	295
15	Rotationskörper im Wasser	299
15.1	Wasserspiegel	299
15.1.1	Die Rechnung	300
15.1.2	3D Simulation als Zugabe	302
15.1.3	Der Graph $w(e)$	303
15.1.4	Konvergiert die Iteration von G_e ?	306
15.1.5	Wir haben keinen Funktionsterm	307
16	Spline-Interpolation	313
16.1	Kubische Splines	314
16.2	Tridiagonal-Matrix Algorithmus(TDMA)	317
16.3	Implementierung in <i>Geogebra (Javascript)</i>	319

16.4	Gegencheck mit <i>wxMaxima</i> (native invert)	322
16.5	Implementierung in <i>Geogebra</i> (native invert)	326
16.6	Zerlegung in LU-Matrizen	328
16.6.1	Rekursion in <i>Geogebra</i>	329
16.7	Spline in <i>Geogebra</i> (LU-Zerlegung)	329
16.8	Erstellen eines Custom Tool in <i>Geogebra</i>	331
16.8.1	2 Verwendungsmöglichkeiten	332
16.9	Theoretische Überlegungen	333
16.9.1	Thomas Algorithmus (TDMA)	333
16.9.2	LU-Faktorisierung	333
17	Computertomografie-Basics	335
17.1	Einleitung	335
17.2	Physikalische Grundlagen	336
17.3	Geometrische Grundlagen	337
17.3.1	Arbeitsmodell	337
17.3.2	Kein Treffer bei $0 \leq \theta \leq 90^\circ$	338
17.3.3	Einfallswinkel $\theta \leq 45^\circ$	339
17.3.4	Einfallswinkel $45^\circ \leq \theta \leq 90^\circ$	339
17.3.5	Kein Treffer bei $90^\circ \leq \theta \leq 180^\circ$	340
17.3.6	Einfallswinkel $90^\circ \leq \theta \leq 135^\circ$	340
17.3.7	Einfallswinkel $135^\circ \leq \theta \leq 180^\circ$	340
17.4	Mathematische Grundlagen	340
17.4.1	Rotationen in der Ebene	340
17.4.2	Mathematik der linearen Gleichungssysteme - LA	340
18	QR Faktorisierung	349
18.1	Die Householder-Transformation	349
18.2	Implementation in <i>wxMaxima</i>	356
18.3	Das lineare Ausgleichsproblem	361
18.4	ANHANG Backtracking	362
18.5	Givens-Rotationen	368
18.5.1	Rotationen in 3D in einer Ebene	368
18.5.2	Rotationen im \mathbb{R}^n in der Ebene aufgespannt durch i, k ($i < k$)	368
19	Der schiefe Wurf	375
19.1	Ohne Luftwiderstand auf der Erdoberfläche	377
19.1.1	Mit Differentialgleichung und <i>wxMaxima</i>	377
19.1.2	Mit Differenzgleichung und <i>wxMaxima</i>	379
19.1.3	Mit LibreOffice-Calc	381
19.1.4	Mit Hand und Kettenregel	382
19.2	Luftwiderstand $\propto \vec{v} $	382
19.2.1	Mit Hand und Variablentransformation	382
19.3	Luftwiderstand $\propto \vec{v} ^2$	386

19.3.1	Mit <i>wxMaxima</i> 1 Bahnkurve	387
19.3.2	Mit <i>wxMaxima</i> mehrere Bahnkurven	388
19.3.3	Mit <i>Geogebra</i> mehrere Bahnkurven	390
19.3.4	Mit Runge-Kutta Verfahren	394
19.3.5	Ermittlung der Bahnkurvengleichung (dimensionslos)	396
19.3.6	Lösung der short-time Näherung im Ortsraum	397
19.3.7	Herleitung und Lösung der short-time Näherung im Zeitraum	400
19.3.8	Herleitung einer “long-time”-Näherung im Zeitraum	403
19.3.9	Implizite Lösung der Bahnkurvengleichung im Zeitraum	408
19.3.10	Allgemeine Lösung der Bahnkurvengleichung im Ortsraum	414
19.3.11	ANHANG: dimensionslose Variablen	420
19.3.12	ANHANG: asymptotische Taylorreihe	421
19.3.13	ANHANG: Invertierung einer Funktion mit Runge-Kutta	422
19.3.14	ANHANG: Lineare Differentialgleichung erster Ordnung	425
20	Gradientenabstieg	429
20.1	Partielles Differenzieren	429
20.2	Eigenschaften des Gradienten	430
20.3	Die Kettenregel	432
20.4	Höhenlinien und Gradient	432
20.5	Gradientenabstieg - gradient descent	436
20.5.1	Erweiterte Kettenregel	441
20.5.2	Erweiterte Kettenregel im Einsatz	442
20.6	Implementation in Python	443
21	Hough-Transformation	449
21.1	Problembeschreibung	449
21.2	Hough Transformation	454
21.2.1	Spezielle Geradendarstellung	454
21.3	NACHTRAG	458
22	Spezielle Relativitätstheorie (SRT)	461
22.1	Postulate der Speziellen Relativitätstheorie	461
22.2	Neue Einheiten	464
22.3	Lichtuhr	464
22.4	Zweidimensionale Raumzeit	466
22.4.1	Zeitdilatation mit neuer Nomenklatur	466
22.4.2	Weltlinien	467
22.4.3	Lorentztransformation	468
22.4.4	Veranschaulichung	469
22.4.5	Einheiten in S'	471
22.4.6	Zeitdilatation im Minkowski-Diagramm	475
22.4.7	Zeitdilatation andersrum - Längenkontraktion	476
22.5	Visualisierung des Zwillingsparadoxons	477

22.6 ANHANG1: Algebraische Eigenschaften von Λ	482
22.7 ANHANG3: Vektoren	485

1 | Rigips Überdeckung

1.1 Problemdarstellung

Ein Liniengitter wird von beinahe(!) senkrechten nicht parallelen Linien geschnitten. Die entsprechenden Trapeze sind aus rechteckigen Platten zuzuschneiden, damit sie überdeckt werden. Was ist wegzuschneiden?

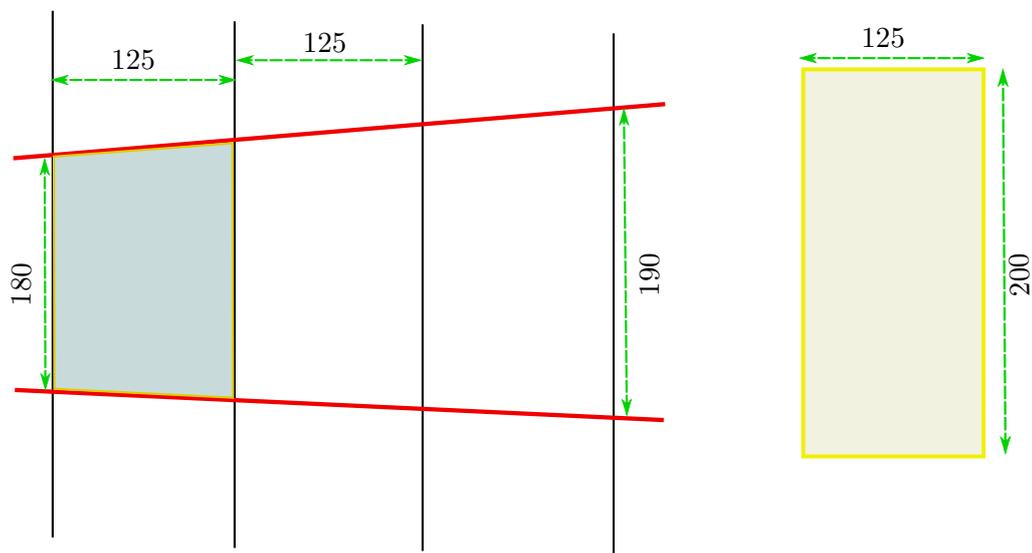


Abb.1 : Rigips-Überdeckung

Wir schlagen hier eine Lösung durch Konstruktion mit *Geogebra* vor als auch eine Übung durch Rechnung mit *wxMaxima* bzw. *GNU-Octave*. Jede dient als Probe für die andere Methode.

1. Rigips Überdeckung

1.2 Lösung durch Konstruktion

Wir benötigen 4 Angabestücke um ein Trapez zu bestimmen, die Höhe h ist hier fix vorgegeben. Wir entscheiden uns für die Länge der parallelen Seiten a und c und eine Diagonale e (z.B. links oben nach rechts unten) - die sind am leichtesten zu messen. Damit fällt die Konstruktion leicht:

1. Wir wählen auf dem Liniengitter einen Punkt A
2. Auf derselben Linie wählen wir B mit der Entfernung a .
3. Von B aus schlagen wir e auf die nächste Linie ab - der untere Schnittpunkt ist D.
4. Von D aus gehen wir c nach oben (Siehe Abb. 2)

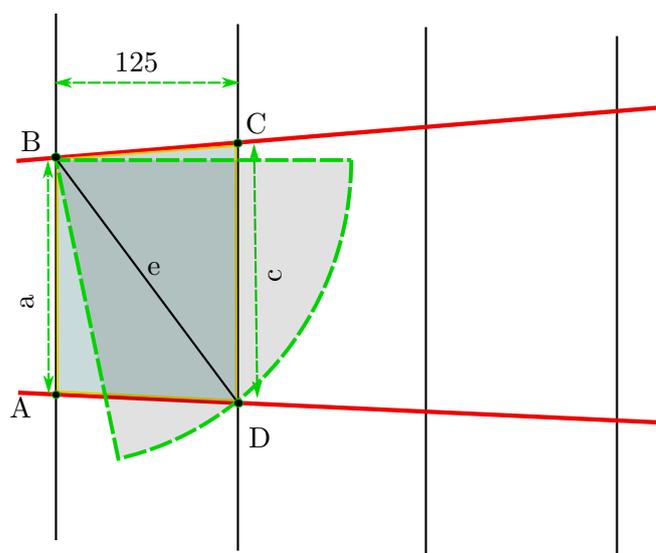


Abb.2 : Rigips-Konstruktion

Diese Konstruktion setzen wir in *Geogebra* um, wobei wir zuallererst das Koordinatensystem "einrichten":

bei gedrückter Shift-Taste mit der linken Maustaste die x-Achse "stauchen" anschl. auf das Zeichenblatt mit rechter Maustaste klicken und im aufpoppenden Kontextmenü den Eintrag *xAchse:yAchse* auswählen und da wiederum *1:1* auswählen. Sonst wird aus dem Konstruktionkreis eine Ellipse.

Das fertige Produkt kann man von <https://www.geogebra.org/m/SqemDbzE> herunterladen.

1.2 Lösung durch Konstruktion

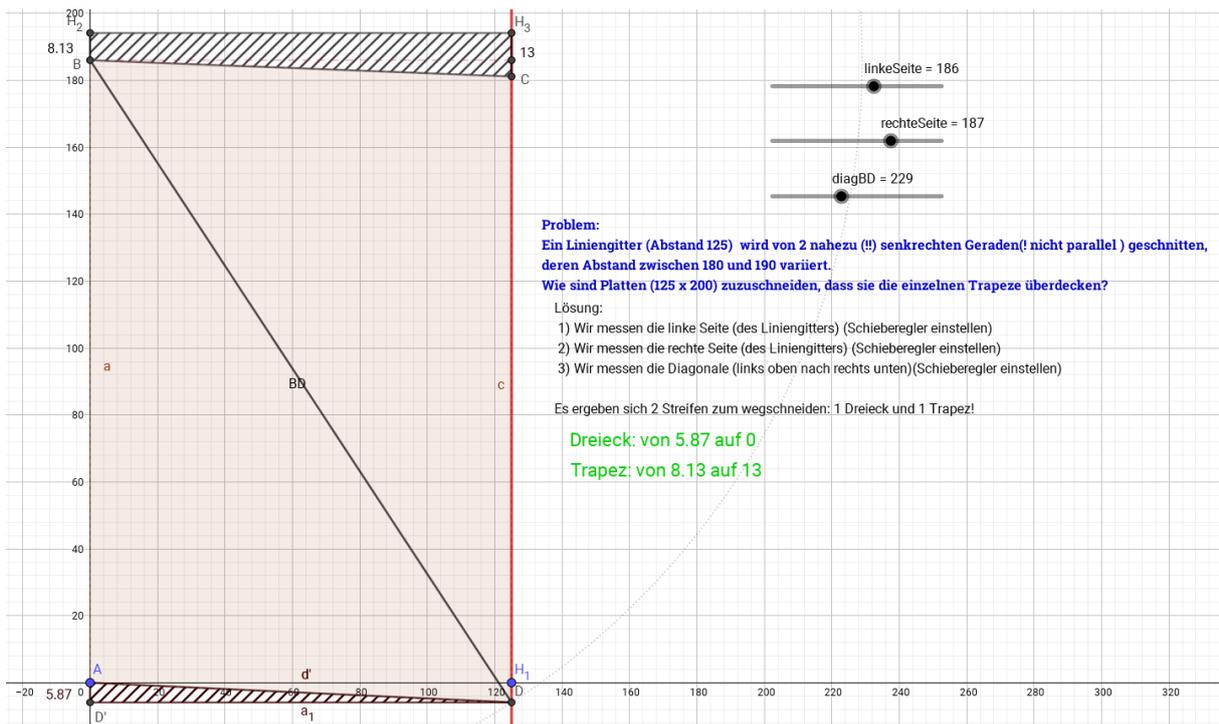


Abb.3 : Geogebra-Arbeitsblatt auf <https://www.geogebra.org/m/SqemDbzE>

Konstruktionsprotokoll		
N	Spalten	Wert
1	Punkt A	$A = (0, 0)$
2	Zahl linkeSeite	linkeSeite = 186
3	Punkt B	$B = (0, 186)$
4	Punkt $H_1(125, 0)$	$H_1 = (125, 0)$
5	Gerade f	Gerade durch H_1 , parallel zu yAchse $f: x = 125$
6	Zahl diagBD	diagBD = 229
7	Kreis k_1	Kreis mit Mittelpunkt B und Radius diagBD $k_1: x^2 + (y - 186)^2 = 52441$
8	Punkt D	Schnittpfad von k_1, f $D = (125, -5.87)$
9	Strecke BD	Strecke B, D $BD = 229$
10	Zahl rechteSeite	rechteSeite = 187
11	Punkt C	$(x(D), y(D) + rechteSeite)$ $C = (125, 181.13)$
12	Viereck Platte	Polygon A, B, C, D Platte = 23312.5

Abb.4 : Geogebra-Konstruktionsprotokoll

1. Rigips Überdeckung

Damit ist zwar das Trapez $ABCD$ festgelegt - allerdings um die "Abschnitte" festzulegen brauchen wir einige Hilfspunkte (H_1 ist in Abb.4 bereits festgelegt).

Übrigens haben wir den zweiten Schnittpunkt von k_1 mit der Geraden auf der c liegt verworfen - die Diagonale würde dann aufwärts statt abwärts gehen und dieses Trapez würde sicher nicht auf unsere Rigipsplatte passen. Der Befehl dafür lautet `Schneide(k_1, f, 1)` - wobei 1 der Index des Schnittpunktes ist!

Die schraffierten schwarzen Abschnittspolygone sind ADD' bzw. BH_2H_3C . Mit folgenden Festlegungen:

- (1) D' : Wenn($y(D) < 0$, ($x(A)$, $y(D)$), (125, 0))
- (2) H_2 : Wenn($y(D) < 0$, $D' + (0, 200)$, (0, 200))
- (3) H_3 : Wenn($y(D) < 0$, $D + (0, 200)$, (125, 200))
- (4) C' : $B + (125, 0)$

1. Solange D unterhalb der x-Achse liegt, liegt D' unterhalb von A (linke untere Plattenecke) andernfalls markiert D' die rechte untere Plattenecke (125, 0)
2. H_2 ist obere linke Plattenbegrenzung
3. H_3 ist obere rechte Plattenbegrenzung (Beachte: H_2 und H_3 haben dieselbe y-Koordinate wegen $y(D) = y(D')$)
Die Platte kann das Trapez überdecken, falls $y(B) \leq y(H_2) \Leftrightarrow y(C') \leq y(H_3)$
4. C' ist der gespiegelte Punkt B

Wir definieren eine Strecke $g : BH_2$ und $h : CH_3$ und zeigen den Text

Trapez: von g auf h (statt g und h zu schreiben Objekte auswählen!) nur unter der Bedingung $y(H_2) > y(B) \wedge y(H_3) > y(C)$ an!

Andernfalls zeigen wir den Text "Platte ist zu klein!"

Das untere Dreieck flippt mit der Spitze auf die andere Seite falls $y(D)$ die x-Achse quert. Wir definieren uns zwei Hilfsgrößen z_1 und z_2 mit

- z_1 : Wenn($y(D) > 0$, 0, Abstand(A , D'))
 z_2 : Wenn($y(D) < 0$, 0, Abstand(D , D'))

Und zeigen unter der Bedingung $y(H_2) > y(B) \wedge y(H_3) > y(C)$ den Text an:

Dreieck: von z_1 auf z_2 (wieder Objekte einfügen!)

Damit ist unser Arbeitsblatt einsatzfähig - auf geht's zur Baustelle und dann heißt es: "messen und schneiden"!

1.3 Lösung durch Rechnung mit *wxMaxima*

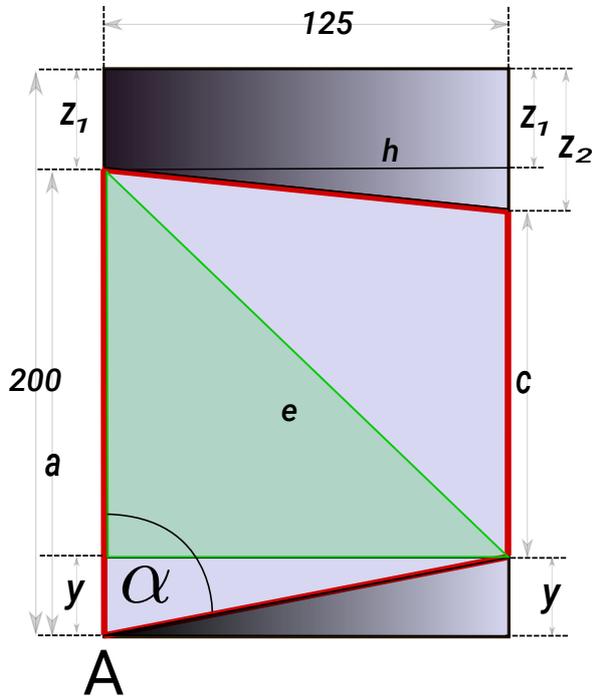


Abb.5 : y ist positiv

Die zugeschnittene Platte ist rot umrandet. Abfall ist das untere schwarze Dreieck und das obere schwarze Trapez!

Das grüne rechtwinkelige Dreieck ist ein guter Ansatzpunkt - daraus ergibt sich die monoton fallende Funktion $y(e)$:

$$y : [h, \infty[\rightarrow] - \infty, a]$$

$$y(e) = a - \sqrt{e^2 - h^2}$$

Ist $y > 0$ dann ist α spitz (Abb. 5), sonst ein stumpfer Winkel (Abb.6). Also A ist linke untere Ecke der Platte oder D ist rechte untere Ecke der Platte!

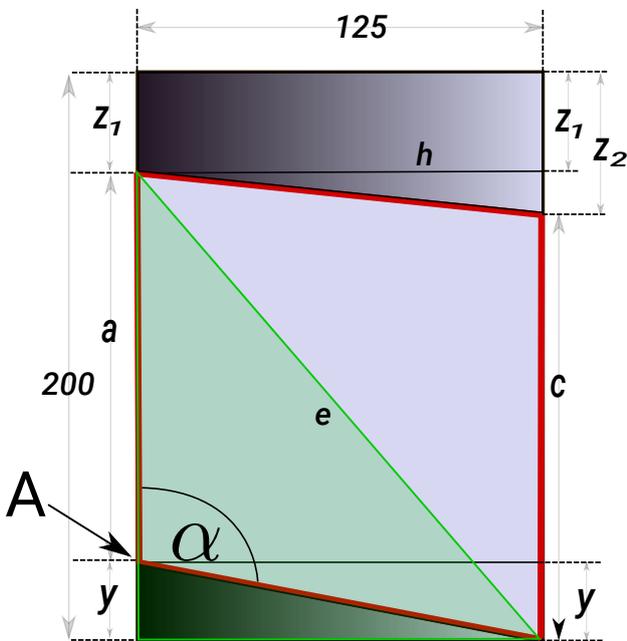


Abb.6 : y ist negativ

Dies führt zu folgender Fallunterscheidung:

1. $y \geq 0$: (Abschnitt rechts unten)
 $z_1 = 200 - a \quad z_2 = 200 - (c + y)$
2. $y < 0$: (Abschnitt links unten)
 $z_1 = 200 - (a - y) \quad z_2 = 200 - c$

z_1 bzw. z_2 sind die Abschnitte links oben bzw, rechts oben - werden sie negativ, ist die Platte zu klein!

1. Rigips Überdeckung

(%i6) ratprint:false\$ fpprintprec : 3 \$ a:186\$c:187\$h:125\$e:229\$

```
(%i7) get_cuts():=block([y,z_1,z_2],
  y: a - sqrt(e\^{2} - h\^{2}),
  if (y >= 0) then block (
    disp("y rechts unten"),
    z_1:200 -a,
    z_2:200 -(c+y))
  else block(
    disp("y links unten"),
    z_1:200 -(a-y),
    z_2:200 -c),
  display(y,z_1,z_2),
  if ((z_1 < 0) or (z_2 < 0)) then disp("Platte zu klein!")
  else disp("")
)\$
```

(%i8) get_cuts(),numer\$

```
y links unten
y = -5.87
z_1 = 8.13
z_2 = 13
```

1.4 Lösung mit *GNU-Octave*

Wir benutzen Abb. 1.1 und legen ein Koordinatensystem(KS) mit Ursprung in B . A hat dann die Koordinaten $A = (0, -a)$. $D = (h, -\sqrt{e^2 - h^2})$, $C = (h, c - \sqrt{e^2 - h^2})$ mit $h = 125$.

Jetzt verschieben wir das KS nach unten, sodass entweder A oder D eine untere Ecke wird - je nachdem, ob a oder $\sqrt{e^2 - h^2}$ größer ist.

Hier jetzt das Octave-Skript (Beachte: Ein Skript darf nicht mit "function" beginnen - sonst liegt eine Funktionsdefinition vor!):

```
1 #minimale/maximale x- bzw. y-Werte; gleich lange Einheiten in x und y
  axis([0, 150, 0, 200], "equal");
3 #h,H Plattenparameter, und Angaben
  a=186; c=187; e=229; h=125;H=200;
5 #Eckpunkte der Platte
  E1=[0,0]; E2=[0,H]; E3=[h,H]; E4=[h,0];
```

```

7 #Loesungen der Eckpunkte mit B als Ursprung
A=[0,-a]; B=[0,0]; C=[h, c-sqrt(e^2-h^2)];
9 D=[h,-sqrt(e^2-h^2)];

11 # Verschiebung um a oder sqrt(e^2-h^2) - was ist groesser?
function yTranslation=getYTranslation(a,e,h)
13   if (a > sqrt(e^2-h^2))
       yTranslation=a;
15   else
       yTranslation=sqrt(e^2-h^2);
17   endif
endfunction

19 #Ausfuehren der Translation A -> A_p ("A primed")
21 function [A_p,B_p,C_p,D_p]=translate(A,B,C,D,vec)
       A_p=A+vec; B_p=B+vec; C_p=C+vec; D_p=D+vec;
23 endfunction

25 #Punkt mit Farbe c zeichnen und bezeichnen, Bezeichnung hat Offset-Vektor
function plotPoint(p,c,letter,offset)
27   plot(p(1),p(2),"o","markersize",12,"markerfacecolor",c);
       text(p(1)+offset(1),p(2)+offset(2),letter,"fontsize",20);
29   hold on;
endfunction

31 # Liste von Punkten wird gezeichnet
33 function plotPoints(pList, cList, letterList, offsetList)
       for i=1:length(pList)
35         plotPoint(pList{i},cList{i}, letterList{i},offsetList{i});
       endfor
37 endfunction

39 ##### MAIN #####
yTranslation=getYTranslation(a,e,h);
41 [A_p,B_p,C_p,D_p]=translate(A,B,C,D,[0,yTranslation]);
pointList={A_p,B_p,C_p,D_p};
43 printf ("Die Eckpunkte\n");
printf ("[ %d , %d ]\n", pointList{:});
45 printf ("links-unten: %d cm auf rechts-unten %d cm\n", A_p(2),D_p(2));
printf ("links-oben: %d cm auf rechts-oben %d cm \n", 200-B_p(2), 200-C_p(2));
47

49 boardHandle=rectangle("Position",[0,0,h,H]);
colorBoard=[0.95,0.95,0.95];
51 set(boardHandle, "Facecolor",colorBoard);
hold on;

53 plotPoints(pointList,{ "b", "b", "b", "b"}, {"A", "B", "C", "D"},
           {[ -5,0],[ -5,0],[ 2,0],[ 2,0]});

55 #Matrix - wobei die Spaltenvektoren die einzelnen Punkte sind
57 linePoints=[A_p;D_p;E4;E1;A_p];
# durch transponieren 1. Zeile x-Werte, 2. Zeile y-Werte der Punkte
59 lowerSegment=linePoints';

```

1. Rigips Überdeckung

```
lh1=line(lowerSegment(1,:),lowerSegment(2,:));
61 set(lh1,"color",[1,0,0],"linestyle","-","linewidth",3);
fill(lowerSegment(1,:),lowerSegment(2,),"r");
63
linePoints=[B_p;E2;E3;C_p;B_p];
65 upperSegment=linePoints';
lh2=line(upperSegment(1,:),upperSegment(2,:));
67 set(lh2,"color",[0,1,0],"linestyle","-","linewidth",3);
fill(upperSegment(1,:),upperSegment(2,),"g");
69
71 th2=title("Rigipsplatte");
set(th2,"fontsize",20);
```

Hier der Output des Skripts:

```
>> rigips
Die Eckpunkte
[ 0 , 5.87496 ]
[ 0 , 191.875 ]
[ 125 , 187 ]
[ 125 , 0 ]
links-unten: 5.87496 cm auf rechts-unten 0 cm
links-oben: 8.12504 cm auf rechts-oben 13 cm
>> |
```

Abb.7 : Octave Output

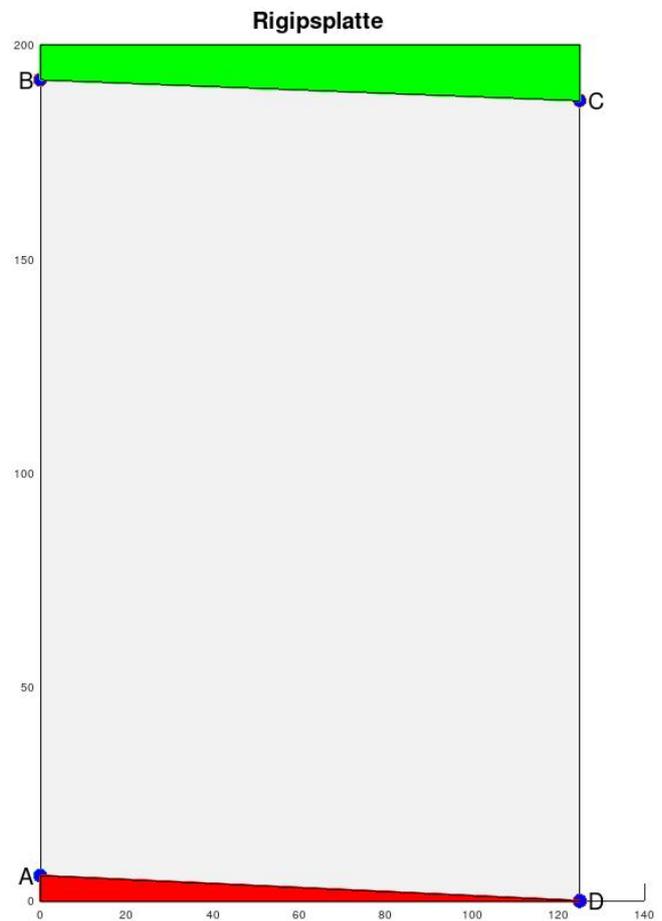


Abb.8 : Octave Figure

2 | Die harmonische Reihe

2.1 Eine unwahrscheinliche Geschichte:

Eine Raupe stößt auf ein 1 km langes Gummiband. Die Raupe selbst hat eine Geschwindigkeit von 1 cm pro Sekunde. Sie versucht das Gummiband zu "überqueren" - leider dehnt ein "böser Geist" das Gummiband pro Sekunde um 1 km. Kann es die Raupe trotzdem schaffen das andere Ende zu erreichen?

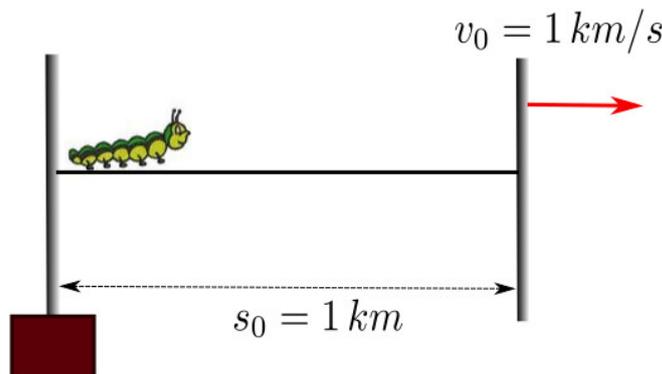


Abb.9 : "... auf dem langen Marsch"

Lösung: Wir vereinfachen den Vorgang des Dehnens: Die Raupe krabbelt 1 Sekunde - dann wird instantan gedehnt! Die Raupe schafft also in der ersten Sekunde 1 cm von 100 000. In der zweiten Sekunde 1 cm von 200 000 cm, usw. also

$$\frac{1}{100000} + \frac{1}{200000} + \frac{1}{300000} + \dots = 1 \Leftrightarrow \frac{1}{100000} \sum_{i=1}^n \frac{1}{i} = 1 \Leftrightarrow \boxed{\sum_{i=1}^n \frac{1}{i} = 10^5}$$

Die letzte Gleichung kann nur dann eine Lösung für n haben, wenn die obige Reihe divergiert oder einen größeren Grenzwert als 10^5 hat! Wir spielen mit einem CAS wie z.B. *wxMaxima* ein bisschen mit der Summe:

`sum(1/i, i, 1, 10^6), numer;` liefert 14.39272672286499

Kann alles bedeuten - was nun? Theorie muss her!

2. Die harmonische Reihe

Definition 2.1 Harmonische Reihe

$\sum_{i=1}^{\infty} \frac{1}{i}$ heißt harmonische Reihe $H_n := \sum_{i=1}^n \frac{1}{i}$ heißt n -te harmonische Zahl

Theorem 2.2 Divergenz der harmonischen Reihe

$$\forall S \in \mathbb{R} \quad \exists N \in \mathbb{N} : \sum_{k=1}^N \frac{1}{k} > S \quad (2.1)$$

Beweis: Wir verwenden die “reductio ad absurdum” mit der Annahme

$$\lim_{n \rightarrow \infty} H_n = H = \lim_{n \rightarrow \infty} H_{2n} \quad \Rightarrow \quad \lim_{n \rightarrow \infty} H_{2n} - \lim_{n \rightarrow \infty} H_n = H - H = 0 \quad (2.2)$$

Andererseits gilt aber auch

$$H_{2n} - H_n = \frac{1}{n+1} + \frac{1}{n+2} + \frac{1}{n+3} + \dots + \frac{1}{2n} > n \frac{1}{2n} = \frac{1}{2}$$

dies ist ein Widerspruch zu 2.2! Die Annahme die Partialsummen hätten einen Grenzwert muss also falsch sein! \square

Unser “Raupenproblem” besitzt also eine Lösung! Wir suchen die kleinste obere Schranke (sup) der Menge L in \mathbb{R} :

$$L := \left\{ x \mid \sum_{i=1}^x \frac{1}{i} < 100000 \right\}$$

Wie aber finden wir dieses Supremum?

Wir stellen $\sum_{i=1}^n \frac{1}{i}$ als Histogramm dar - uns interessiert also der Flächeninhalt **aller** Rechtecke!

Aber diese Rechtecke sind ja Obersumme bzw. Untersumme von Funktionen. Sie sind von $f(x) = \frac{1}{x}$ die Obersumme und von $g(x) = f(x-1) = \frac{1}{x-1}$ die Untersumme. Leicht kann man sich in *Geogebra* davon überzeugen! Die ersten paar Reihenglieder können wir summieren, um den Fehler klein zu halten (hier in der Zeichnung die ersten 3 - aber natürlich werden wir diese Zahl dann größer wählen!)

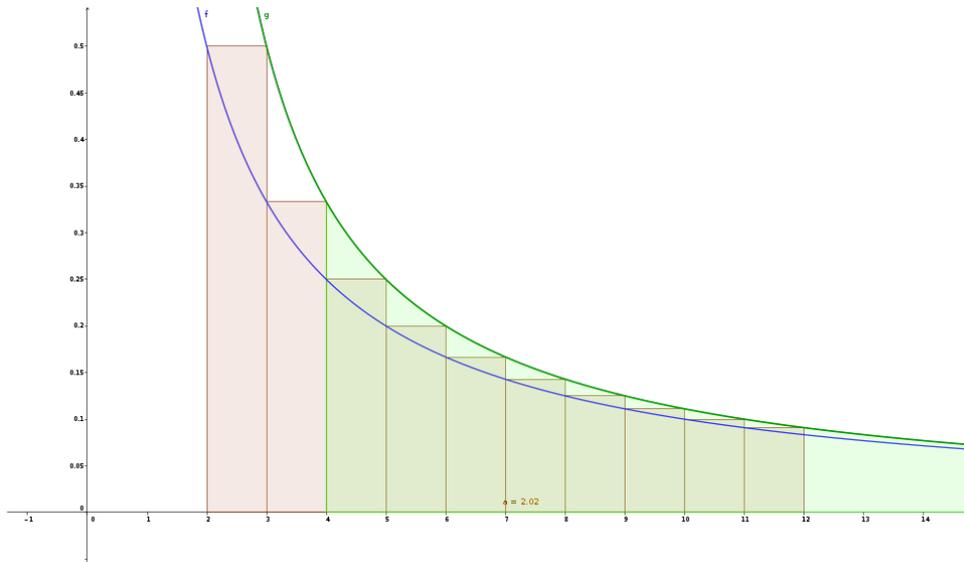


Abb.10 : Ober- und Untersumme

geogebra: $f(x)=1/x$, $g(x)=f(x-1)$, `Obersumme[f, 2, 10, 8]`

$$1 + \frac{1}{2} + \frac{1}{3} + \int_4^N f(x) dx < \sum_{i=1}^N \frac{1}{i} < 1 + \frac{1}{2} + \frac{1}{3} + \int_4^N g(x) dx$$

Statt den ersten 3 Reihengliedern können wir die ersten N_0 nehmen und mit $g(x) = f(x - 1)$ ergibt sich

$$\sum_{i=1}^{N_0} \frac{1}{i} + \int_{N_0+1}^N f(x) dx < \sum_{i=1}^N \frac{1}{i} < \sum_{i=1}^{N_0} \frac{1}{i} + \int_{N_0+2}^{N+1} f(x) dx$$

Einsetzen der Stammfunktion und für $N_0 = 10^6$ (weil da haben wir das Ergebnis schon berechnet) ergibt:

$$14,3927\dots + \ln N - \ln(N_0 + 1) < \sum_{i=1}^N \frac{1}{i} < 14,3927\dots + \ln(N + 1) - \ln(N_0 + 2)$$

Eingesetzt für die Logarithmen ergibt

$$\underbrace{0,5772151649012152}_a + \ln N < \sum_{i=1}^N \frac{1}{i} < \underbrace{0,5772141649027152}_b + \ln(N + 1) \quad (2.3)$$

Unsere Näherungen eingesetzt führt auf die Gleichungen

$$\ln N = 10^5 - a \quad \text{bzw.} \quad \ln(N + 1) = 10^5 - b \quad (2.4)$$

2. Die harmonische Reihe



2.3 kann auch gedeutet werden als $\lim_{N \rightarrow \infty} \left[\sum_{i=1}^N \frac{1}{i} - \ln N \right] = 0,57721 \dots = \gamma$. Diese

Zahl wird als *Euler-Mascheroni Konstante* bezeichnet - sie kommt in Verbindung mit der Digamma-Funktion und der Riemannschen Zetafunktion vor.

Diesen Sachverhalt wissend, hätten wir gleich bei 2.4 starten können: $\ln N \approx 10^5 - \gamma$

In Anbetracht der Größe der Zahl können wir a und b (bzw. γ) vernachlässigen:

$$N \approx \exp(10^5)$$

Um uns besser vorstellen zu können, wo die Zahl im Dezimalsystem zu finden ist, logarithmieren wir mit dem Zehnerlogarithmus

$$\lg N \approx 10^5 \cdot \lg e \approx \frac{10^5}{\ln 10} \approx 43430$$

also

$$N \approx 10^{43430}$$

Nach N Sekunden erreicht die Raupe das Ziel - doch was sollen wir uns unter dieser Zeitspanne vorstellen? Vergleichen wir das mit dem Alter unseres Universums A: 20 Mrd. Erdjahre - rechnen wir in Sekunden um :

$$A \approx 20 \cdot 10^9 \cdot 365,249 \cdot 24 \cdot 3600 \approx 6,3 \cdot 10^{17}$$

Also ein wirklich langer Marsch!

So kurz sieht's in *wxMaxima* aus:

(% i1) `assume(N>1);`

`[N > 1]`

(% i2) `F(N):=integrate(1/x,x,1,N);`

$$F(N) := \int_1^N \frac{1}{x} dx$$

(% i3) `eq:F(N)=10^5;`

$$\log(N) = 100000$$

(% i4) `solve(eq,N);`

$$[N = \%e^{100000}]$$



In *GNU-Octave* (ohne *symbolic package*) ist die Gleichung $\int_1^N \frac{1}{x} dx = 10^5$ für N nicht lösbar, da N größer ist, als die größte darstellbare Zahl - bei meinem Computer e^{308} !

Überprüfe bei deiner *GNU-Octave* -Version:

```
>> realmax
ans = 1.7977e+308
>>
```

2.2 Konvergenz erwingen durch Ausdünnen

(G.H. Behforooz, “Thinning out the Harmonic Series”, Math. Mag., vol. 68, number 4, October 1995)

Was geschieht, wenn wir alle Reihenglieder weglassen, in denen die Ziffer 9 vorkommt?

$$S_9 := 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{8} + \frac{1}{10} + \frac{1}{11} + \dots + \frac{1}{18} + \frac{1}{20} + \frac{1}{21} + \dots + \frac{1}{88} + \frac{1}{100} + \frac{1}{101} + \dots \quad (2.5)$$

Es mag Sie überraschen, aber diese Reihe konvergiert!

Theorem 2.3 Harmonische Reihe ohne Ziffer 9

Die harmonische Reihe ohne Ziffer 9 (wie 2.5) **konvergiert**

Beweis: Wir gruppieren die Reihenglieder nach Anzahl der Ziffern im Nenner:

- Gruppe 1: $1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{8} < 9 \cdot 1$
- Gruppe 2: Wieviel verschiedene 2-er Gruppen der Zahlen $0 \dots 8$ gibt es? 9^2
Nicht alle kommen bei uns vor: z.B.: $1/00$ oder $1/08$ und sie sind kleiner gleich $1/10$,
daraus folgt $9^2 \frac{1}{10}$ ist eine obere Schranke für die 2-er Gruppe!
- Gruppe 3: Es gibt weniger als 9^3 und sie sind kleiner $1/100$
- Gruppe n : Es gibt weniger als 9^n und sie sind kleiner als 10^{n-1}

daraus folgt

$$S_9 < \sum_{n=1}^{\infty} \frac{9^n}{10^{n-1}} = 9 \underbrace{\sum_{n=0}^{\infty} \left(\frac{9}{10}\right)^n}_g$$

Jetzt ist g eine geometrische Reihe mit dem Grenzwert 10 , also muss S_9 zu einem Grenzwert kleiner 90 konvergieren! \square

Statt der Ziffer 9 kann man übrigens jede andere Ziffer nehmen, ohne dass wir an unserer Argumentation etwas ändern müßten.

2.3 Die alternierende harmonische Reihe

$$\sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{k} = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} \pm \dots = ?$$

2.3.1 Experimentalmathematik

Wir schauen uns die Partialsummenfolge $a_n = \sum_{k=1}^n \frac{(-1)^{k+1}}{k}$ der Reihe an, um einen ersten Eindruck zu erhalten.

Mit Geogebra

Wir basteln uns gleich die Folge der Punkte $P_n := (n, a_n)$ mit dem Befehl

```
l1:IterationList( (x(P) + 1, y(P) + (-1)^x(P) / (x(P) + 1)), P, {(1, 1)}, 999)
```

Die Variable ist der Punkt P wird mit $(1, 1)$ initialisiert, also $P_1 = (1, 1)$

$$P_2 = (1 + 1, 1 + (-1)^1/(1 + 1)) = (2, 1 - 1/2)$$

$$P_3 = (2 + 1, 1 - 1/2 + (-1)^2/(2 + 1)) = (2, 1 - 1/2 + 1/3)$$

$$\text{also } P_n = \left(n, \sum_{k=1}^n \frac{(-1)^{k+1}}{k}\right)$$

Die Iteration wird 999-mal wiederholt, mit `Last(l1)` bekommen wir `{(1000, 0.69265)}`!
Erster Eindruck: Schaut nach Konvergenz aus - was ist das für eine Zahl 0.69265... ?

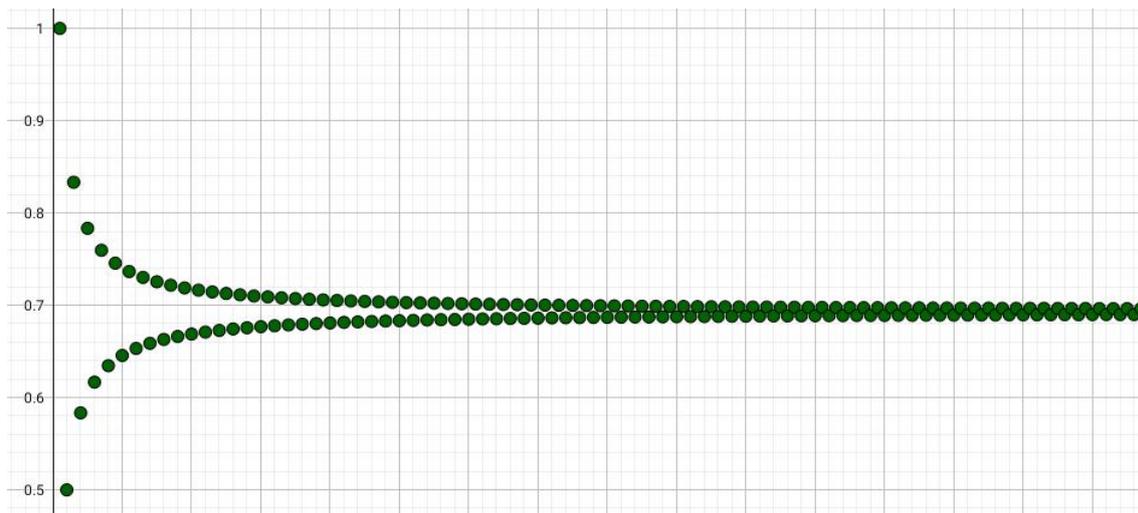


Abb.11 : Was macht die alternierende harmonische Reihe?

Mit *wxMaxima*

Hier präsentieren wir 3 von vielen Möglichkeiten:

- Wir setzen die Vorschrift $a_k := \sum_{n=1}^k \frac{(-1)^{n+1}}{n}$ direkt um in `makelist(sum(float((-1)^(n+1)/n),n,1,k),k,1,500)` und speichern die Ergebnisse in einer Liste - davon geben wir dann die letzten 3 aus. Dies ist einfach - braucht aber viel Zeit, weil Ergebnisse die bereits berechnet wurden, neu berechnet werden. Wir berechnen deshalb auch nur 500 statt 5000 Folgenglieder.
- Berechnen einer Liste der Folgenglieder (nicht Partialsummen) und anschl. durch dazuaddieren des nächsten Folgenglieds die Ermittlung der nächsten Partialsumme - hier wird schon erheblich Zeit gespart.
- Schließlich die Rekursionsformel - es zeigt sich - die schlägt von der Zeit alle anderen!



a_1 wird mit `:` fixiert, die Rekursionsformel mit `:=`

```
(% i1) withSum(m):=lastn(makelist(sum(float((-1)^(n+1)/n),n,1,k),k,1,m),3)$
(% i2) partialSums(k):= block([r:[0], l:[]],
                             l:makelist(float((-1)^(n+1)/n),n,1,k),
                             for i thru length(l) do r:cons(first(r)+l[i],r),
                             firstn(r,3))$
(% i3) ( a[1]:1.0, a[n]:=a[n-1]+(-1)^(n+1)/n)$
(% i4) recursive(n):=lastn(makelist(a[k],k,1,n),3)$
(% i5) showtime:true;
(% i6) withSum(500);
```

Evaluation took **5.4978** seconds (5.4984 elapsed) using 99.786 MB.
[0.6921441725419141, 0.6941481805579461, 0.6921481805579461]

```
(% i7) partialSums(5000);
```

Evaluation took **3.0429** seconds (3.0430 elapsed) using 202.028 MB.
[0.6930471905599515, 0.6932471905599514, 0.6930471505519499]

```
(% i8) recursive(5000);
```

Evaluation took **0.5190** seconds (0.5190 elapsed) using 9.342 MB.
[0.6930471505519499, 0.6932471905599514, 0.6930471905599515]



Das letzte Verfahren ist also mindestens 100-mal so schnell. Man sieht natürlich immer noch das "leichte" Auf und Ab wegen des Vorzeichenwechsels. Aber es schränkt die Lage des Grenzwerts - so es einen gibt - doch ein.

2. Die harmonische Reihe

Mit *GNU-Octave*

Auch hier implementieren wir die rekursive Version!

```
1 # length of the alternating harmonic series
  ahsSize = 5000;
3
4 # get the partial sums by recursion formula
5 function r=getAHS(size)
  r=[1];
7   for i=2:size
     r=[r, r(i-1)+(-1)^(i+1)/i];
9   endfor
10  endfunction
11
12 # display only the last n numbers of array ar
13 function r=dispLastN(n,ar)
  size = length(ar);
15  start = (size -n)+1;
  for i=start:size
17     disp(ar(i));
  endfor
19 endfunction
20
21 profile on; # for evaluation time
23 ahsVals = getAHS(ahsSize);
  profile off;
25
26 suspected_limit = (ahsVals(ahsSize) + ahsVals(ahsSize-1))/2;
27 dispLastN(3,ahsVals);
  plot(1:200,getAHS(200),"linewidth",4,0:200,
29     suspected_limit:suspected_limit,"or","markersize",2);
```

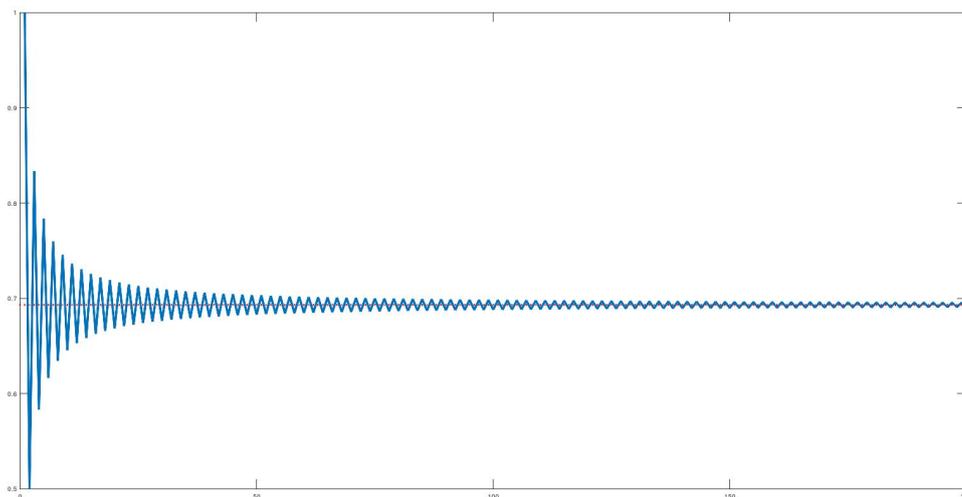


Abb.12 : 0.6931... ein Grenzwert?

Output: `>> alternateHarmonic 0.69305 0.69325 0.69305`

Ich war überrascht, wie schnell (0.077 s) *GNU-Octave* die 5000 Partialsummen berechnet hat:

`>> profshow getAHS Time (s):0.077`

2.3.2 Ist 0.6931... ein (möglicher) Grenzwert?

Theorem 2.4 Grenzwert der alternierenden harmonischen Reihe

$$\lim_{n \rightarrow \infty} \sum_{k=1}^n \frac{(-1)^{k+1}}{k} = \ln 2 = 0.693147180559\dots$$

Beweis:

$$\begin{aligned} \lim_{n \rightarrow \infty} \sum_{k=1}^n \frac{(-1)^{k+1}}{k} &= \lim_{n \rightarrow \infty} \sum_{k=1}^{2n} \frac{(-1)^{k+1}}{k} \stackrel{\text{Theorem 2.10}}{=} \lim_{n \rightarrow \infty} \sum_{j=1}^n \frac{1}{n+j} \\ &= \lim_{n \rightarrow \infty} \sum_{j=1}^n \frac{1/n}{1+j/n} = \lim_{n \rightarrow \infty} \frac{1}{n} \sum_{j=1}^n \frac{1}{1+j/n} \stackrel{\text{Skizze}}{=} \int_0^1 \frac{1}{1+x} dx = \ln 2 - \ln 1 = \ln 2 \end{aligned}$$

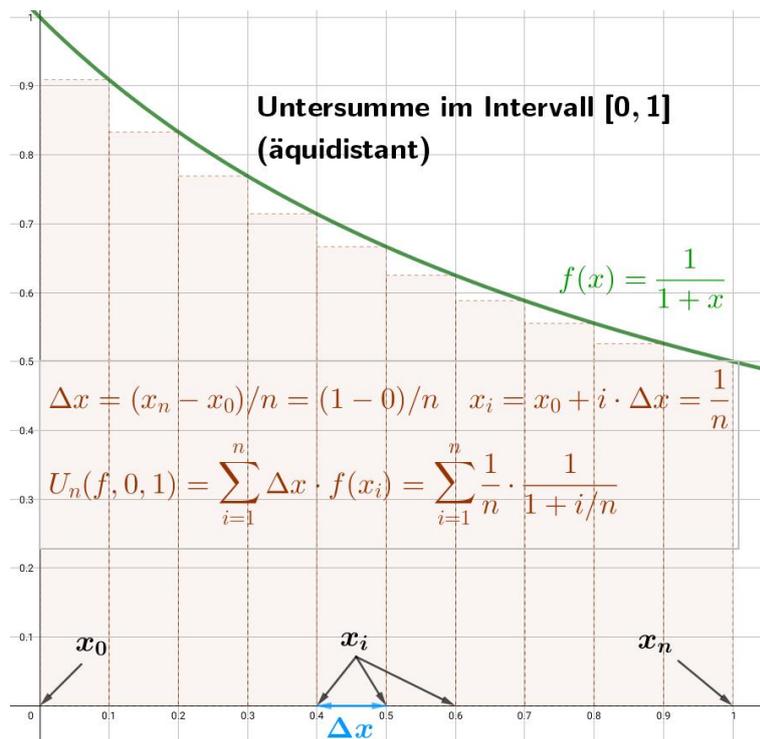


Abb.13 : Untersumme von $f(x) = \frac{1}{1+x}$ im Intervall $[0, 1] =: U_n(f, 0, 1)$

□

2. Die harmonische Reihe

2.3.3 Anderer Grenzwert gefällig?

Wir ordnen die Summanden ab dem dritten Reihenglied um und Klammern:

$$\left(1 - \frac{1}{2}\right) - \frac{1}{4} + \left(\frac{1}{3} - \frac{1}{6}\right) - \frac{1}{8} + \left(\frac{1}{5} - \frac{1}{10}\right) - \frac{1}{12} + \dots$$

Das entspricht folgender Permutation in den Reihengliedern:

$$\begin{pmatrix} 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & \dots \\ 4 & 3 & 6 & 8 & 5 & 10 & 12 & 7 & 14 & 16 & 9 & 18 & \dots \end{pmatrix}$$

Das entspricht folgenden Übergängen für $k \in \mathbb{N}$:

$$\begin{aligned} 3k &\rightarrow 3k + k &= 2(2k) \\ 3k + 1 &\rightarrow 3k + 1 - k &= 2k + 1 \\ 3k + 2 &\rightarrow 3k + 2 + k &= 2(2k + 1) \end{aligned}$$

Durchläuft k alle natürlichen Zahlen ergeben sich auf der linken Tabellenspalte alle natürlichen Zahlen ab 3, auf der rechten Spalte zweite Zeile alle ungeraden natürlichen Zahlen ab 3 und erste und dritte Zeile zusammen liefern alle geraden Zahlen ab 4.

1 und 2 sind Fixpunkte - andere gibt es nicht! Denken sie daran: die geraden Reihenglieder sind negativ (erste und dritte Zeile in Tabelle), die ungeraden positiv (zweite Reihe in Tabelle) - wir haben also:

$$\begin{aligned} \sum_{k=1}^{\infty} \frac{(-1)^{k-1}}{k} &= \frac{1}{2} + \sum_{k=1}^{\infty} \left(\frac{1}{3k} - \frac{1}{3k+1} + \frac{1}{3k+2} \right) = \frac{1}{2} + \sum_{k=1}^{\infty} \left(-\frac{1}{2(2k)} + \frac{1}{2k+1} - \frac{1}{2(2k+1)} \right) = \\ &= \frac{1}{2} + \frac{1}{2} \sum_{k=1}^{\infty} \left(\frac{1}{2k+1} - \frac{1}{2k} \right) = \frac{1}{2} \underbrace{\left(1 + \sum_{k=1}^{\infty} \left(\frac{1}{2k+1} - \frac{1}{2k} \right) \right)}_{\text{alternierende harmonische Reihe}} = \boxed{\frac{1}{2} \ln 2} \end{aligned}$$

Noch andere Möglichkeiten:

In *Knopp, Theory and applications of infinite series* (Seite 150) wird behauptet:

Summiert man von der alternierenden harmonischen Reihe immer wieder p positive Reihenglieder und anschließend q negative Reihenglieder, so konvergiert die Reihe gegen

$$\ln(2) + \frac{1}{2} \ln \left(\frac{p}{q} \right) \quad (2.6)$$

Anstatt dies zu beweisen bemühen wir unsere "experimentelle Mathematik", ob dies richtig sein kann. Die Ausgabe des folgenden *GNU-Octave* - Programms ($p = 10, q = 5$):

```
>> sumPQharmonic(10,5,1000)
Called with paramters 10 and 5
Current sum is now 1.03967
log(2)+1/2*log(p/q) for comparison: 1.03972
```

```

2  ## p positive terms und q negative, repetitions
3  function retval = sumPQharmonic (p, q, rep)
4      printf("Called with paramters %u and %u\n",p,q);
5      ## [current pos index, current neg, current sum,p,q]
6      pars=[1,2,0,p,q];
7      for j=1:rep
8          pars=nextQ(nextP(pars));
9      endfor
10
11     printf("Current sum is now %d\n", pars(3));
12     printf("log(2)+1/2*log(p/q) for comparison: %d\n", log(2)+0.5*log(p/q));
13 endfunction
14
15 function p=nextP(p)
16     for i=1:p(4)
17         p(3)=p(3)+1/p(1);
18         p(1)++; p(1)++;
19     endfor
20 endfunction
21
22 function p=nextQ(p)
23     for i=1:p(5)
24         p(3)=p(3)-1/p(2);
25         p(2)++; p(2)++;
26     endfor
27 endfunction

```

2.3.4 Der Riemann'sche Umordnungssatz

Definition 2.5 Bedingte Konvergenz

Eine Reihe $\sum_{n=0}^{\infty} a_n$ heißt **bedingt konvergent** \Leftrightarrow

$$\Leftrightarrow \sum_{n=0}^{\infty} a_n \text{ ist konvergent und } \sum_{n=0}^{\infty} |a_n| \text{ ist divergent.}$$

Theorem 2.6 Riemann'sche Umordnungssatz

Ist eine Reihe bedingt konvergent, dann gibt es für jedes $S \in \mathbb{R}$ eine Umordnung $\sigma(n)$ der Reihenglieder mit $\sum_{n=0}^{\infty} a_{\sigma(n)} = S$

2. Die harmonische Reihe

Anstatt diesen Satz zu beweisen, gehen wir es gleich mit “experimenteller Mathematik” für unsere alternierende harmonische Reihe an. Folgender Algorithmus:

- Vorgabe einer positiven reellen Zahl S und Schleifendurchläufe rep
- **begin loop:** Wir summieren p_1 positive (ungerader Index) Reihenglieder bis die Partialsumme größer oder gleich S ist :

$$S^{p_1} = \sum_{i=1}^{p_1} \frac{1}{2i-1} \geq S \wedge \sum_{i=1}^{p_1-1} \frac{1}{2i-1} < S$$

- Jetzt subtrahieren wir n_1 Reihenglieder bis wir wieder kleiner S sind
- **end loop:** Schleifenwiederholungen rep erreicht.
- Wir haben eine Folge p_i von positiven und n_i eine von negativen Summanden

Da die Variablen `targetNumber` und `currentSum` global sind, d.h. sie werden in Funktionen verwendet, ohne im Funktionskopf als Parameter übergeben zu werden, müssen wir sie vorher als **global** kennzeichnen (auch in den Funktionen selbst!)

```
## ----- Declarations -----
2 global targetNumber = 1.0;
  global currentSum = 0;
4 rep = 1000; odd=[1]; even=[2];
  format long;
6
## ----- helper-functions -----
8 function next=sumTilGreater(next)
  global targetNumber currentSum;
10 while (currentSum <= targetNumber)
  currentSum=currentSum + 1/next;
12 next++; next++;
  endwhile
14 endfunction

16 function next=sumTilSmaller(next)
  global targetNumber currentSum;
18 while (currentSum > targetNumber)
  currentSum=currentSum - 1/next;
20 next++; next++;
  endwhile
22 endfunction

## ----- MAIN -----
24 for i=1:rep
  odd= [odd, sumTilGreater(odd(end))];
26 even=[even, sumTilSmaller(even(end))];
  endfor
28
  printf("With %u repetitions we have sum: ",rep);
30 disp(currentSum);
```

Hier die Ausführung des Skripts auf der Kommandozeile:

```
>> profile on
>> alternateHarmonicSum
With 1000 repetitions we have sum:      9.996785789421192e-01
```

Wir suchen noch die 3 größten “Zeitfresser”:

```
>> profshow(3)
#           Function Attr      Time (s)   Time (%)   Calls
-----
 1 alternateHarmonicSum          0.075     70.98      1
 4      sumTilGreater           0.013     12.41    1000
 9      sumTilSmaller           0.009      8.73    1000
```

Genauere Betrachtung der Arrays *even* und *odd*:

The screenshot shows the Variable Editor with three variables:

- even**: A 1x1000 array with values from 2000 to 2002. The value at index 1001 is 2002.
- odd**: A 1x3691 array with values from 3691 to 3695. The value at index 1001 is 3695.
- currentSum**: A 1x2 array with values 1 and 2. The value at index 1 is 9.996785789421192e-01.

Abb.14 : Untersuchung von *odd* und *even*

even, *odd* nach dem Ausführen des Skripts (Zeit ca. 0.07s)

Die Arrays *even* und *odd* sind bis zum Index 1000+1 “ausgefüllt”.

Von *even* wurden 2000 Reihenglieder verwendet, von *odd* 3691.

Das größte verwendete Reihenglied ist also $\frac{1}{2000} = 5 \cdot 10^{-4}$ - weiter sind wir vom Ziel nicht entfernt (tatsächlich sind es ca. $3.4 \cdot 10^{-4}$)

Von den Arrays *even* und *odd* kann man gut nachvollziehen wie sich die Reihe zusammensetzt:

$$1 + \frac{1}{3} - \frac{1}{2} + \frac{1}{5} - \frac{1}{4} + \frac{1}{7} + \frac{1}{9} - \frac{1}{6} \pm \dots$$

```
(%i1) oddSum(up):=sum(1.0/(2*n-1), n,1,(up+1)/2)$
(%i2) evenSum(up):=sum(1.0/(2*n),n,1,up/2)$
(%i3) oddSum(3693) - evenSum(2000);
(%o3) 0.9996785789421176
```

Abb.15 : Kontrolle mit *wxMaxima*

2.4 Umordnungssatz von Gilula

Dieser Abschnitt beruht auf dem Paper:

A Class of Simple Rearrangements of the Alternating Harmonic Series,
Maxim Gilula (2018), The American Mathematical Monthly

Theorem 2.7 Umordnungssatz von Gilula

Seien $a, b, c, d \in \mathbb{N}$ mit

- $b < a$,
- $d < c$, and
- $\gcd(a, c) \nmid d - b$ (größter gemeinsamer Teiler von a und c teilt nicht $d - b$)
- Definiere $\phi : \mathbb{N} \rightarrow \mathbb{N}$ mit

$$\phi(n) := \begin{cases} c \frac{n-b}{a} + d & \text{if } n \equiv b \pmod{a} \\ a \frac{n-d}{c} + b & \text{if } n \equiv d \pmod{c} \\ n & \text{sonst} \end{cases}$$

dann gilt:

ϕ ist eine Permutation der natürlichen Zahlen und der Grenzwert der alternierenden harmonischen Reihe

$$\sum_{n=1}^{\infty} \frac{(-1)^{\phi(n)+1}}{\phi(n)} = \log(2) + \log\left(\frac{c}{a}\right) \frac{(-1)^b + (-1)^{a+b}}{2a} + \log\left(\frac{a}{c}\right) \frac{(-1)^d + (-1)^{c+d}}{2c}$$

Für den Beweis von Theorem 2.7 verweise ich auf das oben zitierte Paper. Wir wollen hier die Bedeutung dieses Satzes in einigen Punkten “durchleuchten”.

2.4.1 Was macht ϕ ?

Betrachten wir den oberen Ast:

$$n \equiv b \pmod{a} \Leftrightarrow \exists k \in \mathbb{N} : n - b = k \cdot a \Rightarrow c \frac{n-b}{a} + d \rightarrow c \cdot k + d$$

Also immer wenn $n - b$ ein k -faches von a ist, wird auf $c \cdot k + d$ verschoben!

Der zweite Ast von ϕ verschiebt $c \cdot k + d$ auf $a \cdot k + b$. Dies führt zu einer Vertauschung der beiden Reihenglieder!

Wir vergewissern uns mit einem kleinen *wxMaxima*-Programm:

2. Die harmonische Reihe

Voraussetzungen, N_m ist der maximale Index

(% i1) (a:4,b:2,c:4,d:3,k_m:1000,N_m:k_m*c+d,(d-b)/gcd(a,c)); $\frac{1}{4}$

Vergleicht 2 Listen

(% i2) compareLists(L1,L2):=catch(
 if length(L1)≠length(L2) then throw(false)
 else
 for i thru length(L1) do if (L1[i]≠L2[i]) then throw(false),
 throw(true))\$

Permutation nach Gilula

(% i3) f(n):=catch(
 if (b = mod(n,a)) then throw(c*(n-b)/a+d),
 if (d = mod(n,c)) then throw(a*(n-d)/c+b),
 throw(n))\$

Identische Permutation - vor der Vertauschung

(% i4) ar:makelist(i,i,1,N_m)\$

Test

(% i5) compareLists(ar, makelist(f(n),n,1,N_m)); false

(% i6) swap(i,j):=block([h:0],h:ar[i], ar[i]:ar[j], ar[j]:h)\$

Jetzt wird vertauscht

(% i7) for k:0 thru k_m do swap(a*k+b,c*k+d); done

(% i8) compareLists(ar, makelist(f(n),n,1,N_m)); true

Damit immer eine Vertauschung stattfindet, darf Folgendes **nicht** passieren:

$$\exists k_1, k_2 \in \mathbb{N} : ck_1 + d = ak_2 + b \Leftrightarrow ak_2 - ck_1 = d - b$$

Existiert $\gcd(a, c) > 1$, dann gilt

$$\gcd(a, c) \underbrace{\left[\frac{a}{\gcd(a, c)} k_2 - \frac{c}{\gcd(a, c)} k_1 \right]}_{\in \mathbb{Z}} = d - b \Leftrightarrow \gcd(a, c) \text{ teilt } (d - b)$$

z.B. $a = 8, b = 3, c = 6, d = 5 \Rightarrow \gcd(a, c) = 2 \mid (d - b) = 2$

$$35 \equiv 3 \pmod{8} \Rightarrow \phi(35) = 29 \text{ außerdem gilt } 35 \equiv 5 \pmod{6} \Rightarrow \phi(35) = 43$$

oder man sieht es andersrum $\phi(29) = 35$ und $\phi(43) = 35$

Mit der **“Nicht-Vertauschung”** fallen Surjektivität und Injektivität von ϕ !



Während die ersten beiden Bedingungen in Theorem 2.7 nur Ordnung in die Größe der Variablenamen bringen, ist die dritte Bedingung $\gcd(a, c) \nmid d - b$ wesentlich dafür, dass ϕ eine Permutation darstellt!

2.4.2 Spezieller Grenzwert $\log(2)$ (bzw. $\ln 2$)

Wir schreiben die Behauptung von Theorem 2.7 etwas um:

$$\sum_{n=1}^{\infty} \frac{(-1)^{\phi(n)+1}}{\phi(n)} = \log(2) + \log\left(\frac{c}{a}\right) (-1)^b \frac{1+(-1)^a}{2a} + \log\left(\frac{a}{c}\right) (-1)^d \frac{1+(-1)^c}{2c}$$

Sind a und c ungerade verschwindet die rechte Seite bis auf $\log(2)$

Überprüfen wir das an einem Beispiel in *wxMaxima*:

(% i1) (a:15,b:2,c:9,d:3,(d-b)/gcd(a,c)); $\frac{1}{3}$

(% i2) f(n):=catch(
 if (b = mod(n,a)) then throw(c*(n-b)/a+d),
 if (d = mod(n,c)) then throw(a*(n-d)/c+b),
 throw(n))\$

(% i3) a[n]:=(-1)^(f(n)+1)/f(n); $a_n := \frac{(-1)^{f(n)+1}}{f(n)}$

(% i4) sum(a[n],n,1,10000)-log(2),numer; 3.29203791649135310⁻⁶

Ebenfalls wenn $a = c$ gilt, ergibt sich aus obiger Formel $\log(2)$ als Grenzwert und zwar auch wenn sie gerade sind:

(% i1) (a:6,b:2,c:6,d:3,(d-b)/gcd(a,c)); $\frac{1}{6}$

(% i2) f(n):=catch(
 if (b = mod(n,a)) then throw(c*(n-b)/a+d),
 if (d = mod(n,c)) then throw(a*(n-d)/c+b),
 throw(n))\$

(% i3) a[n]:=(-1)^(f(n)+1)/f(n); $a_n := \frac{(-1)^{f(n)+1}}{f(n)}$

(% i4) sum(a[n],n,1,50000)-log(2),numer; 2.9999700030710⁻⁵

2.4.3 Dichtheit der Grenzwerte

Also für alle reelle Zahlen r gibt es Werte für $a, b, c, d \in \mathbb{N}$, sodass der Grenzwert der Reihe der reellen Zahl r beliebig nahe kommt.

Formal

$$\forall \varepsilon > 0 \forall r \in \mathbb{R} \exists a, b, c, d \in \mathbb{N} : \left| \sum_{n=1}^{\infty} \frac{(-1)^{\phi(n)+1}}{\phi(n)} - r \right| < \varepsilon$$

2. Die harmonische Reihe

Bei $a = 6m$, $b = 2$, $c = 3(2n + 1)$, $d = 3$ ergibt sich als Grenzwert $\log(2) + \log\left(\frac{6n+3}{6m}\right) \frac{1}{6m}$, $m, n \in \mathbb{N}$. Außerdem gilt $\gcd(a, c) \geq 3$, das zur Folge hat: $\gcd(a, c) \nmid (d - b)$

Wir zeigen mit einem *Geogebra*-Programm, dass $\log\left(\frac{6n+3}{6m}\right) \frac{1}{6m}$ mit $6n+3 > 6m$ dicht in \mathbb{R}^+ liegt. Vertauschen der Rollen von $6m$ und $6n+3$ liefern die negativen Werte!

Dabei werden wir sehen, dass das Konvergenzverhalten “fürchterlich” ist, soll heißen, selbst hohe Partialsummen sind noch ein beachtliches Stück vom Grenzwert entfernt.

Außerdem ergeben sich für $r > 5$ Zahlen für a bzw. c , die numerisch mit “normalen Programmen” (*Geogebra*, *wxMaxima*, usw.) nicht mehr beherrschbar sind. Die großen Zahlen besonders für c sind natürlich ein Erbe der “normalen” harmonischen Reihe - wieviel Reihenglieder braucht es, um überhaupt den Summenwert 200 zu erreichen?

z.B.: in *wxMaxima*: `ceiling(%e^60)-floor(%e^60); -617280166`

Geogebra liefert bei der gleichen Rechnung 0 - besser, aber auch falsch!

Theorem 2.8 Dichtheit der Gilula-Grenzwerte

$$\forall \varepsilon > 0 \forall r \in \mathbb{R}^+ \exists m, n \in \mathbb{N} : \left| \log\left(\frac{6n+3}{6m}\right) \frac{1}{6m} - r \right| < \varepsilon$$

Die Funktionen *ceiling* und *floor* werden mit der üblichen Nomenklatur dargestellt:

$\lceil 1.7 \rceil = 2$ und $\lfloor 1.7 \rfloor = 1$ Es gilt $\lceil a + 0.1 \rceil > a$ und $\lfloor a - 0.1 \rfloor < a \quad \forall a \in \mathbb{R}$

Beweisstrategie:

Wir fassen $\Delta x = \frac{1}{6m}$ als Schrittweite auf und wählen diese deutlich kleiner als ε . Damit können wir mit ganzzahligen Schritten vom Ursprung ausgehend nicht über die ε -Umgebung von r “drübersteigen”.

■ $\Delta x = \frac{1}{6m} < \frac{\varepsilon}{6}$ (Schrittweite kleiner ε) $\Leftrightarrow m(\varepsilon) = \lceil \frac{1}{\varepsilon} + 0.1 \rceil$

■ Seien $k_1 \in \mathbb{R}^+$ die “Schritte” um $r - \varepsilon$ zu erreichen, k_2 jene für $r + \varepsilon$ also

$$k_1 \cdot \Delta x = r - \varepsilon \Leftrightarrow \log(e^{k_1}) \cdot \Delta x = r - \varepsilon \quad \text{bzw.} \quad k_2 \cdot \Delta x = r + \varepsilon$$

$$\frac{6n_1 + 3}{6m} = \lceil e^{k_1} + 0.1 \rceil > e^{k_1} \Rightarrow \log\left(\frac{6n_1 + 3}{6m}\right) \cdot \Delta x > r - \varepsilon \Rightarrow n_1 = \lceil e^{k_1} + 0.1 \rceil \cdot m$$

$$\frac{6n_2 + 3}{6m} = \lfloor e^{k_2} - 0.1 \rfloor < e^{k_2} \Rightarrow \log\left(\frac{6n_2 + 3}{6m}\right) \cdot \Delta x < r + \varepsilon \Rightarrow n_2 = \lfloor e^{k_2} - 0.1 \rfloor \cdot m - 1$$

2.5 Der Kochsalzkristall

(siehe auch **Convergence of lattice sums and Madelung's constant**

David Borwein, Jonathan M. Borwein, and Keith F. Taylor)

Dieser besteht aus Na^+ und Cl^- Ionen. In unserem Modell seien diese punktförmig und sie sind auf Gitterplätzen mit ganzzahligen Koordinaten (Wahl der Längeneinheit). Der Kristall habe unendliche Ausdehnung. Wir berechnen die potentielle Energie auf ein Ion im Ursprung - die *Madelung Konstante*. Um uns auf das Problem einzustimmen, verwenden wir vorerst die

2.5.1 2 dimensionale Näherung

Da wir 2 Schleifen benötigen, implementieren wir dies in *Geogebra* mit Javascript. Wir setzen einen Button mit *Caption Lattice*, dessen *Click*-Methode zeichnet das Kristallgitter. Die Schleifen erzeugen die Punktlisten, in Zeile 15/16 werden Leerzeichen und Beistrich abgeschnitten, die Klammer angehängt, Na^+ eingefärbt und Cl^- vergrößert.

```

On Click | On Update | Global JavaScript
1 var pointsNa="NaList=";
2 var pointsCl="ClList=";
3
4 function drawLattice(size){
5   for (k=-size; k<=size; k++){
6     for (j=-size; j<=size; j++){
7       if (Math.pow(-1,j+k)>0) {
8         pointsNa=pointsNa.concat(" "+j+", "+k+", ");
9       }
10      else {
11        pointsCl= pointsCl.concat(" "+j+", "+k+", ");
12      }
13    } // end inner for
14  } // end outer for
15  pointsNa=pointsNa.substring(0, pointsNa.length-2).concat("");
16  pointsCl=pointsCl.substring(0, pointsCl.length-2).concat("");
17 }
18
19 drawLattice(8);
20 ggbApplet.evalCommand(pointsNa);
21 ggbApplet.evalCommand(pointsCl);
22 ggbApplet.evalCommand('SetColor(NaList,"red" ');
23 ggbApplet.evalCommand('SetPointSize(ClList,7) ');
24 ggbApplet.evalCommand('SetVisibleInView(NaList,1,true) ');
25

```

Abb.19 : Javascript für NaCl

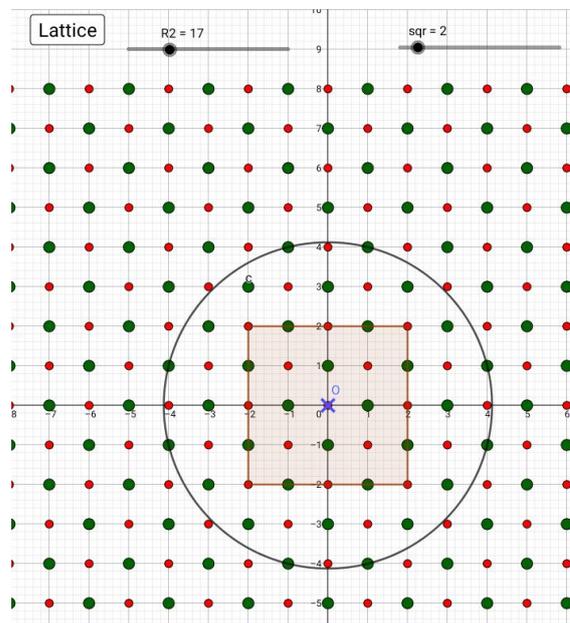


Abb.20 : NaCl-Kristall - 2 dimensional

2 Methoden der Berechnung der potentiellen Energie bieten sich förmlich an:
 Berechnung innerhalb einer **Kreisfläche** bzw. **ladungsneutralen Quadrats** (Abb. 20)

$$E_{pot} = \sum_{i,j \neq 0} \frac{(-1)^{i+j}}{r_{ij}} = \begin{cases} \sum_{n=1} \frac{(-1)^n}{\sqrt{n}} & n \in \mathbb{N} \wedge n = i^2 + j^2 \quad (\text{Kreis}) \\ \sum_{j,k \neq 0} \frac{(-1)^{j+k}}{\sqrt{j^2 + k^2}} & -n \leq j, k \leq n \in \mathbb{N} \quad (\text{Quadrat}) \end{cases} \quad (2.7)$$

Bei voriger Formel für die Kreisflächensummation haben wir verwendet:

$$(-1)^{i+j} = (-1)^{i^2+j^2}$$

denn es gilt: $2|(i+j) \Leftrightarrow 2|(i^2+j^2)$

$$2|(i+j) \Leftrightarrow \begin{cases} i = 2m+1 \wedge j = 2n+1 & \Leftrightarrow i^2+j^2 = (2m+1)^2 + (2n+1)^2 & \Leftrightarrow 2|(i^2+j^2) \\ i = 2m \wedge j = 2n & \Leftrightarrow i^2+j^2 = (2m)^2 + (2n)^2 & \Leftrightarrow 2|(i^2+j^2) \end{cases}$$

Außerdem wirft es folgende Frage auf:

Welche Summation (von den möglichen) wird in der Natur angewendet?

Die Kreisflächensummation

Für die Kreisfläche spricht, dass sich die Wirkung des elektromagnetischen Feldes mit Lichtgeschwindigkeit kreisförmig ausbreitet.

Es wirft aber auch einige unangenehme Fragen auf:

Wenn man den quadratischen Radius des Kreises auf N ausdehnt

- Gibt es Ionen mit $i^2 + j^2 = N$
- Wieviele sind es?
- Sind es Na - oder Cl -Ionen oder beides?

Für die ersten beiden Fragen kommen "brute-force" oder Zahlentheorie in Frage (ich habe beide Methoden) implementiert.

Die dritte Frage wurde oben schon beantwortet, da $i^2 + j^2$ dieselbe Parität hat wie $i + j$.

Hilfsfunktion: Ist $N = 2^{2n}$, dann gibt es nur 1 Lösung von $N = x^2 + y^2$

```
→ isPowerOf4(N):=if (N=4) then true
   else
   if ( (N>4) and integerp(N) ) then isPowerOf4(N/4)$
```

"brute force": Ist ein Summand 0 oder beide gleich, sind 4 Lösungen möglich, sonst 8 zum Schluss wird festgestellt, ob es sich um Na - oder Cl -Ionen handelt(Vorzeichen)

2. Die harmonische Reihe

```
→ sqr_sum(N):=block([up:floor(sqrt(N/2)), p:0, L:[],c:0 ],
  catch(if (isPowerOf4(N)) then throw( [4, [sqrt(N),0]] )),
  for i:0 thru up do (
    p:sqrt(N-i^2),
    if (integerp(p)) then ( L:cons([p,i],L),
      if ((i=0) or (p=i)) then c:c+4 else c:c+8 )
    ),
  if ( (c ≠ 0) and (oddp(L[1][1]+L[1][2])) ) then c: -c,
  cons(c,L))$
```

Das Ergebnis der Zahlentheorie für die Anzahl der Lösungen von $N = x^2 + y^2$ mit $x, y \in \mathbb{N}(x, y \neq 0)$
Wir zerlegen N in Primfaktoren (*inwxMaxima* \rightarrow *ifactors(N)*)

$$N = 2^{a_0} p_1^{n_1} p_2^{n_2} \dots q_1^{m_1} q_2^{m_2} \dots$$

wobei für die $p_i \equiv 1 \pmod{4}$ und für die $q_i \equiv 3 \pmod{4}$ gilt.

Ist eines der m_i ungerade, dann gibt es keine Lösung.

Sei $r = (n_1 + 1) \cdot (n_2 + 1) \cdot \dots$

Ist r gerade, dann ist $r/2$ die Anzahl der Lösungen, sonst wird vorher r inkrementiert oder dekrementiert, je nachdem ob die 2-er Anzahl gerade oder ungerade ist: $1/2(r - (-1)^{a_0})$

Da unsere Lösungen nur in \mathbb{N} gezählt werden, müssen wir mit 8 multiplizieren - aus $r/2$ wird dann $4r$.

Bei $N = 25$ gilt $[5, 0]$ hier müssen wir mit plus 4 korrigieren: $[5, 0], [-5, 0], [0, 5], [0, -5]$.

Sind die beiden Lösungen gleich, wie bei $N = 8$, müssen wir wieder 4 abziehen.

Ist $a_0 > 0$ ist $i + j$ gerade - es handelt sich um ein Na -Ion, die potentielle Energie ist positiv, sonst negativ.

Für die Begründung verweise ich auf die einschlägige Literatur!

```

→ nrOfSqr(pf,corr):=block([r:1, l:length(pf),a0:0 ],
  if ( pf[1][1] =2 ) then a0: pf[1][2],
  for i thru l do (
    if ( (mod(pf[i][1],4)=3) and (oddp(pf[i][2])) ) then (r:0, return()),
    if ( mod( pf[i][1],4)=1) then r: r*(pf[i][2]+1)
  ),
  if (evenp(r)) then r:4*r+corr
  else r:4*(r - (-1)^a0)+corr,
  if (a0 > 0) then r else -r)$

```

Jetzt der Aufruf mit der Primfaktorzerlegung und der Korrektur

```

→ sqr_sum2(N):=block([correction:0 ],
  if (integerp(sqrt(N))) then correction:4,
  if (integerp(sqrt(N/2)) ) then correction:-4,
  nrOfSqr(ifactors(N),correction))$

```

Wir vergleichen "brute force" mit Zahlentheorie

```

→ for i:2 thru 10 do ( printf(true, "N=~d ~d ~d ~%",i,sqr_sum(i)[1], sqr_sum2(i))$

```

N=2	4	4
N=3	0	0
N=4	4	4
N=5	-8	-8
N=6	0	0
N=7	0	0
N=8	4	4
N=9	-4	-4
N=10	8	8

Wir vergleichen die ersten 500 Ergebnisse: keine Ausgabe - gut!

```

→ for i:2 thru 500 do (if ( (sqr_sum2(i)) ≠ (sqr_sum(i)[1] ) ) then disp(i));

```

done

2. Die harmonische Reihe

Wir schauen uns das Laufzeitverhalten beider Implementationen an:

→ `showtime:true$`

Evaluation took 0.0000 seconds (0.0000 elapsed) using 56 bytes.

Die Ergebnisse zeigen, dass sich die Theorie mehr als auszahlt!

→ `sqr_sum2(2^1*5^3*7^2*13^2*17^3);`

Evaluation took 0.0018 seconds (0.0018 elapsed) using 29.992 KB.

192

→ `sqr_sum(2^1*5^3*7^2*13^2*17^3);`

Evaluation took 49.6602 seconds (49.6631 elapsed) using 275.162 MB.

[192,[72905, 69685],[73423, 69139]...

→ `showtime:false$`

→ `s[n]:=sqr_sum(1)[1] + sum(float(sqr_sum2(i)/sqrt(i)),i,2,n);`

$$s_n := [sqr_sum]_1 + \sum_{i=2}^n \text{float} \left(\frac{\text{sqr_sum2}(i)}{\sqrt{i}} \right)$$

2 Stunden Rechenzeit (2020 Intel i3) ergeben

→ `plot2d([discrete,makelist([i,s[i]],i,1,500000,200)],x,1,500000);`

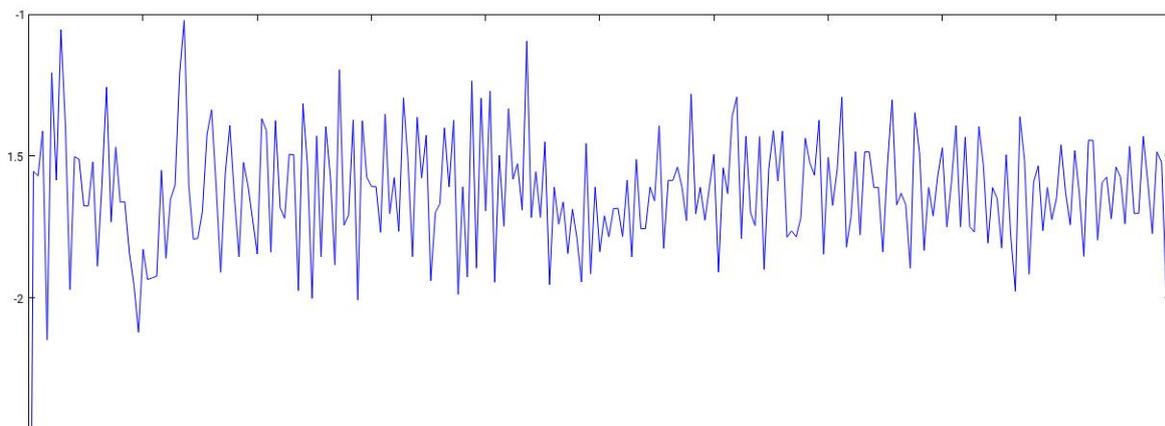


Abb.21 : Madelung Konstante bis R=500 000 - schaut so Konvergenz aus?

David Borwein, Jonathan M. Borwein und Keith F. Taylor zeigen im oben genannten Paper - ja das Ding konvergiert!

Summe über die Quadrate

Dazu verwenden wir jetzt wieder *Geogebra* mit Javascript:

Im Reiter [GlobalJavascript] kürzen wir die Evaluierung der *Geogebra* Kommandos ab, eine Hilfsfunktion `pAr2str(p)`, die aus einem Javascript Array einen *Geogebra*-String erzeugt. Außerdem bestimmen wir den Symmetriefaktor, wie oben mit *wxMaxima* (normalerweise 8, nur wenn eine Lösung verschwindet oder beide gleich sind, dann nur 4)

```

function ggbEv(cmd){
  ggbApplet.evalCommand(cmd);
}

function pAr2str(p){
  return "("+p[0]+", "+p[1]+")";
}

function symFac(s, j){
  var fac=0;
  if ( (j==s) || (j==0) ){
    fac=4;
  }
  else {
    fac=8;
  }
  if ( (s+j) % 2 == 0 ) {
    return fac;
  }
  else {
    return (-1)*fac;
  }
}

function sumUp(s) {
  var sum=0;
  for (var k=0; k <= s; k++){
    sum = sum + symFac(s, k)/Math.sqrt(s*s+k*k);
  }
  return sum;
}

function madelung(upTo){
  var pList="mad={";
  var acc=0;
  var p=[];
  for (var l=1; l<upTo; l++){
    acc=acc+sumUp(l);
    p.push(pAr2str([l, acc]));
  }
  ggbEv(pList.concat(p.join()).concat("}"));
}

madelung(3000);
ggbEv('SetVisibleInView(mad,2,true)');

```

Click-Methode des Button "Madelung"

`sumUp(s)` ist Summe über die "Umfanglinie" eines Quadrates der Seitenlänge s (Abb. 20)

`madelung(upTo)` summiert über die Umfangslinien von 1 bis $upTo$.

`madelung(3000)` erzeugt die Punktliste und anschl. wird sie auf Grafik2 gesetzt.

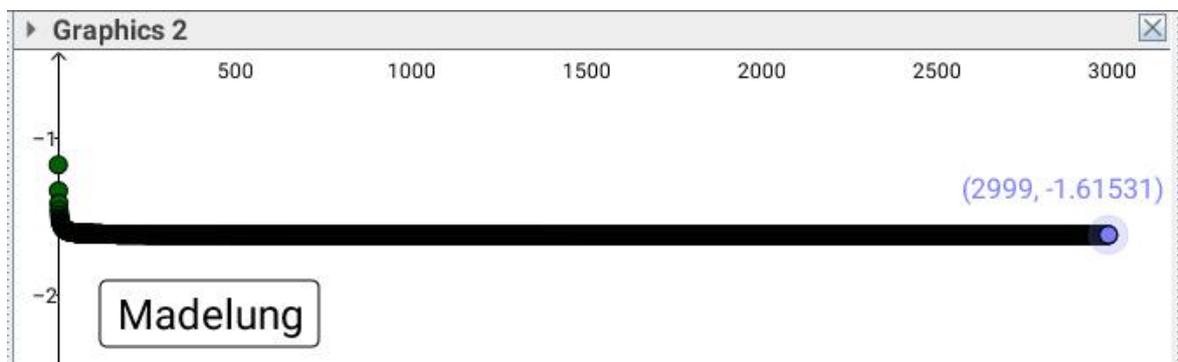


Abb.22 : Summe über die Quadrate - viel besser als "Kreissumme"

2. Die harmonische Reihe

2.5.2 3 dimensionale Näherung

Visualisierung des Problems in *Geogebra*

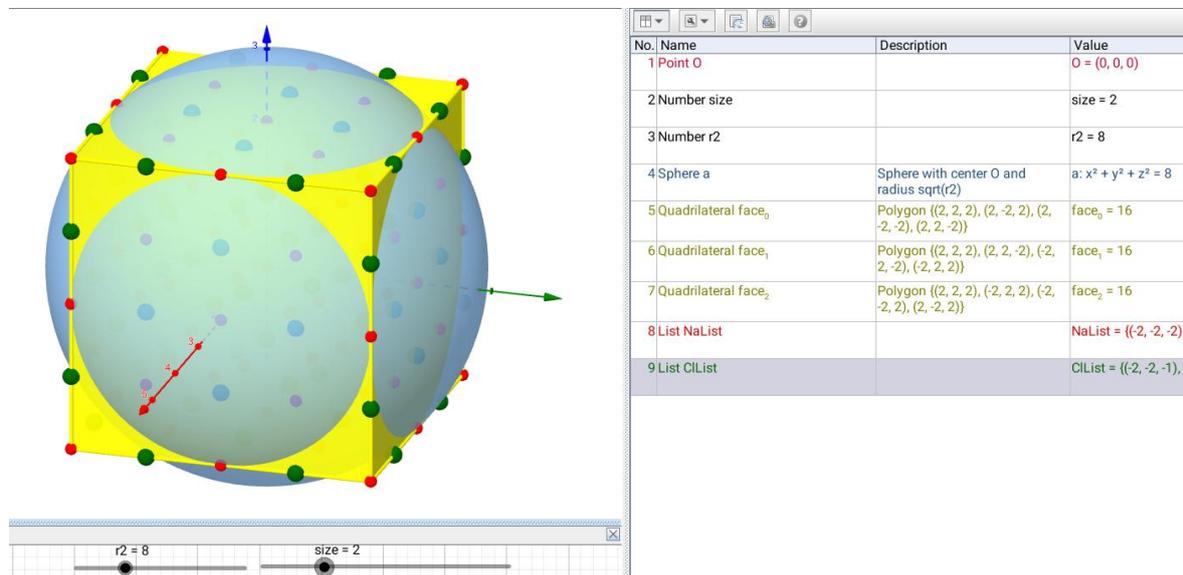


Abb.23 : Visualisierung in *Geogebra*

Wir haben 2 *Slider*(ganzzahlig):

$r2$ für den quadratischen Radius der Kugel um O mit Schrittweite 1
 $size$ für die Seitenlänge des Würfels mit Schrittweite 1

Im Reiter [\[GlobalJavascript\]](#) (von $size$):

```

1 function ggbOnInit () {}
3 var pointsNa=""; var pointsCl="";
5 function ggbEv (cmd) {
6     ggbApplet.evalCommand(cmd);
7 }
9 function ggbDel (obj) {
10    ggbApplet.deleteObject (obj);
11 }
13 function pAr2str (p) {
14    return "(" + p[0] + " , " + p[1] + " , " + p[2] + " )";
15 }

```

Bevor es ans Eingemachte geht, zuvor wieder die üblichen Helferfunktionen:
 Variablendeklarationen, Abkürzungen und Javascript-Array-Triple in *Geogebra*-String konvertieren

```

1 function drawLattice(size){
  pointsNa="NaList={"; pointsCl="ClList={";
3  var pNa=[]; var pCl=[];
  for (j=-size; j<=size; j++){
5    for (k=-size; k<=size; k++){
      for(l=-size; l<=size; l++){
7        if (Math.pow(-1,j+k+l)>0) {
          pNa.push(pAr2str([j,k,l]));
9        }
        else {
11         pCl.push(pAr2str([j,k,l]));
        }
13      } // end inner for
    } // end outer for
15 } // end outer-outer for
  pointsNa=pointsNa.concat(pNa.join()).concat("}");
17  pointsCl=pointsCl.concat(pCl.join()).concat("}");
  }

19 function drawFace(fn, s){
21  var polyStr="face_"+fn+":Polygon({";
  var p=[]; var corner=[s,s,s];
23  p.push(pAr2str(corner)); var c = fn;
  for (i=0; i<3; i++){
25    if ( c == fn ) {
      c = (c+1) % 3;
27    }
    corner[c] = - corner[c];
29    p.push(pAr2str(corner));
    c= (c+1) %3;
31  } //for
  return polyStr.concat(p.join()).concat("}");
33 }

35 function drawFaces(s){
  var cmd="";
37  for (j=0; j<3; j++){
    ggbEv(drawFace(j, s));
39    ggbEv("SetFilling(face_"+j+",0.95)");
    ggbEv('SetColor(face_'+j+', "yellow")');
41  }
  }

43 function drawUpdate(s){
45  drawLattice(s);
  drawFaces(s);
47  ggbEv(pointsNa);
  ggbEv(pointsCl);
49  ggbEv('SetColor(NaList, "red")');
  ggbEv('SetPointSize(ClList,7)');
51 }

```

3 geschachtelte Schleifen erzeugen die Tripel $[j, k, \ell]$, konvertieren sie in einen String und erzeugen so den *Geogebra*-Befehl für eine Punktliste:

$$(-1)^{j+k+\ell} > 0 \quad \begin{cases} \text{ja} \rightarrow & Na^+ \\ \text{nein} \rightarrow & Cl^- \end{cases}$$

Eine Würfel­fläche("face") mit der Nummer fn wird mit dem Eckpunkt $[s, s, s]$ erzeugt (er gehört zu 3 Seiten­flächen mit Nummer 0, 1, 2).

Der Algorithmus benutzt zyklische Vertauschung des Vorzeichens von 0, 1, 2, wobei "Facenummer" fn übersprungen wird.

z.B.: $fn = 1$ erzeugt dann

$$\begin{array}{lcl} [s, s, s] & \rightarrow & [-s, s, s] \rightarrow \\ [-s, s, -s] & \rightarrow & [s, s, -s] \end{array}$$

Zeichnen der 3 Würfel­seiten $drawUpdate(s)$ wird vom Slider $size.Update$ aufgerufen - siehe ↓

```

1 ggbDel('ClList');
  ggbDel('NaList');
3 ggbDel('face_0');
  ggbDel('face_1');
5 ggbDel('face_2');
  var size= ggbApplet.
    getValue('size');
7 drawUpdate(size);

```

Update-Section of slider $size$

2. Die harmonische Reihe

Summe über die Kugelflächen

Wir suchen also $(i, j, k) \in \mathbb{Z}$ mit

$$i^2 + j^2 + k^2 = N \quad N \in \mathbb{N} \quad (2.8)$$

Zitat aus <https://mathworld.wolfram.com/SumofSquaresFunction.html>:

The number of solutions of 2.8 for a given N without restriction on the signs or relative sizes of i , j , and k is given by $r_3(N)$. Gauss proved that if N is squarefree and $N > 4$, then

$$r_3(N) = \begin{cases} 24 h(-N) & \text{for } N \equiv 3 \pmod{8} \\ 12 h(-4n) & \text{for } N \equiv 1, 2, 5, 6 \pmod{8} \\ 0 & \text{for } N \equiv 7 \pmod{8} \end{cases}$$

(Arno 1992), where $h(x)$ is the **class number** of x .

Die Ermittlung der Klassenzahl erfordert viel theoretische Arbeit, sodass ich darauf verzichte!
Was ich allerdings benutze ist

Theorem 2.9 Legendre's three square theorem

A non-negative integer N can be represented as sum of three squares of integers if and only if N is NOT of the form $4^a(8b + 7)$ for some integers a and b .

Alles andere muss die Rechengeschwindigkeit des Prozessors leisten ;(

Hier das *wxMaxima* -Programm

Creates first 50 numbers of Legendre's sequence

```
(% i1) LegendreSeq():=block([L:[]],
    for a:0 thru 5 do
        for b:0 thru 10 do L:cons(4^a*(8*b+7),L),
    sort(L))$

(% i2) Leg:firstn(LegendreSeq(),10); [7, 15, 23, 28, 31, 39, 47, 55, 60, 63] (Leg)
```

production version - explained in debug version below

```
(% i4) sqr_sum(N):=block([up:floor(sqrt(N/2)), p:0, L:[] ],
    catch( if (member(N,Leg)) then throw ([0,[]]),
    for i:0 thru up do block(
        for j: i thru up do block(
            p:sqrt( N- i^2 - j^2),
            if (integerp(p) and p>=j) then L:cons([p,j,i],L) ),
    cons(length(L),L) ) )$
```

debug version - with file logging in *wxMaxima*-workspace in *file.txt*

$i^2 + j^2 + k^2 = N \rightarrow$ if $i^2 = N/2$ greatest value for j^2 is $N/2$, but k^2 and j^2 may flip - we only take the value for k^2 , where $k^2 \geq j^2$ - the other pair is rejected!

```
(% i3) sqr_sum_debug(N):=block([up:floor(sqrt(N/2)), p:0, L:[] ],
  /* if Legendre-number no triple exists */
  catch( if (member(N,Leg)) then throw ([0,[]]),
  /*LISP-command for deleting "file.txt" (must exist in workspace!->touch file.txt)*/
  ?delete\~file ("file.txt"),
  /* open stream s */
  s:openw("file.txt"),
  for i:0 thru up do block(
    printf(s, i= ~d up=~d j=~d up2=~d ~%",i,up,i,up),
    for j: i thru up do block(
      p:sqrt( N- i^2 - j^2),
      if (integerp(p) and p<j) then printf(s,"found and rejected: [~d,~d,~d] ~%",p,j,i),
      if (integerp(p) and p>=j) then (L:cons([p,j,i],L),
      printf(s,"found: [~d,~d,~d] ~%",p,j,i) ) ),
    close(s,[length(L),L]))$
```

```
(% i5) sqr_sum_debug(1125);
[10,[ [24,18,15],[23,20,14],[25,20,10],[30,12,9],[31,10,8],[26,20,7],[25,22,4],[32,10,1],[30,15,0],[33,6,0] ]
```

```
1  i= 0  up=23  j=0  up2=23
   found: [33,6,0]
3  found: [30,15,0]
i= 1  up=23  j=1  up2=23
5  found: [32,10,1]
i= 2  up=23  j=2  up2=23
7  i= 3  up=23  j=3  up2=23
   i= 4  up=23  j=4  up2=23
9  found: [25,22,4]
i= 5  up=23  j=5  up2=23
11 i= 6  up=23  j=6  up2=23
   i= 7  up=23  j=7  up2=23
13 found: [26,20,7]
i= 8  up=23  j=8  up2=23
15 found: [31,10,8]
i= 9  up=23  j=9  up2=23
17 found: [30,12,9]
i= 10 up=23  j=10 up2=23
19 found: [25,20,10]
```

```
1  i= 11 up=23  j=11 up2=23
   i= 12 up=23  j=12 up2=23
3  i= 13 up=23  j=13 up2=23
   i= 14 up=23  j=14 up2=23
5  found: [23,20,14]
   found and rejected: [20,23,14]
7  i= 15 up=23  j=15 up2=23
   found: [24,18,15]
9  i= 16 up=23  j=16 up2=23
   i= 17 up=23  j=17 up2=23
11 i= 18 up=23  j=18 up2=23
   i= 19 up=23  j=19 up2=23
13 i= 20 up=23  j=20 up2=23
   found and rejected: [14,23,20]
15 i= 21 up=23  j=21 up2=23
   i= 22 up=23  j=22 up2=23
17 i= 23 up=23  j=23 up2=23
```

output in "file.txt"

- 1) die größte Möglichkeit für i, j ist $\sqrt{N/2}$ z.B.: $[k, j, i] = [\sqrt{N/2}, \sqrt{N/2}, 0]$
- 2) ist $[k, 3, 1]$ eine Lösung dann auch $[k, 1, 3]$, also startet die j -Schleife bei i ($j \geq i$)
- 3) ist $[2, 3, 1]$ eine Lösung dann auch $[3, 2, 1]$, also nehmen wir jene $k \geq j$

Durch diese 3 Regeln bekommen wir alle Lösungen $[k, j, i]$ mit $k \geq j \geq i$ - alle anderen Lösungen bekommt man durch Symmetrieüberlegungen. Natürlich gilt: $\|(k, j, -i)\| = \|(k, j, i)\|$.

2. Die harmonische Reihe

Wir ermitteln die Symmetriefaktoren an Hand der Nullen(z und wenn Lösungen gleich sind(eq)
 Außerdem ermitteln wir die Parität - Na^+ oder Cl^- (p)

```
(% i7) /*input:tripel, vars: zeros, equalities, last non zero, parity, formula params */
symFac(t):=block( [z:0,eq:0, l:0,p:0,k:0,a:0],
    /* detemine parity of tripel [i,j,k] --> i+j+k */
    if (evenp(reduce("+",t))) then p:1
    else p:-1,
    /* determine zeros and equalities */
    for i :3 thru 1 step (-1) do block (
        if (t[i]=0) then z:z+1
        else
        /* t[i] is NOT zero now! */
        if (l≠t[i]) then l:t[i]
        else eq:eq+1
    ),
    if (z=2 and eq=0) then k:1, /* 6 positions */
    if (z=1 and eq=1) then k:2, /* 12 positions */
    if (z=1 and eq=0) then k:4, /* 24 positions */
    if (z=0 and eq=0) then k:8, /* 48 positions */
    if (z=0 and eq=1) then k:4, /* 24 positions */
    if (z=0 and eq=2) then (k:1, a:2), /* 8 positions */
    (k*6+a)*p) $ /* return the number of positions with the same norm */
```

Jetzt werden die Potentiale der einzelnen Kugelflächen summiert

```
(% i8) madelung(N):=block([sum:0.0, L:[], plot:[]],
    for j thru N do (
        L:sqr_sum(j),
        for k: 2 thru length(L) do (
            t:L[k],
            if (t ≠ []) then sum: sum + ( symFac(t)/sqrt(t[1]^2 + t[2]^2 + t[3]^2) ) ,
            plot:cons([j,sum],plot) ) ,
    plot)$

(% i9) plot2d([discrete,madelung(500)],[x,1,500]);
```

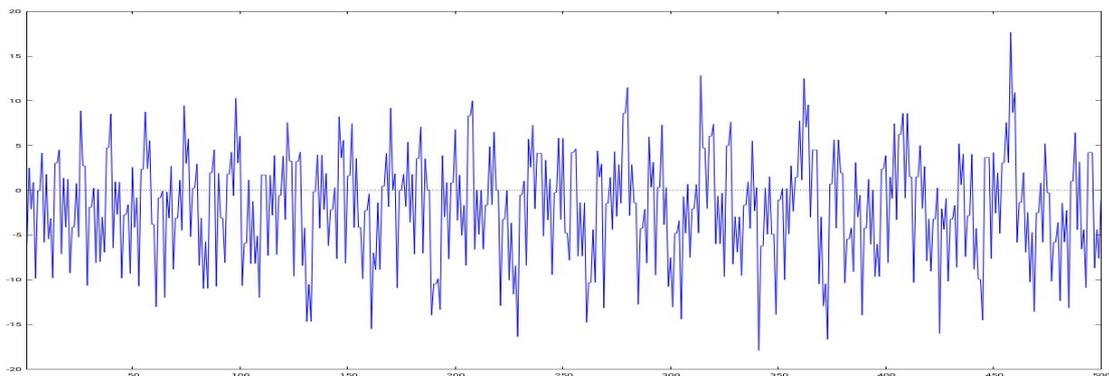


Abb.24 : Summen über Kugelflächen - sieht so Konvergenz aus?

Tatsächlich haben David Borwein, Jonathan M. Borwein, and Keith F. Taylor im oben genannten Paper gezeigt, dass diese Folge divergent ist, obwohl in "alten" Lehrbüchern der Festkörperphysik diese Methode zur "Bestimmung" der Madelung-Konstante herangezogen wird - berechnet hat man sie mit dieser Methode offensichtlich nicht!

Noch einige Worte zur Bestimmung der Symmetriefaktoren:

Entweder man lässt sie dem Computer mit "brute-force" selbst bestimmen, indem man die Werte für (i, j, k) nicht auf positive Werte einschränkt und auch auf die Monotonie verzichtet - außerdem werden für k auch die negativen Lösungen hinzugefügt:

```
(%i5) sqr_sum_simple(N):=block([up:ceiling(sqrt(N)), p:0, L:[ ] ],
  catch( if (member(N,Leg)) then throw ([0,[ ]]),
    for i:-up thru up do block(
      for j: -up thru up do block(
        p:sqrt( N- i^2 - j^2),
        if (integerp(p) ) then
          (L:cons([p,j,i],L), if p#0 then L:cons([-p,j,i],L))
        ) ),
    cons(length(L),L) )
)$
```

```
sqr_sum_simple(0^2+0^2+2^2); [6,[0,0,2],[0,2,0],[-2,0,0],[2,0,0],[0,-2,0],[0,0,-2]]
sqr_sum_simple(0^2+1^2+1^2); [12,[0,1,1],[-1,0,1],[1,0,1],...]
sqr_sum_simple(0^2+1^2+2^2); [24,[0,1,2],[-1,0,2],[1,0,2],[0,-1,2],[0,2,1],[-2,0,1],[2,0,1],...]
sqr_sum_simple(3^2+1^2+2^2); [48,[-1,2,3],[1,2,3],[-2,1,3],[2,1,3],[-2,-1,3],[2,-1,3],[-1,-2,3],...]
sqr_sum_simple(1^2+3^2+3^2); [24,[-1,2,3],[1,2,3],[-2,1,3],[2,1,3],[-2,-1,3],[2,-1,3],[-1,-2,3],...]
sqr_sum_simple(1^2+1^2+1^2); [8,[-1,1,1],[1,1,1],[-1,-1,1],[1,-1,1],[-1,1,-1],[1,1,-1],[-1,-1,-1],...]
```

Natürlich führen auch kombinatorische oder wahrscheinlichkeitstheoretische Ansätze zum (hoffentlich) selben Ziel:

z.B.: $(\pm 1)^2 + (\pm 2)^2 + (\pm 3)^2 = 14$ also keine Nullen und keine Gleichheit!

Ein wahrscheinlichkeitstheoretischer Ansatz könnte lauten: Es wird aus 3 Urnen (in der i -ten befinden sich $+i$ und $-i$) jeweils eine Zahl gewählt. Die Wahrscheinlichkeit für eine bestimmte Zahl ist dann

$$P(Z) = \underbrace{\frac{1}{3}}_{\text{Urne}} \overbrace{\frac{1}{2}}^{\text{Zahl wählen}} \underbrace{\frac{1}{2}}_{\text{Urne}} \underbrace{\frac{1}{2}}_{\text{Urne}} \frac{1}{2} = \frac{1}{48} \Rightarrow 48 \text{ Möglichkeiten weil gleiche W.}$$

Bei $(\pm 1)^2 + (\pm 1)^2 + (\pm 1)^2 = 3$ hat man nur eine Urne mit Zurücklegen: $\frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{8}$

Oder mit Kombinatorik: Bei $0^2 + (\pm 1)^2 + (\pm 1)^2 = 2$ ergeben sich 12 Möglichkeiten:

$(0, 1, 1) \rightarrow \frac{3!}{2!}$ Umstellungen $(0, -1, 1) \rightarrow 3!$ Umstellungen $(0, -1, -1) \rightarrow \frac{3!}{2!}$ Umstellungen

2. Die harmonische Reihe

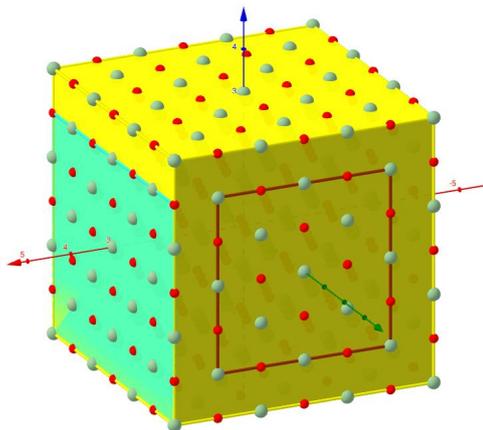
Summe über die Würfel­flächen

Wieviel Ionen befinden sich auf einer Würfel­oberfläche?

- Auf einer Würfel­oberfläche mit Schwerpunkt im Ursprung und Eckpunkt (N, N, N) befinden sich

$$4(2N + 1)2N + 2(2N - 1)^2 = 24N^2 + 2 \text{ Ionen}$$

In *Geogebra*: `nrByFormula=24*size^2 2+`



Mit dem türkisen Rechteck mit

$$(2N + 1)2N \text{ Ionen}$$

lässt sich ein "Mantel" um den Würfel legen.

es bleiben dann noch die Ionen in Deck- und Grundfläche

$$2(N - 1)^2$$

Abb.25 : Ionenzahl der Würfel­oberfläche

- Wir lassen sie zählen!

```

1 function drawLattice(size) {
2   ...
3   csp="cL={"; \\cube surface points
4
5   var pNa=[]; var pCl=[]; sp=[];
6   for (j=-size; j<=size; j++){
7     for (k=-size; k<=size; k++){
8       for (l=-size; l<=size; l++){
9         if (Math.pow(-1,j+k+l)>0){
10          pNa.push(pAr2str([j,k,l]));
11        }
12        else {
13          pCl.push(pAr2str([j,k,l]));
14        } // end if-else
15        if (Math.max(j*j,k*k,l*l) == size*size){
16          sp.push(pAr2str([j,k,l]));
17        } // end - if
18      } // end inner for
19    } // end outer for
20  } // end outer-outer for
21  pointsNa=pointsNa.concat(pNa.join()).concat("}");
22  pointsCl=pointsCl.concat(pCl.join()).concat("}");
23  csp=csp.concat(sp.join()).concat("}");
24  }

```

Bei der Zeichnung der Ionengitterplätze fügen wir Zeile 15 ein:

in *csp* wird eine Liste der Oberflächenpunkte erstellt - mindestens eine Koordinate muss die Größe von *size* haben - diese Liste brauchen wir nur mehr abzuzählen!

In *Geogebra*:

`nrByCount=Length(cL)`

🔴 Einteilung nach ihrem Abstand vom Ursprung

Es sind 7 Fälle möglich (sie alle haben Repräsentanten im Dreieck von Abb. 26) - man muss "nur" überlegen, wieviele Punkte von dieser Sorte es gibt!

Fallunterscheidungen:

1)	$(N, 0, 0)$	6		$r = N$	Mittelpunkt 1 Quadrates
2)	$(N, N, 0)$	12		$r = \sqrt{N^2 + N^2}$	Schnitt der Koord.-Ebenen mit Würfel; 3 Quadrate mit 4 Eckpunkten
3)	(N, N, N)	8		$r = \sqrt{N^2 + N^2 + N^2}$	8 Eckpunkte des Würfels
4)	$(N, i, 0)$	24	$0 < i < N$	$r = \sqrt{N^2 + i^2}$	x -Richtung - 4 pro Fläche
5)	(N, i, i)	24	$0 < i < N$	$r = \sqrt{N^2 + i^2 + i^2}$	Diagonale - 4 pro Fläche
6)	(N, N, i)	24	$0 < i < N$	$r = \sqrt{N^2 + N^2 + i^2}$	1/2 Kante - 12 Kanten \times 2
7)	(N, i, j)	48	$0 < i < N$ $\wedge j < i$	$r = \sqrt{N^2 + i^2 + j^2}$	"grünes Dreieck"; 8 pro Fläche

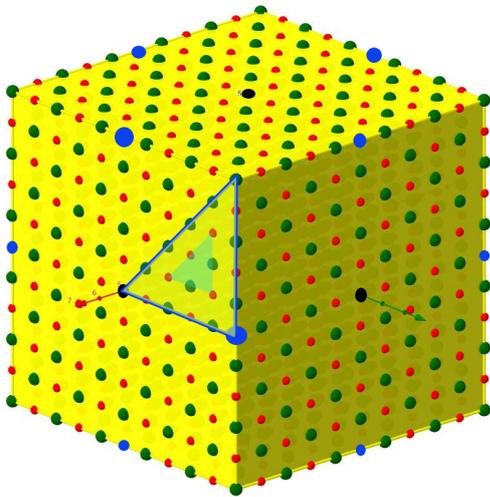


Abb.26 : Symmetrie

Jetzt zählen wir die Punkte zusammen:

```
function ionsOnSurface(N) {
2   var sum=0;
   var sym=[0,6,12,8,24,24,24,48];
   // (N,0,0)+(N,N,0)+(N,N,N)
   sum = sum + sym[1]+sym[2]+sym[3];
   for (var i=1; i<N; i++){
6     // (N,i,0)+(N,i,i)+(N,N,i)
       sum = sum + sym[4] + sym[5] + sym[6];
       for (var j = 1; j<i; j++){
10        sum = sum + sym[7]; // (N,i,j)
       }
12  }
   return sum;
14 }
```

In `drawUpdate(s)` fügen wir hinzu: `ggbApplet.setValue("nrByCases", ionsOnSurface(s));`

Madelung Konstante

Zuerst setzen wir einen Button "Madelung by Cubes" und in dessen `Click`-Methode schreiben wir folgenden Javascript Code:

```
var madGraph="";
2
function pAr2str(p) {
4   return "("+p[0]+ " , " +p[1]+ ")"
   ;
6 }
function ggbEv(cmd) {
8   ggbApplet.evalCommand(cmd);
}
```

← zuerst einige Hilfsfunktionen↓

```
1 function parity(nr) {
   if (nr % 2 == 0) return 1;
3   return -1;
   }
5
function radiusSqr(x,y,z) {
7   return x*x+y*y+z*z;
   }
```

2. Die harmonische Reihe

In gewisserweise gleicht die Funktion $\text{cubeSurface}(N)$ der $\text{ionsOnSurface}(N)$, nur dass jetzt die Potentiale (Parität mal reziproker Radius) addiert werden:

```

function cubeSurface(N){
2  var sum=0; var r2 = 0; var p=0;
3  var sym=[0,6,12,8,24,24,24,48];
4
5  r2 = radiusSqr(N,0,0);
6  p= parity(r2);
7  sum += sym[1]*p/Math.sqrt(r2);
8
9  r2 = radiusSqr(N,N,0);
10 p= parity(r2);
11 sum += sym[2]*p/Math.sqrt(r2);
12
13 r2 = radiusSqr(N,N,N);
14 p= parity(r2);
15 sum += sym[3]*p/Math.sqrt(r2);
16
17 for (var i=1; i<N; i++){
18   r2 = radiusSqr(N,i,0);
19   p= parity(r2);
20   sum += sym[4]*p/Math.sqrt(r2);
21
22   r2 = radiusSqr(N,i,i);
23   p= parity(r2);
24   sum += sym[5]*p/Math.sqrt(r2);
25
26   r2 = radiusSqr(N,N,i);
27   p= parity(r2);
28   sum += sym[6]*p/Math.sqrt(r2);
29
30 // now the inner loop -->
}

for (var j = 1; j<i; j++){
2   r2 = radiusSqr(N,i,j);
3   p= parity(r2);
4   sum += sym[7]*p/Math.sqrt(r2);
5 } // end for j
6 } // end for i
7
8 return sum;
9 } // end func cubeSurface

function mad(s) {
10 var m=0; var p=[];
11 var madGraph="madG={";
12 for (var i=1; i <= s; i++) {
13   m = m+cubeSurface(i);
14   p.push(pAr2str([i,m]));
15 } // end for i
16 // data for pointlist in string-array p now!
17 ggbEv(madGraph.concat(p.join()).concat("}"));
18 ggbEv('SetVisibleInView(madG,1,true)');
19 ggbEv('SetColor(madG,"red")');
20 ggbEv('SetPointSize(madG,2)');
21 alert("M-Const by 400 cubes: "+m);
22 } // end func mad

/* ----- MAIN -----*/
mad(400);

```

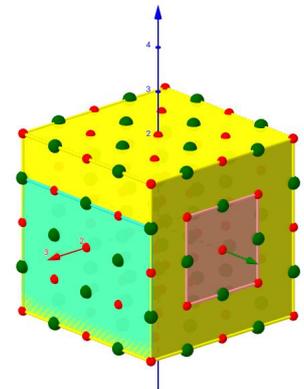
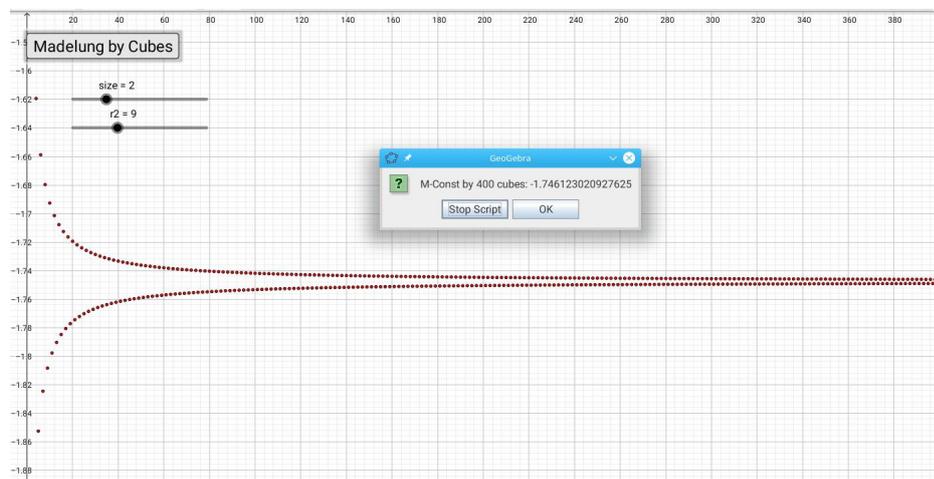


Abb.27 : Summe von 400 Würfelflächen

2.6 ANHANG

Theorem 2.10 Endliche harmonische Reihe

$$\sum_{k=1}^{2n} \frac{(-1)^{k+1}}{k} = \sum_{j=1}^n \frac{1}{n+j}$$

Beweis: Wir beweisen mit vollständiger Induktion:

• Anfang $n = 1$ ergibt: $1 - 1/2 = 1/2$

• Induktionsschritt:

$$\sum_{k=1}^{2(n+1)} \frac{(-1)^{k+1}}{k} \stackrel{!}{=} \sum_{j=1}^{n+1} \frac{1}{n+1+j} \quad \text{wir zerlegen die Summen und fassen zusammen} \quad \Downarrow$$

$$\underbrace{\sum_{k=1}^{2n} \frac{(-1)^{k+1}}{k}}_{\text{Induktionsvoraussetzung}} + \frac{1}{2n+1} - \frac{1}{2n+2} = \sum_{j=1}^n \frac{1}{n+1+j} + \frac{1}{2n+2} - \frac{1}{2n+2} \Rightarrow$$

Induktionsvoraussetzung

$$\sum_{k=1}^n \frac{1}{n+k} - \frac{1}{n+1} = \sum_{j=1}^n \frac{1}{n+1+j} - \frac{1}{2n+1} \Rightarrow \sum_{k=2}^n \frac{1}{n+k} = \sum_{j=1}^{n-1} \frac{1}{n+1+j} \Rightarrow$$

$$\stackrel{\ell = j+1}{\Downarrow} \sum_{k=2}^n \frac{1}{n+k} = \sum_{\ell=2}^n \frac{1}{n+\ell}$$

□

2. Die harmonische Reihe

3 | Zykloiden

Abstract:

Herleitung der Formeln für Epizykloide (Kreis rollt außen auf Kreis) und Hypozykloide (Kreis rollt innen auf Kreis). Benötigt(zumindest von Vorteil) für die Durcharbeitung dieses Schriftstücks ist das Programm *Geogebra*. Am Mittelpunkt des abrollenden Kreises sei noch eine mitrotierende “Stange” angebracht. Damit stehen 3 frei wählbare Parameter zur Verfügung: Kreis k_2 mit Radius r rollt auf k_1 mit Radius R ab. Länge der “Stange” ist ρ
Bevor wir uns mit dem “Rollen” beschäftigen, machen wir es etwas einfacher - wir beschäftigen uns mit

3.1 Gleiten auf einem Kreis

Konzentrieren wir uns vorerst auf den Fall ein Kreis k_2 gleitet außen auf einen anderen Kreis k_1 . Auf k_2 befindet sich ein Objekt - dies wird durch den roten Vektor veranschaulicht. Denken Sie sich eine Münze mit einer Zahl oder einem “Kopf” darauf. Damit keine Rollbewegung im Spiel ist fixieren wir beim Gleiten k_2 mit einer blauen Stange! Hier das *Geogebra* Arbeitsblatt <https://www.geogebra.org/m/dnc4J32A> - mit dem Schieberegler können Sie den äußeren Kreis gleiten lassen!

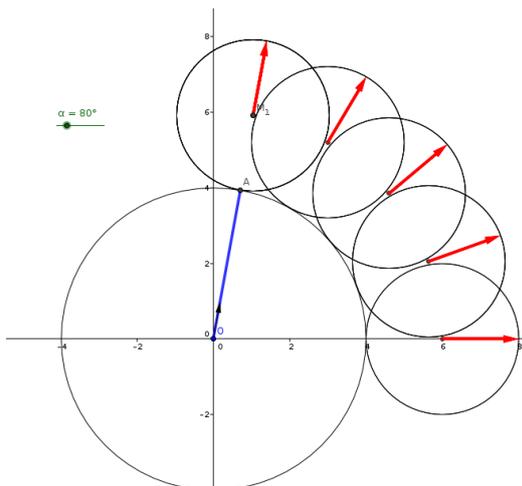


Abb.28 : Gleiten

Halten wir fest: Während in Bezug auf ein mitrotierendes Bezugssystem K_r die Position des Objekts (roter Vektor) unverändert bleibt, führt es im “Zeichenblattsystem” auch eine Drehung aus - und zwar um den gleichen Winkel. Die neue Lage von k_2 ergibt sich also aus einer Translation **plus** einer Drehung - und das **ohne** Rollen. Dasselbe gilt natürlich auch für einen Kreis k_3 , der innen gleitet: im Arbeitsblatt M_2 und k_3 auf sichtbar schalten. Wer Lust hat kann auch wieder einen Vektor als “Objekt” einzeichnen.

3. Zykloiden

Es sei

$$\vec{e}_\alpha = \begin{pmatrix} \cos \alpha \\ \sin \alpha \end{pmatrix} \quad \text{der Einheitsvektor in Richtung } \alpha$$

Ein Punkt P am Spitzenpunkt des roten Vektor hat dann die Koordinaten

$$\vec{p} = \vec{m}_1 + \rho \vec{e}_\alpha = (R + r) \vec{e}_\alpha + \rho \vec{e}_\alpha$$

In unserem *Geogebra*-Arbeitsblatt heißt \vec{e}_α einfach v_0 . Damit können wir leicht die Probe machen, ob wir mit unserer Formel richtig liegen: Wir geben in der Befehlszeile ein:

$$P=(R+r)*v_0 + \rho*v_0$$

Schalten den Schieberegler für ρ sichtbar und betätigen den für α .

Jetzt zum

3.2 Kreis rollt auf Kreis - außen: Epizykloide

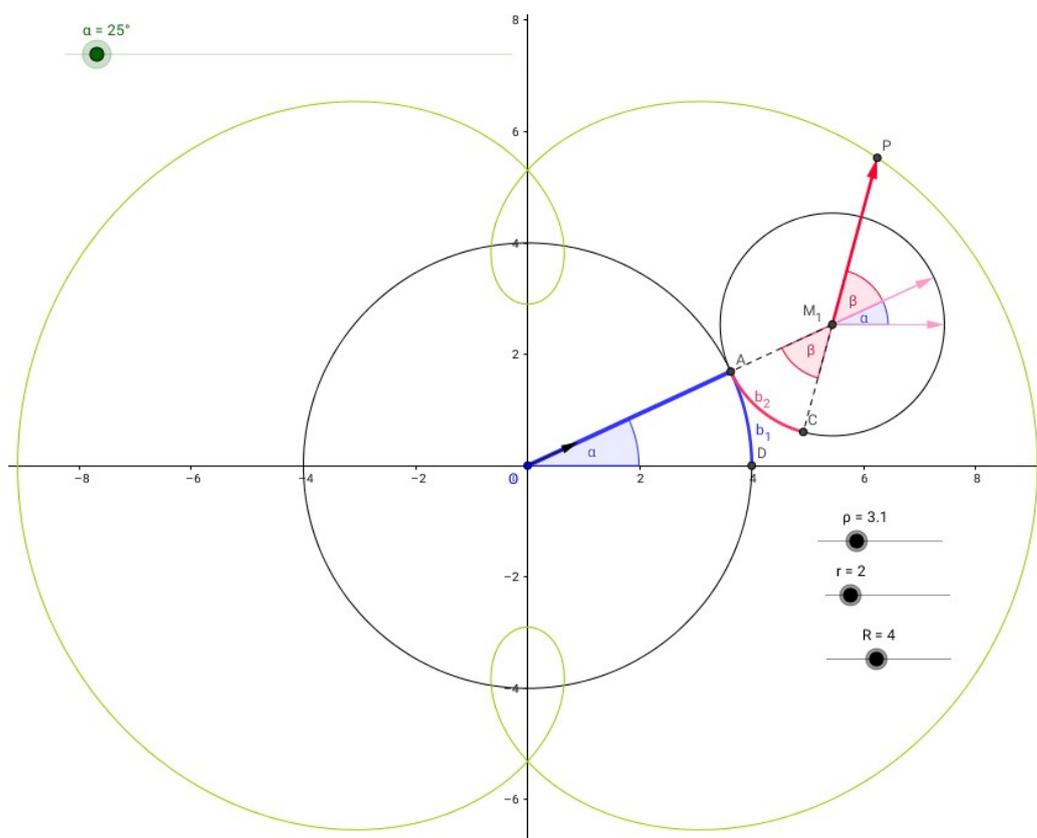


Abb.29 : Rollen - außen

Beim Rollen müssen die zurückgelegten Strecken gleich sein. Es muss also gelten

$$\boxed{b_1 = b_2} \Rightarrow R\alpha = r\beta \Rightarrow \beta = \frac{R}{r}\alpha$$

Schauen wir uns die “roten” Vektoren an:

waagrecht war die Ausgangsrichtung, um α gedreht ergibt sich allein durch das Gleiten, dazu kommt jetzt eine Drehung um β .

Damit ergeben sich für den Punkt P folgende Koordinaten

$$P = M_1 + \rho \vec{e}_{\alpha+\beta} \Leftrightarrow P = (R+r) \begin{pmatrix} \cos \alpha \\ \sin \alpha \end{pmatrix} + \rho \begin{pmatrix} \cos \left(\left(1 + \frac{R}{r}\right) \alpha \right) \\ \sin \left(\left(1 + \frac{R}{r}\right) \alpha \right) \end{pmatrix} \quad (3.1)$$

Im *Geogebra* Arbeitsblatt <https://www.geogebra.org/m/apcgy7jq> ist diese Formel bereits als parametrisierte Kurve k (hellgrün) eingegeben. Wenn man außerdem die Spur von P einschaltet, lässt sich beim Betätigen des Schiebereglers für α leicht die Richtigkeit obiger Formel bestätigen!(oder wählt das Werkzeug *Ortslinie* klickt anschl. auf P und α)

Mit dem Einstellen der Schieberegler für R , r und ρ lassen sich die verschiedensten *Epizykloiden* erzeugen!

Das Konstruktionsprotokoll in Kürze:

- Slider für R , r , ρ und α erzeugen; Ursprung $O = (0, 0)$ festlegen
- $\beta = (R/r) * \alpha$;
- Einheitsvektoren: $v_0 = \text{Vector}(\cos(\alpha), \sin(\alpha))$; $z_0 = \text{Vector}(\cos(\alpha + \beta), \sin(\alpha + \beta))$
- Jetzt die Kreise: $M_1 = (R+r) * v_0$ $k_1: \text{Circle}(0, R)$ $k_2: \text{Circle}(M_1, r)$
- Punkte: $A = R * v_0$ $P = M_1 + \rho * z_0$ $C = M_1 - r * z_0$ $D = (R, 0)$
- Kreisbögen: $b_1 = \text{CircularArc}(0, D, A)$ $b_2 = \text{CircularArc}(M_1, A, C)$
- Strecken zeichnen: $a = \text{Segment}(0, A)$ $b = \text{Segment}(M_1, A)$ $e = \text{Segment}(M_1, C)$
- Hilfspunkte (unsichtbar): $H_1 = M_1 + (r, 0)$ $H_2 = M_1 + r * v_0$
- Einzeichnen der restlichen Vektoren und Winkel
- Zum Schluss die Formel für unsere Kurve:
 $k: \text{Curve}((R+r) * \cos(\alpha) + \rho * \cos((1+R/r) * \alpha), (R+r) * \sin(\alpha) + \rho * \sin((1+R/r) * \alpha), \alpha, 0, 6.28319)$

3. Zykloiden

3.2.1 Spezielle Epizykloiden

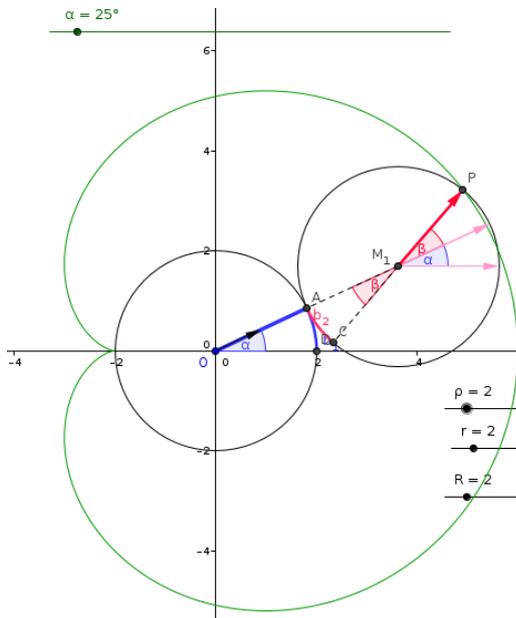


Abb.30 : Kardoide: $R = r = \rho$

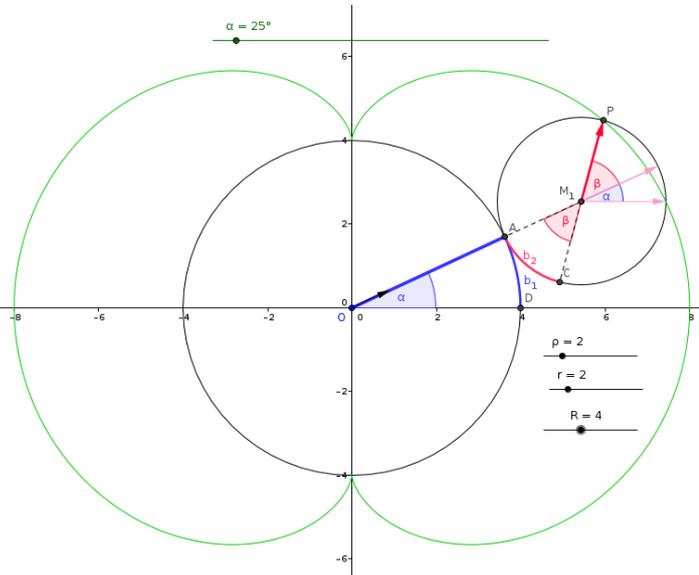


Abb.31 : Nephroide: $r = \rho \quad R = 2r$

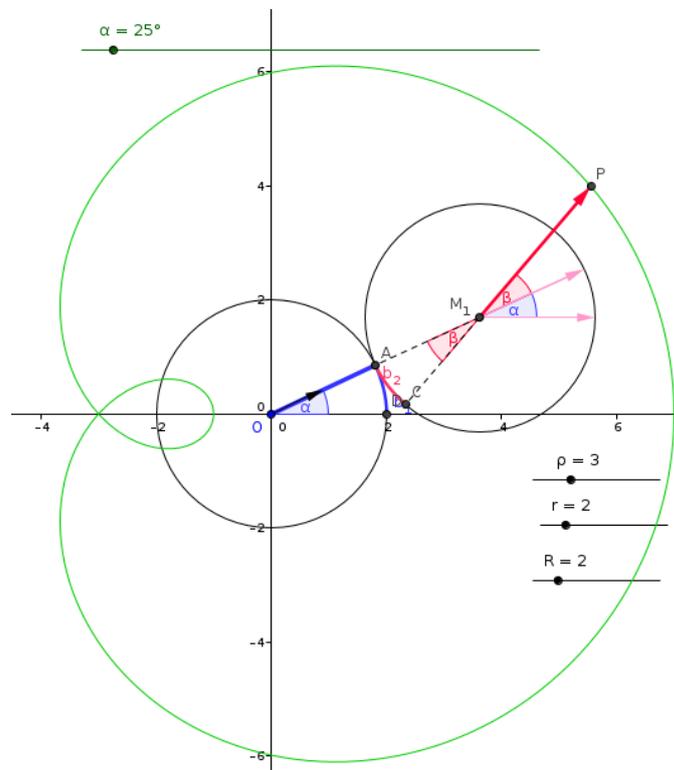


Abb.32 : Pascal-Schnecke(Limacon): $r = R \quad \rho > R$

3.3 Kreis rollt auf Kreis - innen: Hypozykloide

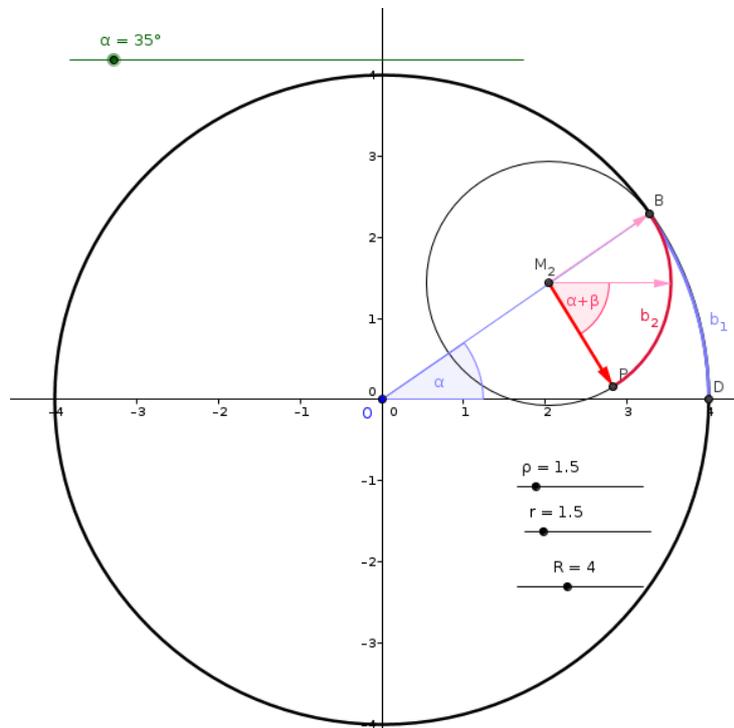


Abb.33 : Rollen - innen

Beim Rollen müssen die zurückgelegten Strecken gleich sein. Es muss also gelten

$$\boxed{b_1 = b_2} \Rightarrow R\alpha = r|\beta| \Rightarrow |\beta| = \frac{R}{r}\alpha$$

Allerdings ist der Effekt der Gleitbewegung jetzt der Rollbewegung entgegengerichtet! Schauen wir uns die "roten" Vektoren an:

waagrecht war die Ausgangsrichtung, um α gedreht ergibt sich allein durch das Gleiten, dazu kommt jetzt eine negative Drehung um β .

Damit ergeben sich für den Punkt P folgende Koordinaten

$$P = M_2 + \rho \vec{e}_{\alpha-|\beta|} \Leftrightarrow P = (R+r) \begin{pmatrix} \cos \alpha \\ \sin \alpha \end{pmatrix} + \rho \begin{pmatrix} \cos \left(\left(1 - \frac{R}{r}\right) \alpha \right) \\ \sin \left(\left(1 - \frac{R}{r}\right) \alpha \right) \end{pmatrix}$$

Hier wieder das Arbeitsblatt dazu (<https://www.geogebra.org/m/kwKVTQft>).

Spitzen ergeben sich für $\rho = r$ und die Anzahl der Spitzen $n = R/r$ - also wenn r ein Teiler von R ist.

3. Zykloiden

3.3.1 Spezielle Hypozykloiden

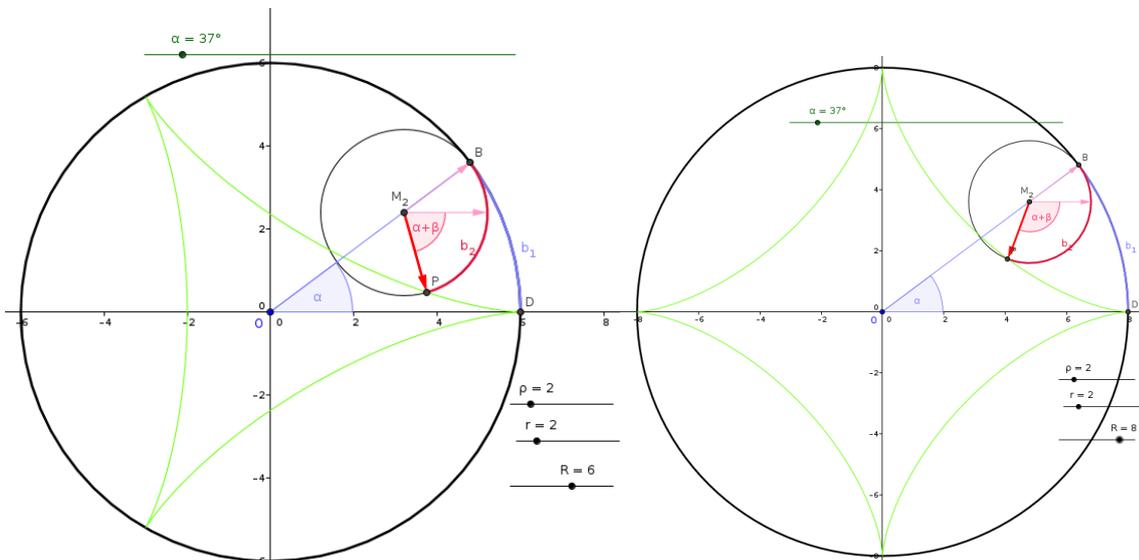


Abb.34 : Tricuspoide: $r = \rho = \frac{R}{3}$

Abb.35 : Astroide: $r = \rho = \frac{R}{4}$

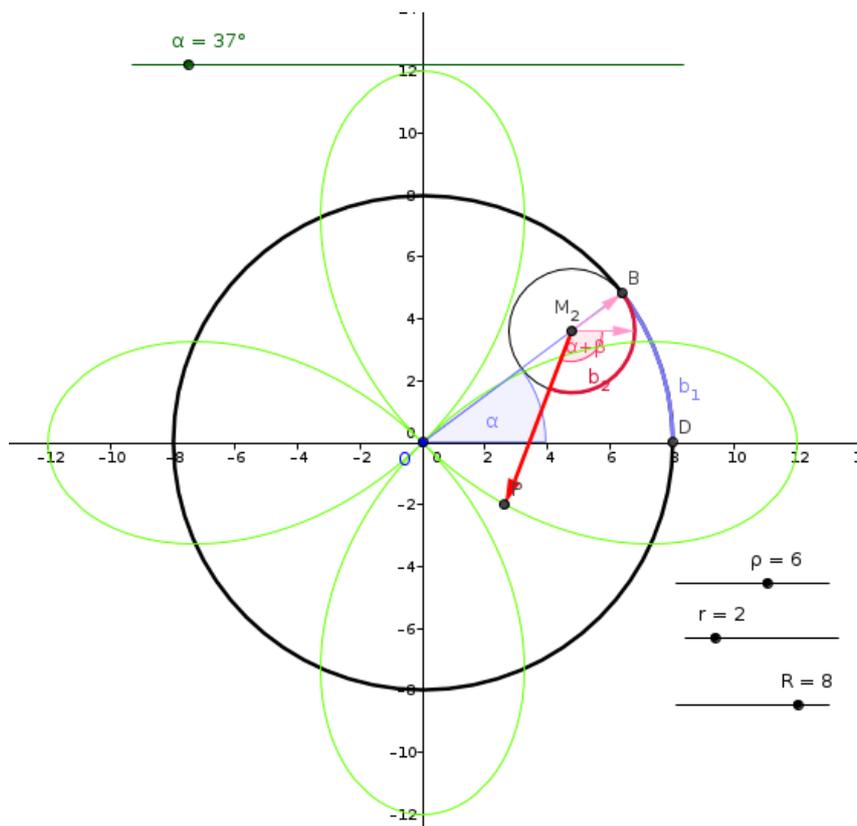


Abb.36 : NoName: $r = \frac{R}{4}$ und $\rho > r$

3.4 ANHANG: Rotationsmatrizen

Mit dem Wissen von Kapitel 5 über homogene Koordinaten, Rotations- und Translationsmatrizen lässt sich die Formel 3.1 algebraisch herleiten:

- Der Punkt $P(r + R + \rho, 0)$ wird um α um den Ursprung rotiert: $R_\alpha \cdot P$
- Anschließend um β um den Punkt M_1 ($M_1 = R_\alpha \cdot (r + R, 0)$)

Sei T_A die Translationsmatrix zum Punkt A - dann ergeben sich für den Punkt P insgesamt folgende Transformationen:

$$T_{M_1} \cdot R_\beta \cdot T_{(-M_1)} \cdot R_\alpha \cdot P$$

Ausführung in *wxMaxima*:

Rotationsmatrix für homogene Koordinaten

```
(% i1) R(%alpha):=matrix([cos(%alpha),-sin(%alpha),0],
                        [sin(%alpha),cos(%alpha),0],
                        [0,0,1])$
```

Translationsmatrix um Spaltenvektor v , dessen Struktur ist: $[[.],[.],[1]]$ - wir zerlegen sie mit `args(v): matrix → nested list, h_1 is first list, pop(h_1) is first element in list h_1`

```
(% i2) T(v):=block([h:args(v)],
                  h_1:pop(h), h_2:pop(h),
                  matrix([1,0,pop(h_1)],
                        [0,1,pop(h_2)],
                        [0,0,1]))$
```

```
(% i3) P:[R+r+%rho,0,1]$
```

```
(% i4) M_1:R(%alpha) . [r+R,0,1]$
```

```
(% i5) A:trigreduce(T(M_1) . R(%beta) . T(-M_1) . R(%alpha));
```

$$\begin{pmatrix} \cos(\beta + \alpha) & -\sin(\beta + \alpha) & -\cos(\beta + \alpha)r + \cos(\alpha)r - \cos(\beta + \alpha)R + \cos(\alpha)R \\ \sin(\beta + \alpha) & \cos(\beta + \alpha) & -\sin(\beta + \alpha)r + \sin(\alpha)r - \sin(\beta + \alpha)R + \sin(\alpha)R \\ 0 & 0 & 1 \end{pmatrix}$$

```
(% i6) P_s:trigreduce(A . P);
```

$$\begin{pmatrix} \cos(\alpha)r + \cos(\alpha)R + \cos(\beta + \alpha)\rho \\ \sin(\alpha)r + \sin(\alpha)R + \sin(\beta + \alpha)\rho \\ 1 \end{pmatrix} \Leftrightarrow (R + r) \begin{pmatrix} \cos \alpha \\ \sin \alpha \\ 1 \end{pmatrix} + \rho \begin{pmatrix} \cos(\alpha + \beta) \\ \sin(\alpha + \beta) \\ 1 \end{pmatrix}$$

4 | Rotationen - Basics

4.1 Matrixmultiplikation

Seien A und B zwei (3×3) Matrizen dann ist $C = (c_{ij})$

$$C = A \cdot B \Leftrightarrow c_{ij} := a_{ik} b_{kj} \quad \text{Summationskonvention: doppelter Index bedeutet Summe}$$

Die Multiplikation ist nur definiert, wenn Spaltenanzahl von A gleich Zeilenanzahl von B ist! Diese Verknüpfung erlaubt 2 Deutungen:

4.1.1 Skalare Produkte

Bei fixem i und j durchläuft k bei A den Spaltenindex (greift also den i -ten Zeilenvektor heraus) und bei B den Zeilenindex (greift also den j -ten Spaltenvektor heraus).

Seien $\vec{a}_1, \vec{a}_2, \vec{a}_3$ die Zeilenvektoren von A und $\vec{b}_1, \vec{b}_2, \vec{b}_3$ die Spaltenvektoren von B , also

$$A = \begin{pmatrix} \vec{a}_1 \\ \vec{a}_2 \\ \vec{a}_3 \end{pmatrix} \quad B = \begin{pmatrix} \vec{b}_1 & \vec{b}_2 & \vec{b}_3 \end{pmatrix} \Rightarrow A \cdot B = \begin{pmatrix} \vec{a}_1 \cdot \vec{b}_1 & \vec{a}_1 \cdot \vec{b}_2 & \vec{a}_1 \cdot \vec{b}_3 \\ \vec{a}_2 \cdot \vec{b}_1 & \vec{a}_2 \cdot \vec{b}_2 & \vec{a}_2 \cdot \vec{b}_3 \\ \vec{a}_3 \cdot \vec{b}_1 & \vec{a}_3 \cdot \vec{b}_2 & \vec{a}_3 \cdot \vec{b}_3 \end{pmatrix}$$

Beachte, dass das skalare Produkt zweier Vektoren als Matrixmultiplikation gedeutet werden kann. Hat man zwei Spaltenvektoren \vec{a}, \vec{b} , so ist das skalare (innere) Produkt

$$\vec{a} \cdot \vec{b} := \vec{a}^T \cdot \vec{b} \quad \text{wobei links das skalare Produkt gemeint ist, rechts die Matrixmultiplikation}$$

Manchmal ist auch das äußere (dyadische) Produkt zweier Vektoren von Interesse:

$$\vec{a} \otimes \vec{b} := \vec{a} \cdot \vec{b}^T \quad \text{ergibt eine 3 mal 3 Matrix; rechts ist die Matrixmultiplikation gemeint}$$

4.1.2 Linearkombinationen

Nochmals die Definition:

$$C = A \cdot B \Leftrightarrow c_{ij} := a_{ik} b_{kj} \quad \text{Summationskonvention: doppelter Index bedeutet Summe}$$

Lassen wir in obiger Formel i alle Werte durchlaufen, dann wird aus c_{ij} der j -te Spaltenvektor \vec{c}_j der Produktmatrix, aus a_{ik} der k -te Spaltenvektor \vec{a}_k von A , wobei über k summiert wird, also

$$\vec{c}_j = \sum_k \vec{a}_k b_{kj} \quad \underbrace{\quad}_{\text{Summenkonvention}} \quad \vec{a}_k b_{kj}$$

4. Rotationen - Basics

Also in der Produktmatrix steht in der j -ten Spalte die Linearkombination der Spaltenvektoren von A , wobei die Koeffizienten der Linearkombination in der j -ten Spalte von B stehen! Also

$$\vec{a}_i \in \mathbb{R}^3 : (\vec{a}_1 \ \vec{a}_2 \ \vec{a}_3) \cdot \begin{pmatrix} b_1 & b_4 \\ b_2 & b_5 \\ b_3 & b_6 \end{pmatrix} = A_{(3 \times 3)} \cdot B_{(3 \times 2)} \left(\sum_{i=1}^3 \vec{a}_i b_i \quad \sum_{i=4}^6 \vec{a}_i b_i \right) = C_{(3 \times 2)}$$

Spezialfall "Selektionsmatrizen":

$$(\vec{a}_1 \ \vec{a}_2 \ \vec{a}_3) \cdot \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} = (\vec{a}_3 \ \vec{a}_2 \ \vec{a}_1) \quad \text{Erste und dritte Spalte werden vertauscht}$$

Mit Theorem 5.3 und der Tatsache, dass Rotationen lineare Abbildungen sind, lassen sich leicht deren Matrixdarstellungen herleiten - z.B. die Rotation um die x -Achse um $\alpha \rightarrow R_{x,\alpha}$. Sei $R_{x,\alpha}(\vec{e}_i) =: \vec{r}_i$, dann gilt

$$\begin{aligned} \vec{p}' &= R_{x,\alpha}(\vec{p}) = R_{x,\alpha}(\lambda \vec{e}_1 + \mu \vec{e}_2 + \nu \vec{e}_3) = \lambda R_{x,\alpha}(\vec{e}_1) + \mu R_{x,\alpha}(\vec{e}_2) + \nu R_{x,\alpha}(\vec{e}_3) = \\ &= (\vec{r}_1 \ \vec{r}_2 \ \vec{r}_3) \cdot \begin{pmatrix} \lambda \\ \mu \\ \nu \end{pmatrix} = R_x \vec{p} = \vec{p}' \Rightarrow \text{Rotationsmatrix } R_x \text{ besteht aus } \vec{r}_i \end{aligned}$$

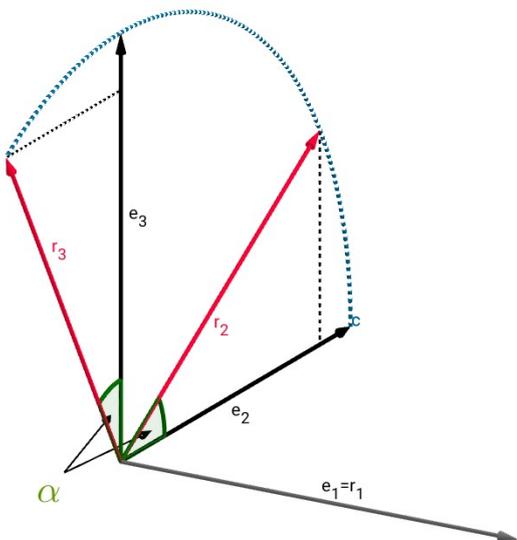


Abb.37 : Rotation der Basisvektoren

$$\begin{aligned} \vec{r}_1 &= \vec{e}_1 = (1, 0, 0)^t \\ \vec{r}_2 &= (0, \cos \alpha, \sin \alpha)^t \\ \vec{r}_3 &= (0, -\sin \alpha, \cos \alpha)^t \end{aligned}$$

Damit ergibt sich die Rotationsmatrix

$$R_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \alpha & -\sin \alpha \\ 0 & \sin \alpha & \cos \alpha \end{pmatrix} \quad (4.1)$$

Siehe auch Theorem 5.2

Definition 4.1 Mathematisch positive-aktive Drehung

Wir schauen in Richtung des Drehachsenvektors und drehen nach **rechts**
 Unterschied zwischen aktiver und passiver Drehung siehe Kapitel 5.6

4.2 Rotation um eine beliebige Achse durch den Ursprung

Wir drehen unser KS derart, dass die Rotationsachse auf die x -Achse zu liegen kommt:
 $\vec{e}_1, \vec{e}_2, \vec{e}_3$ seien unsere Standardbasisvektoren in Spaltenschreibweise. \vec{e}'_1 sei der normalisierte Vektor in Richtung der Drehachse. Wir konstruieren ein neues Orthonormalsystem:

$$\vec{e}'_2 = \frac{\vec{e}_1 \times \vec{e}'_1}{\|\vec{e}_1 \times \vec{e}'_1\|} \quad \text{und} \quad \vec{e}'_3 = \vec{e}'_1 \times \vec{e}'_2$$

Es gelten folgende Gleichungen (mit Summationskonvention):

$$\vec{e}'_i = a_{ji} \vec{e}_j \quad \Leftrightarrow \quad (\vec{e}'_1, \vec{e}'_2, \vec{e}'_3) = (\vec{e}_1, \vec{e}_2, \vec{e}_3) \cdot A \Big| A^{-1} \quad (4.2)$$

Matrixschreibweise

$$\vec{e}_k = b_{jk} \vec{e}'_j \quad \Leftrightarrow \quad (\vec{e}'_1, \vec{e}'_2, \vec{e}'_3) \cdot A^{-1} = (\vec{e}_1, \vec{e}_2, \vec{e}_3) = I \quad (4.3)$$

Matrixschreibweise

Aus 4.3 folgt unmittelbar $A = (\vec{e}'_1, \vec{e}'_2, \vec{e}'_3)$

Die Zahlen b_{jk} stellen die Zahlen der invertierten Matrix von A dar!

Da beide Basissysteme Orthonormalsysteme sind gilt:

$$\vec{e}_i \cdot \vec{e}_j = \vec{e}'_i \cdot \vec{e}'_j = \delta_{ij} \quad (\text{Kronecker Delta})$$

Wir multiplizieren nun obige Gleichung 4.2 mit \vec{e}_k bzw. Gleichung 4.3 mit \vec{e}'_i , dann ergibt sich

$$\vec{e}'_i \cdot \vec{e}_k = a_{ji} \delta_{jk} = a_{ki} \quad (4.4)$$

$$\vec{e}'_i \cdot \vec{e}_k = b_{jk} \delta_{ij} = b_{ik} \quad (4.5)$$

Einerseits wissen wir aus 4.4 wie die Matrix A zu berechnen ist und da $b_{ik} = a_{ki}$ gilt in Matrixschreibweise

$$B = A^T = A^{-1}$$

Um den Vektor \vec{x} um \vec{e}'_1 um α zu rotieren liegt jetzt unsere Vorgangsweise klar vor uns:

1. Wir transformieren \vec{x} ins gestrichene Koordinatensystem:

$$x_k \vec{e}_k = \underbrace{x_k b_{jk}}_{x'_j} \vec{e}'_j \Rightarrow x'_j = x_k b_{jk} \Rightarrow \vec{x}' = B \cdot \vec{x} = A^T \vec{x}$$

4. Rotationen - Basics

2. Rotieren dort mit $R_x(\alpha)$

3. Wir machen die Transformation rückgängig mit $(A^T)^{-1} = (A^T)^T = A$

$$\vec{x}_\alpha = \underbrace{A R_x(\alpha) A^T}_T \vec{x} = T \vec{x} \quad (4.6)$$

Dies wäre die Formel für Spaltenvektoren, für Zeilenvektoren müssen wir linke und rechte Seite transponieren (Reihenfolge der Multiplikation dreht sich um):

$$\vec{x}_\alpha^T = \underbrace{(A R_x(\alpha) A^T \vec{x})^T}_T = \vec{x}^T \underbrace{A R_x(-\alpha) A^T}_{T_Z}$$

Wir probieren das gleich in *GNU-Octave* bzw. *wxMaxima* aus, wobei wir notgedrungen die Nomenklatur etwas ändern müssen:

$\vec{e}'_i \rightarrow$ `e_ip` “p” for primed, in German “Strich”
zuerst das *GNU-Octave* - Skript dann das *wxMaxima* - Programm

```

1 # all vectors are column vectors here
2 # rotation axis, point to rotate and angle of rotation
3 axis=[2;-2;1]; X=[0.5;0;0.5]; rotAngle=pi/3;

5 # we build a unit vector
6 function v0=unitVec(v)
7     n=norm(v);
8     v0=1/n*v;
9 endfunction

11 # build the rotation-matrix for x-axis about angle a
12 function Rx=R_x(a)
13     Rx=[1,0,0; 0,cos(a),-sin(a); 0,sin(a),cos(a)];
14 endfunction
15

17 e_1=[1;0;0]; e_2=[0;1;0]; e_3=[0;0;1];
18 standardBasis=[e_1, e_2, e_3];
19 #we build the primed basis
20 e_1p=unitVec(axis); e_2p=unitVec(cross(e_1p,e_1));
21 e_3p=cross(e_1p, e_2p);
22 # we use here that [e_1,e_2,e_3] is the identity matrix
23 # if we use any other basis then we need to build a matrix
24 # with the dot-product of primed and unprimed basis
25 # — we have done this in wxMaxima
26 primedBasis=[e_1p, e_2p, e_3p];
27 A=primedBasis;
28 # we construct the transformation matrix
29 T=A * R_x(rotAngle) * A';
30 # and transform the point X
31 X_p=T * X;
32 #we display the result
33 disp("Rotated point is:"),disp(X_p);

```

Output:

>> rot_axis1

Rotated point is:

0.12799

-0.31100

0.62201

>>

Nun zum *wxMaxima* - Programm, bei dem wir nicht voraussetzen, dass die Ausgangsbasis die Standardbasisvektoren sind :

(%i1) load("vect")\$

Axis of rotation and point X to be rotated (arbitrary)

(%i3) axis:[2,-2,1]\$ X:[0.5,0,0.5]\$

We use 60° for rotation-angle

(%i4) rotAngle:%pi/3\$

Conversion row- to columnvector and unitvector respectively

```

(%i6) row2col(v):=matrix([v[1]], [v[2]], [v[3]])$
      unitVec(v):= 1/sqrt(v . v)*v$

```

General rotationmatrix around x-axis by α

4. Rotationen - Basics

```
(%i7) R_x(%alpha):=matrix([1,0,0],
                        [0,cos(%alpha),-sin(%alpha)],
                        [0,sin(%alpha),cos(%alpha)])$
```

Specific rotationmatrix here

```
(%i8) R_x:R_x(rotAngle),numer$
```

```
(%i11) e_1:[1,0,0]$e_2:[0,1,0]$e_3:[0,0,1]$
```

Define standardbasis

```
(%i12) standardBase:[e_1,e_2,e_3];
```

```
(%i15) e_1p:unitVec(axis), numer$ e_2p:unitVec(express(e_1p ~ e_1))$
      e_3p:express(e_1p ~ e_2p)$
```

```
(%i16) primedBase:[e_1p,e_2p,e_3p]$
```

Generate matrix A with 4.4 - we don't use that \vec{e}_i is standardbasis

```
(%i17) A:genmatrix(lambda([k,i], primedBase[i] . standardBase[k]), 3, 3)$
```

Now the transformationmatrix

```
(%i18) T:A . R_x . transpose(A)$
```

Construct the rotated point X_p , output as row-vector only because of space-usage

```
(%i19) X_p:transpose(T . row2col(X));
```

```
(%o19) (0.128 -0.311 0.622)
```

Hier jetzt die *Geogebra*-3d Zeichnung dazu

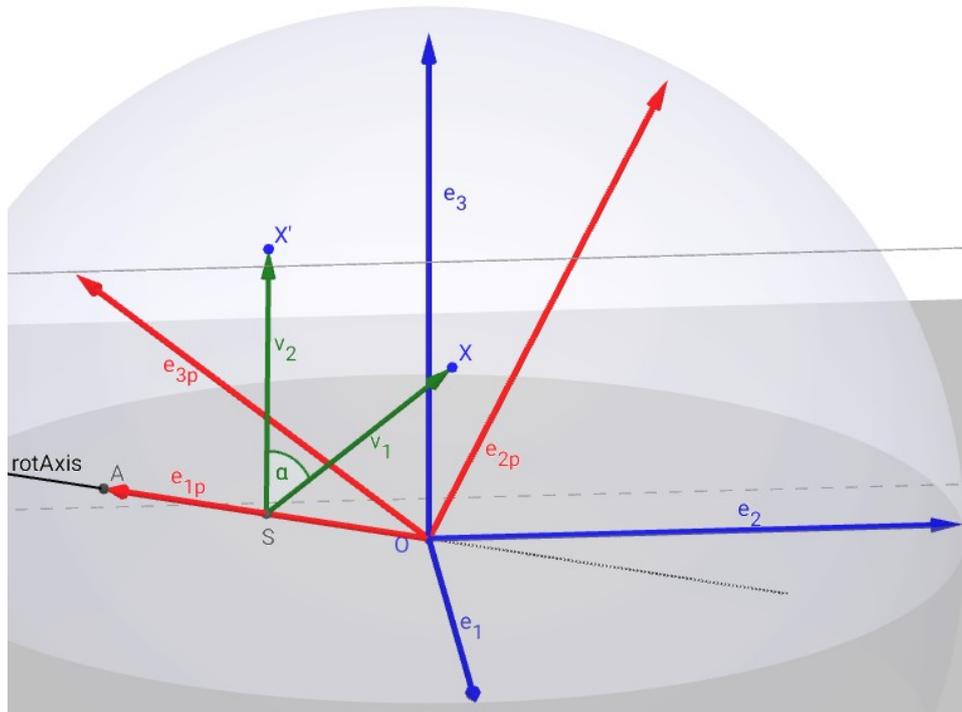


Abb.38 : Rotation um eine Gerade durch den Ursprung in *Geogebra*

No.	Name	Description	Value
1	e_1	Vector((1, 0, 0))	$e_1 = (1, 0, 0)$
2	e_2	Vector((0, 1, 0))	$e_2 = (0, 1, 0)$
3	e_3	Vector((0, 0, 1))	$e_3 = (0, 0, 1)$
4	Point O		$O = (0, 0, 0)$
5	Sphere k_1	Sphere with center O and radius 1	$k_1: x^2 + y^2 + z^2 = 1$
6	rotAxis	Vector((2, -2, 1))	rotAxis = (2, -2, 1)
7	Point A	$O + \text{UnitVector}(\text{rotAxis})$	$A = (0.6667, -0.6667, 0.3333)$
8	e_{1p}	Vector(O, A)	$e_{1p} = (0.6667, -0.6667, 0.3333)$
9	e_{2p}	Unit vector of $e_{1p} \otimes e_1$	$e_{2p} = (0, 0.4472, 0.8944)$
10	e_{3p}	$e_{1p} \otimes e_{2p}$	$e_{3p} = (-0.7454, -0.5963, 0.2981)$
11	Point X		$X = (0.5, 0, 0.5)$
12	Line b	Line O, A	$b: X = (0, 0, 0) + \lambda (0.6667, -0.6667, 0.3333)$
13	Plane ϵ_1	Plane through X perpendicular to b	$\epsilon_1: 0.6667x - 0.6667y + 0.3333z = 0.5$
14	Point S	Intersection point of b, ϵ_1	$S = (0.3333, -0.3333, 0.1667)$
15	Point X'	X rotated by angle 60° about b	$X' = (0.128, -0.311, 0.622)$
16	v_1	Vector(S, X)	$v_1 = (0.1667, 0.3333, 0.3333)$
17	v_2	Vector(S, X')	$v_2 = (-0.2053, 0.0223, 0.4553)$
18	Angle α	Angle between v_1, v_2	$\alpha = 60^\circ$

Abb.39 : Konstruktionsprotokoll

Gehen wir das Konstruktionsprotokoll durch:

4. Rotationen - Basics

1. 1-4 Standardbasisvektoren und Ursprung festlegen
2. 5 Einheitskugel festlegen (jeder Endpunkt eines Orthonormalsystems muss sich darauf befinden!)
3. 6-7 Endpunkt A von \vec{e}'_1 (Rotationsachse) festlegen
4. 8-10 gestrichenes Orthonormalsystem festlegen (Kreuzprodukt mit `Cross[e_{1p}, e_1]`)
5. 11 X festlegen (zu rotierender Punkt um Achse "axis")
6. 12 Rotationsachsen-Gerade b festlegen
7. 13 Ebene ε_1 festlegen: b ist Normale und geht durch X
8. 14 Schneide ε_1 mit b (x' -Achse) : C
9. 15 Drehe X um b um 60° : X' – hier kann man sich von der Richtigkeit unserer Rechnung überzeugen!
10. 16-18 Bestimme den Winkel α zwischen X und X' (Probe)

Hier noch der Link zum obigen *Geogebra*-Arbeitsblatt

<http://www.angsuesser.at/docs/math/geogebra/rot-axis4.ggb>

zum Experimentieren (X ist frei verschiebbar)

4.3 Rotation um eine beliebige Achse außerhalb des Ursprungs(1)

Am einfachsten ist die Sache in *Geogebra*:

Alles bleibt gleich - nur bestimmen wir einen Punkt M , durch den wir die Rotationsachse durchlegen - das Konstruktionsprotokoll ist analog wie oben. Sowohl M wie auch X sind "freie Punkte" und können in der Konstruktion beliebig verschoben werden!

Hier noch der Link zum diesem *Geogebra*-Arbeitsblatt

<http://www.angsuesser.at/docs/math/geogebra/rot-axis5.ggb>

Die mathematische Theorie um X' zu berechnen ist etwas aufwendiger:

Wir können diesen Fall auf obigen (Achse im Ursprung) zurückführen, indem wir eine Verschiebung durchführen. Mit Matrizen macht man das indem man auf homogene Koordinaten umstellt:

$$(x_1, x_2, x_3) \rightarrow (x_1, x_2, x_3, 1)$$

Man bettet also den \mathbb{R}^3 in den \mathbb{R}^4 ein, hält dabei die 4.-te Koordinate bei 1 fest! Damit lässt sich eine Translation als Matrizenmultiplikation darstellen - die Matrix V verschiebt jeden Vektor um (v_x, v_y, v_z) wie man leicht feststellt:

$$V = \begin{pmatrix} 1 & 0 & 0 & v_x \\ 0 & 1 & 0 & v_y \\ 0 & 0 & 1 & v_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \Rightarrow V \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + v_x \\ y + v_y \\ z + v_z \\ 1 \end{pmatrix} \quad V^{-1} = \begin{pmatrix} 1 & 0 & 0 & -v_x \\ 0 & 1 & 0 & -v_y \\ 0 & 0 & 1 & -v_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

damit ist der Algorithmus klar: wir verschieben die Lage der Drehachse in den Ursprung (mit V^{-1}) führen dann die Prozedur für “Drehachse im Ursprung” durch und anschl. verschieben wir mit V zurück!

Damit ergibt sich die Transformationsmatrix $T = V A R_x(\alpha) A^T V^{-1}$

$$\boxed{\vec{x}_\alpha = \underbrace{V A R_x(\alpha) A^T V^{-1}}_T \vec{x}} \quad \text{wobei } A = \left(\begin{array}{ccc|c} A_{3 \times 3} & & & 0 \\ & & & 0 \\ & & & 0 \\ \hline & & & 1 \end{array} \right) \quad \text{analog die anderen Matrizen!}$$

Jetzt versuchen wir das in *wxmaxima* bzw. *GNU-Octave* hinzubekommen, außer Verschiebungsmatrix V und Punkt M und anderes X bleibt ja alles gleich!

```
(%i1) load("vect")$
direction and point of rotationaxis, X point to rotate
(%i3) axis:[2,-2,1]$ M:[0.3,0.2,0.2]$ X:[1,0.5,0.5] $
(%i5) rotAngle:%pi/3 /* amount of rotation around axis */$
(%i7) row2col4(v):=matrix([v[1]], [v[2]], [v[3]], [1])$ unitVec(v):= 1/sqrt(v . v)*v$
(%i8) R_x(%alpha):=matrix([1,0,0],
                          [0,cos(%alpha),-sin(%alpha)],
                          [0,sin(%alpha),cos(%alpha)])$
adds to matrix M the columnvector(3D) v and the row [0,0,0,1]
(%i9) extend3DimMat(M,v):=block([unit:[0,0,0,1]],
                                M:addrow(addcol(M,v), unit))$
so this makes a 4-dim translationmatrix
(%i10) mk_Trans_Mat(v):=extend3DimMat(diagmatrix(3,1),v)$
convert to 4-dim without any translation
(%i11) mk_extended_Mat(M):= extend3DimMat(M,[0,0,0])$
remaining commands are known from previous program
(%i12) R_x:R_x(rotAngle),numer$
(%i15) e_1:[1,0,0]$e_2:[0,1,0]$e_3:[0,0,1]$
(%i16) standardBase:[e_1,e_2,e_3]$
(%i19) e_1p:unitVec(axis), numer$e_2p:unitVec(express(e_1p ~ e_1))$
      e_3p:express(e_1p ~ e_2p)$
(%i20) primedBase:[e_1p,e_2p,e_3p]$
(%i21) A:genmatrix(lambda ([k, i], primedBase[i] . standardBase[k] ), 3, 3)$
3-dimensional Transformationmatrix (around the origin)
(%i22) T3_origin: A . R_x . transpose(A)$
```

4. Rotationen - Basics

Transformation with explicit Translation - correct result in cart. coordinates

```
(%i23) X_p: transpose(T3_origin . (X - M)+M);
```

```
( 0.512  0.257  0.988 )
```

Do it all with one 4-dim transformationmatrix $T4$ - result in homogenous coordinates

```
(%i24) T4:mk_Trans_Mat(M) . mk_extended_Mat(T3_origin) . mk_Trans_Mat(-M)$
```

```
(%i25) X_p:transpose(T4 . row2col4(X));
```

```
( 0.512  0.257  0.988 )
```

Jetzt mit *GNU-Octave*

```
# all vectors are column vectors here
# rotation axis (direction and support
  point),
axis=[2;-2;1]; M=[0.3;0.2;0.2];
#point to rotate and angle of rotation
X=[1;0.5;0.5;1]; rotAngle=pi/3;
```

```
# we build a unit vector
```

```
function v0=unitVec(v)
```

```
  n=norm(v);
```

```
  v0=1/n*v;
```

```
endfunction
```

```
# build the rotation-matrix for x-axis
  about angle 'a'
```

```
# for homogenous coordinates
```

```
function Rx=Rx_hom(a)
```

```
  Rx=[1,0,0,0; 0,cos(a),-sin(a),0; 0,sin(
```

```
  a),cos(a),0;0,0,0,1];
```

```
endfunction
```

```
function tr=getTranslationMatrix(t)
```

```
  tr=eye(4);
```

```
  for i=1:3
```

```
    tr(i,4)=t(i);
```

```
  endfor
```

```
endfunction
```

```
V=getTranslationMatrix(M);
```

```
V_inv=getTranslationMatrix(-M);
```

```
e_1=[1;0;0]; e_2=[0;1;0]; e_3=[0;0;1];
```

```
standardBasis=[e_1, e_2, e_3];
```

```
#we build the primed basis
```

```
e_1p=unitVec(axis); e_2p=unitVec(cross(
```

```
  e_1p,e_1));
```

```
e_3p=cross(e_1p, e_2p);
```

```
# we use here that [e_1,e_2,e_3] is the
```

```
  identity matrix
```

```
primedBasis=[e_1p, e_2p, e_3p];
```

```
#we extend for homogenous coordinates -
```

```
  column of zeros and unit row!
```

```
A=[primedBasis [0;0;0];[0,0,0,1]];
```

```
# we construct the transformation matrix
```

```
T= V * A * Rx_hom(rotAngle) * A' * V_inv;
```

```
# and transform the point X
```

```
X_p=T * X;
```

```
#we display the result and omitting the
```

```
  fourth coordinate
```

```
disp("Rotated point is:"); disp(X_p(1:3));
```

Output:

```
>> rot_axis2
      Rotated point is:
      0.51241  0.25665  0.98846
```

4.4 Rotation um eine Achse(2) - Eulerwinkel

Eine andere Möglichkeit um die Transformationsmatrix zu berechnen, wäre die Zerlegung in Achsenrotationen. Also durch M gehe eine Rotationsachse $\vec{v} = (v_x, v_y, v_z)^t$ mit $|\vec{v}| = 1$. Die Rotation soll einen Winkel α betragen. (Frage: Ist die aus den Rotationen gewonnene Matrix C ident mit der Matrix A^T aus dem vorigen Abschnitt ?)

Wichtig dabei sind die Projektionen von \vec{v} auf die Koordinatenebenen π_i mit $\vec{v}_{xy} = (v_x, v_y, 0)^t$, $\vec{v}_{xz} = (v_x, 0, v_z)^t$ und $\vec{v}_{yz} = (0, v_y, v_z)^t$. Die Idee ist \vec{v} (roter Pfeil) auf eine Koordinatenachse zu drehen, anschl. die Rotation durchzuführen und dann zurückzudrehen. Nehmen wir als "Zielachse" die x-Achse, dann können wir mit obiger Rechnung direkt vergleichen (rot - x-Achse, grün - y-Achse, blau - z-Achse).

Überlege, dass es mehrere Möglichkeiten gibt dieselbe Achsenrotation in 3 Rotationen zu zerlegen. Außerdem muss \vec{v}_{xy} und damit β_1 nicht existieren (\vec{v} schaut dann in z-Richtung). Die Beschreibung einer Rotation mit Euler-Winkel (hier eigentlich Tait-Bryan Winkel) ist daher nicht unbedingt "das gelbe vom Ei", weil keine Eindeutigkeit (Bijektivität) gewährleistet ist. Wir werden noch andere Möglichkeiten später aufzeigen.

In der Navigation hat man meist das umgekehrte Problem: Ein Flugzeug fliegt einen bestimmten Kurs (x-Achse) horizontal, es soll ein neuer Kurs mit Abweichung β_1 ("Gier-Winkel") geflogen werden, die "Nase" des Flugzeugs soll sich um β_2 nach oben richten und das Flugzeug soll sich um den Winkel α (Rollwinkel) um die eigene Achse drehen (Roll-Nick-Gier-Winkel, englisch roll-pitch-yaw angle).

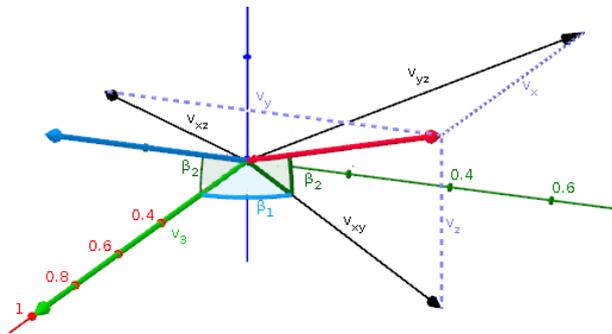


Abb.40 : Zerlegung in Euler-Winkel

1. Wir drehen im Uhrzeigersinn (math. negativ $\Leftrightarrow v_2 > 0$) um die z-Achse um β_1 (ergibt blauen Pfeil) - dadurch wird $|\vec{v}_{xy}|$ die x-Koordinate des blauen Pfeils!
2. Wir drehen entgegen den Uhrzeigersinn (math. positiv $\Leftrightarrow v_3 \geq 0$) um die y-Achse um β_2 (ergibt grünen Pfeil)
3. Jetzt drehen wir um die x-Achse um α und anschl. machen wir die vorigen Transformationen wieder rückgängig

$$T = \underbrace{R_z^{-1}(-\operatorname{sgn}(v_2)\beta_1) \cdot R_y^{-1}(\operatorname{sgn}(v_3)\beta_2)}_{C^{-1}=C^t \text{ siehe Theorem 5.1}} \cdot R_x(\alpha) \cdot \underbrace{R_y(\operatorname{sgn}(v_3)\beta_2) \cdot R_z(-\operatorname{sgn}(v_2)\beta_1)}_C$$

Beachte R_z im Uhrzeigersinn, R_y entgegen!

Das setzen wir jetzt mit *wxMaxima* um und machen damit zugleich die "Probe" für den vorigen Abschnitt!

```
(%i1) load("vect")$
specification as in previous section
(%i3) axis:[2,-2,1]$ M:[0.3,0.2,0.2]$ X:[1,0.5,0.5]$
```

4. Rotationen - Basics

```
(%i5) rotAngle:%pi/3 /* amount of rotation around axis */$
(%i7) row2col(v):=matrix([v[1]],[v[2]],[v[3]])$ unitVec(v):= 1/sqrt(v . v)*v$
rotationmatrices
(%i8) R_x(%alpha):=matrix([1,0,0],
                        [0,cos(%alpha),-sin(%alpha)],
                        [0,sin(%alpha),cos(%alpha)])$
(%i9) R_y(%alpha):=matrix([cos(%alpha),0,sin(%alpha)],
                        [0,1,0],
                        [-sin(%alpha),0, cos(%alpha)])$
(%i10) R_z(%alpha):=matrix([cos(%alpha),-sin(%alpha),0],
                        [sin(%alpha),cos(%alpha),0],
                        [0,0,1])$

standard basis
(%i13) e_1:[1,0,0]$e_2:[0,1,0]$e_3:[0,0,1]$

“working horse” - we determine a set of angles (one of many)
(%i14) mk_Transform_Mat(v, angle):=block([%beta1:0, %beta2:0, signBeta1:1,
                                        signBeta2:1, C:diagmatrix(3,1) ],
    if ((v[1] = 0) and (v[2] = 0)) then float(R_z(angle))
    else
    block(
        %beta1:float(acos(unitVec([v[1],v[2],0]) . e_1)),
        %beta2:float(acos(unitVec(v) . unitVec([v[1],v[2],0]))),
        C:R_y(signum(v[3]) * %beta2) . R_z(-signum(v[2]) * %beta1),
        transpose(C) . float(R_x(angle)) . C /* return value */
    ) )$
(%i15) T:mk_Transform_Mat(axis, rotAngle)$
(%i16) X_p: T . (X-M)+M;
```

$$(\%o16) \begin{pmatrix} 0.5124146010868906 \\ 0.2566452912372587 \\ 0.9884613803007369 \end{pmatrix}$$

In *GNU-Octave* berechnen wir die Eulerschen Winkel anders:

Anstatt die Fallunterscheidungen für die Vorzeichen der Winkel selbst vorzunehmen, transformieren wir die entsprechenden Vektoren auf die komplexe Ebene und lassen uns mit der Funktion *arg* den Winkel berechnen!

Demonstrieren wir das an Hand von Abb. 40 für β_1 und β_2 :

β_1 ist das Argument der komplexen Zahl v_{xy} also $(v_x + i v_y) = v(1) + i v(2)$ (in *GNU-Octave*), wobei das Ergebnis aus $[-\pi, \pi]$ ist (je nach Vorzeichen von $v(2)$) und nicht wie bei der Ermittlung durch das skalare Produkt aus $[0, \pi]$!

Für die Berechnung von β_2 legen wir reelle Achse durch \vec{v}_{xy} und die imaginäre Achse von \mathbb{C} durch die z -Achse:

β_2 ist dann das Argument der komplexen Zahl $\sqrt{v_x^2 + v_y^2} + i v_z$

```

1 # euler angles
2 # all vectors are column vectors here
3 # rotation axis, point to rotate and angle of rotation
  axis=[2;-2;1]; M=[0.3;0.2;0.2]; X=[1;0.5;0.5]; rotAngle=pi/3;
5 v=axis;
  z_1=v(1)+i*v(2);
7 z_2=sqrt(v(1)^2+v(2)^2)+i*v(3);

9 beta_1=arg(z_1); beta_2=arg(z_2);

11 # build the rotation-matrix for x/y/z-axis about angle a
  function r=R(axis,a)
13   switch (axis)
15     case 'x'
       r=[1,0,0; 0,cos(a),-sin(a); 0,sin(a),cos(a)];
17     case 'y'
       r=[cos(a),0,sin(a); 0,1,0; -sin(a),0,cos(a)];
       otherwise
19     r=[cos(a),-sin(a),0; sin(a),cos(a),0; 0,0,1];
  endswitch
21 endfunction

23 function t=getTransformMat(z1,z2,angle)
  C=R('y',arg(z2))*R('z',-arg(z1));
25 t=C'*R('x',angle)*C;
  endfunction
27
29 #Calculate the rotated point
  X_p=getTransformMat(z_1,z_2,rotAngle)*(X-M)+M

```

Output:

```

>> euler_angles
X_p =
    0.51241
    0.25665
    0.98846
>>

```

4.5 Rotation um eine beliebige Achse(3) - Rodrigues

Eine andere Möglichkeit der Darstellung einer Drehung (besonders in der Physik verwendet) ist die durch einen Vektor

$$\vec{\omega} = \theta \hat{n} \quad \text{mit} \quad |\hat{n}| = 1$$

\hat{n} ist dabei die Drehachse, θ entspricht dem Drehwinkel.

Bevor wir die Rodrigues(z)-Formel anschreiben noch eine kurze Vorbemerkung zur Matrixschreibweise eines Vektorprodukts(Summationskonvention):

$$(\vec{a} \times \vec{b})_i = \underbrace{\varepsilon_{ijk} a_j}_{c_{ik}} b_k = c_{ik} b_k \Rightarrow \vec{a} \times \vec{b} = [\vec{a}]_{\times} \vec{b}$$

Definition 4.2 Definition der Matrix $[\vec{a}]_{\times}$ bzw. \vec{a}_{\times}

$$[\vec{a}]_{\times} := \varepsilon_{ijk} a_j = \begin{pmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{pmatrix} \quad \text{und} \quad \vec{v}(\vec{n})_{\times} := [\vec{n}]_{\times} \vec{v} = \vec{n} \times \vec{v}$$

Falls die Rotationsachse \vec{n} klar ist, schreiben wir einfach \vec{v}_{\times} statt $\vec{v}(\vec{n})_{\times}$!

Die Matrix $[\vec{a}]_{\times}$ erbt die Antisymmetrie vom Levi-Civita Symbol und die Hauptdiagonale verschwindet überall.

Es gilt dann natürlich

$$[\vec{a}]_{\times}^2 \vec{b} = \vec{a} \times (\vec{a} \times \vec{b})$$

Nun zur Rodrigues-Formel in Matrixschreibweise und darunter in Vektorschreibweise

$$R(\hat{n}, \theta) = I + \sin \theta [\hat{n}]_{\times} + (1 - \cos \theta) [\hat{n}]_{\times}^2 \quad (4.7)$$

$$\vec{v}_{rot} = \vec{v} \cos \theta + (\hat{n} \times \vec{v}) \sin \theta + \hat{n}(\hat{n} \cdot \vec{v})(1 - \cos \theta) \quad (4.8)$$

$$\vec{v}_{rot} = \vec{v}_{\perp} \cos \theta + (\hat{n} \times \vec{v}) \sin \theta + \vec{v}_{\parallel} \quad (4.9)$$

Wir “probieren” 4.7 in *wxMaxima* aus:

(% i1) `load("vect")$`

Angaben wie vorher

(% i2) `(axis:[2,-2,1], M:[0.3,0.2,0.2], X:[1,0.5,0.5], rotAngle:float(%pi/3))$`

Hilfsfunktionen: Erstelle Spaltenvektor und berechne Einheitsvektor

```
(% i3) row2col(v):=matrix([v[1]],[v[2]],[v[3]]), unitVec(v):= 1/sqrt(v . v)*v)$
```

Einheitsvektor der Drehachse als Zeilenvektor

```
(% i4) n_0:unitVec(axis)$
```

Benutzerdefinierten Operator festlegen mit Anfang- und Endesymbolen

```
(% i5) matchfix("|", "|_x")$
```

Erstellt Matrix für Kreuzprodukt, Input: Zeilenvektor!

```
(% i6) f(x):=matrix([0,-x[3],x[2]], [x[3],0,-x[1]],[-x[2],x[1],0])$
```

Verknüpft Operatorsymbole mit auszuführender Funktion

```
(% i7) |n|_x:=f(n)$
```

Erstellung der Rodriguez-Formel - beachte den Operator $\hat{\cdot}$ für das Quadrieren der Matrix

```
(% i8) Rod(n,theta):=diagmatrix(3,1)+sin(theta)*|n|_x+(1-cos(theta))*(|n|_x )2$
```

Anwenden der Formel auf unsere Angaben

```
(% i9) Rod(n_0, rotAngle) . row2col(X-M) +row2col(M);
```

$$\begin{pmatrix} 0.5124146010868907 \\ 0.2566452912372592 \\ 0.9884613803007369 \end{pmatrix}$$

Wie man sieht, stimmt das Ergebnis mit den vorhergehenden ein!

Zuerst zeigen wir, dass 4.7 und 4.8 denselben Sachverhalt darstellen, also

$$R(\hat{n}, \theta) \vec{v} = \vec{v}_{rot}$$

- nachdem der Sinusterm offensichtlich derselbe ist, kürzen wir ihn gleich raus und erhalten

$$\begin{aligned} \vec{v} + (1 - \cos \theta) \hat{n} \times (\hat{n} \times \vec{v}) &= \vec{v} \cos \theta + \hat{n}(\hat{n} \cdot \vec{v})(1 - \cos \theta) \\ \cancel{(1 - \cos \theta)} [\vec{v} + \hat{n} \times (\hat{n} \times \vec{v})] &= \hat{n}(\hat{n} \cdot \vec{v})\cancel{(1 - \cos \theta)} \end{aligned} \quad (4.10)$$

Mit der Graßmann-Identität (siehe Anhang) folgt die Behauptung!

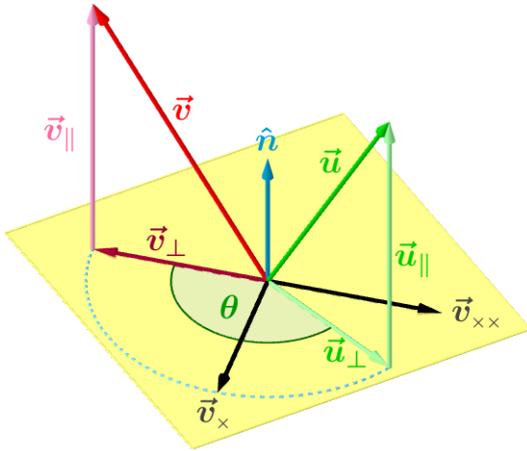
4.8 und 4.9 sind ident, da $\vec{v}_{\parallel} = \hat{n}(\hat{n} \cdot \vec{v})$ und $\vec{v} = \vec{v}_{\perp} + \vec{v}_{\parallel}$ gilt!

4. Rotationen - Basics

Jetzt zur Herleitung der Formel von Rodrigues:

Aus Gleichung 4.10 oben folgt unmittelbar

$$\vec{v} + \vec{v}_{\times\times} = [(I + [\hat{n}]_{\times}^2)\vec{v}] = \vec{v}_{\parallel} \quad \text{und da gilt} \quad \vec{v}_{\parallel} = \vec{v} - \vec{v}_{\perp} \Rightarrow \vec{v}_{\times\times} = -\vec{v}_{\perp}$$



Wegen $\vec{v}_{\times\times} = \hat{n} \times \vec{v}_{\times}$ gilt:

$$\vec{v}_{\times\times} \cdot \vec{v}_{\times} = 0 \quad \text{und} \quad |\vec{v}_{\times\times}| = |\vec{v}_{\times}|$$

also spannen \vec{v}_{\perp} und \vec{v}_{\times} ein kartesisches Koordinatensystem auf und da gilt (mit $\vec{u} = \vec{v}_{rot}$)

$$|\vec{u}_{\perp}| = |\vec{v}_{\perp}| = |\vec{v}_{\times}| \Rightarrow$$

$$\vec{u}_{\perp} = \cos \theta \vec{v}_{\perp} + \sin \theta \vec{v}_{\times} = -\cos \theta [\hat{n}]_{\times}^2 \vec{v} + \sin \theta [\hat{n}]_{\times} \vec{v}$$

$$\vec{v}_{rot} = \vec{u} = \vec{u}_{\perp} + \vec{v}_{\parallel} = -\cos \theta [\hat{n}]_{\times}^2 \vec{v} + \sin \theta [\hat{n}]_{\times} \vec{v} + (I + [\hat{n}]_{\times}^2)\vec{v}$$

Zusammenfassen der letzten Zeile liefert die Behauptung.

Abb.41 : Rodrigues-Formel

Beachte: Für $|\theta| \ll 1$ geht die Rodrigues-Formel über in:



$$\sin \theta \approx \theta \quad \wedge \quad \cos \theta \approx 1 \Rightarrow R(\hat{n}, \theta) = I + \theta [\hat{n}]_{\times} \Rightarrow \vec{v}_{rot} = \vec{v} + \vec{\omega} \times \vec{v}$$

Eine interessante Möglichkeit unsere Formel zu bestätigen ist folgender Sachverhalt:

Eine Rotation um θ ist dasselbe wie k -Rotationen um θ/k :

$$R(\hat{n}, \theta) = \lim_{k \rightarrow \infty} \left[R\left(\hat{n}, \frac{\theta}{k}\right) \right]^k = \lim_{k \rightarrow \infty} \left(I + \frac{\theta}{k} [\hat{n}]_{\times} \right)^k = \exp([\omega]_{\times}) = I + \frac{\theta [\hat{n}]_{\times}}{1} + \frac{\theta^2 [\hat{n}]_{\times}^2}{2} + \frac{\theta^3 [\hat{n}]_{\times}^3}{3!} + \dots$$

mit $[\hat{n}]_{\times}^{k+2} = -[\hat{n}]_{\times}^k$ (siehe ANHANG Theorem 4.8) können wir umformen

$$\begin{aligned} R(\hat{n}, \theta) &= I + \frac{\theta [\hat{n}]_{\times}}{1} + \frac{\theta^2 [\hat{n}]_{\times}^2}{2} - \frac{\theta^3 [\hat{n}]_{\times}}{3!} - \frac{\theta^4 [\hat{n}]_{\times}^2}{4!} + \frac{\theta^5 [\hat{n}]_{\times}}{5!} + \dots \\ &= I + [\hat{n}]_{\times} \underbrace{\left(\frac{\theta}{1} - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} \pm \dots \right)}_{\sin \theta} + [\hat{n}]_{\times}^2 \underbrace{\left(\frac{\theta^2}{2} - \frac{\theta^4}{4!} + \frac{\theta^6}{6!} \pm \dots \right)}_{1 - \cos \theta} \end{aligned}$$

also schließlich wie behauptet

$$R(\hat{n}, \theta) = I + \sin \theta [\hat{n}]_{\times} + (1 - \cos \theta) [\hat{n}]_{\times}^2$$

4.6 Rotation um eine Achse(4) - Quaternionen

Quaternionen lassen sich als Verallgemeinerung von \mathbb{C} als auch von \mathbb{R}^3 auffassen:

$$q := q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3 = (q_0, q_1, q_2, q_3) = \underbrace{q_0}_{\text{Skalarteil}} + \underbrace{\vec{q}}_{\text{Vektorteil}} = (q_0, \vec{q})$$

Die Addition ist isomorph zu der in \mathbb{R}^4 :

$$p + q = (p_0 + q_0) + (\vec{p} + \vec{q}) = (p_0 + q_0, \vec{p} + \vec{q})$$

Für die Multiplikation standen die kartesischen Einheitsvektoren (mit dem Kreuzprodukt im Hinterkopf) und \mathbb{C} Pate:

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$$

$$\begin{aligned} \mathbf{ijk} = -1 \mid \cdot \mathbf{k} &\Rightarrow \mathbf{ij} = \mathbf{k} \\ \mathbf{ijk} = -1 \mid \cdot \mathbf{i} &\Rightarrow -\mathbf{jk} = -\mathbf{i} \mid \mathbf{j} \cdot \Rightarrow \mathbf{k} = -\mathbf{ji} \end{aligned}$$

daraus folgt die Nichtkommutativität

$$\mathbf{ij} = \mathbf{k} = -\mathbf{ji} \quad \text{und analoge Ausdrücke}$$

Berechnet man das Produkt zweier Quaternionen ist dies ein längerer Ausdruck, der allerdings mit Skalar- und Vektorprodukt etwas kompakter geschrieben werden kann:

$$pq = (p_0 + \mathbf{i}p_1 + \mathbf{j}p_2 + \mathbf{k}p_3)(q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3) = (p_0q_0 - \vec{p} \cdot \vec{q}, p_0\vec{q} + q_0\vec{p} + \vec{p} \times \vec{q}) \quad (4.11)$$

Wieder sieht man, dass durch das Vektorprodukt die Kommutativität verletzt wird. Untersucht man die beiden Rechenoperationen genauer ergibt sich als Struktur ein Nicht-Abelscher Ring. Mit der komplexen Konjugation $q^* := q_0 - \vec{q}$ lässt sich eine Norm $|q|$ festlegen:

$$q^*q = (q_0, -\vec{q})(q_0, \vec{q}) = q_0^2 + \vec{q} \cdot \vec{q} = q_0^2 + q_1^2 + q_2^2 + q_3^2 = qq^* =: |q|^2 \quad (4.12)$$

Man kann mit obiger Multiplikation leicht zeigen:

$$(pq)^* = q^*p^* \Rightarrow |pq|^2 = (pq)(pq)^* = (pq)(q^*p^*) = p|q|^2p^* = |p|^2|q|^2$$

Mit der Definition des Inversen ergibt sich

$$q^{-1} := \frac{q^*}{|q|^2} \Rightarrow qq^{-1} = q^{-1}q = 1$$

Für Einheitsquaternionen gilt also:

$$q^{-1} = q^* \quad \text{und mit der Beziehung} \quad 1 = |q|^2 = qq^* \stackrel{4.12}{=} q_0^2 + \vec{q} \cdot \vec{q} = q_0^2 + \|\vec{q}\|^2$$

4. Rotationen - Basics

gibt es daher für jedes q mit $|q| = 1$ ein $\theta \in [0, \pi]$ und einen Einheitsvektor $\hat{n} \in \mathbb{R}^3$ sodass gilt:

$$q = (s, \vec{q}) = (\cos \theta, \sin \theta \hat{n})$$

Damit können wir jetzt den Rotationsoperator festlegen:

$$R_q(\vec{v}) = q \vec{v} q^{-1} = q \vec{v} q^* = (s, \vec{q}) (0, \vec{v}) (s, -\vec{q}) \quad (s, \vec{q}) = (\cos \theta, \sin \theta \hat{n})$$

Wir zeigen, dass $R_q(\vec{v})$ eine Rotation des Vektors \vec{v} um die Achse \hat{n} mit Betrag 2θ darstellt:

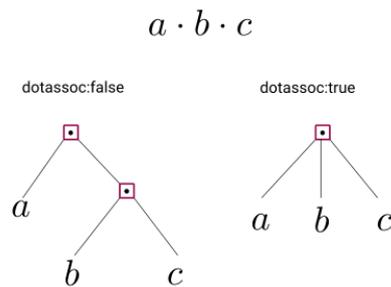
$$\begin{aligned} R_q(\vec{v}) &= (s, \vec{q}) (0, \vec{v}) (s, -\vec{q}) \stackrel{4.11}{=} (s, \vec{q}) (\vec{v}\vec{q}, s\vec{v} + \vec{q} \times \vec{v}) \stackrel{4.11}{=} \\ &= (s\vec{v}\vec{q} - \vec{q} \cdot (s\vec{v} + \vec{q} \times \vec{v}), s(s\vec{v} + \vec{q} \times \vec{v}) + (\vec{v}\vec{q})\vec{q} + \vec{q} \times (s\vec{v} + \vec{q} \times \vec{v})) = \\ &= (0, s^2\vec{v} + 2s\vec{q} \times \vec{v} + (\vec{v}\vec{q})\vec{q} + \vec{q} \times \vec{q} \times \vec{v}) = \text{für den letzten Ausdruck verwenden wir 4.13} \\ &= (0, \vec{v}(s^2 - \vec{q} \cdot \vec{q}) + 2(\vec{v} \cdot \vec{q})\vec{q} + 2s\vec{q} \times \vec{v}) \quad \text{mit } (s, \vec{q}) = (\cos \theta, \sin \theta \hat{n}) \quad \text{ergibt sich} \\ &= (0, \vec{v}(\cos^2 \theta - \sin^2 \theta) + (\vec{v} \cdot \hat{n})\hat{n} 2 \sin^2 \theta + 2 \cos \theta \sin \theta (\hat{n} \times \vec{v})) \\ &= (0, \vec{v}(\cos 2\theta) + (\vec{v} \cdot \hat{n})\hat{n} (1 - \cos(2\theta)) + \sin 2\theta (\hat{n} \times \vec{v})) \end{aligned}$$

Die letzte Zeile entspricht der Rodrigues-Formel 4.8 für 2θ .

Im folgenden *wxMaxima* - Programm zeigen wir auch die Darstellung eines Quaternions als Matrix. Einige Teile gehen auf das Macro "quaternion.mac" von Gosei Furuya (war im Verzeichnis "share/contrib/quaternion.mac" meiner Maxima-Version zu finden). Wobei einige grundlegende Änderungen von mir gemacht wurden - auf die auch hingewiesen wird. Sehen wir uns dieses "Ungetüm" an!

4.6.1 Quaternionen in *wxMaxima*

Wir benutzen für das Produkt der Quaternionen das "Matrixprodukt" (Punkt) und setzen es **nichtassoziativ** - das stellt sicher, dass beim Arithmetik-Baum beim Punkt immer nur 2 Kindknoten vorhanden sind (wichtig!) - damit kann *applyb2* den Baum "von unten" (bottom) nach den Regeln ersetzen. Würden mehr als 2 Kindknoten möglich sein, gingen die Regeln für $\mathbf{i} \cdot \mathbf{j}$ und Co ins "Leere". Außerdem sollen Produkte wie $\mathbf{i} \cdot \mathbf{i}$ nicht durch \mathbf{i}^2 ersetzt werden (= `dotexptsimp:false`)!

Abb.42 : Einfluss von *dotassoc*

Punktregeln festlegen

```
(% i1) (dotdistrib:true, dotexptsimp:false, dotscrules:true, dotassoc:false)$
```

i, j, k sind besondere Dinge

```
(% i2) declare([i,j,k],nonscalar)$
```

Jetzt die Produktregeln - beachte die Nichtkommutativität

```
(% i3) (defrule(f1,j,j,-1),defrule(f2,i,i,-1),defrule(f3,k,k,-1),defrule(f4,i,j,k),defrule(f5,j,i,-k))$
```

```
(% i4) (defrule(f6,j,k,i), defrule(f7,k,j,-i), defrule(f8,k,i,j), defrule(f9,i,k,-j))$
```

Das Arbeitspferd für die Multiplikation - rekursiv werden die Regeln - von den "Blättern" des Baums ausgehend (*applyb2* - "b" wie bottom) - angewandt und anschl. i, j, k herausgehoben

```
(% i6) expand4(_ex):=block([_eex],_eex:applyb2(expand(_ex),f1,f2,f3,f4,f5,f6,f7,f8,f9),
collectterms(_eex,i,j,k))$
```

Jetzt folgen einige "utility"-Routinen wie z.B.: is Quaternion?

```
(% i7) quaternionp(q):= if (abs(coeff(q,i)) + abs(coeff(q,j)) + abs(coeff(q,k)) # 0) then
true else false$
```

$q \rightarrow q^*$

```
(% i8) conj4(_ex1):=block([_eex1],_eex1:expand4(_ex1),
if (listp(_ex1)) then [_ex1[1], -_ex1[2], -_ex1[3], -_ex1[4]]
else scalarpart4(_eex1) - vectorpart4(_eex1) )$
```

Wir legen die Norm fest - gilt auch bei Übergabe als Liste!

```
(% i9) norm4(_ex1):=if (listp(_ex1)) then sqrt(lreduce("+",_ex1 * _ex1))
else sqrt(expand4(_ex1 . conj4(_ex1)))$
```

$q \rightarrow q^{-1}$

4. Rotationen - Basics

```
(% i10) inv4(_ex):= block( if (norm4(_ex)≠0) then conj4(_ex)/norm4(_ex)^2 else false)$
```

Skalaranteil zurückgeben

```
(% i11) scalarpart4(_ex1):=block([_eex1],_eex1:expand4(_ex1),
    _ex1:coeff(_eex1,i)*i+coeff(_eex1,j)*j+coeff(_eex1,k)*k,(_eex1-_ex1))$
```

Vektoranteil zurückgeben

```
(% i12) vectorpart4(_ex1):=block([_eex1],_eex1:expand4(_ex1),
    (_eex1,i)*i+coeff(_eex1,j)*j+coeff(_eex1,k)*k)$
```

Konvertierung in Zeilenvektor (Liste), Vorsicht: keine Matrix!

```
(% i13) convert2vec(_ex1):=block([_eex1],_eex1:expand4(_ex1),
    [scalarpart4(_eex1),coeff(_eex1,i),coeff(_eex1,j),coeff(_eex1,k)])$
```

Definition von Skalaren: $\text{inpart}(a \cdot b, 0) \rightarrow *$, dot wird "normale" Multiplikation

```
(% i14) declare([a,b,c,d,e,f,g,h],scalar)$
```

4 Dezimalen-Ausgabe, keine Konvertierungswarnungen, brauchen Levi-Civita Symbol ε_{ijk}

```
(% i15) (fpprintprec:4,ratprint:false, load(itensor))$
```

i,j und k werden fett gedruckt bei *tex* - Ausgabe

```
(% i16) (texput(i, "\\textbf{i}"), texput(j, "\\textbf{j}"), texput(k, "\\textbf{k}"))$
```

Wir konstruieren 2 "Probe"-Quaternionen und berechnen das Produkt

```
(% i17) (q1:a+b*i+c*j+d*k, q2:e+f*i+g*j+h*k)$
```

```
(% i18) p_q1q2:expand4(q1 . q2);
```

$$(ah + bg - cf + de) \mathbf{k} + (-bh + ag + df + ce) \mathbf{j} + (ch - dg + af + be) \mathbf{i} - dh - cg - bf + ae$$

Ausgabe wurde oben verwendet

```
(% i19) tex(p_q1q2,false);
```

Nur für Testzwecke

```
(% i20) vectorpart4(p_q1q2)+scalarpart4(p_q1q2);
```

$$(ah + bg - cf + de) \mathbf{k} + (-bh + ag + df + ce) \mathbf{j} + (ch - dg + af + be) \mathbf{i} - dh - cg - bf + ae$$

Ausgabe siehe oben

```
(% i21) tex(vectorpart4(p_q1q2)+scalarpart4(p_q1q2),false);
```

Konvertierung in einen Zeilen- bzw. Spaltenvektor

(% i22) `v1:convert2vec(p_q1q2);`

$$[-dh - cg - bf + ae, ch - dg + af + be, -bh + ag + df + ce, ah + bg - cf + de]$$

(% i23) `v2:transpose(v1);`
$$\begin{pmatrix} -dh - cg - bf + ae \\ ch - dg + af + be \\ -bh + ag + df + ce \\ ah + bg - cf + de \end{pmatrix}$$

Diesen Vektor kann man offensichtlich als Linearkombination mit den Linearfaktoren h, g, f, e darstellen - und somit nach 4.1.2 als Produkt einer Matrix mit dem Vektor $(h, g, f, e)^t$

(% i24) `q2AsVec:convert2vec(q2);` $[e, f, g, h]$

Die Zerlegung von (% i23) gelingt nur auf "Umwegen":

`inpart(v1[2],3)` → holt vom 2.-ten Listeneintrag den 3.-ten Summanden also → $-dg$

`partition(inpart(v1[2],3),d)` → erstes Listenelement ist Faktor von d also → $-g$

Wir holen die Ausdrücke wo e, f, g und h vorkommen als Vektoren:

`partition(inpart(v1[1],2),q2AsVec[2])` → $[-b, f]$

Wir verallgemeinern:

`partition(inpart(v1[i],j),q2AsVec[j])` → mit $i \in \{1 \dots 4\}, j \in \{1 \dots 4\}$

nehmen nur das erste Element der Liste (den Faktor) und stellen damit eine Liste zusammen (*makelist*), bauen einen Spaltenvektor (*transpose*) und stellen eine Liste der Spaltenvektoren zusammen (*makelist*) - nicht gerade elegant, aber funktioniert.

(% i25) `vecList:makelist(transpose(makelist(first(partition(inpart(v1[i],j),q2AsVec[j])),i,1,4)),j,1,4));`

$$\left[\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix}, \begin{pmatrix} -b \\ a \\ d \\ -c \end{pmatrix}, \begin{pmatrix} -c \\ -d \\ a \\ b \end{pmatrix}, \begin{pmatrix} -d \\ c \\ -b \\ a \end{pmatrix} \right]$$

Wir schalten den "Simplifier" ab, sonst multipliziert er wieder aus!

(% i26) `simp:false$`

Damit können wir das Produkt $q1 \cdot q2$ wie unten schreiben - nach 4.1.2 ist das gleich dem Produkt einer Matrix mit dem Spaltenvektor $[e,f,g,h]$

(% i27) `ex2:vecList[1]*e+vecList[2]*f+vecList[3]*g+vecList[4]*h;`

$$\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} e + \begin{pmatrix} -b \\ a \\ d \\ -c \end{pmatrix} f + \begin{pmatrix} -c \\ -d \\ a \\ b \end{pmatrix} g + \begin{pmatrix} -d \\ c \\ -b \\ a \end{pmatrix} h$$

(% i28) `simp:true$`

4. Rotationen - Basics

(% i29) is (equal(ex2 ,v2)); **true** Bestätigung der Gleichheit

Wir bauen aus den Spaltenvektoren eine Matrix

(% i30) buildQMatrix():=block([A:matrix([])],for i: 1 thru 4 do A: addcol(A,vecList[i],A)\$

(% i31) B:buildQMatrix();

$$\begin{pmatrix} a & -b & -c & -d \\ b & a & -d & c \\ c & d & a & -b \\ d & -c & b & a \end{pmatrix}$$

(% i32) is (equal(B . q2AsVec, v2)); **true** Passt ja!

Konvertierung für später

(% i35) Vec2Complex(v):=v[1]+i*v[2]+j*v[3]+k*v[4]\$

Wie ist diese Matrix M aufgebaut: 1) antisymmetrisch: $M = L - L^t$ L ist linke Dreiecksmatrix

2) Hauptdiagonale ist der Skalarteil: $M + D$ (wobei D Diagonalmatrix)

3) 1. Spalte die Vektordarstellung: $L[i, 1] \leftarrow q[i]$, damit sind bis auf 3 alle Elemente bestimmt:

$$\begin{pmatrix} a & -b & -c & -d \\ b & a & \cdot & \cdot \\ c & \cdot & a & \cdot \\ d & \cdot & \cdot & a \end{pmatrix}$$

Es fällt auf $B[3,2]=+q[4]$, $B[4,2]=-q[3]$, $B[4,3]=+q[2]$. Dies erinnert an den ε -Tensor:

Genauere Untersuchung ergibt: $B[i, j] := \sum_{k=2}^4 -\varepsilon_{kij} * q[k]$ $i \in \{3, 4\}, j \in \{2 \dots (i - 1)\}$

dabei ergeben sich die richtigen Vorzeichen. (Man könnte auch die 3 Werte einfach in die Matrix schreiben, aber ...). Damit haben wir die *Minor*-Matrix erledigt.

```
(% i36) qAsMat(q):=block([q_row, m:zeromatrix(4,4), d ],
/* convert input in row-vector (list)!
and leave inner block, when finished */
block(
if (listp(q)) then (q_row:q, return(0)),
if (scalarp(q)) then (q_row:cons(q,[0,0,0]), return(0)),
if (matrixp(q)) then (q_row:flatten(args(q)), return(0)),
if (quaternionp(q)) then (q_row:convert2vec(q))
),
/* end of inner block - q_row is a list now! */
d:diagmatrix(4,q_row[1]),
/* first column of matrix is set to q */
for i:2 thru 4 do m[i,1]: q_row[i],
/* Now formula above is applied */
for i:3 thru 4 do
for j:2 thru (i-1) do m[i,j]: sum(-levi_civita([k,i,j])*q_row[k],k,2,4),
/* skew matrix is built with q[1] as diagonal-elements */
m-transpose(m)+d
)$
```

Jetzt der Test:

```
(% i37) q:[a,b,c,d]$
```

```
(% i38) q1Mat:qAsMat(q);
```

$$\begin{pmatrix} a & -b & -c & -d \\ b & a & -d & c \\ c & d & a & -b \\ d & -c & b & a \end{pmatrix}$$

Jetzt die Matrix des inversen Quaternionen

```
(% i39) q1_inv_Mat:qAsMat(1/norm4(q)2*conj4(q))$
```

```
(% i40) q1_inv_Mat . q1Mat;
```

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \text{ Also gilt auch } Q^{-1} \cdot Q = I$$

Jetzt lösen wir unsere Testaufgabe(Rotation) mit Quaternionen - zuerst die Angaben

```
(% i41) (axis:[2,-2,1], M:[0.3,0.2,0.2], X:[1,0.5,0.5] , rotAngle:float(%pi/3) )$
```

Einheitsvektor \hat{n} in Richtung Drehachse

```
(% i42) n_0:float(1/norm4(cons(0,axis)) * axis)$
```

Wir erstellen das Rotations-Quaternion - Achtung halber Drehwinkel!

```
(% i44) q_rot_vec:cons(cos(rotAngle/2), sin(rotAngle/2)*n_0)$
```

4. Rotationen - Basics

```
(% i45) q_rot:Vec2Complex(q_rot_vec)$
```

Der rotierte Punkt als Quaternion - Realteil (Rundungsfehler) verschwindet!

```
(% i51) expand4(q_rot . Vec2Complex(cons(0,X-M)) . conj4(q_rot)) + Vec2Complex(cons(0,M));
```

$$0.9885 * k + 0.2566 * j + 0.5124 * i + 3.469 * 10^{-18}$$

Jetzt mit Quaternion-Matrizen - *transpose* nur aus Platzgründen

```
(% i53) transpose(qAsMat( qAsMat(q_rot) . cons(0,X-M) ) . conj4(q_rot_vec) + cons(0,M));
```

$$(0.0 \ 0.5124 \ 0.2566 \ 0.9885)$$

4.6.2 Quaternionen in *GNU-Octave*

In *GNU-Octave* braucht man das Rad nicht neu erfinden - es existiert ein Paket, welches die wesentlichen Operationen zur Verfügung stellt - insbesondere die Funktion *rot2q* ist bereits vorhanden - Eingabe: Einheitsvektor der Rotationsachse und Rotationswinkel.

Hier das Listing von Skript *rotByQuat.m*:

```

1  ## make sure that the package is loaded
  pkg load quaternion;
3
4  ## we use the test-data
5  axis=[2;-2;1]; M=[0.3;0.2;0.2]; X=[1;0.5;0.5]; rotAngle=pi/3;
6
7  ## we need the unitvector for axis
  function v0=unitVec(v)
9      n=norm(v);
      v0=1/n*v;
11 endfunction
12
13 ## convert vector to quaternion
  function q=vec2q(v)
15     q=quaternion(0,v(1),v(2),v(3));
  endfunction
17
18 ## convert quaternion to column-vector
19 function v=q2vec(q)
      c={"x","y","z"};
21     v=zeros(3,1);
      for i=1:3
23         v(i,1)=get(q,c{i});
      endfor
25 endfunction
26
27 ## create the rotation quaternion with the build-in function "
    rot2q"
  function qr=mkQRot(axis,angle)
29     qr=rot2q(unitVec(axis),angle);
  endfunction
31
32 ## we put it all together
  function X_r=rotPoint(point,angle,axis_direction,axis_point)
35     # axis_point is optional!
      if nargin == 3
37         axis_point=zeros(3,1);
      endif
39     q_r=mkQRot(axis_direction,angle);
      # this is our formula from the text
41     X_r=q2vec(q_r*vec2q(point-axis_point)*conj(q_r))+axis_point;
  endfunction
43 ## output
  X_r=rotPoint(X,rotAngle,axis,M)

```

Output:

```

>> rotByQuat
X_r =
    0.51241
    0.25665
    0.98846
>>

```

Auf weitergehende Betrachtungen sei hier verzichtet (Differentiation, Integration, Exponential- und Logarithmusfunktion, algebraische Eigenschaften, usw.) .

Fazit: Sind Translationen im Spiel haben die Matrizen leichte Vorteile, da sie ebenfalls über homogene Koordinaten als Matrizen darstellbar sind, ansonsten sind Quaternionen wegen ihrer reichhaltigen algebraischen Struktur beinahe unschlagbar.

4.7 ANHANG

Definition 4.3 Levi-Civita Symbol

Seien \vec{e}_1, \vec{e}_2 und \vec{e}_3 die Standardbasisvektoren des 3 dim Vektorraums, dann definieren wir

$$\varepsilon_{ijk} := \begin{vmatrix} - & \vec{e}_i & - \\ - & \vec{e}_j & - \\ - & \vec{e}_k & - \end{vmatrix}$$

ε_{ijk} ist die Determinante der Matrix, die aus obigen Zeilenvektoren besteht!



Nach den Eigenschaften der Determinante gilt $\varepsilon_{ijk} = (-1) \cdot \varepsilon_{jik}$. 2 Zeilenvertauschungen führen wieder zur Identität, also $\varepsilon_{jkl} = \varepsilon_{lkj}$
Dies gilt natürlich für alle(!) Vertauschungen.

Wir benutzen jetzt folgende Eigenschaften der Determinante:

$$\det(A \cdot B) = \det(A) \cdot \det(B) \quad \text{und} \quad \det(A^T) = \det(A)$$

Theorem 4.4 Multiplikation von Levi-Civita Symbolen

$$\varepsilon_{ijk} \varepsilon_{lmn} = \begin{vmatrix} - & \vec{e}_i & - \\ - & \vec{e}_j & - \\ - & \vec{e}_k & - \end{vmatrix} \begin{vmatrix} - & \vec{e}_l & - \\ - & \vec{e}_m & - \\ - & \vec{e}_n & - \end{vmatrix} = \begin{vmatrix} - & \vec{e}_i & - \\ - & \vec{e}_j & - \\ - & \vec{e}_k & - \end{vmatrix} \begin{vmatrix} | & | & | \\ \vec{e}_l & \vec{e}_m & \vec{e}_n \\ | & | & | \end{vmatrix} = \begin{vmatrix} \delta_{il} & \delta_{im} & \delta_{in} \\ \delta_{jl} & \delta_{jm} & \delta_{jn} \\ \delta_{kl} & \delta_{km} & \delta_{kn} \end{vmatrix}$$

Theorem 4.5 Korollar aus 4.4

Mit dem Laplace'schen Entwicklungssatz (Entwicklung nach erster Zeile) und $i = \ell$ folgt mit Theorem 4.4

$$\varepsilon_{ijk} \varepsilon_{imn} := \sum_{i=1}^3 \varepsilon_{ijk} \varepsilon_{imn} = \delta_{jm} \delta_{kn} - \delta_{km} \delta_{jn}$$

Theorem 4.6 Graßmann-Identität

$$\vec{a} \times \underbrace{(\vec{b} \times \vec{c})}_{\vec{d}} = (\vec{a} \cdot \vec{c})\vec{b} - (\vec{a} \cdot \vec{b})\vec{c} \quad (4.13)$$

Beweis: Die j -te Komponente des Kreuzproduktes (mit Summenkonvention) ist

$$(\vec{a} \times \vec{d})_j = \varepsilon_{jkl} a_k d_l$$

Wir berechnen die j -te Komponente von 4.13:

$$\underbrace{\varepsilon_{jkl}}_{= \varepsilon_{ljk}} a_k \underbrace{\varepsilon_{lmn} b_m c_n}_{d_l} = (\delta_{jm}\delta_{kn} - \delta_{jn}\delta_{km}) a_k b_m c_n = a_k b_j c_k - a_k b_k c_j$$

wobei wir Theorem 4.5 benutzt haben. □

Theorem 4.7 Orthogonalität des Kreuzproduktes

$$(\vec{a} \times \vec{b}) \perp \vec{a} \quad \wedge \quad (\vec{a} \times \vec{b}) \perp \vec{b}$$

Beweis:

$$(\vec{a} \times \vec{b}) \cdot \vec{a} = \varepsilon_{ikl} a_k b_l a_i$$

Da gilt $\varepsilon_{ikl} = -\varepsilon_{kil}$ kommen in obiger Summe die Summanden $a_k b_l a_i$ sowohl mit positiven als auch negativen Vorzeichen vor - die Summe verschwindet also!

Die analoge Situation ergibt sich für \vec{b} . □

Theorem 4.8 Wiederholtes Kreuzprodukt mit einem Einheitsvektor

Sei \hat{n} ein Einheitsvektor des \mathbb{R}^3 , dann gilt

$$[\hat{n}]_{\times}^3 = -[\hat{n}]_{\times} \Leftrightarrow \vec{v}_{\times \times \times} = -\vec{v}_{\times} \Leftrightarrow \hat{n} \times (\hat{n} \times (\hat{n} \times \vec{v})) = -(\hat{n} \times \vec{v})$$

aus der ersten Identität folgt unmittelbar

$$[\hat{n}]_{\times}^{k+2} = -[\hat{n}]_{\times}^k \quad k \in \{1, 2, 3 \dots\}$$

Beweis: Wir setzen $\vec{a} := \hat{n} \times \vec{v}$ wobei wir aus Theorem 4.7 wissen: $\vec{a} \perp \hat{n}$

Wir müssen also zeigen: $\hat{n} \times (\hat{n} \times \vec{a}) = -\vec{a}$

Die i-te Komponente von $\hat{n} \times \vec{a}$ ist:

$$(\hat{n} \times \vec{a})_i = \varepsilon_{ijk} n_j a_k =: x_i$$

Die α -Komponente von $\hat{n} \times \vec{x}$ ist:

$$(\hat{n} \times \vec{x})_{\alpha} = \varepsilon_{\alpha\beta i} n_{\beta} x_i = \varepsilon_{\alpha\beta i} n_{\beta} \varepsilon_{ijk} n_j a_k$$

Beachte: Der einzige unabhängige Index ist α - über alle anderen wird summiert. Es handelt sich also um einen Vektor!

Mit Theorem 4.5 lösen wir das Produkt der Levi-Civita Symbole auf zu

$$\hat{n} \times (\hat{n} \times \vec{a})_{\alpha} = [\delta_{\alpha j} \delta_{\beta k} - \delta_{\alpha k} \delta_{\beta j}] n_{\beta} n_j a_k = \underbrace{n_k a_k}_0 n_{\alpha} - \underbrace{n_j n_j}_1 a_{\alpha} = -a_{\alpha}$$

Der erste Term verschwindet wegen der Orthogonalität von \hat{n} und \vec{a} und $n_j n_j = 1$, da \hat{n} ein Einheitsvektor ist!

□

4.8 Installation des Quaternion-Pakets

4.8.1 GNU-Octave

1. Runterladen des Pakets (gezipptes tar-Archiv) von <https://octave.sourceforge.io/quaternion/index.html>
2. Bevor wir dieses Paket installieren können, muss das Program *mkoctfile* und entsprechende Compiler zur Verfügung stehen - bei meiner OpenSuSE (Leap 15.1) ist dies im Paket *octave-devel* alles zusammengefasst, also als Root

```
zypper install octave-devel
```

3. Jetzt gilt es zu überlegen, ob das Paket “privat” ist oder für alle Benutzer zu Verfügung steht (UNIX ist ja bekanntlich ein Multiuser-System). Ich habe mich für alle Benutzer entschieden - dafür brauchen wir wieder Root-Rechte! Wir starten *GNU-Octave* ohne Grafik:

```
sudo /usr/bin/octave  
pkg install -global /home/<username>/Downloads/quaternion-2.4.0.tar.gz
```

Das sollte ohne Fehlermeldungen vonstatten gehen. Man beendet *GNU-Octave*

4. Man startet *GNU-Octave* von seinem privaten Desktop und überprüft, ob das Paket zur Verfügung steht:

```
pkg describe
```

Unser Paket Quaternion sollte zur Verfügung stehen - jetzt noch laden

5.

```
pkg load quaternion  
test_quaternion
```

Falls alle Tests positiv durchlaufen, ist das Paket einsatzfähig!

5 | Rotation einer Schachtel

Ziel dieses Kapitels ist die Darstellung einer rotierenden Schachtel mit Hilfe von *Geogebra-2D*. Dazu braucht es im Wesentlichen 2 Schritte:

1. Die Darstellung der Schachtel im Kamera-System (Augen-System)
2. Die Projektion dieser transformierten Schachtel auf ein Zeichenblatt - wobei sich die Frage ergibt: welche Flächen sind sichtbar?

5.1 Transformation ins Augensystem

Die mathematischen Grundlagen, die wir benötigen, ist die Darstellung von Translationen, Spiegelungen und Drehungen von Punkten bzw. Koordinatensystemen (KS). Diese Abbildungen sind linear und daher sind sie mit Matrizen darstellbar (siehe ANHANG Theorem 5.3).

Zuerst zu den **Drehungen**. Ein guter Überblick befindet sich in der Wikipedia.

Wichtig sind folgende Fakten:

- Passive Drehungen (also Drehung des KS) sind invers zu aktiven (siehe Anhang).
- $(R_\alpha)^{-1} = R_{-\alpha}$

Theorem 5.1 Invarianz des skalaren Produkts impliziert Orthogonalität

$$\langle Ax, Ay \rangle = \langle x, y \rangle \Rightarrow A^{-1} = A^T \quad \text{wobei } \langle x, y \rangle := \vec{x} \cdot \vec{y}$$

Winkel- und längentreue Abbildungen implizieren eine orthogonale Matrix

Beweis:

$$(A\vec{x}) \cdot \vec{y} = \sum_{j,i} (a_{ij}x_j)y_i = \sum_{i,j} x_j(a_{ij}y_i) = \vec{x} \cdot (A^T\vec{y}) \quad \text{bzw.} \quad \langle Ax, y \rangle = \langle x, A^T y \rangle \quad (*)$$

$$\langle Ax, Ay \rangle = \langle x, y \rangle \stackrel{(*)}{\Rightarrow} \langle x, A^T Ay \rangle = \langle x, y \rangle \Rightarrow A^T A = I \Rightarrow A^{-1} = A^T$$

□

5. Rotation einer Schachtel

Wir beschränken uns im Folgenden auf den \mathbb{R}^3 .

Führt man homogene Koordinaten ein, lassen sich auch Translationen als Matrixmultiplikation darstellen. Die "Umrechnung" ist trivial - man führt eine 4.-te Koordinate ein und setzt sie 1.

$$\vec{x}_h = \begin{pmatrix} \vec{x} \\ 1 \end{pmatrix}$$

Alle Transformationsmatrizen leiten sich jetzt von (4×4) Einheitsmatrizen ab. Die für die Translation lässt sich jetzt schreiben:

$$\vec{c} = \begin{pmatrix} 1 \\ 2 \\ 3 \\ 1 \end{pmatrix} \quad T = \begin{pmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{pmatrix} \Rightarrow T \vec{c} = \begin{pmatrix} x+1 \\ y+2 \\ z+3 \\ 1 \end{pmatrix}$$

Auch bei den Drehmatrizen führt das zu analogen Ergebnissen (hier wird \vec{e}_1 um die z-Achse um α gedreht):

$$R_{z,\alpha} = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \vec{e}_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} \Rightarrow R_{z,\alpha} \vec{e}_1 = \begin{pmatrix} \cos(\alpha) \\ \sin(\alpha) \\ 0 \\ 1 \end{pmatrix}$$

(Hinweis: Wer die Drehmatrizen herleiten möchte, braucht sie nur unbekannt ansetzen und auf die Basisvektoren \vec{e}_i ansetzen mit bekanntem Ergebnis! Dadurch ergeben sich die 9 Variablen; Übrigens mit $R_{z,\alpha} R_{z,\beta} = R_{z,\alpha+\beta}$ ergeben sich die Sumsensätze der cos- und sin-Funktion!)
Was wir jetzt noch brauchen ist die Spiegelung um die y-z-Ebene M_{yz} - sie dreht das Vorzeichen der x-Koordinaten.

Hier noch die Zusammenfassung:

Transformationsmatrizen für homogene Koordinaten

$$T = \begin{pmatrix} 1 & 0 & 0 & x_T \\ 0 & 1 & 0 & y_T \\ 0 & 0 & 1 & z_T \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad M_{yz} = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad R_z = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad R_y = \begin{pmatrix} \cos(\alpha) & 0 & \sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Abb.43 : T-Translation; M_{yz} -Spiegelung um y-z-Ebene; R_i -Drehung um i -Achse

Was wir jetzt noch brauchen, um mit dem Rechnen zu beginnen, ist der Punkt, wo die Kamera (Augen) steht: dazu verwenden wir Kugelkoordinaten (ρ, θ, φ) . Die Umrechnung in kartesische Koordinaten sollte durch wiederholte Anwendung des ‘Pythagoras’ keine Schwierigkeit darstellen:

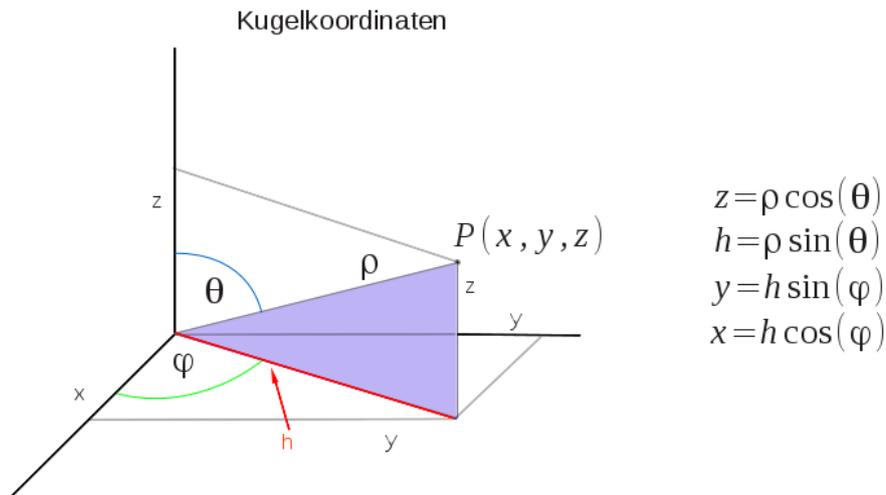


Abb.44 : Kugelkoordinaten

Das Augensystem wählen wir so, dass die x_A -Achse waagrecht nach rechts geht, die y_A -Achse nach oben (wie gewohnt) und die z_A -Achse von den Augen weg zum Ursprung des anderen Koordinatensystems.

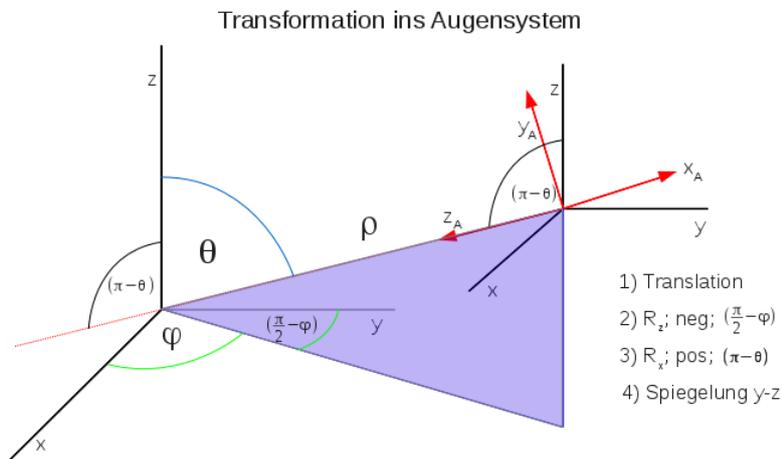


Abb.45 : Transformation ins Augensystem

Für die Transformationsmatrix T gibt es zwei Interpretationen(wie eingangs erwähnt):



- *aktiv*: $T \cdot P = P'$ P' sind die Koordinaten eines **neuen** Objekts
- *passiv*: $T^{-1} \cdot P = P'$ P' sind die Koordinaten **desselben** Objekts bei transformiertem Koordinatensystem (siehe Anhang: aktive und passive Transformation)

5. Rotation einer Schachtel

Hier ist einer der “springenden Punkte” in unserer Rechnung, darum schauen Sie sich die Zeichnung genau an und versuchen Sie die einzelnen Schritte nachzuvollziehen (auf den “alten” Koordinaten-Ursprung konzentrieren):

1. Translation - eh klar!
2. Drehung um die z -Achse im Uhrzeigersinn (math. negativ) bis y -Achse im blauen Dreieck liegt.
3. Jetzt Drehung um die x -Achse gegen den Uhrzeigersinn bis die z -Achse auf der Verbindungslinie der KS-Ursprünge “einrastet”
4. Zum Schluss das Umklappen der x -Achse (Spiegelung) - wir erhalten ein Linkssystem!

Zur Veranschaulichung kann man sich das *Geogebra* Arbeitsblatt auf <https://www.geogebra.org/m/NnKQP7Xq> anschauen bzw. runterladen.

Die “Gesamtmatrix” - die das alles macht - lassen wir uns mit einem CAS berechnen (z.B. *wxMaxima* siehe Abschnitt 5.4)

$$T = \begin{pmatrix} -\sin(\phi) & \cos(\phi) & 0 & 0 \\ -\cos(\phi) \cos(\theta) & -\sin(\phi) \cos(\theta) & \sin(\theta) & 0 \\ -\cos(\phi) \sin(\theta) & -\sin(\phi) \sin(\theta) & -\cos(\theta) & \rho \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Hier endlich das Endergebnis - 1 Matrix für Alles!

Damit wäre der erste Teil - Darstellung der Schachtel im Augensystem - erledigt.

5.2 Projektion auf das Zeichenblatt

Dies ist eigentlich nur mehr eine Angelegenheit von “ähnlichen Dreiecken”. Schauen wir uns dazu die folgende Zeichnung an (siehe auch <https://www.geogebra.org/m/qmk2zmKe>):

- Unser “Auge” sitzt im Ursprung
- Das Zeichenblatt ist parallel zur xy -Ebene mit Abstand D
- Die Koordinaten (x_S, y_S) der perspektivischen Projektion von P ergeben sich aus dem blauen und grünen Dreieck (,die z_P gemeinsam haben) zu:

$$\xi_S = \frac{D}{z_P} \xi_P \quad \xi \in \{x, y\} \quad 0 < D < z_P \Rightarrow \text{Verkleinerung}$$

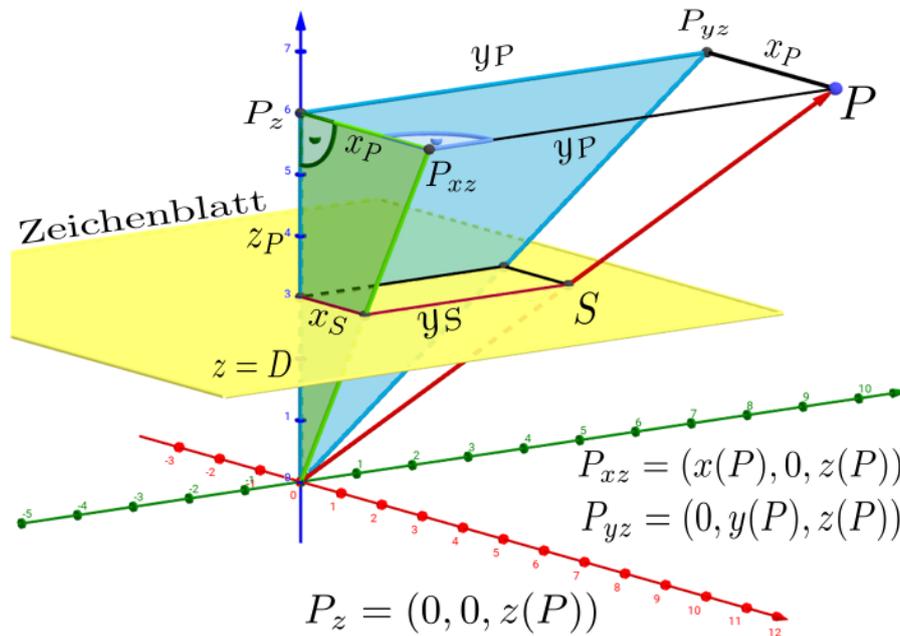


Abb.46 : perspektivische Projektion; Ursprung ist "Augenposition"

5.3 Sichtbarkeit

Wie man auf der Zeichnung sehen kann, ist eine Fläche nur dann sichtbar, wenn der Winkel zwischen dem Vektor $\vec{\sigma}$ (von einem Flächenpunkt zum Auge) und dem Normalvektor der Fläche (nach außen) spitz ist! Bilden die Schachtelkanten ein Basis-Dreibein gilt $\vec{n}_0 = \pm \vec{e}_i$ außerdem reichen für die 6 Flächenpunkte 2 Eckpunkte (Endpunkte der Raumdiagonalen) \vec{p}_1 und \vec{p}_2 aus:

$$\pm \vec{e}_i \cdot \underbrace{\left[\rho \begin{pmatrix} \cos \theta \cos \varphi \\ \sin \theta \sin \varphi \\ \cos \theta \end{pmatrix} - \vec{p}_k \right]}_{\vec{\sigma}} > 0$$

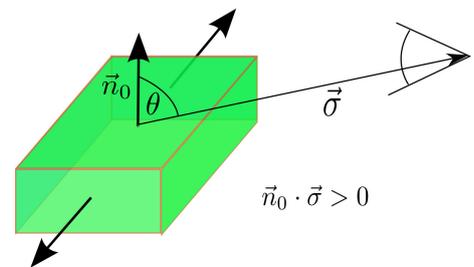


Abb.47 : Sichtbarkeit einer Fläche

Da bei den Basisvektoren \vec{e}_i nur jeweils 1 Koordinate nicht verschwindet, entsteht eine skalare Sichtbarkeitsbedingung, die man in *Geogebra* bei den Eigenschaften der Polygonfläche eintragen kann!

Hier die "fertige" Datei zum runterladen auf

<http://www.angsuesser.at/docs/math/pdf/3dbox-rotating/rotation.ggb>.

5.4 Berechnung der Transformationsmatrix

Die Multiplikation einer Matrix mit einem Vektor kann als Linearkombination der Spaltenvektoren gedeutet werden, also

$$\underbrace{\begin{pmatrix} \vdots & \vdots & \vdots \\ \vec{a}_1 & \vec{a}_2 & \vec{a}_3 \\ \vdots & \vdots & \vdots \end{pmatrix}}_A \cdot \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{pmatrix} = \lambda_1 \vec{a}_1 + \lambda_2 \vec{a}_2 + \lambda_3 \vec{a}_3 = A \cdot \vec{\lambda}$$

Damit ergibt sich folgender Sachverhalt:

Theorem 5.2 Bedeutung der Spaltenvektoren

A sei eine lineare Transformation (Matrix) von $\mathbb{R}^3 \rightarrow \mathbb{R}^3$
 und
 \vec{e}_i seien die Standardbasisvektoren
 $\left. \vphantom{\begin{matrix} A \\ \text{und} \\ \vec{e}_i \end{matrix}} \right\} \Rightarrow \begin{pmatrix} \vdots & \vdots & \vdots \\ A\vec{e}_1 & A\vec{e}_2 & A\vec{e}_3 \\ \vdots & \vdots & \vdots \end{pmatrix} = A$

Die Spaltenvektoren von A sind die Transformation angewandt auf die Basisvektoren! (siehe auch ANHANG 5.6)

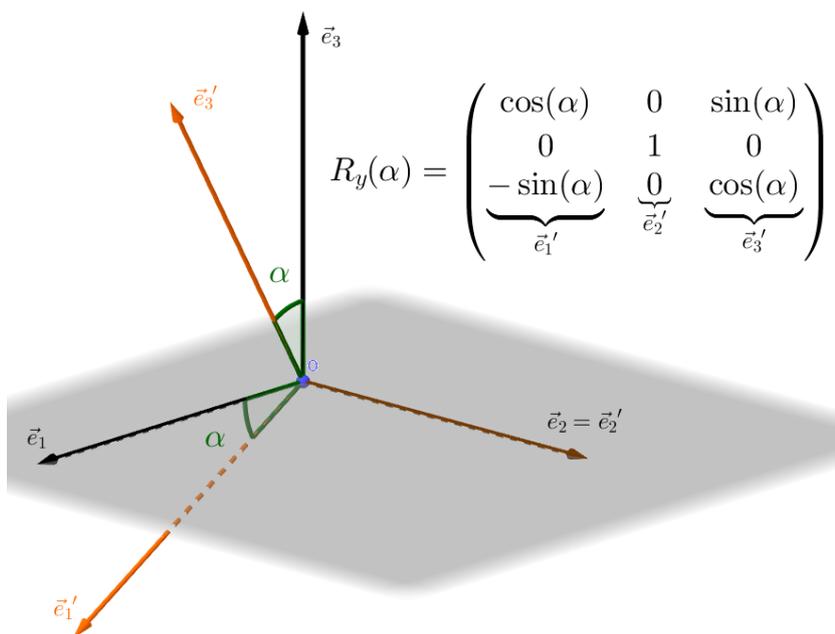


Abb.48 : Als Beispiel die Erstellung von $R_y(\alpha)$



Zum Beweis reicht einfaches ausrechnen!

Obiges Verfahren verwenden wir für die Darstellung von $R_z(\alpha)$ - für die homogenen Vektoren ergänzen wir noch mit der Einheitsmatrix

```
(%i1) R_z(%alpha):=matrix([cos(%alpha),-sin(%alpha),0,0],
                          [sin(%alpha),cos(%alpha),0,0],
                          [0,0,1,0],
                          [0,0,0,1]);
```

$$R_z(\alpha) := \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (\%o1)$$

Dasselbe für R_x

```
(%i2) R_x(%alpha):=matrix([1,0,0,0],
                          [0,cos(%alpha),-sin(%alpha),0],
                          [0,sin(%alpha),cos(%alpha),0],
                          [0,0,0,1]);
```

$$R_x(\alpha) := \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (\%o2)$$

Wegen der Translation als Matrixmultiplikation haben wir die homogenen Koordinaten eingeführt!

```
(%i3) T(x_T,y_T,z_T):=matrix([1,0,0,x_T],
                             [0,1,0,y_T],
                             [0,0,1,z_T],
                             [0,0,0,1]);
```

$$T(x_T, y_T, z_T) := \begin{pmatrix} 1 & 0 & 0 & x_T \\ 0 & 1 & 0 & y_T \\ 0 & 0 & 1 & z_T \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (\%o3)$$

Spiegelung um die yz -Ebene - die x -Koordinate wechselt ihr Vorzeichen!

```
(%i4) M_yz:matrix([-1,0,0,0],
                  [0,1,0,0],
                  [0,0,1,0],
                  [0,0,0,1]);
```

$$\begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (M_{yz})$$

5. Rotation einer Schachtel

Verschiebung in Kugelkoordinaten des ursprünglichen Koordinatensystems

$$\begin{aligned} (\%i10) \quad & x_t:\%rho * \sin(\%theta) * \cos(\%phi) \\ & y_t:\%rho * \sin(\%theta) * \sin(\%phi) \\ & z_t:\%rho * \cos(\%theta) \end{aligned}$$

Translation zum Auge anschl. Rotation

$$(\%i15) \quad R_zT:\text{trigsimp}(R_z(\%pi/2-\%phi) . T(-x_t,-y_t,-z_t));$$

$$\begin{pmatrix} \sin(\phi) & -\cos(\phi) & 0 & 0 \\ \cos(\phi) & \sin(\phi) & 0 & -\rho \sin(\theta) \\ 0 & 0 & 1 & -\rho \cos(\theta) \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (R_zT)$$

Zum Vergleich: Rotation und anschl. Translation - die Koordinaten des Auges haben sich verändert:
 $\phi = \pi/2$

$$(\%i18) \quad TR_z:\text{trigsimp}(T(0,-\%rho * \sin(\%theta),-\%rho * \cos(\%theta)) . R_z(\%pi/2-\%phi));$$

$$\begin{pmatrix} \sin(\phi) & -\cos(\phi) & 0 & 0 \\ \cos(\phi) & \sin(\phi) & 0 & -\rho \sin(\theta) \\ 0 & 0 & 1 & -\rho \cos(\theta) \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (TR_z)$$

Jetzt die gesamte Transformationsmatrix

$$(\%i22) \quad Tr:M_yz . \text{trigsimp}(R_x(\%theta - \%pi) . TR_z) ;$$

$$\begin{pmatrix} -\sin(\phi) & \cos(\phi) & 0 & 0 \\ -\cos(\phi) \cos(\theta) & -\sin(\phi) \cos(\theta) & \sin(\theta) & 0 \\ -\cos(\phi) \sin(\theta) & -\sin(\phi) \sin(\theta) & -\cos(\theta) & \rho \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (Tr)$$

5.5 Implementation in *Geogebra*

Eine Matrix ist in *Geogebra* eine Liste von Listen - also : $\{ \{1,2,3\},\{4,5,6\},\{7,8,9\} \}$
 ein Spaltenvektor (benötigt für die Matrizenmultiplikation) ist dann
 $\{ \{1\},\{4\},\{7\} \}$

- Die Transformationsmatrix eingeben:

$$T = \left\{ \left\{ -\sin(\varphi), \cos(\varphi), 0, 0 \right\}, \left\{ -\cos(\varphi) \cos(\theta), -\sin(\varphi) \cos(\theta), \sin(\theta), 0 \right\}, \left\{ -\cos(\varphi) \sin(\theta), -\sin(\varphi) \sin(\theta), -\cos(\theta), \rho \right\}, \left\{ 0, 0, 0, 1 \right\} \right\}$$

Die 3 Schieberegler für ρ, θ, φ , die dadurch anfallen, abnicken und einstellen!

- Die 3 Zahlen für Länge (l), Breite (w) und Höhe (h) des Quaders festlegen, außerdem die Distanz des “Zeichenblatts” vor dem Auge (D) - je mehr sich D von ρ unterscheidet, umso geringer die “perspektivische Verzerrung” (Übergang in den Schrägriss)!
- Eine Liste L_1 der 4 Grundflächenpunkte erstellen:
 $\{(0, 0, 0), (l, 0, 0), (l, w, 0), (0, w, 0)\}$
- Eine Liste L_2 der 4 Deckflächenpunkte erstellen:
 $Sequence(Element(L_1, i) + (0, 0, h), i, 1, 4)$
- Eine Liste L_P aller Punkte erstellen:
 $Join(L_1, L_2)$
- Jetzt erstellen wir uns ein Werkzeug $hom[< Point >]$, welches aus einem Punkt einen homogenen Spaltenvektor erzeugt:
 - (a) Punkt $P = (0, 0, 0)$ erstellen
 - (b) $P_h = \{\{x(P)\}, \{y(P)\}, \{z(P)\}, \{1\}\}$
 - (c) “Neues Werkzeug” erstellen, Eingabe P , Ausgabe P_h und Name hom
- Damit können wir eine Liste L_h der homogenen Spaltenvektoren erstellen:
 $Sequence(hom(Element(L_P, i)), i, 1, 8)$
- Jetzt wird ins “Augensystem” transformiert:
 $L_T=Sequence(T * Element(L_h, i), i, 1, 8)$
- Jetzt die 2D-Punkte der perspektivischen Projektion:

```
L_{Prj}= Sequence( (
    D*Element(L_T, i, 1, 1) / Element(L_T, i, 3, 1),
    D*Element(L_T, i, 2, 1) / Element(L_T, i, 3, 1)
) ,i, 1, 8)
```

Das ist sicher die schwierigste Befehlszeile, deshalb habe ich sie hier etwas strukturiert!



Die roten Klammern erzeugen einen Punkt in Geogebra

- Jetzt die einzelnen Oberflächen (Polygone) festlegen, dazu erstellen wir eine Tabelle über die Eckpunkte und welchen Punkt wir für die Sichtbarkeit hernehmen:

Polygon	Eckpunkt-Indices	Punkt für Sichtbarkeit
P_1	1 2 3 4	1
P_2	5 6 7 8	7
P_3	1 2 6 5	1
P_4	2 3 7 6	7
P_5	3 4 8 7	7
P_6	1 4 8 5	1

5. Rotation einer Schachtel

- Legen wir jetzt die einzelnen Polygone fest und um die Sichtbarkeit kümmern wir uns anschließend. Das ist mühsame stupide Schreiarbeit - ich zeige das am Beispiel von P_4 :
`Polygon(Element(L_{Prj},2), Element(L_{Prj},3), Element(L_{Prj},7), Element(L_{Prj},6))`
(Siehe auch: NACHSCHLAG)

- Jetzt werden die Oberflächen(Polygone) verschieden eingefärbt (Durchsichtigkeit vermindert, opacity erhöht) und die Segmente(Kanten) auf "unsichtbar" geschaltet.

- Jetzt zur Sichtbarkeit: Wir legen uns eine Liste von Einheitsnormalvektoren für die einzelnen Polygone an:

```
NList = {(0, 0, -1), (0, 0, 1), (0, -1, 0), (1, 0, 0), (0, 1, 0), (-1, 0, 0)}
```

also $(0, -1, 0)$ ist der Einheitsnormalvektor von P_3 !

Wir legen uns eine Liste von Punkten an, die auf der Fläche liegen (siehe Tabelle):

```
PInPoly = {(0, 0, 0), (4, 3, 2), (0, 0, 0), (4, 3, 2), (4, 3, 2), (0, 0, 0)}
```

Wir berechnen die "Augenkoordinaten":

$$A = \rho * (\sin(\theta) * \cos(\phi), \sin(\theta) * \sin(\phi), \cos(\theta))$$

Nun die Sehvektoren $\vec{\sigma}$ von der Fläche zu den Augen:

```
 $\sigma$ List = Sequence(A - Element(PInPoly, i), i, 1, 6)
```

So nun noch die Liste der skalaren Produkte:

```
visibleL=Sequence(Element(NList, i) Element( $\sigma$ List, i), i, 1, 6)
```

Hier verbirgt sich das skalare Produkt, das auch mit `dot(<Vector1>, <Vector2>)` geschrieben werden kann!

- Praktisch geschafft:
Wir tragen bei den einzelnen Polygonen die Sichtbarkeitsbedingung ein - z.B. für P_3 :

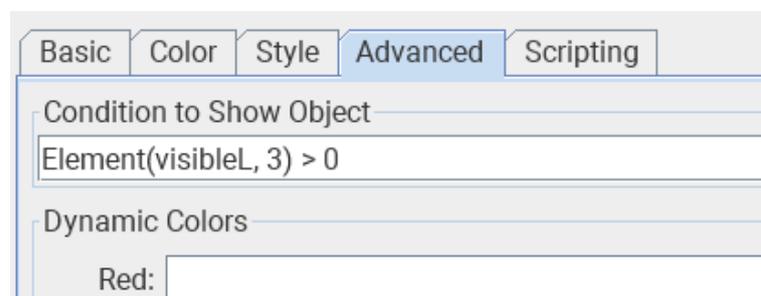


Abb.49 : Sichtbarkeitsdialog

- Animation für die Schieberegler einschalten und die Früchte der Arbeit genießen!

NACHSCHLAG:

Wie meist immer, wenn es sich um stupide Schreibarbeit handelt - wie oben, wo einfach 6 Polygone mit Eigenschaften versehen werden - bietet sich ein Programm in Javascript an. Zeitersparnis bringt das zwar ganz gewiss nicht - denn bis das Programm fehlerfrei läuft - hat man die Schreibarbeit auch erledigt. Aber es ist allemal interessanter und man lernt dabei einiges über das Javascript-Interface von *Geogebra*!

Daher an dieser Stelle auch die Version mit Javascript:

Der Reiter "Global Javascript" sieht dann bei mir folgendermaßen aus:

```

function ggbOnInit() { main(); }
2
function main(){
4   var nr=0;
      for (j=1; j<7; j++){
6       initHelperLists(j);
          getPointListForPoly(j, 1);
8       makePolygon(j);
      }
10  ggbApplet.deleteObject("PointIndices"); // not needed anymore
    }
12  // ----- main ends here -----

14  initHelperLists(j){
      ggbApplet.evalCommand("Q_{"+j+"0}={}")
16      ggbApplet.setAuxiliary("Q_{" + j + "0}", true);
      ggbApplet.evalCommand("PointIndices=Element(PointList,"+j+"");
18  }

20  makePolygon(j){
      //Q_j4 ist point-list for j-th polygon denoted by P_j
22      ggbApplet.evalCommand("P_{"+j+"}=Polygon(Q_{"+j+"4})");
      ggbApplet.evalCommand("SetConditionToShowObject(P_{"+j+"",
24          Element(visibleL,"+j+") > " +Number(0)+")");
      ggbApplet.setAuxiliary("P_{"+j, true);
26      ggbApplet.setColor("P_{"+j, j*100, 250-j*30, j*10);
      ggbApplet.setFilling("P_{"+j, 0.5);
28  }

30  getPointListForPoly(j, i){
      if (i < 5) {
32      // get point-indices for j-th polygon
          nr=ggbApplet.getListValue("PointIndices", i);
34      // get the current projection point as list and
          //join it with its predecessors (4 points)
36      ggbApplet.evalCommand("Q_{" + j + i + "} =Join(Q_{" + j + (i-1) +
          "}, {Element(L_{Prj}," + nr + ")}");
38      ggbApplet.setVisible("Q_{" + j + i + "}", false);
      ggbApplet.setAuxiliary("Q_{" + j + i + "}", true);
40      getPointListForPoly(j, i+1); //recur until 4 points
      }
42  }

```

5. Rotation einer Schachtel



In Zeile 36 wird mit verschachteltem *Join* die Punktliste Q_{j4} erstellt (ich habe die zweite Schleife hinter einer Rekursion "verborgen"). Man könnte meinen, dass man Q_{j0} , Q_{j1} , Q_{j2} , Q_{j3} löschen kann, da man sie ja nicht mehr braucht. Aber weit gefehlt. *Join* ist in *Geogebra* mit Zeigerverweisen implementiert (wie eine verkettete Liste) und es werden nicht die Werte kopiert. Mit dem Löschen von Q_{j0} verliert man den Listenanfang und damit die gesamte Liste. Man pflastert sich damit das Algebra-Fenster ganz schön zu - aber mit der Markierung als Hilfsobjekt verschwinden diese Teillisten wenigstens aus der Normalansicht!

Dazu gibt es noch einen Schaltfläche mit dem Titel "Start/Stop Animation" mit folgender Javascript Click-Methode:

```
2 startStop();
4 function startStop(){
6   if (! ggbApplet.isAnimationRunning()){
8     ggbApplet.setAnimating("\u03b8", true); //  $\theta$ 
9     ggbApplet.setAnimating("\u03A6", true); //  $\Phi$ 
10    ggbApplet.startAnimation();
11  }
12  else ggbApplet.stopAnimation();
13 }
```

Die utf-8 Codes für die griechischen Buchstaben in Javascript besorgt man sich aus dem Internet. Im *listing*-package von \LaTeX (wie hier verwendet) ist dazu bei der Initialisierung des packages *mathescape=true* anzugeben:

```
\lstset{ %
  backgroundcolor=\color{white},
  .....
  mathescape=true
  .....
```

Dann kann man im Listing die griechischen Buchstaben wie gewohnt mit θ schreiben!

5.6 ANHANG: Aktive und passive lineare Transformationen

Sei die lineare Transformation durch die Matrix A symbolisiert - ohne Beschränkung der Allgemeinheit sei sie hier (3×3) .

Ihre Darstellung mit Spaltenvektoren lautet:

$$A = \begin{pmatrix} \vdots & \vdots & \vdots \\ \vec{a}_1 & \vec{a}_2 & \vec{a}_3 \\ \vdots & \vdots & \vdots \end{pmatrix}$$

- Zuerst die **aktive** Transformation:

Das Bild von \vec{p} unter A sei \vec{p}' , also

$$A \cdot \vec{p} = \vec{p}' \quad \text{es wird ein neues Objekt erschaffen!}$$

- Jetzt die **passive** Transformation:

Was sind die Koordinaten λ_i von \vec{p} mit der Basis $\{A \cdot \vec{e}_1, A \cdot \vec{e}_2, A \cdot \vec{e}_3\}$?

$$\lambda_1 \underbrace{(A\vec{e}_1)}_{\vec{a}_1} + \lambda_2 (A\vec{e}_2) + \lambda_3 (A\vec{e}_3) = \vec{p}$$

$$\lambda_1 \vec{a}_1 + \lambda_2 \vec{a}_2 + \lambda_3 \vec{a}_3 = \vec{p}$$

$$A \cdot \vec{\lambda} = \vec{p} \Rightarrow \boxed{\vec{\lambda} = A^{-1} \cdot \vec{p}}$$

Die **passive** Transformation ist die **inverse** der **aktiven** Transformation

Theorem 5.3 Jede lineare Transformation ist als Matrix darstellbar

$$\left. \begin{array}{l} T \text{ sei eine lineare Transformation von } \mathbb{R}^3 \rightarrow \mathbb{R}^3 \\ \text{und} \\ T(\vec{e}_i) = \vec{c}_i \quad \vec{e}_i \text{ seien die Standardbasisvektoren} \end{array} \right\} \Rightarrow T(\vec{x}) = C \cdot \vec{x}$$

Beweis:

$$T(\vec{x}) = T\left(\sum_i x_i \vec{e}_i\right) = \sum_i x_i T(\vec{e}_i) = \sum_i x_i \vec{c}_i = C \cdot \vec{x} \quad \text{mit } C := \begin{pmatrix} \vdots & \vdots & \vdots \\ \vec{c}_1 & \vec{c}_2 & \vec{c}_3 \\ \vdots & \vdots & \vdots \end{pmatrix}$$

□

5. Rotation einer Schachtel

5.6.1 Lineare Transformationen in verschiedenen Basisdarstellungen

Sei $B = \{\vec{v}_1, \vec{v}_2, \vec{v}_3\}$ ebenfalls eine Basis des \mathbb{R}^3 .

\vec{x} sei die Darstellung des Vektors in der Standardbasis

$[\vec{x}]_B = (\lambda_1, \lambda_2, \lambda_3)^T$ die Darstellung in der Basis B . Es gilt also:

$$\vec{x} = \lambda_1 \vec{v}_1 + \lambda_2 \vec{v}_2 + \lambda_3 \vec{v}_3 = \underbrace{\begin{pmatrix} \vdots & \vdots & \vdots \\ \vec{v}_1 & \vec{v}_2 & \vec{v}_3 \\ \vdots & \vdots & \vdots \end{pmatrix}}_C \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{pmatrix} = C [\vec{x}]_B \quad (5.1)$$

Der Wechsel zwischen den Basisdarstellungen findet also folgendermaßen statt:

$$\vec{x} = C [\vec{x}]_B \quad (5.2)$$

$$[\vec{x}]_B = C^{-1} \vec{x} \quad (5.3)$$

Haben wir eine lineare Transformation $T(\vec{x}) = A \cdot \vec{x}$ mit der Standardbasis, ergibt sich die Frage, welche Darstellung hat diese Transformation in der Basis B ?

$$T_B([\vec{x}]_B) = D \cdot [\vec{x}]_B \quad D = ?$$

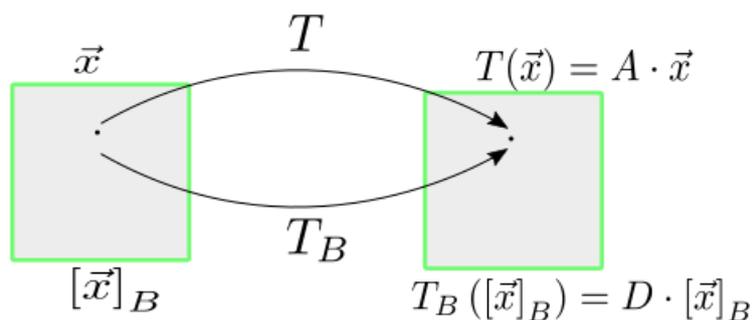


Abb.50 : Verschiedene Basisdarstellungen desselben Objekts

$$T_B([\vec{x}]_B) = [T(\vec{x})]_B \quad \text{linke und rechte Seite bezeichnen dasselbe Objekt}$$

$$D \cdot [\vec{x}]_B = [T(\vec{x})]_B \stackrel{5.3}{=} C^{-1} \cdot T(\vec{x}) = C^{-1} \cdot A \cdot \vec{x} \stackrel{5.2}{=} C^{-1} \cdot A \cdot C \cdot [\vec{x}]_B$$

also

$$D = C^{-1} \cdot A \cdot C \quad \Leftrightarrow \quad A = C \cdot D \cdot C^{-1} \quad \text{wobei } C \text{ die "Basismatrix" von } B \text{ ist}$$

Beispiel 5.4



Ein Anwendungsbeispiel im \mathbb{R}^2 :

Punkte S sollen entlang einer Geraden g , die durch den Ursprung mit dem Steigungswinkel α geht, gespiegelt werden. Wie lautet die Transformationsmatrix A ?

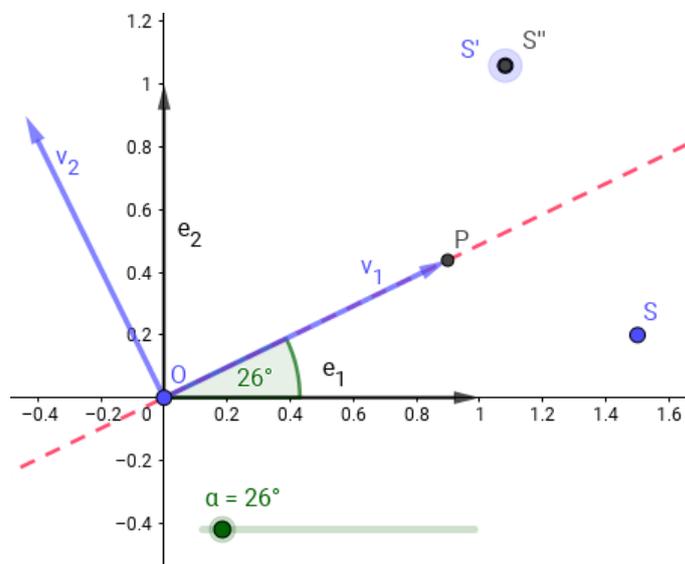


Abb.51 : Spiegelung eines Punktes S an einer Geraden mit Steigungswinkel α

In *Geogebra* ist die Aufgabe leicht gelöst:

- Punkte $O(0,0)$ und $P(\cos(\alpha), \sin(\alpha))$ festlegen und Schieberegler “abnicken”
- Gerade g festlegen: `g=Line(O,P)`
- Zu spiegelnder Punkt setzen: `S=(x_s,y_s)` und Schieberegler “abnicken”
- Jetzt spiegeln an g : `Reflect(S,g) → S'`

Welche Transformation wird mit dem letzten Befehl ausgeführt? *Ohne* etwas über Drehungen zu wissen, können wir diese Aufgabe knacken:

Bezüglich der Basisvektoren \vec{v}_1 und \vec{v}_2 ist die Transformation leicht: die y -Koordinate wechselt ihr Vorzeichen, also

$$D = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \quad \vec{v}_1 = \overrightarrow{OP} = \begin{pmatrix} \cos(\alpha) \\ \sin(\alpha) \end{pmatrix} \quad \vec{v}_2 = \begin{pmatrix} -y(\vec{v}_1) \\ x(\vec{v}_1) \end{pmatrix} = \begin{pmatrix} -\sin(\alpha) \\ \cos(\alpha) \end{pmatrix}$$

damit ergibt sich die orthogonale (vgl. Theorem 5.1) Basismatrix C zu

$$C = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \stackrel{\text{orthogonal}}{\Downarrow} C^{-1} = C^T = \begin{pmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{pmatrix} \Rightarrow$$

$$A = C \cdot D \cdot C^{-1} = \begin{pmatrix} \cos(2\alpha) & \sin(2\alpha) \\ \sin(2\alpha) & -\cos(2\alpha) \end{pmatrix}$$

5. Rotation einer Schachtel

5.6.2 Unser eigenes *Geogebra*-Werkzeug

Zuerst die Eingabe der Transformationsmatrix und anschl. auf S anwenden:

$$A = \left\{ \left\{ \cos(2\alpha), \sin(2\alpha) \right\}, \left\{ \sin(2\alpha), -\cos(2\alpha) \right\} \right\}$$
$$S'' = A * S$$

Wie man in Abb. 51 erkennen kann, macht unsere Matrix genau das, was auch *Geogebra* macht: $S'' = S'$

Jetzt sind die "Vorarbeiten" abgeschlossen. Wir gehen auf "Werkzeug erzeugen":

- output: S''
- input: α - A hängt ja von α ab - und S
- name: `flip` \rightarrow `flip[<Angle>, <Point>]`

Damit können wir unser Werkzeug verwenden:

- mit `R=flip[30°,Q]` - der Punkt Q muss natürlich existieren
- oder auf Werkzeug-Icon klicken und anschl. auf einen Winkel und Punkt - der Name wird dann von *Geogebra* gewählt.

6 | Rotationskörper mit *Geogebra*

Um mit *Geogebra*-Rotationskörper(revolution of solid) zu zeichnen hier einige Punkte, die man wissen sollte. Wenn die Sachverhalte bekannt sind - einfach überspringen!

1. Wie lautet der Funktionsterm einer Funktion g deren Graph gegenüber einer Funktion f um a nach rechts und b nach oben verschoben ist:

$$\begin{pmatrix} x \\ f(x) \end{pmatrix} + \begin{pmatrix} a \\ b \end{pmatrix} = \begin{pmatrix} x' \\ g(x') \end{pmatrix}$$

aus der 2.-ten Komponente folgt

$$g(x') = f(x) + b \quad - \text{daraus folgt mit der 1. Komponente} \Rightarrow g(x') = f(x' - a) + b$$

und da x' nur eine „dummy-Variable“ darstellt, können wir schreiben

$$g(x) = f(x - a) + b$$

2. Lineare (reelle) Funktionen f haben die Eigenschaft

$$f(\alpha a) = \alpha f(a) \quad \alpha \in \mathbb{R} \quad (6.1)$$

$$f(a + b) = f(a) + f(b) \quad a, b \in D_f \quad (6.2)$$

Leicht kann man zeigen, dass $f(x) = kx$ diese Eigenschaften besitzt.

3. Obige Definition lässt sich erweitern:

Sei A eine $n \times n$ Matrix mit den Elementen $a_{ik} \in \mathbb{R}$, so ist

$$\begin{aligned} \vec{f} : \mathbb{R}^n &\rightarrow \mathbb{R}^n \\ \vec{x} &\mapsto A\vec{x} := \sum_k a_{ik}x_k \end{aligned} \quad (6.3)$$

eine lineare Funktion. Durch Einsetzen lassen sich Eigenschaft (1) und (2) leicht beweisen. Beachten Sie, dass sich die Matrixmultiplikation auch deuten lässt als Linearkombination der Matrixspaltenvektoren:

$$y_i = \sum_k a_{ik}x_k \quad \text{oder} \quad \vec{y} = \sum_k \vec{a}_k x_k$$

wobei \vec{a}_k der k -te Spaltenvektor der Matrix A ist!

6. Rotationskörper mit *Geogebra*

4. Umgekehrt gilt auch: Ist eine Funktion von \mathbb{R}^n in \mathbb{R}^n linear, dann ist sie durch eine Matrixmultiplikation darstellbar:

$$\vec{y} = f(\vec{x}) = f\left(\sum_k \vec{e}_k x_k\right) = \sum_k \vec{f}(e_k) x_k = A\vec{x} \quad \text{siehe oben!}$$

Die Matrix A hat als Spaltenvektoren die Werte der Standardeinheitsvektoren

$$\vec{e}_i = (0, \dots, \underbrace{1}_{i\text{-te Pos.}}, 0, \dots)$$

5. Die Rotation um eine Achse ist eine lineare Abbildung \Rightarrow ist als Multiplikation mit einer (3×3) -Matrix darstellbar!

Wie man sich mit *Geogebra3d* leicht veranschaulicht, gilt sowohl

$$R(\vec{a} + \vec{b}) = R(\vec{a}) + R(\vec{b}) \quad \text{als auch} \quad R(\lambda \vec{a}) = \lambda R(\vec{a})$$

6. Um eine Rotationsmatrix um die x-Achse $R_x = r_{ij}$ festzulegen, reicht es nach obigen Überlegungen diese Abbildung f (wir lassen wegen der Kürze das Vektorzeichen bei f weg) bei den Basisvektoren $\vec{e}_1, \vec{e}_2, \vec{e}_3$ zu kennen.

$$f(\vec{e}_1) = (1, 0, 0) \quad \leftarrow \quad \text{1. Spaltenvektor von } R_x$$

Dieser Vektor gehört zu den Invarianten, da er auf der Rotationsachse liegt (Fixpunkt)

Mit nebenstehender Abbildung ergibt sich $f(\vec{e}_2)$:

$$f(\vec{e}_2) = (0, \cos \theta, \sin \theta) \quad \leftarrow \quad \text{2. Spaltenvektor von } R_x$$

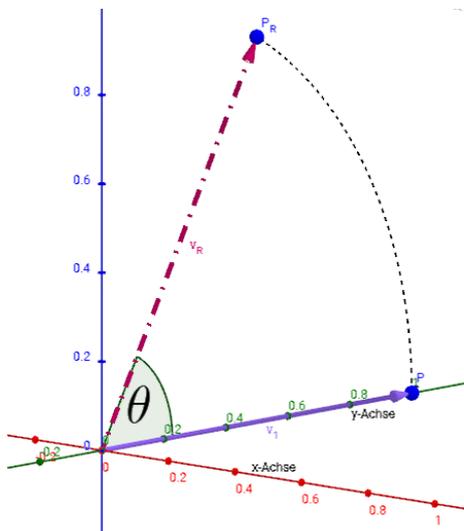
Es sei dem Leser als Übung überlassen, dass gilt

$$f(\vec{e}_3) = (0, -\sin \theta, \cos \theta) \quad \leftarrow \quad \text{3. Spaltenvektor von } R_x$$

Damit ergibt sich R_x und analog R_y zu

$$R_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{pmatrix}$$

$$R_y = \begin{pmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{pmatrix}$$



7. So jetzt haben wir unser Ziel “einen Rotationskörper in *Geogebra*” zu zeichnen fast erreicht: Wir brauchen jetzt nur mehr den Graphen einer Funktion g um die x- bzw. y-Achse zu rotieren:

$$R_x \underbrace{\begin{pmatrix} t \\ g(t) \\ 0 \end{pmatrix}}_{\text{Graph von } g} = \begin{pmatrix} t \\ \cos \theta g(t) \\ \sin \theta g(t) \end{pmatrix} = S_x \text{ wobei } t \in [a, b], \theta \in [0, 2\pi]$$

Dies ist also die Parameterdarstellung der Oberfläche (engl. *surface*) eines Rotationskörpers (Rotationsachse ist x-Achse). Analog ergibt sich für die y-Achse

$$R_y \begin{pmatrix} t \\ g(t) \\ 0 \end{pmatrix} = \begin{pmatrix} t \cos \theta \\ g(t) \\ -t \sin \theta \end{pmatrix} = S_y \text{ wobei } t \in [a, b], \theta \in [0, 2\pi]$$

8. **Beispiel in *Geogebra*:** Wir verschieben einen Kreis mit Radius 2 um 4 Einheiten in Richtung positiver x-Achse und lassen ihn sowohl um die x-Achse (es entsteht eine Kugel) als auch um die y-Achse (es entsteht ein Torus) rotieren:

pos./neg. Halbkreis im Ursprung: ${}_1g_2(x) = \pm\sqrt{2^2 - x^2}$

Verschiebung: ${}_1k_2(x) = {}_1g_2(x - 4) = \pm\sqrt{4 - (x - 4)^2}$

- (a) Wir definieren den Funktionsbereich $D_g = [a, b] = [2, 6]$ in *Geogebra*(Eingabezeile) und die Funktionen g_i :

```
a=2
b=6
g_1(x)=Funktion[+sqrt(4-(x-4)^2), a, b]
g_2(x)=Funktion[-sqrt(4-(x-4)^2), a, b]
```

- (b) Jetzt auf 3D-Ansicht wechseln und den Oberflächenbefehl verwenden (bei S_x nachschauen):

```
S_x=Oberfläche[t, cos(theta)*g_1(t), sin(theta)*g_1(t), t, a, b, theta, 0, 2*pi]
```

- (c) Damit wir diesen “umständlichen” Befehl nicht immer eingeben müssen, basteln wir uns einen Makro (in *Geogebra* “Werkzeug” genannt):

- In Menüzeile *Werkzeuge* → *Neues Werkzeug erstellen*
- Reiter *Eingabe Objekte*: Im darunterliegender Eingabezeile (*Objekte in der Konstruktion*) nacheinander g_1 , a und b wählen anschl. *Weiter* klicken.

6. Rotationskörper mit *Geogebra*

- Jetzt den Namen des Makros eingeben: `rot_x` ,
den Hilfetext z.B.: *Rotationkörper um die xAchse*
- Ausgabeobjekt wählen: `S_x`
und *Fertigstellen* → dies wird hoffentlich mit einer Erfolgsmeldung abgeschlossen!
- In Menüzeile *Werkzeuge* → *Werkzeuge verwalten* kann man dieses Makro als **.ggt-Datei (Geogebra Tool)* abspeichern. Wenn man das Werkzeug braucht, dann wie eine “normale *Geogebra-Datei*” (*.ggb) öffnen - **nicht** in *Werkzeuge verwalten Öffnen* wählen!

So jetzt können wir das neue Werkzeug gleich ausprobieren:

Wir erzeugen einen Grammophontrichter mit

```
rot_x[0.1*x^2,1,5]
```

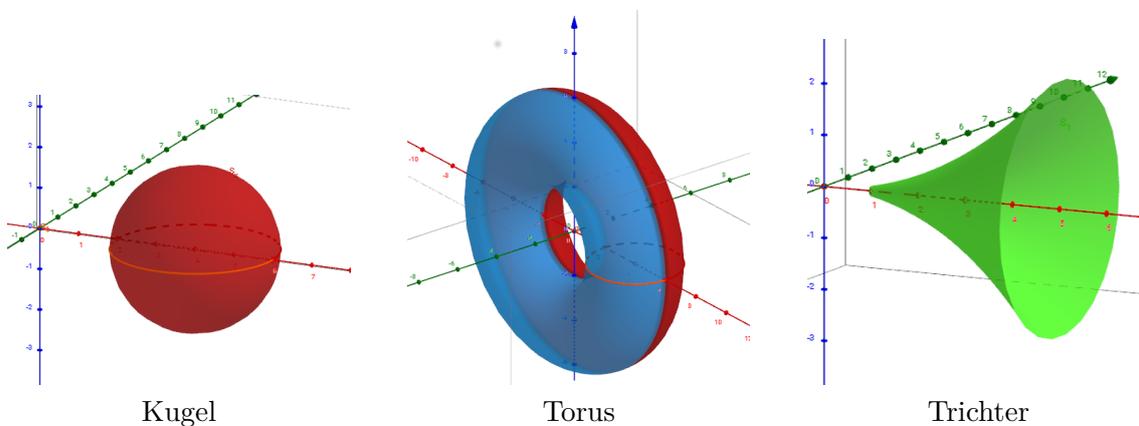
in der Eingabezeile - und schon wird der Rotationskörper angezeigt.

So jetzt noch der Torus - also g_1 und g_2 um die y -Achse rotieren:

```
S_y1=Oberfläche[t*cos(θ),g_1(t), -t*sin(θ), t, a, b, θ, 0, 2*pi]
```

```
S_y2=Oberfläche[t*cos(θ),g_2(t), -t*sin(θ), t, a, b, θ, 0, 2*pi]
```

Natürlich kann man sich auch wieder ein Werkzeug `rot_y` nach obigem Muster erstellen!



7 | Moivre-Laplace und Stetigkeitskorrektur

Abstract:

Vorstellung und Veranschaulichung des Satzes (mit *wxMaxima*), Stetigkeitskorrektur, Beispiel für seine Benutzung

7.1 Moivre-Laplace Theorem

Sei $b_{n,p}(k) = \binom{n}{k} p^k (1-p)^{n-k}$ die Dichtefunktion einer Binomialverteilung mit $\mu = np$ und $\sigma^2 = np(1-p)$ dann gilt

$$\lim_{n \rightarrow \infty} b_{n,p}(k) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{(k-\mu)^2}{2\sigma^2}\right] = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{k-\mu}{\sigma}\right)^2} = \varphi_{\mu,\sigma}(k)$$

wobei der Ausdruck auf der rechten Seite der Dichte der Normalverteilung entspricht. Verbal ausgedrückt:

Die diskrete Binomialverteilung kann für große n durch die stetige Normalverteilung angenähert werden.

“Große n” heißt als Faustregel

$$\sigma^2 > 9 \Leftrightarrow \sigma > 3$$

7.2 GNU-Plot

Bevor wir uns das in *wxMaxima* veranschaulichen, vielleicht noch eine kurze Einführung in gnuplot (das *wxMaxima* verwendet):

Es verfügt über einen Linienzeichenprogramm, das sich sehr leicht scripten lässt - die Syntax sieht so aus: `wxplot2d([discrete, Liste[Liste1, Liste2, ...]])`

Liste1, Liste2 sind Koordinatenlisten von Punkten abwechselnd x und y. Diese Punkte werden

7. Moivre-Laplace und Stetigkeitskorrektur

durch Linien verbunden. 'wxplot2d' nimmt als Input wiederum eine Liste von Funktionen (diskret oder kontinuierlich). Am besten wir schauen uns das an einem Beispiel an:

```
(%i15) wxplot2d([0.5*x,[discrete, [[0,0,1,1], [1,0.5,2,0]]]],  
               [x,0,2], [legend,"f","g"]);
```

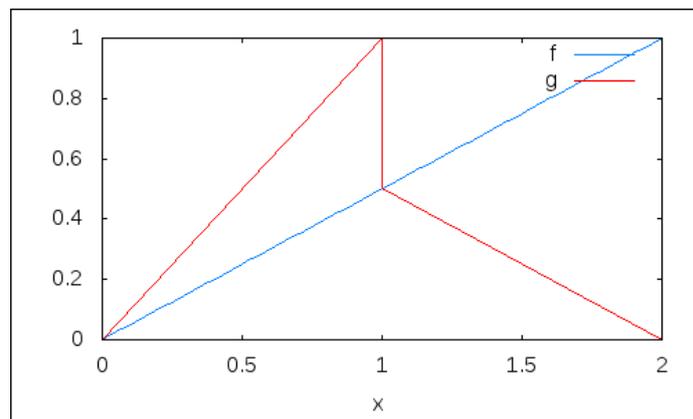


Abb.52 : Funktionsplot

Es werden 2 Funktionen gezeichnet: $0.5x$ und eine diskrete aus den Punkten $(0,0)$ und $(1,1)$ und aus den Punkten $(1,0.5)$ und $(2,0)$. Alle diese Punkte werden durch Linien verbunden (default - Einstellung; sie lässt sich auch ändern - aber das ist kein Kurs über Gnuplot) Außerdem wird er x-Bereich der Zeichnung (Intervall $[0,2]$) festgelegt und wie die Legende auszuschauen hat.

7.3 Binomial- und Normalverteilung mit *wxMaxima*

Wir nehmen z.B. eine Binomialverteilung mit den Parametern $p = 0,4 \Rightarrow q = 0,6$ und erhöhen n so, dass obige Faustregel erfüllt ist: z.B. $n = 50$

Geben wir das im *wxMaxima* ein:

```
(%i1) n:50$p:0.4$q:1-p$
```

```
(%i4) bin(k):=binomial(n,k)*p^k*q^(n-k);
```

```
(%i5) w1:makelist([i,0,i,bin(i),i+1,bin(i)],i,5,35)$
```

zur Erinnerung: `makelist(expr,var,start,end)` erstellt eine Liste, indem die Variable `var` alle Integerwerte von `start` bis `end` durchläuft: Also Linie von $(i,0)$ nach $(i,\text{bin}(i))$ weiter nach $(i+1,\text{bin}(i))$ - das ist eine Senkrechte und eine Waagrechte und so gehts weiter für alle $i \in [5,35]$. Bei allen übrigen sind die Wahrscheinlichkeiten so klein, dass sie im Diagramm verschwinden!

```
(%i6) %mu:n*p;
```

```
(%o6) 20.0
```

```
(%i7) %sigma:sqrt(%mu*q);      (%o7) 3.4641016      Man sieht σ > 3
(%i8) f(x):=1/(%sigma*sqrt(2*%pi))*e^(-0.5*((x-%mu)/%sigma)^2),numer;
(%o8) f(x) :=  $\frac{1}{\sigma \sqrt{2\pi}} e^{(-0.5) \left(\frac{x-\mu}{\sigma}\right)^2}$ 
(%i9) wxplot2d([f(x), [discrete,w1]], [x,5,35], [xlabel,"x-Werte"],
[legend,"gauss","binomial"]);
```

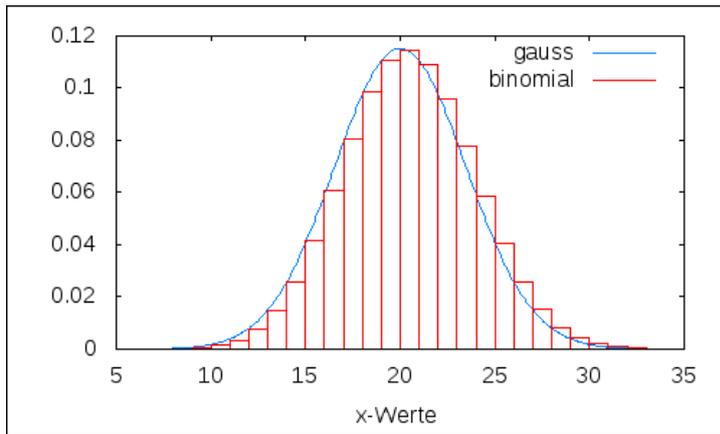


Abb.53 : Gauß-Binomialverteilung

Man sieht leicht, wie die Wahrscheinlichkeiten der Binomialverteilung gut gegen die Gaußkurve “konvergieren” - nur hier und da gibt es leichte Abweichungen. Also wäre ein erster Ansatz:

$$b_{n,p}(k) \approx \int_k^{k+1} \varphi(x) dx$$

aber es geht noch besser, dazu zoomen wir in das Diagramm!

```
(%i10) wxplot2d([f(x), [discrete,w1]], [x,14,17], [legend,"f(x)","bin"] );
```

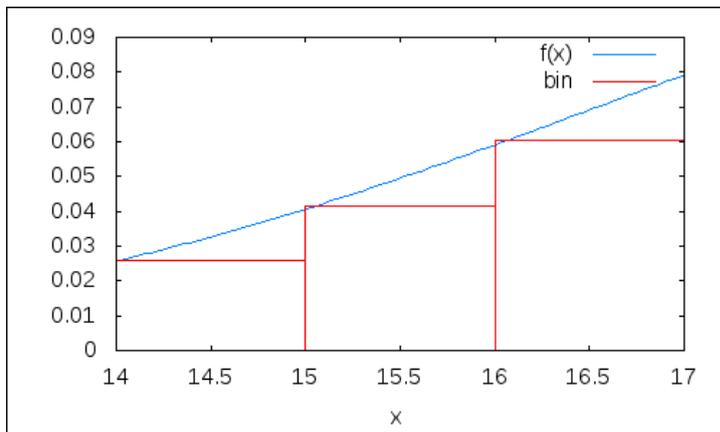


Abb.54 : Gauß-Binomialverteilung-zoomed

deutlich kann man erkennen:

$$b_{50;0,4}(15) < \int_{15}^{15+1} \varphi(x) dx$$

dies gilt natürlich für alle Werte der Zufallsvariablen $X < \mu$ - rechts von μ drehen sich die Größenverhältnisse um.

Würde der Graph von φ eine Gerade sein, könnten wir ein flächengleiches Rechteck “basteln”, indem wir die Integrationsgrenzen um $\frac{1}{2}$ verschieben - wie die unteren Zeichnungen deutlich machen:

7. Moivre-Laplace und Stetigkeitskorrektur

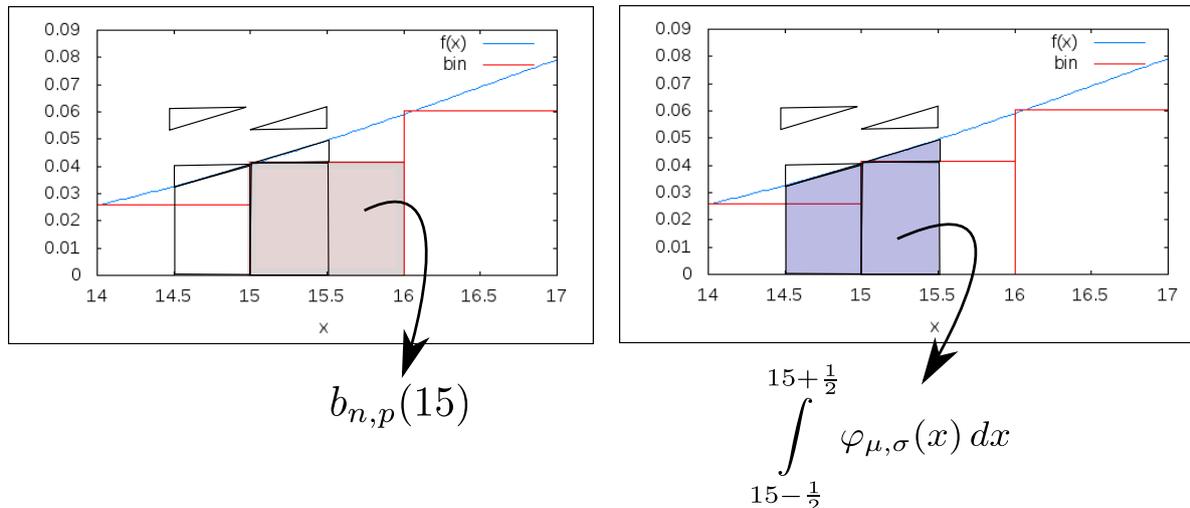


Abb.55 : Stetigkeitskorrektur

Halten wir fest: $b_{n;p}(k) \approx \int_k^{k+1} \varphi(x) dx$ **gut**

$$b_{n;p}(k) \approx \int_{k-\frac{1}{2}}^{k+\frac{1}{2}} \varphi(x) dx = \int_{k-\frac{1}{2}}^{k+\frac{1}{2}} \varphi(x) dx \quad \text{ist besser!}$$

Wie man sich leicht überzeugen kann, gilt für Summen:

$$P(a \leq X \leq b) = \sum_{i=a}^b b_{n;p}(i) \approx \int_{a-\frac{1}{2}}^{b+\frac{1}{2}} \varphi(x) dx$$

Wir schauen uns die Ergebnisse jetzt numerisch in *wxMaxima* an:

(%i11) `sum(bin(i),i,15,17);` $\rightarrow \sum_{i=15}^{17} b_{n,p}(i)$ (%o11) 0.182920766322365

`quad_qags` ist einer der numerischen Integrationsalgorithmen von *wxMaxima*, als output liefert es den Wert des Integrals und den Fehler und einige andere Dinge, die hier nicht wesentlich sind (bedenken Sie, dass die Gaußfunktion über keine Stammfunktion verfügt und daher die "übliche Methode" `integrate(expr, var, start, end)` nicht zum Ziel führt!

(%i12) `quad_qags (f(x), x , 15, 17+1);` $\rightarrow \int_{15}^{17+1} f(x) dx$
 (%o12) [.2073940942360036, 2.302536985920908 10⁻¹⁵, 21, 0]

```
(%i13) quad_qags ( f(x), x , 15-0.5, 17+0.5);
```

```
(%o13) [.1790676121841628, 1.988049860115059 10-15, 21, 0]
```

```
(%i14) sum(bin(i), i, 24, 26);
```

```
(%o14) .1247627778007346
```

```
(%i15) quad_qags ( f(x), x , 24, 26+1);
```

```
(%o15) [.1024524680895658, 1.137450890027101 10-15, 21, 0]
```

```
(%i16) quad_qags ( f(x), x , 24-0.5, 26+0.5);
```

```
(%o16) [.1258597259821049, 1.39732365658346 10-15, 21, 0]
```

Jetzt das Ganze noch mit dem Maxima-Modul “distrib” (distribution - Verteilung)
 pdf_* steht für “probability density function” also die Dichtefunktion
 cdf_* steht für “cumulative density function” als die Verteilungsfunktion; bei der Normalverteilung oft mit $\Phi(z; \mu; \sigma)$ bezeichnet

```
(%i17) load("distrib");
```

```
(%o17) /usr/share/maxima/5.29.1/share/distrib/distrib.mac
```

```
(%i18) cdf_normal(18,%mu,%sigma)-cdf_normal(15,%mu,%sigma), numer;
```

```
(%o18) .2073940942360036
```

```
(%i19) cdf_binomial(17, n, p)- cdf_binomial(14, n, p);
```

```
(%o19) .1829207663223664
```

```
(%i20) sum(pdf_binomial(i,n,p), i, 15, 17);
```

```
(%o20) 0.182920766322365
```

```
(%i24) wxplot2d(pdf_normal(x,%mu,%sigma), [x, 5, 35], [legend, Phi(z)]);
```

7. Moivre-Laplace und Stetigkeitskorrektur

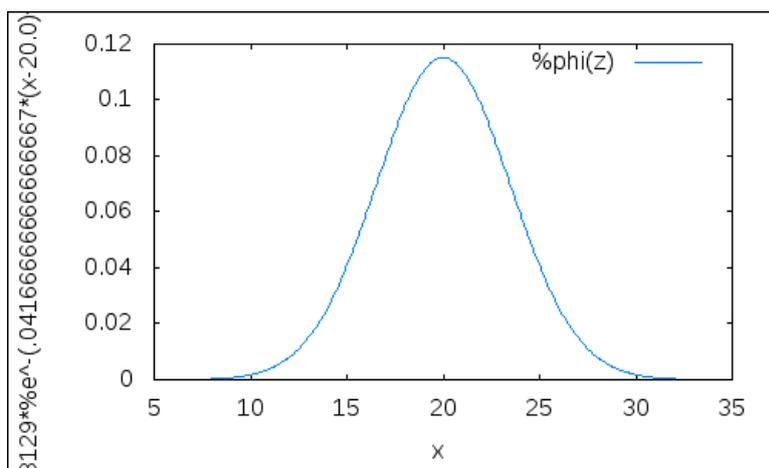


Abb.56 : Normalverteilung

Ein weiterer Grund für die Benutzung des Theorems ist, dass $b_{n;p}(k)$ für große n praktisch ohne Spezialsoftware unberechenbar wird:

$$b_{n;p}(k) = \underbrace{\binom{n}{k}}_{\rightarrow \infty} \underbrace{p^k (1-p)^{n-k}}_{\rightarrow 0} = ?$$

Ein Summenzeichen davor macht die Sache auch nicht besser! Bleiben wir bei unserem obigen Beispiel $p = 0,4$ und erhöhen wir n auf 500. Sei X die Anzahl mit der unser spezifisches Ereignis eintritt und wir möchten wissen, wie groß die Wahrscheinlichkeit ist, dass diese Anzahl um weniger als eine Standardabweichung vom Erwartungswert abweicht:

$$P(|X - \mu| < \sigma) = P(\mu - \sigma < X < \mu + \sigma) = P(190 \leq X \leq 210) = \sum_{i=190}^{210} \binom{500}{i} 0.4^i 0.6^{500-i}$$

Die Summanden der letzten Summe sind für einen "normalen" Taschenrechner praktisch nicht berechenbar, z.B.:

$$\binom{500}{200} \approx 5 \cdot 10^{144} \quad \text{und} \quad 0.4^{200} \cdot 0.6^{300} \approx 7 \cdot 10^{-147}$$

$$\Phi_{\mu,\sigma}(210, 5) - \Phi_{\mu,\sigma}(189, 5) \quad \underbrace{=}_{\text{standardisieren}} \quad \Phi\left(\frac{10,5}{\sigma}\right) - \Phi\left(-\frac{10,5}{\sigma}\right) = 2\Phi\left(\frac{10,5}{\sigma}\right) - 1 = 2\Phi(0.9585) - 1$$

Während obige Binomial-Wahrscheinlichkeiten nicht mehr mit jedem x-beliebigen Taschenrechner zu berechnen sind, ist das letzte Ergebnis mit jeder Normalverteilungstabelle leicht zu ermitteln!

Wie sieht das in *wxMaxima* aus?

```
(%i22) n:500;
```

```
(%o22) 500
```

```
(%i23) %mu:n*p;
```

```
(%o23) 200.0
```

```
(%i24) %sigma:sqrt(%mu*q);
```

```
(%o24) 10.95445115010332
```

```
(%i25) f(x):=1/(%sigma*sqrt(2*%pi))*e^(-0.5*((x-%mu)/%sigma)^2)$
```

Hier das Integral mit Stetigkeitskorrektur!

```
(%i26) quad_qags ( f(x), x , 189.5, 210.5);
```

```
(%o26) [.6621966043761834, 7.351859170070347 10-15, 21, 0]
```

Mit der “eingebauten” Normalverteilungstabelle

```
(%i27) 2*cdf_normal(0.9585,0,1)-1, numer;
```

```
(%o27) .6621893085136601
```

Hier die Summe mit der “eingebauten” Verteilungsfunktion: Beachte die Grenze 189

```
(%i28) cdf_binomial(210, 500, p)- cdf_binomial(189, 500, p);
```

```
(%o28) .6621956027530275
```

Hier die Summe “zu Fuß”

```
(%i29) display(sum(bin(i), i, 190, 210))$
```

$$\sum_{i=190}^{210} \text{bin}(i) = .6621956027529964$$

Fazit: Es ist gleichgültig, welche Methode man wählt, Hauptsache man macht es richtig!

Apropos “gleichgültig, welche Methode man wählt”. Sehr einfach gehts jetzt auch mit *Geogebra* 4.2+. Hier verbirgt sich im Menü “ABC” ein Wahrscheinlichkeitsrechner, der extrem einfach zu bedienen ist. In 1 stellt man die Parameter ein ($b_{n;p}$), in 2 die Intervallgrenzen und schon kann man das Ergebnis in 3 ablesen. Als “Zucker” gibts in 4 μ und σ , in 5 die “Teilwahrscheinlichkeiten” und in 6 eine grafische Aufbereitung. Dieser Rechner ist extrem schnell und macht eigentlich dieses Dokument über Stetigkeitskorrektur beinahe überflüssig!

7. Moivre-Laplace und Stetigkeitskorrektur

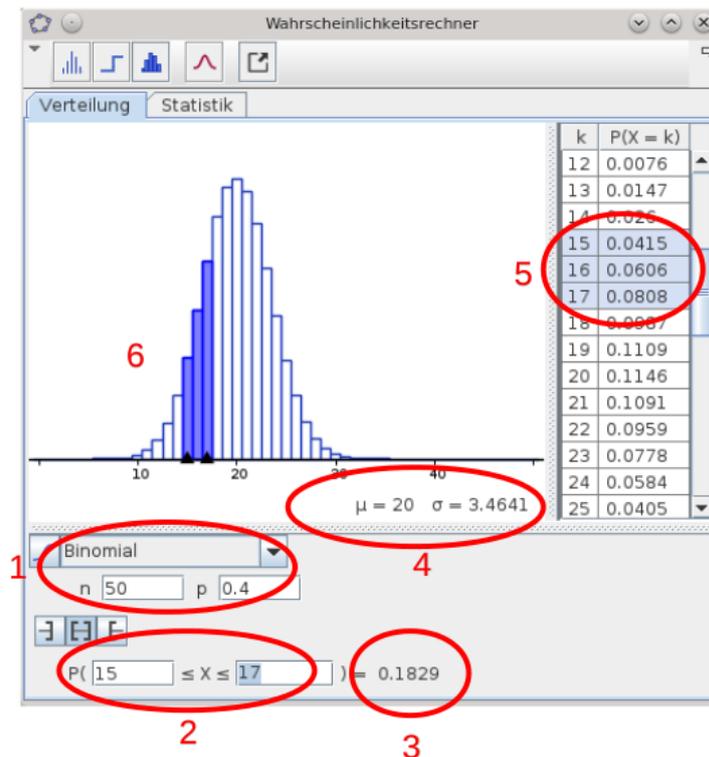


Abb.57 : Geogebra Wahrscheinlichkeitsrechner

Nachsatz: Vergessen Sie nicht auf die Voraussetzung $\sigma^2 > 9$. Im andern Fall (p weicht weit von 0,5 ab und/oder n ist nicht allzu groß), kann die Binomialverteilung "sehr schief" sein und die Symmetrie bricht zusammen!

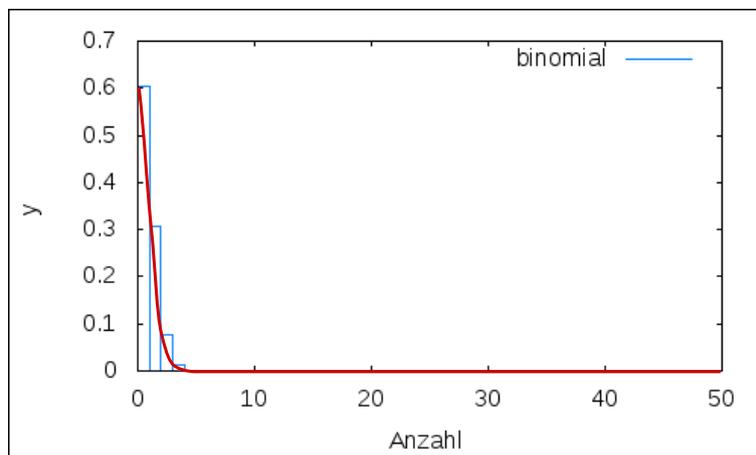


Abb.58 : Keine Stetigkeitskorrektur möglich

Hier nur als Warnung eine Binomialverteilung mit den Parametern $n = 50$, $p = 0.05$. An den Eckpunkten ist eine kontinuierliche rote Kurve gezogen - von einer Glockenkurve ist hier nichts zu sehen. Irgendwie erinnert das an eine fallende Exponentialfunktion - aber die Poissonverteilung ist wieder eine andere Geschichte!

8 | Lagrange Calculus

Vor einigen Monaten stieß ich bei einem Youtube-Video auf die Differentialrechnung von Lagrange - diese bezieht sich zwar nur auf Polynome, aber das bemerkenswerte daran ist, dass man weder Grenzwerte braucht noch Regeln über die Differentialrechnung - die "Taylorreihe" für das Polynom ergibt sich ganz von selbst.

Außerdem so eine Einschränkung ist das ja nicht, da sich ja nach dem Satz von Weierstrass jede stetige Funktion in einem Intervall beliebig genau durch ein Polynom annähern lässt!

8.1 Der Grundgedanke

Sei $p(x) = \sum_{i=0}^n a_i x^i$ ein Polynom n-ten Grades mit $a_i \in \mathbb{R}$. Wir verschieben jetzt p um r -Einheiten nach links

$$p_r(x) = p(x+r) = \sum_{i=0}^n a_i (x+r)^i = \sum_{i=0}^n a_i \sum_{k=0}^i \binom{i}{k} x^k r^{i-k}$$

wobei die Potenzen expandiert und nach den Exponenten von x geordnet werden. p_r wird jetzt wieder r -Einheiten nach rechts verschoben, sodass sich das ursprüngliche p ergibt:

$$p(x) = p_r(x-r) = \sum_{i=0}^n a_i \sum_{k=0}^i \binom{i}{k} (x-r)^k r^{i-k}$$

wobei letztes Polynom nicht mehr expandiert wird (wozu auch - wir wissen was herauskommt!) Wir haben also $p(x)$ als Ausdruck einer beliebigen "Nachbarstelle" r ausgedrückt. Berechnet man mit obigen Ausdruck $p(r)$ verschwinden in der zweiten Summe alle Summanden außer $k=0$ damit ergibt sich $p(r) = \sum_{i=0}^n a_i r^i$ - was nicht verwunderlich ist.

Ist x in der Nähe von r , gilt insbesondere $|x-r| < 1$, dann werden die Ausdrücke $(x-r)^k$ immer unbedeutender. Man kann sich also obige Darstellung von p in der Nähe von r nach Größenordnung gereiht denken!

Folgendes Beispiel in *wxMaxima* soll das verdeutlichen :

8. Lagrange Calculus

8.2 Ein Beispiel

```
(%i1) p(x):=2*x^3+3*x^2-12*x+2;
```

```
(%o1) p(x) := 2 - 12 · x + 3 · x2 + 2 · x3
```

```
(%i3) s(x):=ratsimp(expand(p(x+r)))$ display(s(x))$
```

$$s(x) = 2 \cdot x^3 + (3 + 6 \cdot r) \cdot x^2 + (-12 + 6 \cdot r + 6 \cdot r^2) \cdot x + 2 \cdot r^3 + 3 \cdot r^2 - 12 \cdot r + 2$$

$p(x)$ in neuer Schreibweise; Beachte, dass $\lim_{x \rightarrow r} p(x)$ leicht zum Ausführen ist!

```
(%i5) p_neu(x):=subst(x-r,x,s(x))$ display(p_neu(x))$
```

$$p(x) = 2 \cdot (x - r)^3 + (3 + 6 \cdot r) \cdot (x - r)^2 + (-12 + 6 \cdot r + 6 \cdot r^2) \cdot (x - r) + \overbrace{2 \cdot r^3 + 3 \cdot r^2 - 12 \cdot r + 2}^{p(r)}$$

Nun die linearen Terme in $(x - r)$ ausmultipliziert!

```
(%i6) h_1(x):=(-12+6*r+6*r^2)*(x-r)+2*r^3+3*r^2-12*r+2$
```

```
(%i8) p_1(x):=ratsimp(h_1(x))$ display(p_1(x))$
```

```
(%o8) p_1(x) = (-12 + 6 · r + 6 · r2) · x - 4 · r3 - 3 · r2 + 2
```

Nur bis zu den quadratischen Termen (Kegelschnitte - conics) - ausmultipliziert!

```
(%i9) h_2(x):=(3+6*r)*(x-r)^2+(-12+6*r+6*r^2)*(x-r)+2*r^3+3*r^2-12*r+2$
```

```
(%i11) p_2(x):=ratsimp(h_2(x))$ display(p_2(x))$
```

```
(%o11) p_2(x) = (3 + 6 · r) · x2 + (-12 - 6 · r2) · x + 2 · r3 + 2
```

Wir wählen eine spezielle Stelle: $r=-1$

```
(%i12) r:-1$
```

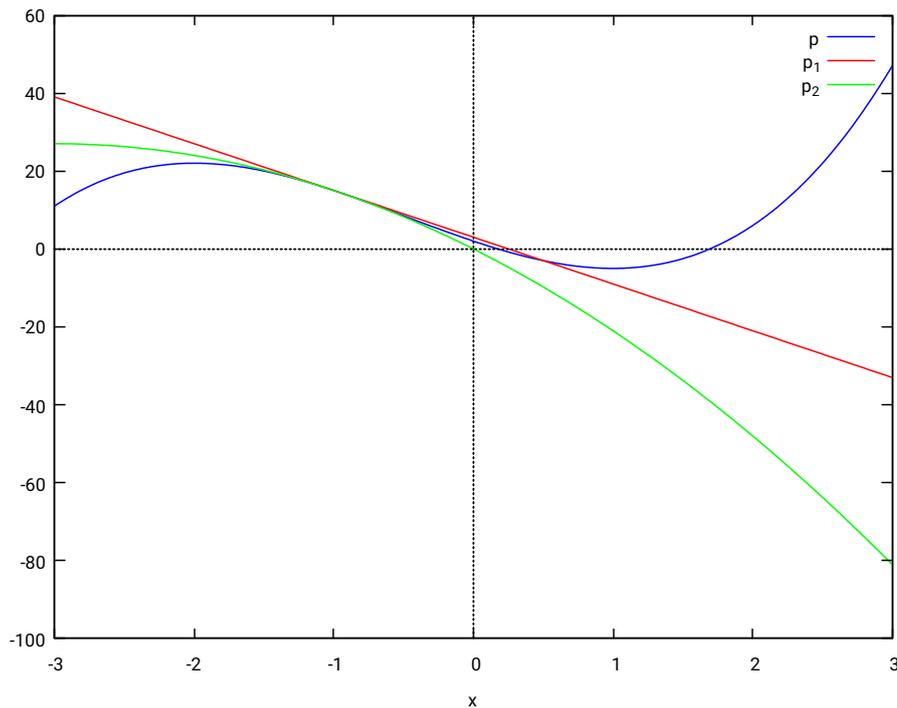
```
(%i13) p_1(x);
```

```
(%o13) 3 - 12 · x
```

```
(%i14) p_2(x);
```

```
(%o14) -3 · x2 - 18 · x
```

```
(%i15) plot2d([p,p_1,p_2],[x,-3,3])$
```

Abb.59 : p mit den Näherungen p_1 und p_2 bei $r = -1$

Wichtig ist die Ausgabe von **(%i5)**

Klar erkennt man wieder $p(x = r) = 2 \cdot r^3 + 3 \cdot r^2 - 12 \cdot r + 2 = p(r)$

Aber man kann $p(x)$ auch schreiben als

$$p(x) = D_3(r)(x - r)^3 + D_2(r)(x - r)^2 + D_1(r)(x - r) + D_0(r) = \sum_{i=0}^n D_i(r) (x - r)^i$$

Welche Bedeutung haben die einzelnen $D_i(r)$ -Funktionen? $D_0(r) = p(r)$ wie man durch einfaches Einsetzen sehen kann. Bringen wir $D_0(r) = p(r)$ auf die andere Seite und dividieren durch $x - r$ ergibt sich die Gleichung

$$\frac{p(x) - p(r)}{x - r} = D_3(r)(x - r)^2 + D_2(r)(x - r) + D_1(r) \quad | \quad \lim_{x \rightarrow r}$$
 auf beiden Seiten ergibt

$$\lim_{x \rightarrow r} \frac{p(x) - p(r)}{x - r} = p'(r) = D_1(r)$$

Wir können auch (1) auf jeder Seite differenzieren und gehen vor wie oben, um ein Ergebnis für D_2 zu erhalten

$$\frac{p'(x) - p'(r)}{x - r} = D_3(r)3(x - r) + 2D_2(r) \quad | \quad \lim_{x \rightarrow r} \text{ auf beiden Seiten ergibt } D_2(r) = \frac{p''(r)}{2}$$

8. Lagrange Calculus

Man sieht leicht, dass sich allgemein ergibt

$$D_i(r) = \frac{p^{(i)}(r)}{i!}$$

Die $D_i(r)$ sind also die „Taylorkoeffizienten“. Zur Erinnerung Taylor für Polynome vom Grad n :

$$p(r+h) = \sum_{i=0}^n \frac{p^{(i)}(r)}{i!} h^i \quad \text{mit } r+h = x \text{ folgt}$$

$$p(x) = \sum_{i=0}^n \underbrace{\frac{p^{(i)}(r)}{i!}}_{D_i(r)} (x-r)^i$$



Beachte : Die $D_i(r)$ von Lagrange sind leichter zu handhaben als die Ableitungen, da die Fakultät bereits „integriert“ ist - damit ergeben sich i.a. kleinere Zahlen.

Auch verschiedene Ableitungsregeln lassen sich leicht zeigen. Hier am Beispiel der Produktregel:

$$p(x) = \sum_{i=0}^n D_{i,p}(r) (x-r)^i \quad \text{und} \quad q(x) = \sum_{i=0}^n D_{i,q}(r) (x-r)^i \quad \Rightarrow$$

$$p(x) \cdot q(x) = D_{0,p}D_{0,q} + \underbrace{(D_{0,p}D_{1,q} + D_{0,q}D_{1,p})}_{D_{1,pq}}(x-r) + \dots$$

Da wir D_1 mit $\frac{d}{dx}$ identifiziert haben, ergibt sich:

$$\boxed{\frac{d}{dx}(pq) = p \frac{dq}{dx} + \frac{dp}{dx} q}$$

Im nächsten Kapitel sieht man, dass die Einschränkung auf Polynome für diese Rechnungsart gilt, nicht so stark ist wie sie klingt!

9 | Satz von Weierstrass

Die Polynome nehmen unter den stetigen Funktionen eine wichtige Rolle ein - genau wie mit anderen Funktionsgruppen (z.B.: Fourier-Reihen) - kann man damit in einem beliebigen Intervall $[a, b]$ stetige Funktionen beliebig genau annähern. Um dies zu zeigen nehmen wir einen "kleinen" Umweg über Folgen von Funktionen und einen kurzen Überblick über die Binomialverteilung.

9.1 Funktionsfolgen

Definition 9.1 Punktweise Konvergenz

Sei $f_n : A \rightarrow \mathbb{R}$ eine Folge von Funktionen. Wir sagen f_n konvergiert punktweise gegen $f : A \rightarrow \mathbb{R}$ wenn gilt:

$$\forall x \in A : \lim_{n \rightarrow \infty} f_n(x) = f(x) \Leftrightarrow \forall x \in A \exists N(x, \varepsilon) \in \mathbb{N} \forall n > N : |f_n(x) - f(x)| < \varepsilon$$

Die punktweise Konvergenz ist vermutlich der Konvergenzbegriff, der sich bei der Beschäftigung mit dem Thema zuerst aufdrängt. Trotzdem werden wir an Hand der folgenden Beispiele zeigen, dass damit einige "Pferdefüße" verbunden sind - die man so nicht vermutet hätte!

Beispiel 9.2



Sei $f_n :]0, 1[\rightarrow \mathbb{R}$ definiert durch

$$f_n(x) := \frac{n}{nx + 1} \quad (\text{Beachte: } f_n(x) < n)$$

$$\text{Wegen } x \neq 0 : \lim_{n \rightarrow \infty} f_n(x) = \lim_{n \rightarrow \infty} \frac{1}{x + \frac{1}{n}} = \frac{1}{x} = f(x)$$

Obwohl alle f_n beschränkt sind, ist f unbeschränkt!
Beschränktheit wird bei der punktweisen Konvergenz nicht erhalten!

Beispiel 9.3



Sei $f_n : [0, 1] \rightarrow \mathbb{R}$ definiert durch $f_n(x) := x^n$ dann gilt:

$$\forall x \in [0, 1[: \lim_{n \rightarrow \infty} x^n = 0 \quad \text{und für } x = 1 : \lim_{n \rightarrow \infty} x^n = 1 \quad \text{also}$$

$$f(x) = \begin{cases} 0 & 0 \leq x < 1 \\ 1 & x = 1 \end{cases}$$

Obwohl alle f_n stetig sind, ist f unstetig!
 Stetigkeit wird bei der punktweisen Konvergenz nicht erhalten!

Beispiel 9.4



Sei $f_n : [0, 1] \rightarrow \mathbb{R}$ definiert durch:

$$f_n(x) = \begin{cases} 2n^2x & \text{für } 0 \leq x \leq \frac{1}{2n} \\ 2n^2\left(\frac{1}{n} - x\right) & \text{für } \frac{1}{2n} < x < \frac{1}{n} \\ 0 & \text{für } \frac{1}{n} \leq x \leq 1 \end{cases}$$

Schauen wir uns die Graphen der ersten 10 Folgenglieder an:

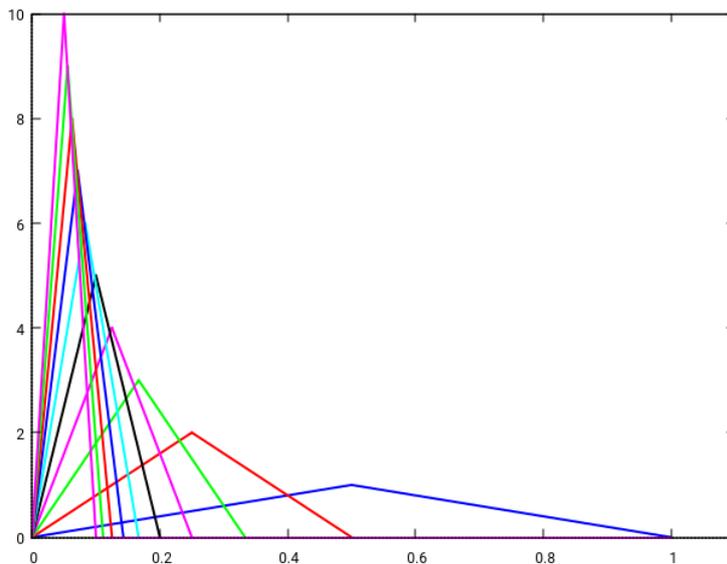


Abb.60 : Die ersten 10 Funktionen der Folge

- Das Maximum der f_n wird am Ende des 1. Astes erreicht:

$$f_n\left(\frac{1}{2n}\right) = n \Rightarrow \max\left\{\lim_{n \rightarrow \infty} f_n\right\} = \infty \Rightarrow \text{die } f_n \text{ sind unbeschränkt}$$

-

$$\forall x \in]0, 1] \exists n \in \mathbb{N} : \frac{1}{n} < x \Rightarrow \text{für jedes } x \text{ ist schließlich der 3.-te Ast zuständig}$$

$$\Rightarrow \forall x \in]0, 1] : \lim_{n \rightarrow \infty} f_n(x) = 0 \quad \wedge \quad f_n(0) = 0 \Rightarrow \forall x \in [0, 1] : \lim_{n \rightarrow \infty} f_n(x) = 0$$

also $f(x) = 0$ für $x \in [0, 1]$

Obwohl $\lim_{n \rightarrow \infty} f_n(x) = 0$ für $x \in [0, 1]$ ist die Folge der f_n unbeschränkt!

- Hier nur kurz der *wxMaxima*-Code für obigen Graph!

Hier die Liste der Funktionsäste und ihre Definitionsmengen:

(%i1) `f_n_List:[2*n^2*t, 2*n^2*(1/n-t), 0] $`

(%i2) `domainList:[[t,0,1/(2*n)], [t,1/(2*n),1/n], [t,1/n,1.001]] $`

Wir erstellen die Plotliste (in Parameterform können wir die Definitionsmengen einzeln festlen!)

(%i3) `plotList(N):=block([size:length(f_n_List),L:[]],
for n thru N do
L:append(L,makelist([parametric,t,ev(f_n_List[i]),ev(domainList[i]),i,1,size)],
L)) $`

Eine alternative Form zu obigem *plotList(N)*: Nested List wird mit *apply* zusammengefügt)

inneres *makelist* \rightarrow `[L(1)(n),L(2)(n),L(3)(n)]` mit $L(i)(n) = [\text{parametric}, t, \text{term}(i)(n), \text{domain}(i)(n)]$

äußeres *makelist* \rightarrow `[[L(1)(1),L(2)(1),L(3)(1)], [L(1)(2),L(2)(2),L(3)(2)], [. . . L(1)(N),L(2)(N),L(3)(N)]]`

Beachte: Durch *ev(.)* werden die n -Terme bei jedem Durchlauf ausgewertet

apply(+, [x, y, z]) \rightarrow $x + y + z$ auf eine Liste wird eine Funktion angewendet - in unserem Fall *append* - damit wird eine "Schachtelebene" entfernt!

apply(append, [[1], [2]]) \rightarrow `[1, 2]`

(%i4) `plotLList(N):=apply(append,makelist(
makelist([parametric,t,ev(f_n_List[i]),ev(domainList[i]),i,1,length(f_n_List)),n,1,N)) $`

Wie oben in der "alternativen Form" werden die Linienfarben für die die Funktionsäste erstellt!

(%i5) `makeStyleList(N):=cons(style,apply(append,makelist(
makelist([lines,2,color],i,1,length(f_n_List)),color,1,N))) $`

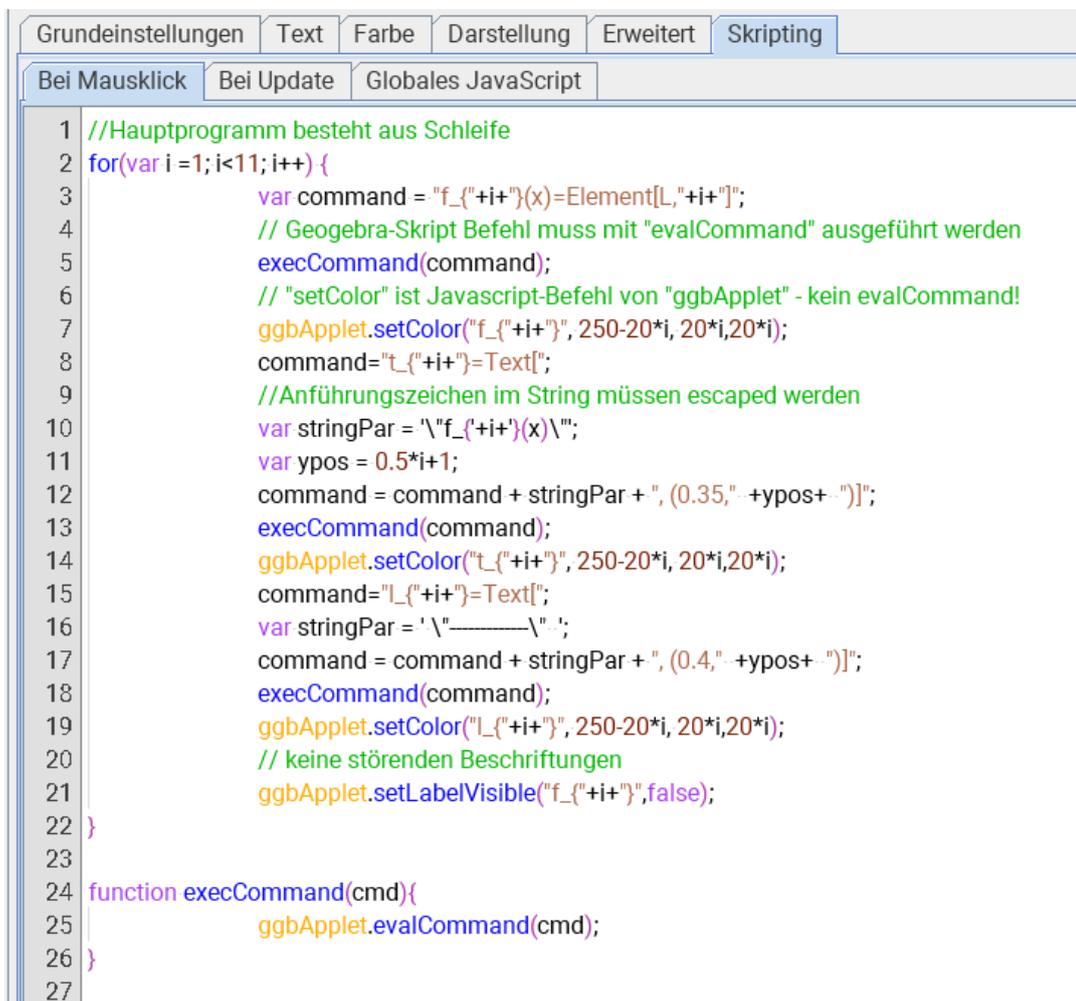
(%i6) `plot2d(plotLList(10),makeStyleList(10),[x,0,1.1],[legend,""] /*,[same_xy]*/);`

9. Satz von Weierstrass

- Auch in *Geogebra* lässt sich die Funktionsfolge veranschaulichen - wir erstellen eine Liste der ersten 10 Funktionen mit einer verschachtelten WENN-Funktion:

```
L=Folge[Wenn[0 <= x <= 1 / (2*n), 2n^2*x,  
  Wenn[1/(2*n)<x<1/n, 2*n^2*(1/n - x),  
  Wenn[1/n <= x <= 1, 0]], n, 1, 10]
```

Allerdings lassen sich den Folgengliedern keine Farben zuordnen. Wir setzen daher die Liste L auf unsichtbar und erstellen einen Button "Show Func" mit folgendem Javascript (bei Mausklick):



```
Grundeinstellungen Text Farbe Darstellung Erweitert Skripting
Bei Mausklick Bei Update Globales JavaScript
1 //Hauptprogramm besteht aus Schleife
2 for(var i =1;i<11;i++){
3     var command = "f_{"+i+"}(x)=Element[L,"+i+"]";
4     // Geogebra-Skript Befehl muss mit "evalCommand" ausgeführt werden
5     execCommand(command);
6     // "setColor" ist Javascript-Befehl von "ggbApplet" - kein evalCommand!
7     ggbApplet.setColor("f_{"+i+"}", 250-20*i, 20*i,20*i);
8     command="t_{"+i+"}=Text[";
9     //Anführungszeichen im String müssen escaped werden
10    var stringPar = "\"f_{"+i+"}(x)\\";
11    var ypos = 0.5*i+1;
12    command = command + stringPar + ", (0.35," +ypos+ ")";
13    execCommand(command);
14    ggbApplet.setColor("t_{"+i+"}", 250-20*i, 20*i,20*i);
15    command="l_{"+i+"}=Text[";
16    var stringPar = "\"-----\\";
17    command = command + stringPar + ", (0.4," +ypos+ ")";
18    execCommand(command);
19    ggbApplet.setColor("l_{"+i+"}", 250-20*i, 20*i,20*i);
20    // keine störenden Beschriftungen
21    ggbApplet.setLabelVisible("f_{"+i+"}",false);
22 }
23
24 function execCommand(cmd){
25     ggbApplet.evalCommand(cmd);
26 }
27
```

Abb.61 : Javascript für Funktionsfolgen in *Geogebra*

```
1 for(var i =1;i<11;i++){
2     ggbApplet.deleteObject("f_{"+i+"}"); ggbApplet.deleteObject("t_{"+i+"}");ggbApplet.deleteObject("l_{"+i+"}");
3 }
```

Abb.62 : Manchmal ganz brauchbar:Javascript für Button-Löschen

■ *Geogebra*-Graph

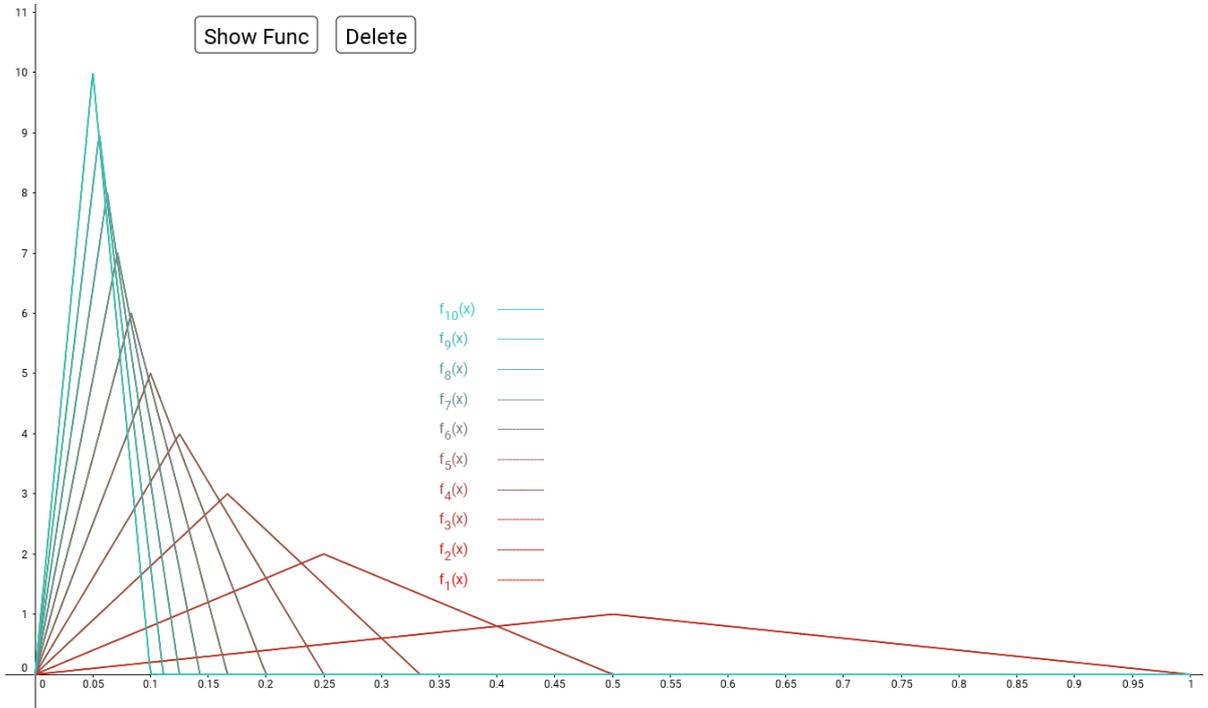


Abb.63 : Funktionen-Folge in *Geogebra*

Beispiel 9.5



Sei $f_n : \mathbb{R} \rightarrow \mathbb{R}$ definiert durch: $f_n(x) = \frac{\sin(nx)}{n}$

Jetzt gilt:

$$\forall x \in \mathbb{R} \exists N \in \mathbb{N} \forall n > N : |f_n(x) - 0| < \varepsilon \Rightarrow \lim_{n \rightarrow \infty} f_n(x) = 0$$

Falls man die Folge differenziert $f'_n(x) = \cos(nx)$ - konvergiert sie an manchen Stellen nicht mehr, z.B.:

$$\cos(n\pi) = (-1)^n$$

Für punktweise konvergente Funktionsfolgen **gilt nicht**:

$$\lim_{n \rightarrow \infty} \frac{d}{dx} f_n(x) = \frac{d}{dx} \lim_{n \rightarrow \infty} f_n(x)$$

Beispiel 9.6



Sei $f_n : \mathbb{R} \rightarrow \mathbb{R}$ definiert durch: $f_n(x) = \frac{x^2}{\sqrt{x^2 + \frac{1}{n}}}$

$$\forall x \neq 0 : \lim_{n \rightarrow \infty} f_n(x) = \frac{x^2}{|x|} = |x| \quad \wedge \quad \lim_{n \rightarrow \infty} f_n(0) = 0 \Rightarrow \boxed{\forall x : \lim_{n \rightarrow \infty} f_n(x) = |x|}$$

Während alle f_n bei 0 differenzierbar sind, ist es die Grenzwertfunktion nicht

$$\lim_{n \rightarrow \infty} \frac{d}{dx} f_n(x) \neq \frac{d}{dx} f(x) \quad \text{falls } f_n \text{ punktweise konvergent}$$

Theorem 9.7

Sei $f_n : \mathbb{R} \rightarrow \mathbb{R}$ definiert durch: $f_n(x) = \left(1 + \frac{x}{n}\right)^n \Rightarrow \lim_{n \rightarrow \infty} f_n(x) = e^x$

Beweis 1: Wir logarithmieren obige Behauptung:

$$\ln \left(\lim_{n \rightarrow \infty} \left(1 + \frac{x}{n}\right)^n \right) \stackrel{!}{=} \ln(e^x) \stackrel{\text{Stetigkeit}}{\implies} \lim_{n \rightarrow \infty} n \ln \left(1 + \frac{x}{n}\right) \stackrel{!}{=} x \quad (9.1)$$

Der Grenzwert ist von der Form "0 · ∞" und den kann man mit L'Hospital behandeln:

$$\lim_{n \rightarrow \infty} \frac{\frac{d}{dn} \ln \left(1 + \frac{x}{n}\right)}{\frac{d}{dn} \frac{1}{n}} = \lim_{n \rightarrow \infty} \left(1 + \frac{x}{n}\right)^{-1} (-x n^{-2}) (-n^2) = x$$

Für negative x in 9.1 lässt man die ersten N -Folglieder weg, bis gilt: $\left|\frac{x}{N}\right| < 1$.

Das ändert nichts am Konvergenzverhalten! □

Beweis 2: Eine andere Möglichkeit zu zeigen, dass gilt: $\lim_{n \rightarrow \infty} n \ln \left(1 + \frac{x}{n}\right) = x$

Nach Taylor gilt für $x \rightarrow 0$ bzw. $n \rightarrow \infty$:

$$\ln(1+x) = x + \mathcal{O}(x^2) \Rightarrow n \left[\ln \left(1 + \frac{x}{n}\right) \right] = n \left[\frac{x}{n} + \mathcal{O} \left(\left(\frac{x}{n}\right)^2 \right) \right] = x + \mathcal{O} \left(\frac{x^2}{n} \right)$$

Beweis 3: Dieser Beweis nutzt den Binomischen Lehrsatz (Theorem 9.16) □

$$\lim_{n \rightarrow \infty} \left(1 + \frac{x}{n}\right)^n = \lim_{n \rightarrow \infty} \sum_{k=0}^n \binom{n}{k} \left(\frac{x}{n}\right)^k 1^{n-k} = \lim_{n \rightarrow \infty} \sum_{k=0}^n \frac{1}{k!} x^k \underbrace{\frac{n!}{(n-k)! n^k}}_{=1}$$

Die übrigbleibende Reihe $\sum_{k=0}^{\infty} \frac{x^k}{k!}$ ist die Taylorreihe von e^x □

Definition 9.8 Gleichmäßige Konvergenz

Sei $f_n : A \rightarrow \mathbb{R}$ eine Folge von Funktionen. Wir sagen f_n konvergiert gleichmäßig gegen $f : A \rightarrow \mathbb{R}$ wenn gilt:

$$\exists N(\varepsilon) \in \mathbb{N} \forall x \in A \forall n > N : |f_n(x) - f(x)| < \varepsilon$$

Beachten Sie den Unterschied in der Stellung der Quantoren gegenüber der punktweisen Konvergenz. Hier: es existiert ein ε für alle x gegenüber "für alle x existiert ein ε " bei der punktweisen Konvergenz! Sollte der Unterschied nicht klar sein, denken Sie an: Für alle Männer existiert eine Frau gegenüber es existiert eine Frau für alle Männer!

Beispiel 9.9

Wir sehen uns noch einmal Beispiel 9.3 an:

Sei $f_n : [0, 1] \rightarrow \mathbb{R}$ definiert durch $f_n(x) := x^n$ dann gilt in $[0, 1[$:

$$|f_n(x) - f(x)| = |x^n| < \varepsilon \Rightarrow x < \varepsilon^{\frac{1}{n}}$$

Wie groß man n auch wählt - es gilt nicht für alle x . (Übrigens 1 macht überhaupt keine Probleme!). Allerdings ist diese Funktion im Intervall $[0, b]$ mit $0 \leq b < 1$ gleichmäßig konvergent:

$$\forall n > N = \frac{\ln \varepsilon}{\ln b} : x^n < \varepsilon$$

Hier nur 2 Theoreme(ohne Beweis):

Theorem 9.10 Erhaltung der Beschränktheit

$f_n \rightarrow f \Leftrightarrow f_n$ konvergiert gegen f

Sei $f_n : A \rightarrow \mathbb{R}$

$$\left. \begin{array}{l} \forall n \in \mathbb{N} : f_n \text{ ist beschränkt in } A \\ \wedge \\ f_n \rightarrow f \text{ gleichmäßig in } A \end{array} \right\} \Rightarrow f \text{ ist beschränkt in } A$$

Also gleichmäßige Konvergenz erhält Beschränktheit!

Theorem 9.11 Erhaltung der Stetigkeit

Sei $f_n : A \rightarrow \mathbb{R}$

$$\left. \begin{array}{l} \forall n \in \mathbb{N} : f_n \text{ ist stetig in } A \\ \wedge \\ f_n \rightarrow f \text{ gleichm\u00e4\u00dfig in } A \end{array} \right\} \Rightarrow f \text{ ist stetig in } A$$

Also gleichm\u00e4\u00dfige Konvergenz erh\u00e4lt Stetigkeit!

Beispiel 9.12



Das Beispiel von 9.2 Sei $f_n :]0, 1[\rightarrow \mathbb{R}$ definiert durch

$$f_n(x) := \frac{n}{nx + 1} \quad (\text{Beachte: } f_n(x) < n)$$

(Alle f_n beschr\u00e4nkt und $f(x) = \frac{1}{x}$ unbeschr\u00e4nkt) $\stackrel{9.10}{\Rightarrow} f_n \rightarrow f$ **nicht** gleichm\u00e4\u00dfig

In $[a, 1[$ allerdings gilt: $f_n \rightarrow f$ gleichm\u00e4\u00dfig ($N = 1/(a^2\varepsilon)$)

F\u00fcr die Erhaltung der Differenzierbarkeit haben wir die Bedingungen zuverst\u00e4rken:

Theorem 9.13 Erhaltung der Differenzierbarkeit

Sei $f_n :]a, b[\rightarrow \mathbb{R}$

$$\left. \begin{array}{l} f_n \rightarrow f \text{ punktweise in }]a, b[\\ \text{und} \\ f'_n \rightarrow g \text{ gleichm\u00e4\u00dfig in }]a, b[\end{array} \right\} \Rightarrow f' = g \text{ in }]a, b[$$

Die f'_n brauchen nicht stetig zu sein! Setzt man die Stetigkeit der f'_n zus\u00e4tzlich voraus (die f'_n sind dann Riemann-integrierbar) geht der Beweis einfach mit dem Fundamentalsatz der Analysis (Fundamental Theorem of Calculus FTC):

$$\int_{x_0}^x f'(t) dt = f(x) - f(x_0) \tag{9.2}$$

Beweisskizze:

$$\begin{aligned} \int_{x_0}^x f'_n(t) dt &\rightarrow \int_{x_0}^x g(t) dt && \text{wegen gleichm\u00e4\u00dfiger Konvergenz} \\ f(x) - f(x_0) &\rightarrow \int_{x_0}^x g(t) dt && \text{wegen FTC 9.2 (links vom Pfeil angewendet)} \\ & && f' = g \text{ wegen FTC 9.2 (rechts vom Pfeil angewendet)} \end{aligned}$$

□

9.2 Die sup-Norm

Ein äquivalenter und vielleicht sogar klarerer Weg die gleichmäßige Konvergenz zu beschreiben.

Definition 9.14 sup-Norm

Sei $f_n : A \rightarrow \mathbb{R}$. Unter gleichmäßiger Norm oder sup-Norm $\|f\|$ von f auf A versteht man:

$$\|f\| = \sup_{x \in A} |f(x)|$$

Eine Funktion f ist beschränkt $\Leftrightarrow \|f\| < \infty$

Definition 9.15 gleichmäßige Konvergenz einer Funktionenfolge

Eine Folge von Funktionen $f_n : A \rightarrow \mathbb{R}$ konvergiert gleichmäßig gegen f auf $A \Leftrightarrow \lim_{n \rightarrow \infty} \|f_n - f\| = 0$

Ist A ein abgeschlossenes Intervall wird natürlich aus der Supremums-Norm eine Maximums-Norm!

Um den Satz von Weierstrass zu beweisen, brauchen wir einige Ergebnisse der Binomialverteilung. Deshalb jetzt ein kleiner statistischer Einschub!

9.3 Binomialverteilung

Theorem 9.16 Binomischer Lehrsatz

$$(a + b)^n = \sum_{k=0}^n \binom{n}{k} b^k a^{n-k} \quad \text{wobei} \quad \binom{n}{k} := \frac{n!}{k!(n-k)!}$$

Beweis: In den meisten Fällen wird der Beweis mit vollständiger Induktion geführt. Ich möchte hier einen Beweis mit Hilfe der "Permutation" führen. Wir betrachten vorerst den Aufbau

9. Satz von Weierstrass

der Summe für den Spezialfall $n = 4$ indem wir den "a's" und "b's" durch Indices Individualität verleihen:

$$\begin{aligned}
 (a + b)^4 &= (a_1 + b_1)(a_2 + b_2)(a_3 + b_3)(a_4 + b_4) = \\
 &a_1 a_2 a_3 a_4 + \\
 &\quad + b_1 a_2 a_3 a_4 + a_1 b_2 a_3 a_4 + a_1 a_2 b_3 a_4 + a_1 a_2 a_3 b_4 + \\
 &\quad + b_1 b_2 a_3 a_4 + b_1 a_2 b_3 a_4 + b_1 a_2 a_3 b_4 + a_1 b_2 b_3 a_4 + a_1 b_2 a_3 b_4 + a_1 a_2 b_3 b_4 + \\
 &\quad + a_1 b_2 b_3 b_4 + b_1 a_2 b_3 b_4 + b_1 b_2 a_3 b_4 + b_1 b_2 b_3 a_4 + \\
 &b_1 b_2 b_3 b_4
 \end{aligned}$$

Wie sind die einzelnen Summanden aufgebaut?

In jedem Term kommen immer alle 4 Indices von 1 bis 4 vor - ich hab sie oben nach dem Vorkommen der Anzahl von "b's" gereiht.

1. Zeile:

Keine b. Term lautet $b^0 a^4$. Anzahl dieser Terme: $1 = \frac{4!}{0!4!}$
 (Umordnung von 1234 dividiert durch Umordnung der b und a)

2. Zeile:

Ein(1) b. Term lautet $b^1 a^3$. Anzahl dieser Terme: $4 = \frac{4!}{1!3!}$
 (Umordnung von 1234 dividiert durch Umordnung der b und a)

3. Zeile:

Zwei b. Term lautet $b^2 a^2$. Anzahl dieser Terme: $6 = \frac{4!}{2!2!}$
 (Umordnung von 1234 dividiert durch Umordnung der b und a)

4. Zeile:

Drei b. Term lautet $b^3 a^1$. Anzahl dieser Terme: $4 = \frac{4!}{3!1!}$
 (Umordnung von 1234 dividiert durch Umordnung der b und a)

5. Zeile:

Vier b. Term lautet $b^4 a^0$. Anzahl dieser Terme: $1 = \frac{4!}{4!0!}$
 (Umordnung von 1234 dividiert durch Umordnung der b und a)

Allgemein gilt: Es kommen alle n Indices vor. Die Zeile mit den k b's liefert:

Term lautet $b^k a^{n-k}$, Anzahl: $\frac{n!}{k!(n-k)!} = \binom{n}{k}$ □



Beachte: Mit $a = 1$ und $b = 1$ liefert obiger Satz $2^n = \sum_{k=0}^n \binom{n}{k}$ - mit jeder Multiplikation verdoppeln sich die Anzahl der Terme

9.4 Binomialverteilung einer Zufallsvariablen X

Der Ausfallsraum (Wahrscheinlichkeitsraum) eines Zufallversuchs sei $\Omega = A_1 \cup A_1'$ ($A_1' = \Omega \setminus A_1$).

Die Wahrscheinlichkeit, dass $\omega \in A_1$ sei p , formal: $P(\omega \in A_1) = p$.

Klarerweise gilt: $P(\omega \in A_1') = 1 - p =: q$. Wird nun dieser Zufallversuch n -mal wiederholt, kann man sich fragen mit welcher Wahrscheinlichkeit k -mal das Ereignis $\omega \in A_1$ eintritt? Sei die Zufallsvariable $X \in \{0, 1, 2, \dots, n\}$ das Auftreten von $\omega \in A_1$. Wir fragen uns also $P(X = k) = ?$

Ein mögliches Ereignis hat die Wahrscheinlichkeit:

$$\underbrace{p \cdot p \cdot \dots \cdot p}_{k\text{-mal}} \cdot \underbrace{q \cdot q \cdot \dots \cdot q}_{(n-k)\text{-mal}} = p^k q^{n-k}$$

Alle Terme mit k -mal p 's und $(n - k)$ -mal q 's erhält man durch Permutation der n -Faktoren, wobei die Permutationen unter den p 's und q 's nicht zu unterscheiden sind!

Definition 9.17 Binomialverteilung

Eine Zufallsvariable X heißt binomialverteilt, wenn gilt:

$$P(X = k) = \frac{n!}{k!(n-k)!} p^k q^{n-k} = \binom{n}{k} p^k q^{n-k}$$

Theorem 9.18 Axiome der Wahrscheinlichkeitsrechnung

Obige Definition 9.17 erfüllt die Axiome der Wahrscheinlichkeitsrechnung

Beweis: Nichtnegativität und Additivität sind klar.

Für $p \in [0, 1]$ und dem binomischen Satz ergibt sich

$$1 = (p + q)^n = \sum_{k=0}^n \binom{n}{k} p^k q^{n-k} = \sum_{k=0}^n P(X = k)$$

□

In vielen Lehrbüchern fehlt die Berechnung des Erwartungswertes $E(X)$ bzw. der Varianz $V(X)$.

Wir erinnern uns in der beschreibenden Statistik gilt

$$\mu = \sum_i a_i p_i \quad \text{bzw.} \quad \sigma^2 = \sum_i (a_i - \mu)^2 p_i = \sum_i \left(a_i^2 p_i - \underbrace{2\mu a_i p_i}_{2\mu^2} + \underbrace{\mu^2 p_i}_{\mu^2} \right) = \sum_i a_i^2 p_i - \mu^2$$

9. Satz von Weierstrass

hier werden diese Formeln zu

$$E(X) = \sum_{k=0}^n k \binom{n}{k} p^k q^{n-k} \quad \text{bzw.} \quad V(X) = \sum_{k=0}^n k^2 \binom{n}{k} p^k q^{n-k} - [E(X)]^2 \quad (9.3)$$

Theorem 9.19 Erwartungswert und Varianz von X

$$P(X = k) = \binom{n}{k} p^k q^{n-k} \Rightarrow E(X) = np \quad \text{und} \quad V(X) = npq$$

Beweis:

$$f(p) = (p + q)^n = \sum_{k=0}^n \binom{n}{k} p^k q^{n-k} \left| \frac{d}{dp} \right. \quad (9.4)$$

$$n(p + q)^{n-1} = \sum_{k=0}^n \binom{n}{k} k p^{k-1} q^{n-k} \left| \cdot p \right.$$

$$np(p + q)^{n-1} = \sum_{k=0}^n \binom{n}{k} k p^k q^{n-k} \left| \quad q = (1 - p) \right.$$

$$np = \sum_{k=0}^n k \binom{n}{k} p^k (1 - p)^{n-k} = E(X) \quad (9.5)$$

Auf 9.4 wenden wir $\frac{d^2}{dp^2}$ an und multiplizieren mit p^2 :

$$n(n-1)(p+q)^{n-2} p^2 = \sum_{k=0}^n \binom{n}{k} k(k-1) p^k q^{n-k} = \sum_{k=0}^n \binom{n}{k} k^2 p^k q^{n-k} - \underbrace{\sum_{k=0}^n \binom{n}{k} k p^k q^{n-k}}_{np}$$

Wenn wir in obiger Zeile wieder $q = 1 - p$ setzen ergibt sich

$$n(n-1)p^2 + np = \sum_{k=0}^n k^2 \binom{n}{k} p^k (1-p)^{n-k} \quad (9.6)$$

Diese Identität verwenden wir in 9.3 zusammen mit 9.5 für die Berechnung der Varianz:

$$V(X) = n(n-1)p^2 + np - (np)^2 = np(1-p)$$

□

9.5 Approximationssatz von Weierstrass

Theorem 9.20 Satz von Weierstrass

Gegeben sei eine stetige Funktion

$$f : [a, b] \rightarrow \mathbb{R} \quad \Rightarrow \quad \forall \varepsilon \exists n \in \mathbb{N} : \|f - p_n\| < \varepsilon$$

Wobei p_n ein Polynom vom Grad n ist, also

$$p_n(x) = \sum_{i=0}^n a_i x^i \quad \text{wobei } a_i \in \mathbb{R}$$

Wobei bei obiger Norm die Maximum-Norm (bzw. sup-Norm) gemeint ist:

$$\|f - p_n\| := \|f - p_n\|_\infty = \max\{|f(x) - p_n(x)| \mid x \in [a, b]\}$$

Damit ist sichergestellt, dass sich p_n innerhalb eines “ ε -Schlauchs” um f befindet! (Siehe Abschnitt sup-Norm 9.2)

Bevor wir uns dem etwas längerem Beweis dieses Theorems zuwenden, der zeigt, dass z.B. die Bernsteinpolynome die Behauptung dieses Theorems erfüllen, können wir den Sachverhalt insofern vereinfachen, weil wir **jede** stetige Funktion $f : [a, b] \rightarrow \mathbb{R}$ mit einer bijektiven Funktion g (d.h. g^{-1} existiert) auf das Intervall $[0, 1]$ “komprimieren” können, also

$$\bar{f} : [0, 1] \rightarrow \mathbb{R} \quad \wedge \quad \bar{f} = f \circ g^{-1} \quad \wedge \quad g : [a, b] \rightarrow [0, 1]$$

es gibt dann für jedes $x \in [a, b]$ ein $g(x) = t \in [0, 1]$ sodass gilt:

$$f(x) = f(g^{-1}(g(x))) = \bar{f}(g(x)) = \bar{f}(t) \Rightarrow \sup_{x \in [a, b]} |f(x)| = \sup_{t \in [0, 1]} |\bar{f}(t)| \Rightarrow \|f\| = \|\bar{f}\|$$



Da die Normen von f und \bar{f} übereinstimmen, reicht es den Satz von Weierstrass für das Intervall $[0, 1]$ zu beweisen. Die ursprüngliche Funktion $f = \bar{f} \circ g$ ist leicht wiederhergestellt! (Selbstverständlich gilt dann auch $\|\bar{f} - \bar{p}_n\| < \varepsilon \Leftrightarrow \|f - p_n\| < \varepsilon$)

Wir demonstrieren das graphisch in $[a, b] = [2, 6]$ mit den Funktionen

$$\begin{array}{lll} f : [a, b] \rightarrow \mathbb{R} & g : [a, b] \rightarrow [0, 1] & g^{-1} : [0, 1] \rightarrow [a, b] \\ x \rightarrow a - (x - (a + 1))^2 & x \rightarrow \frac{x - a}{b - a} & t \rightarrow a + (b - a)t \end{array}$$

9. Satz von Weierstrass

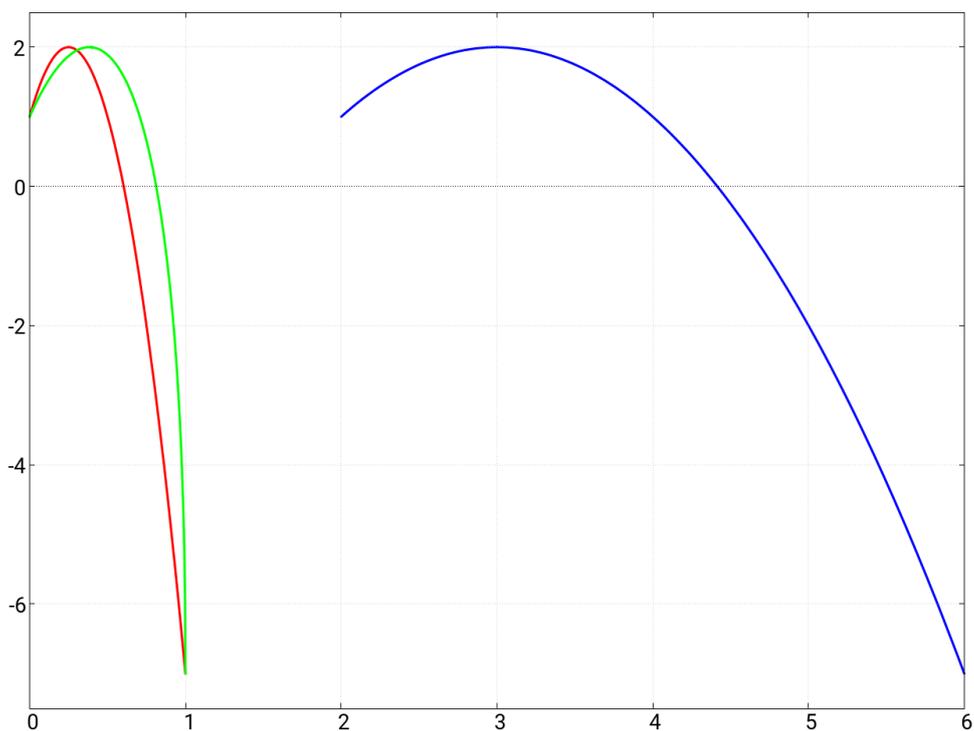


Abb.64 : Komprimierte Funktion $\bar{f} = f \circ g^{-1}$ (rot) und Original f (blau); $\|f\| = \|\bar{f}\| = 7$

Während wir mit g eine lineare “Kompressionsfunktion” verwendet haben - ist das nicht notwendig. Wir haben im obigen Graph auch eine grüne Funktion eingezeichnet, die folgendermaßen entstanden ist: $f \circ h^{-1}$ mit

$$\begin{array}{lll}
 f : [a, b] \rightarrow \mathbb{R} & h : [a, b] \rightarrow [0, 1] & h^{-1} : [0, 1] \rightarrow [a, b] \\
 x \rightarrow a - (x - (a + 1))^2 & x \rightarrow \sin\left(\frac{\pi x - a}{2(b - a)}\right) & t \rightarrow \frac{2(b - a)}{\pi} \arcsin(t) + a
 \end{array}$$

Obwohl sich die x-Lage des Maximums verändert hat, so ist doch die Norm wieder unverändert geblieben!

Auf der nächsten Seite noch der “Code” in *wxMaxima* zur Erzeugung des obigen Graphen - gegenüber den Voreinstellungen wurde der Schriftfont der x- und y-Achse vergrößert, die Legende weggelassen und die Linienstärke der 3 Funktionen auf 3 vergrößert! Der 2.-te Parameter von `lines` ist die Zeichenfarbe.

(%i2) a:2\$b:6\$

(%i3) f(x):=a-(x-(a+1))^2;

$$f(x) := a - (x - (a + 1))^2 \quad (\%o3)$$

(%i4) g(x):=(x-a)/(b-a);

$$g(x) := \frac{x - a}{b - a} \quad (\%o4)$$

(%i5) `g_inv(x):=(b-a)*x+a;`

$$g_{inv}(x) := (b - a)x + a \quad (\%o5)$$

(%i6) `h(x):=sin(%pi*(x-a)/(2*(b-a)));`

$$h(x) := \sin\left(\frac{\pi(x-a)}{2(b-a)}\right) \quad (\%o6)$$

(%i7) `h_inv(x):=2*(b-a)/%pi*asin(x)+a;`

$$h_{inv}(x) := \frac{2(b-a)}{\pi} \operatorname{asin}(x) + a \quad (\%o7)$$

(%i8) `plot2d([[parametric,t,f(t),[t,a,b]], [parametric,t,f(g_inv(t)),[t,0,1]],
[parametric,t,f(h_inv(t)),[t,0,1]]], [x,0,b],[y,-7.5,2.5],
[legend,""], [style,[lines,3,1],[lines,3,2],[lines,3,3]],
[gnuplot_preamble,"set xtics \"", 20\"; set ytics font\", 20\" "])`

Beweis: Wir zeigen, dass die Bernsteinpolynome b_n den Satz von Weierstrass erfüllen:

$$b_n(x) := \sum_{k=0}^n f\left(\frac{k}{n}\right) r_k(x) \quad \text{mit} \quad r_k(x) := \binom{n}{k} x^k (1-x)^{n-k}$$

Wir wissen aus obigen Überlegungen (binom. Lehrsatz und Varianz der Binomialverteilung), dass gilt

$$\sum_{k=0}^n r_k(x) = 1 \quad (9.7)$$

$$\sum_{k=0}^n (k - nx)^2 r_k(x) = nx(1-x) \quad (9.8)$$

außerdem gilt: $f(x) = f(x) \sum_{k=0}^n r_k(x) = \sum_{k=0}^n f\left(\frac{k}{n}\right) r_k(x)$ wegen 9.7

Wir werden zeigen, dass für alle ε ein $N \in \mathbb{N}$ existiert, sodass gilt (falls $n > N$):

$$\|f - b_n\| < \varepsilon$$

Wegen der Stetigkeit von f gilt: $|x_1 - x_2| < \delta \Rightarrow |f(x_1) - f(x_2)| < \frac{\varepsilon}{2}$

wir zerlegen \mathbb{N}_0 in zwei diskunkte Teilmengen K_1 und K_2 :

$$\begin{aligned} K_1 &:= \left\{ k \in \mathbb{N}_0 : \left| \frac{k}{n} - x \right| < \delta \right\} \\ K_2 &:= \mathbb{N}_0 \setminus K_1 = \{k \in \mathbb{N}_0 : |k - nx| \geq n\delta\} \end{aligned} \quad (9.9)$$

9. Satz von Weierstrass

Dann gilt:

$$\begin{aligned} \|f - b_n\| &= \left\{ \left| \sum_{k=0}^n \left[f(x) r_k(x) - f\left(\frac{k}{n}\right) r_k(x) \right] \right| \right\}_{max} \leq \left\{ \sum_{k=0}^n r_k(x) \left| f(x) - f\left(\frac{k}{n}\right) \right| \right\}_{max} = \\ &= \underbrace{\sum_{k \in K_1} r_k(x) \left| f(x) - f\left(\frac{k}{n}\right) \right|}_{S_1} + \underbrace{\sum_{k \in K_2} r_k(x) \left| f(x) - f\left(\frac{k}{n}\right) \right|}_{S_2} \end{aligned}$$



Die Summen S_1 und S_2 lassen sich für jedes $x \in [0, 1]$ erstellen! Wir zeigen, dass diese Summen für jedes x sich unter eine vorgegebene ε -Schranke drücken lassen, wenn nur n groß genug gewählt wird!

$$S_1 < \sum_{k \in K_1} r_k(x) \frac{\varepsilon}{2} = \frac{\varepsilon}{2} \sum_{k \in K_1} r_k(x) \leq \frac{\varepsilon}{2} \Rightarrow S_1 < \frac{\varepsilon}{2} \quad (9.10)$$

Mit 9.8 lässt sich folgende Abschätzung durchführen:

$$\begin{aligned} nx(1-x) &= \sum_{k=0}^n (k-nx)^2 r_k(x) \geq \sum_{k \in K_2} (k-nx)^2 r_k(x) \underset{9.9}{\geq} (n\delta)^2 \sum_{k \in K_2} r_k(x) \Rightarrow \\ \sum_{k \in K_2} r_k(x) &\leq \frac{x(1-x)}{n\delta^2} \leq \frac{1}{4n\delta^2} \end{aligned} \quad (9.11)$$

Die letzte Ungleichung gilt, da das Maximum von $x(1-x)$ in $[0, 1]$ $\frac{1}{4}$ ist!

Wegen der Kompaktheit von $[0, 1]$ existiert $M \in \mathbb{R}^+$ mit $\|f\| = M$ damit gilt:

$$\left| f(x) - f\left(\frac{k}{n}\right) \right| \leq 2M$$

S_2 lässt sich damit abschätzen:

$$S_2 \leq 2M \sum_{k \in K_2} r_k(x) \underset{9.11}{\leq} \frac{M}{2n\delta^2}$$

für alle $n > \frac{M}{\varepsilon\delta^2}$ wird letzte Gleichung zu

$$S_2 < \frac{\varepsilon}{2} \quad (9.12)$$

□

Wir bilden die Bernsteinpolynome in *Geogebra* nach, wobei wir als zu approximierende Funktion die “komprimierte” Sinusfunktion $cs(x)$ hernehmen:

$$cs(x) = \sin\left(\frac{\pi}{2}x\right)$$

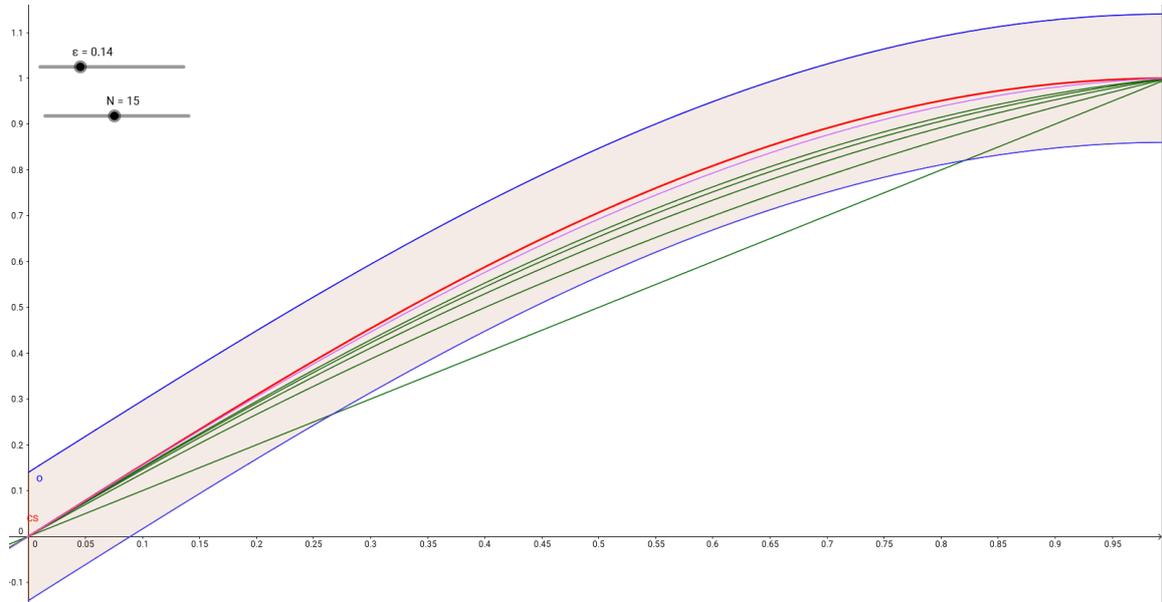


Abb.65 : Approximation des Sinus(rot) durch Bernsteinpolynome

Der Graph zeigt den ϵ -Schlauch(einstellbar mit dem Schieberegler) um die “komprimierte” Sinusfunktion $cs(x)$, die ersten 5 Bernsteinpolynome und das 15.-te Bernsteinpolynom.

Nr.	Name	Beschreibung	Wert
1	Funktion cs		$cs(x) = \text{Wenn}[0 \leq x \leq 1, \sin(\pi / 2 x)]$
2	Zahl ϵ		$\epsilon = 0.14$
3	Funktion o	$o(x) = cs(x) + \epsilon$	$o(x) = \text{Wenn}[0 \leq x \leq 1, \sin(\pi / 2 x)] + 0.14$
4	Funktion u	$u(x) = cs(x) - \epsilon$	$u(x) = \text{Wenn}[0 \leq x \leq 1, \sin(\pi / 2 x)] - 0.14$
5	Zahl a	IntegralZwischen[o, u, 0, 1]	$a = 0.28$
6	Liste Liste1	Folge[Summe[Folge[cs(k / n) n! / (k! (n - k)!) x^k (1 - x)^(n - k), k, 0, n]], n, 1, 5]	Liste1 = {Wenn[0 ≤ 0 ≤ 1, sin(π / 2 (0))] 1! / (0! (1)!) x^0 (1 - x)^(1) + ...
7	Zahl N		$N = 15$
8	Funktion f	Summe[Folge[cs(k / N) N! / (k! (N - k)!) x^k (1 - x)^(N - k), k, 0, N]]	$f(x) = \text{Wenn}[0 \leq 0 / 15 \leq 1, \sin(\pi / 2 (0 / 15))] 15! / (0! (15)!) x^0 (1 - ...$

Abb.66 : Konstruktionsprotokoll für obigen Graphen

Mit dem Schieberegler N lässt sich das N -te Bernsteinpolynom zeichnen. Man sieht die gleichmäßige Konvergenz.

9. Satz von Weierstrass

Zur Erklärung:

$$\sum_{k=0}^N cs \left(\frac{k}{N} \right) r_k(x)$$

entspricht in *Geogebra*

`Summe[Folge[cs(k / N) N! / (k! (N - k)!) x^k (1 - x)^(N - k), k, 0, N]]`

Bei $N \approx 130$ ist auf dem Bildschirm (falls man nicht hineinzoomt) die Originalfunktion $cs(x)$ nicht mehr vom Bernsteinpolynom zu unterscheiden. Wie "vorsichtig" im oberen Beweis unsere Abschätzung ist, lässt sich an diesem Beispiel demonstrieren:

Da die maximale Steigung in $[0, 1]$ $\pi/2$ beträgt gilt:

$$\frac{cs(x_2) - cs(x_1)}{x_2 - x_1} \leq \frac{\pi}{2} \Rightarrow cs(x_2) - cs(x_1) \leq \frac{\pi}{2} (x_2 - x_1)$$

Für die Summen S_1 und S_2 ergaben unsere Abschätzungen ($M = 1$ bei $cs(x)$):

$$\underbrace{\frac{M}{2n\delta^2}}_{\frac{\varepsilon}{2}} + \underbrace{\frac{\pi}{2}\delta}_{\frac{\varepsilon}{2}} = \varepsilon$$

Bei $\varepsilon = 0.2$ ergibt sich aus dem zweiten Summanden $\delta \approx 0.06$, damit liefert der erste Summand einen Wert für $n \approx 1400$ - der weit überzogen ist (wie wir aus der Zeichnung sehen). Wichtig dabei ist nur, dass wir den zweiten Summand mit δ beliebig klein machen können, und obwohl δ im ersten Summand quadratisch im Nenner steht, können wir mit einem "großen" n auch diesen beliebig klein machen!

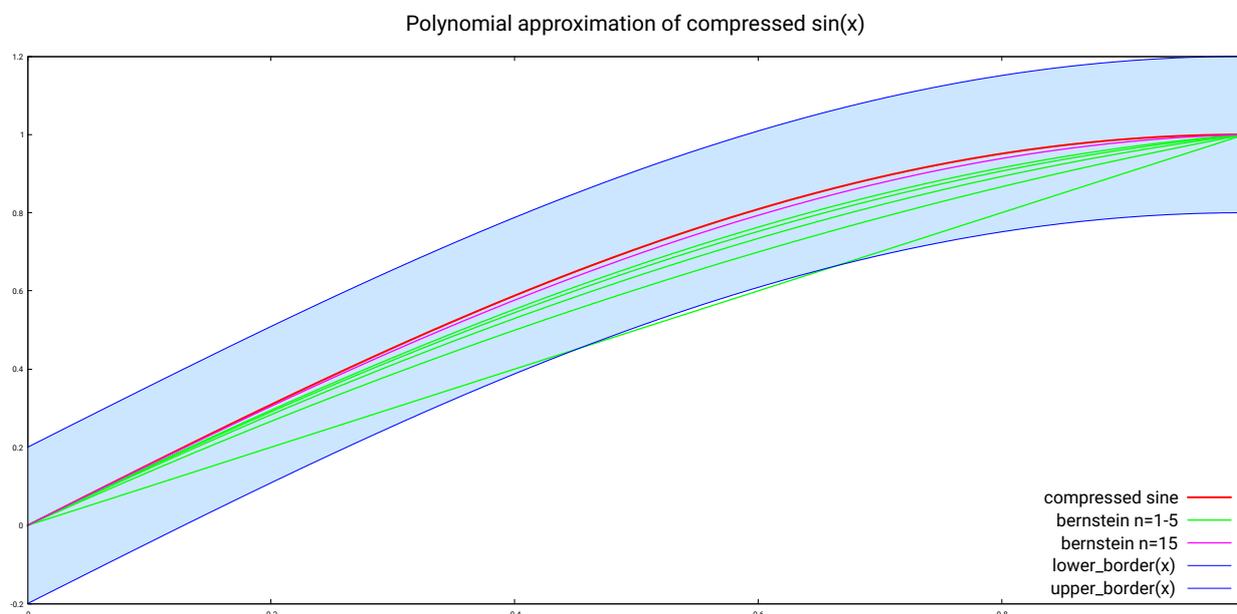


Abb.67 : Hier die Konstruktion mit *gnuplot*

```

# write out the tables for the first 5 Bernstein-polynomials
2 set key font ",20"
  set xrange [0:1]
4
set sample 250
6 b(n,k,x)=sin(pi/2*k/n)*gamma(n+1)/(gamma(k+1)*gamma(n-k+1))*x**k*(1-x)**(n-k)
  set table "bernstein-data.gnu"
8   plot for [n=1:5] (sum[k=0:n] b(n,k,x))
  unset table
10
  set table "bernstein-dataN15.gnu"
12   plot (sum[k=0:15] b(15,k,x))
  unset table
14
set key bottom right
16 set style data lines
  set xtics 0.2
18 set title font ",25"
  set title "Polynomial approximation of compressed sin(x)"
20 eps=0.2
  cs(x)=sin(pi/2*x)
22 lower_border(x) = cs(x)-eps
  upper_border(x) = cs(x)+eps
24 ## Last datafile plotted: "+"
  plot '+' using 1:(cs(1):(lower_order(1))) with filledcurve notitle lt 2 lc rgb "#
    cce6ff", \
26   '+' using 1:(cs(1):(upper_order(1))) with filledcurve notitle lt 2 lc rgb "#
    cce6ff", \
    cs(x) with lines title "compressed sine" lw 3 lc rgb "red", \
28   "bernstein-data.gnu" title "bernstein n=1-5" lw 2 lc rgb "green", \
    "bernstein-dataN15.gnu" title "bernstein n=15" lw 2 lc rgb "magenta", \
30   lower_border(x) with lines lw 1 lc rgb "blue", \
    upper_border(x) with lines lw 1 lc rgb "blue"

```

Die Veranschaulichung lässt sich natürlich auch mit *gnuplot* darstellen, das ja auch *wxMaxima* verwendet - allerdings in abgespeckter Form, sodass nicht alle Möglichkeiten zur Verfügung stehen. Darunter ist das zugehörige "Programm" abgebildet - die meisten Befehle sind selbst-erklärend.

Hier das Wichtigste:

- 5 → es wird die Sampling-Rate für die Datentabellen **bernstein*** festgelegt
- 6 → Der Bernsteinpolynom-Term für sin wird mit der Gamma-Fkt. festgelegt:

$$B_{k,n}(x) = \sin\left(\frac{\pi k}{2n}\right) \binom{n}{k} x^k (1-x)^{n-k} \quad \text{mit} \quad \binom{n}{k} := \frac{\Gamma(n+1)}{\Gamma(k+1)\Gamma(n-k+1)}$$

- 7 – 13 → Die Sample-Datentabellen für die ersten 5 bzw. das 15.-te Bernsteinpolynom werden auf die Platte geschrieben

9. Satz von Weierstrass

21 – 23 → Die Funktionen werden festgelegt (Sinus mit ε -Schlauch)

25 – 31 → Plotten der einzelnen Funktionen; “+” ist ein Pseudofile

Jetzt schätzen wir noch die sup-Norm für das 15.-tes Näherungspolynom mit Bernsteinpolynomen ab:

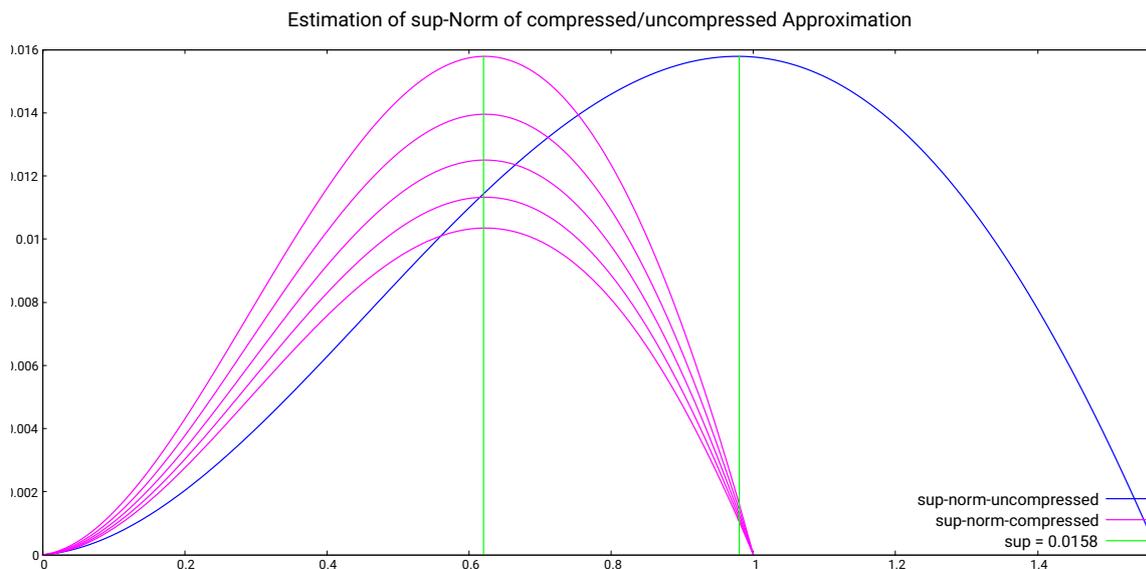


Abb.68 : Abschätzung der sup-Norm für 15.-tes Näherungspolynom mit *gnuplot*

$$\text{Es gilt offensichtlich: } \|cs - \bar{b}_{15}\| = \|\sin - b_{15}\|$$

also die Normen der komprimierten und unkomprimierten Funktionen stimmen überein. Um die gleichmäßige Konvergenz zu betonen, wurde noch \bar{b}_{17} , \bar{b}_{19} , \bar{b}_{21} , \bar{b}_{23} eingezeichnet - man kann das stetige Abnehmen der sup-Norm erkennen!

Auf der nächsten Seite noch der *gnuplot*-Code für obigen Graph.

In diesem Listing ist

$$c(n, k, x) = b(n, k, x) \circ g \quad \text{mit} \quad g(x) := \frac{x - a}{b - a}$$

die erweiterte Bernsteinpolynom-Näherung wie vorher besprochen, hier natürlich mit $a = 0$ und $b = \pi/2$

```

#plot the sup-norm of (un)compressed approximations
2 set key font ",20"
  set tics font ",16"
4 #set sample 200 — not necessary -> it's default
  b(n,k,x)=sin(pi/2*k/n)*gamma(n+1)/(gamma(k+1)*gamma(n-k+1))*x**k*(1-x)**(n-k)
6 c(n,k,x)=sin(pi/2*k/n)*gamma(n+1)/(gamma(k+1)*gamma(n-k+1))*(2*x/pi)**k*(1-(2*x/pi)
  )**(n-k)
  cs(x)=sin(pi/2*x)
8
# the next rows have to be uncommented on the first run to produce data for
  plotting
10
#set xrange [0:1]
12 #set table "sin-bern-norm-data.gnu"
  # plot for [n=0:4] abs( (sum[k=0:15+2*n] b(15+2*n,k,x)) - cs(x))
14 #unset table

16 set xrange [0:pi/2]
  #set table "sin-bern-norm-data1.gnu"
18 # plot abs( (sum[k=0:15] c(15,k,x)) - sin(x))
  #unset table
20

  set key bottom right
22 set style data lines
  set xtics 0.2
24 set title font ",25"
  set title "Estimation of sup-Norm of compressed/uncompressed Approximation"
26 ### Last datafile plotted: "+"
  plot "sin-bern-norm-data1.gnu" title "sup-norm-uncompressed" lw 2 lc rgb "blue",\
28     "sin-bern-norm-data.gnu" title "sup-norm-compressed" lw 2 lc rgb "magenta",\
     "sin-bern-norm-data2.gnu" title "sup = 0.0158" lw 2 lc rgb "green"

```


10 | Räuber-Beute Modell

Wenn hier von Räuber-Beute-Modell die Rede ist, ist das Modell nach Lotka-Volterra gemeint:

$$\begin{aligned}\dot{p} &= a p(t) - b p(t) f(t) \\ \dot{f} &= c p(t) f(t) - d f(t) \quad a, b, c, d \in \mathbb{R}^+\end{aligned}\tag{10.1}$$

Inspiziert wurde ich zu diesem Kapitel von einem Artikel in Computermagazin *c't 2019, Heft 15*. Die Beutetiere sind dort Pinguine(p), die Raubtiere Füchse(f). a, b, c, d sind Sterbe- bzw. Zuwachsraten. Ich habe diese Nomenklatur (trotz biologischen Unsinns) beibehalten. Rein formal handelt es sich bei 10.1 um ein nichtlineares gewöhnliches Differentialgleichungssystem - geschlossene Lösungsformeln für $p(t)$ und $f(t)$ für Anfangswerte p_0 und f_0 darf man da nicht erwarten.

10.1 “Brute force” Attacke

In *GNU-Octave* wird ein gewöhnliches Differentialgleichungssystem (ordinary differential equation - system ODE) numerisch mit dem Solver

`lsode(fh, x0, t)` → `y` gelöst.

Dafür ist das ODE-System auf folgende Form zu bringen

$$\frac{dx_i}{dt} = f_i(x_1, x_2, \dots, x_n, t) \quad i \in \{1, 2, \dots, n\}\tag{10.2}$$

Bedeutung der Parameter:

- fh ist ein Funktionsbezeichner (function handle) für die $f_i(x_1, x_2, \dots, x_n, t) = f_i(\vec{x}, t)$
- x_0 ist der Anfangswertvektor für die x_1, x_2, \dots, x_n
- t ist der Vektor, an dem die Werte der f_i berechnet werden (Stützstellen)
- y die Lösung ist eine Matrix, wobei die i -te Zeile $x_i(t_i)$ angibt

Ohne etwas über das Lösen von ODE(-Systemen) zu wissen (*lsode* ist einfach eine black box, die wir füttern), können wir uns anschauen wie sich Räuber und Beute für gewisse Parameter a, b, c, d entwickeln:

10. Räuber-Beute Modell

```
# Parameter of model equations
2
a=1; # increase of penguins
4 b=0.2; # penguins death rate caused by foxes
c=0.04; # increase of foxes
6 d=0.8; # death rate of foxes

8 # initial values for x_1 und x_2
x0=[50;10];

10
# xdot is an arbitrary name, nbut why not?
12 function xdot=setf(a,b,c,d,x,t)
    xdot=zeros(2,1); # init matrix
14    xdot(1)=a*x(1)-b*x(1)*x(2); #die Gleichungen
    xdot(2)=c*x(1)*x(2)-d*x(2);
16 endfunction

18 # provides the function-handle
function fh=getFH(a,b,c,d,func)
20    # fh is the handle for a function of vector x and scalar t
    fh=@(x,t) func(a,b,c,d,x,t);
22 endfunction

24
funcHandle=getFH(a,b,c,d,@setf);
26
t=linspace(0,30,1000);
28 y=lsode(funcHandle, x0, t);
plot(y);
```

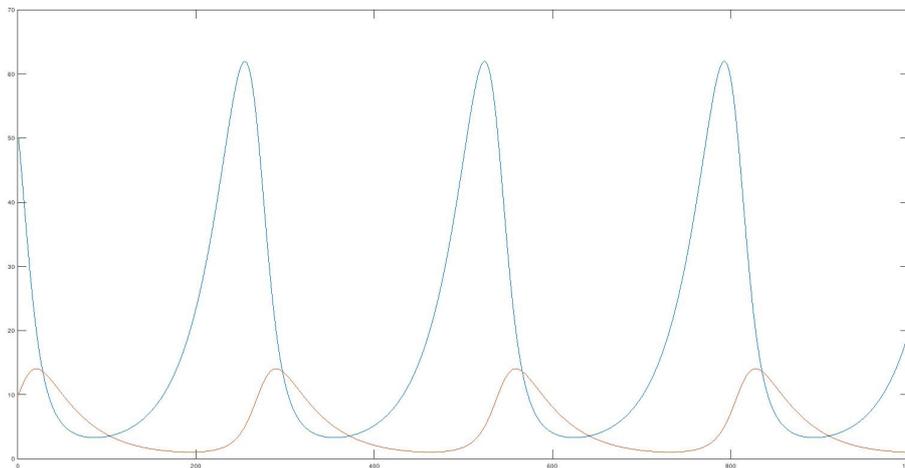


Abb.69 : Räuber-Beute Modell nach Lotka-Volterra

Es fällt auf, dass beide Populationen mit derselben Periode schwanken - allerdings phasenverschoben. Außerdem gibt es kein Aussterben - auch wenn eine Population auf 1 (oder weniger?)

Individuum geschrumpft ist - auch 0.4 "Individuen" können sich langsam wieder vermehren. Dies ist natürlich der "Kontinuität" des Modells zu verdanken.

In *wxMaxima* geht es auch ganz kurz mit dem Runge-Kutta Verfahren (*rk*) - es benötigt eine Liste der Änderungsterme (rechte Seite unseres ODE-Systems), Liste der Variablen, Anfangswertvektor und Stützstellen - hier allerdings nicht Schrittzahl (*linspace*: 1000) sondern Schrittweite 0.03.

rk liefert eine Liste mit den Tripeln $[t_i, x_i, y_i]$, daraus extrahieren wir die Pinguin-Population zum Zeitpunkt t_i : $[t_i, x_i]$ und die der Füchse $[t_i, y_i]$ mit *map*. Beachte auch den Unterschied in der *x*-Achsen-Skalierung!

```
(%i1) (a:1,b:0.2,c:0.04,d:0.8)$
(%i2) points:rk([a*x-b*x*y,c*x*y-d*y],[x,y],[50,10],[t,0,30,0.03])$
(%i3) penguins:map(lambda([x],[first(x),second(x)]),points)$
(%i4) foxes:map(lambda([x],[first(x),third(x)]),points)$
(%i5) plot2d([ [discrete, penguins], [discrete, foxes] ],
              [legend, "Pinguine", "Füchse"])$
```

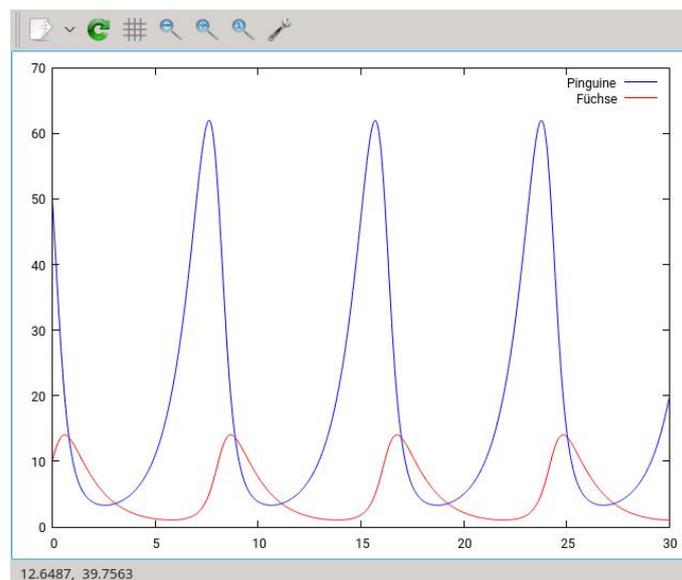


Abb.70 : *wxMaxima* mit *gnuplot* im Einsatz

In *Geogebra* heißt das numerische Verfahren für gewöhnliche Differentialgleichungen

$$NSolveODE(\{d_1, d_2, \dots, d_n\}, t_0, \{y_1, y_2, \dots, y_n\}, t_f)$$

- dabei ist $\{d_1, d_2, \dots, d_n\}$ die Liste der Differentialgleichungen nach 10.1
- t_0 bzw. t_f der Anfangs- bzw. Endzeitpunkt der Berechnung
- $\{y_1, y_2, \dots, y_n\}$ sind die Werte der Funktionen bei t_0

10. Räuber-Beute Modell

Construction Protocol	
No.	Value
1	$a = 1$
2	$b = 0.2$
3	$c = 0.04$
4	$d = 0.8$
5	$p_0 = 50$
6	$f_0 = 10$
7	$t_0 = 0$
8	$t_1 = 30$
9	$p'(t, p, f) = p - 0.2p f$
10	$f'(t, p, f) = 0.04p f - 0.8f$
11	$\text{penguins} = \text{NSolveODE}(\{p', f\}, t_0, \{p_0, f_0\}, t_f)$
11	$\text{foxes} = \text{NSolveODE}(\{p', f\}, t_0, \{p_0, f_0\}, t_f)$

Abb.71 : Protokoll in *Geogebra*

Zeile 1 bis 8 die üblichen Parameter

Zeile 9/10 - die Differentialgleichungen in dieser expliziten Form (Änderungsrate links vom Gleichheitszeichen!)

Zeile 11 wird nur ein einziges Mal eingegeben (obwohl sie im Protokoll doppelt vorhanden ist) und liefert die Lösung als Ortskurve (engl. Locus). Die Namen (penguins/foxes) wurden erst wurden erst später geändert!

Wie wir daraus einzelne Punkte extrahieren können, erklären wir in einem späteren Abschnitt.

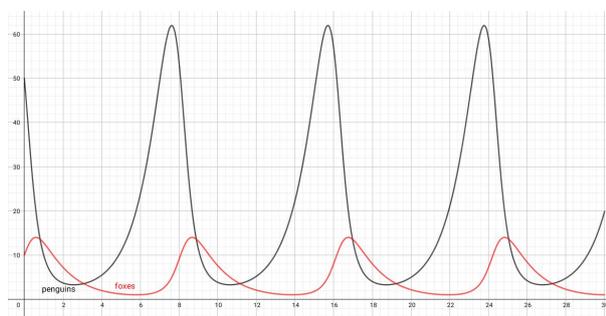


Abb.72 : Lösungen als Ortslinie (mit *NSolveODE* erzeugt)

10.2 Lösung mit Differenzengleichungen

Wir nähern die Differentiale durch Differenzen an:

$$\dot{p}(t) \approx \frac{p_1 - p_0}{(\Delta t)_0} \quad p_i := p(t_i), \quad t \in [t_0, t_1], \quad (\Delta t)_0 = t_1 - t_0 \rightarrow 0$$

Setzen wir jetzt nach 10.1 (Zeitintervall wird in n Teilintervalle zerlegt: $t_0, t_1, t_2, \dots, t_n$):

$$\frac{p_1 - p_0}{(\Delta t)_0} \approx \dot{p}(t) \approx p(t_0) = a p(t_0) - b p(t_0) f(t_0) = a p_0 - b p_0 f_0$$

ergibt sich

$$p_1 \approx p_0 + (\Delta t)_0 [a p_0 - b p_0 f_0] = F_p(p_0, f_0, (\Delta t)_0)$$

Eine analoge Näherung kann man leicht für f_1 aus den Startwerten (p_0, f_0) herleiten!

$$p_2 = F_p(p_1, f_1, (\Delta t)_1) \quad \text{bzw.} \quad f_2 = F_f(p_1, f_1, (\Delta t)_1)$$

Die Lösungen von 10.1 lassen sich näherungsweise rekursiv berechnen

$$\begin{aligned} p_{i+1} &= p_i + (\Delta t)_i \cdot [a \cdot p_i - b \cdot p_i \cdot f_i] && \text{Startwerte } (p_0, f_0) \\ f_{i+1} &= f_i + (\Delta t)_i \cdot [c \cdot p_i \cdot f_i - d \cdot f_i] && i \in \{0, 1, \dots, (n-1)\} \end{aligned} \quad (10.3)$$

Hier das das *GNU-Octave* Programm dazu

```

1 # Here the parameters you can experiment with
3 a=1; # increase of penguins
  b=0.2; # penguins death rate caused by foxes
5 c=0.04; # increase of foxes
  d=0.8; # death rate of foxes
7
  #initial values for penguins and foxes
9 p=50; f=10;
  #decomposition of simulation interval [0,30]
11 I=0:.005:30;
13 function results=solve_LV(a,b,c,d,start ,steps)
  num=length(steps);
15 #matrix initialization: 2 rows, (number of steps) columns
  results=zeros(2,num);
17 # first column is initialized with starting values
  results(:,1)=start;
19   for i=2:num
     Delta_t=steps(i)-steps(i-1);
21     x=results(1,i-1);
     y=results(2,i-1);
23     pfnew=[a*x-b*x*y;c*x*y-d*y]; # column-vector
     results(:,i)=results(:,i-1)+Delta_t*pfnew;
25   endfor
  endfunction
27
  #invoking the solver: first row penguins, second row foxes
29 populations= solve_LV(a,b,c,d,[p;f],I);
  rangeY=[0,70]; #range for vertical axis
31 #syntax: plot(xVector,yVector, ....)
  hold on;
33   plot(I,populations(1,:), 'b', 'linewidth',2.5)
     plot(I,populations(2,:), 'r', 'linewidth',2.5)
35   legend('Penguins','Foxes')
     set(gca, 'linewidth',3, 'fontsize',15, 'ylim',rangeY)
37   set(legend, 'fontsize',15)
  hold off

```

Der Algorithmus von 10.3 wird mit der Funktion *solve_LV* in Zeile 13 umgesetzt - allerdings iterativ statt rekursiv. Rückgabewert ist eine Matrix (*results*)

An sich ist das Programm selbsterklärend. In Zeile 21/22 werden jeweils die Pinguin- und Fuchspopulationen aus der *results*-Matrix geholt und auf *x* bzw. *y* gelegt (wäre an sich nicht notwendig), als Bonus ergibt sich in Zeile 23 die **Änderung** des Pinguin-Fuchs-Vektors *pfnew* in diesem Zeitintervall als Kopie unseres Gleichungssystems 10.1.

Die neuen Populationen werden dann wieder in der *results*-Matrix in Zeile 24 abgespeichert. Wobei die Besonderheit der *GNU-Octave*-Sprache wiederholt verwendet wird:

Der Doppelpunkt steht "gilt für alle" – also *results(:,i)*
alle Zeilen der *i*-ten Spalte!

10.3 Stationäre Lösung und Richtungsfeld

Wenn \dot{p} und \dot{f} gleichzeitig verschwinden in 10.1, ändern sich die Populationen nicht mehr!

$$\begin{array}{l} 0 = a \cdot p - b \cdot p \cdot f \\ 0 = -d \cdot f + c \cdot p \cdot f \end{array} \left. \begin{array}{l} \cdot c \\ \cdot b \end{array} \right\} + \Rightarrow a \cdot c \cdot p = b \cdot d \cdot f \Rightarrow p = \frac{bd}{ac} f$$

Substitution in die obere Gleichung ergibt

$$0 = \frac{bd}{c} f - \frac{bd}{a} f \cdot f$$

Dies führt zum trivialen Lösungspaar $(p, f) = (0, 0)$ und zur stationären Lösung

$$(p_s, f_s) = \left(\frac{d}{c}, \frac{a}{b} \right) \stackrel{\text{hier}}{\stackrel{\perp}{=}} (20, 5)$$

Man kann sich leicht vergewissern, wenn man in obigen Programmen als Anfangsvektor $\left(\frac{d}{c}, \frac{a}{b} \right)$ vorgibt, bleiben beide Populationen konstant.

Jede Populationsentwicklung $\vec{x}(t) := (p(t), f(t))$ lässt sich auch als parametrisierte Kurve im (p, f) -Raum interpretieren. 10.1 gibt dann an, wie die Tangentenvektoren einer solchen Kurve auszuschauen haben.

$$\dot{\vec{x}} = \vec{F}(\vec{x}(t)) \tag{10.4}$$

Zeichnet man in jedem Punkt des (p, f) -Raumes (Phasenraum) den Tangentenvektor ein (Richtungsfeld) bekommt man einen guten Überblick wie die Kurve (Trajektorie = Entwicklung der Population) ausschauen wird - sie ist ja jene Kurve, die die Vektoren als Tangente besitzt. Man wählt also einen Startpunkt und folgt mit einem Stift einfach den Vektoren. Wobei wir durchaus die Vermutung hegen, dass die zyklischen Populationen "irgendwie" um die stationäre Lösung "kreisen" werden.

In *GNU-Octave* heißt der Befehl für das Zeichnen eines Richtungsfelds $quiver(\vec{x}, \vec{y}, F_x, F_y)$
Dabei ist

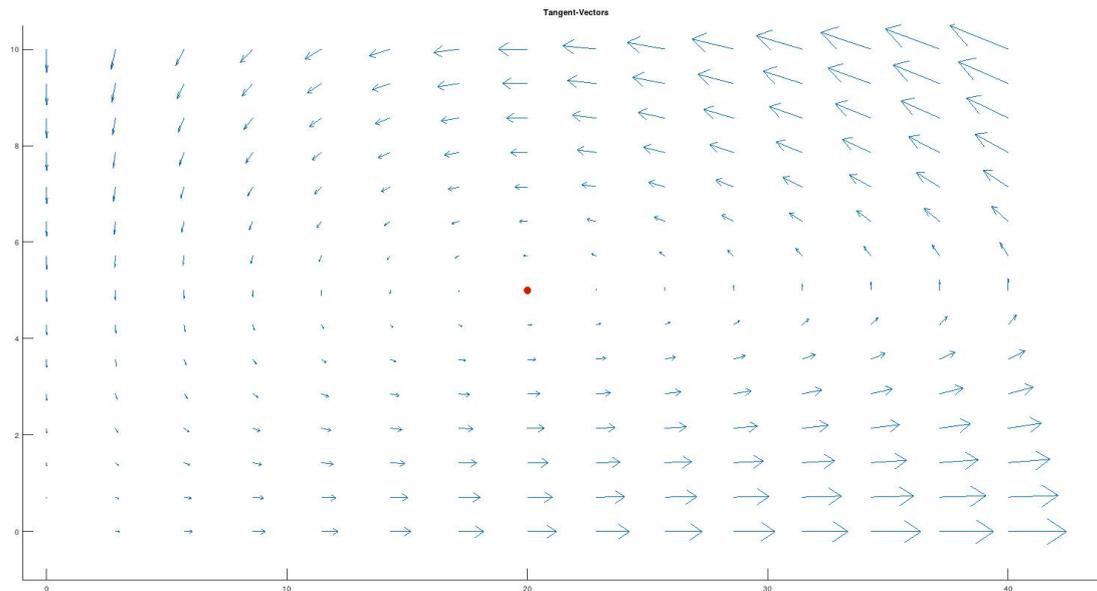
- \vec{x} (bzw. \vec{y}) ist der Stützstellenvektor in x (bzw. y)-Richtung
- F_x (bzw. F_y) sind die Terme von 10.4 (rechte Seite)

Als Output bekommt man an den Stellen $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$ die Tangentenvektoren gezeichnet. Wir haben im folgenden *GNU-Octave*-Skript auch deutlich die stationäre Lösung eingezeichnet:

```

a=1;
2 b=0.2;
c=0.04;
4 d=0.8;
   f_s=a/b; # stationary solution f_s = 5
6   p_s=d/c; # stationary solution p_s= 20
# linspace(lowerBound, upperBound, number of intersection-points)
8 rangeP = linspace (0, 2*p_s, 15);
rangeF = linspace (0, 2*f_s, 15);
10 # get a 2-dim grid for [x,y] -> here [p,f]
[p,f]=meshgrid(rangeP, rangeF);
12 # expression for change in x-direction
dp=a*p-b*p.*f;
14 # expression for change in y-direction
df=c*p.*f-d*f;
16 hold on ;
   axis ([-1 45 -1 10.5])
18 quiver(p,f,dp,df);
   plot(p_s,f_s, "o", "markersize", 10, "markerfacecolor", "red");
20 title ("Tangent-Vectors");
hold off;

```

Abb.73 : Richtungsfeld um die stationäre Lösung mit *quiver*

Deutlich kann man das Wirbelfeld um die stationäre Lösung erkennen (in der Physik spielen Wirbelfelder eine große Rolle, z.B. erzeugen zeitlich variable magnetische Wirbelfelder elektrische Wirbelfelder). Wir erkennen auch, dass die Vektoren nicht symmetrisch um die stationäre Lösung verteilt sind - welche Formen haben die Populationskurven?

10. Räuber-Beute Modell

In *wxMaxima* heißt der Befehl zur Darstellung eines Richtungsfelds (directionfield or slope-field) *plotdf*. Eingabeparameter sind: Liste der Ableitungsterme, Variablenlist, Trajektorie durch die Anfangswerte, Intervalle in denen die Variablen gezeichnet werden. Klickt man mit der Maus auf die Zeichnung wird dies als Ausgangspunkt einer Trajektorie gemacht (hier wurde in die Nähe der stationären Lösung geklickt). Das sieht dann bei uns so aus:

```
(%i1) (a:1,b:0.2,c:0.04,d:0.8)$  
(%i2) dpdt:a*p-b*p*f$  
(%i3) dfdt:c*p*f-d*f$  
(%i6) plotdf([dpdt,dfdt],[p,f],[trajectory_at,50,10],[p,-1,75],[f,-1,15])$
```

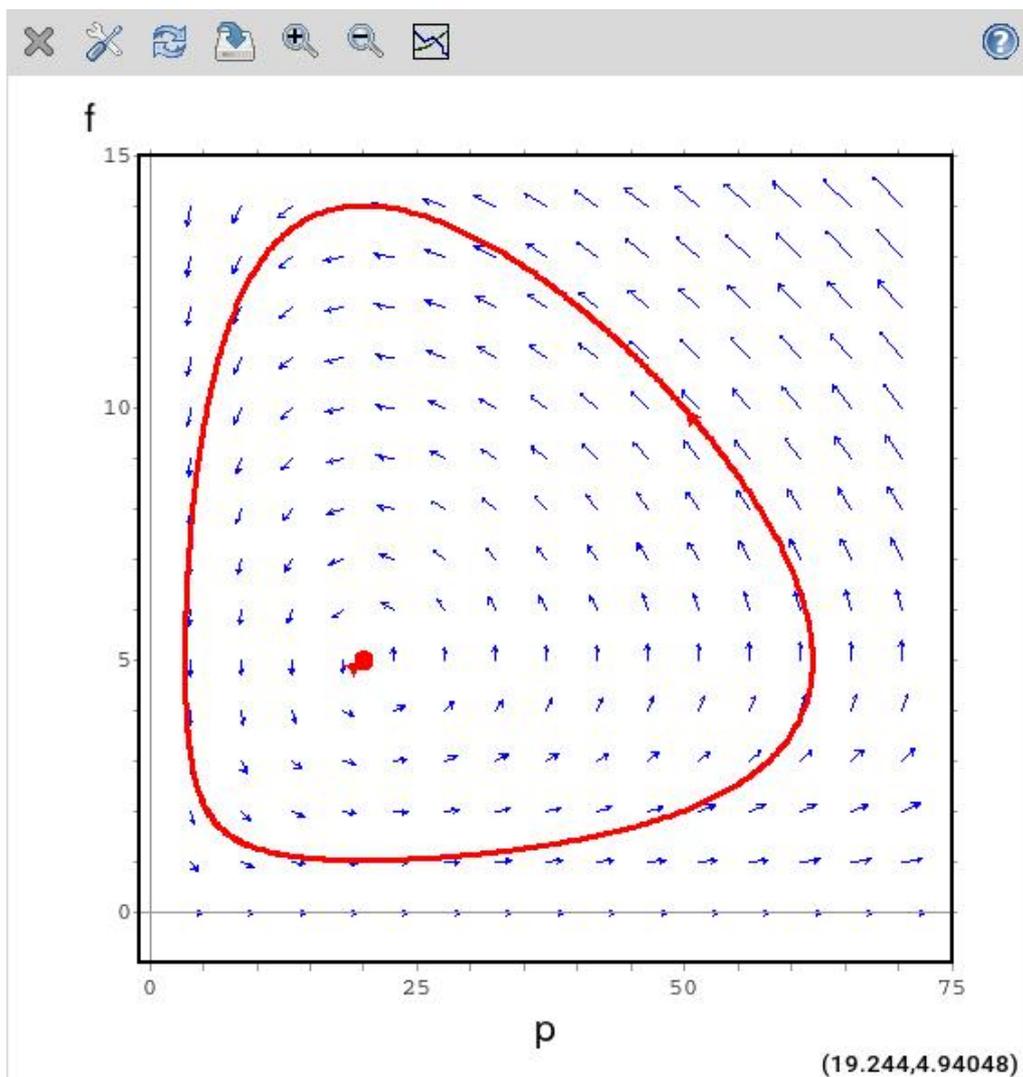


Abb.74 : Richtungsfeld und Trajektorie (50,10) in *wxMaxima* mit *plotdf*



Damit der Befehl *plotdf* funktioniert, muss *wxMaxima* installiert sein - *plotdf* kann man zwar von *wxMaxima* aufrufen, schickt dann aber die Ausgabe immer an *wxMaxima*.

10.3.1 Richtungsfeld in *Geogebra*

Befehl *Slopefield*

Bei der englischen Version von *Geogebra* (die ich benutze) lautet der eingebaute Befehl für das Zeichnen eines Richtungsfeldes

$$\text{SlopeField}(\langle f(x,y) \rangle, \langle \text{Number } n \rangle, \langle \text{Length Multiplier } a \rangle, \langle \text{Min } x \rangle, \langle \text{Min } y \rangle, \langle \text{Max } x \rangle, \langle \text{Max } y \rangle)$$

Folgende Angaben werden (für die größtmögliche Kontrolle) benötigt:

- die Differentialgleichung $\frac{dy}{dx} = f(x,y)$ - wir haben allerdings $\frac{dx}{dt}$ und $\frac{dy}{dt}$ also müssen wir umformen $\frac{dy}{dx} = \frac{dy}{dt} \frac{dt}{dx} = \frac{dy}{dt} \frac{1}{\frac{dx}{dt}}$
- das Richtungsfeld wird in einem $n \times n$ grid (Gitter) angegeben
- die Länge der Steigungslinien wird mit dem nächsten Parameter bestimmt
- der rechteckige Bereich (left bottom - right top) für das Richtungsfeld

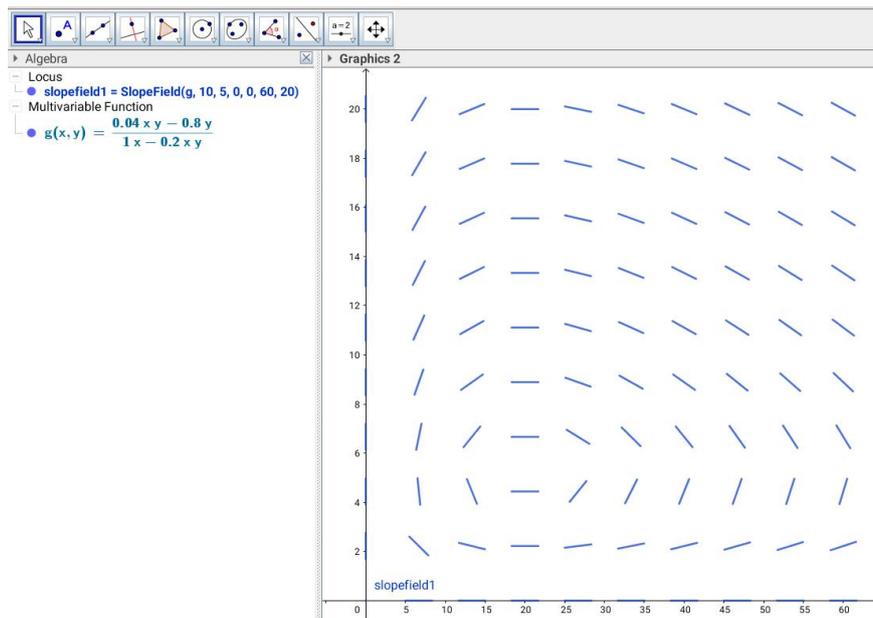


Abb.75 : Der *Slopefield* - Befehl und seine Ausgabe

Nachteile:

- ☞ keine Orientierung des Feldes
- ☞ alle Linien gleich lang (Stärke nicht abschätzbar → stationäre Lösung kaum sichtbar)
- ☞ umschreiben der Gleichung bzw. Gleichungssystems

10. Räuber-Beute Modell

Selbstgebastelt - um zu lernen

- Mit Corner(1) bzw. Corner(3) holen wir uns die Koordinaten der linken unteren Ecke (lb - left bottom) bzw. der rechten oberen Ecke (rt - right top)
- $xDensity$ - wieviele Punkte in x -Richtung wollen wir haben?
- Der $scaleFactor$ verkürzt/verlängert die Vektoren im Vektorfeld
- Δx ist die Abstand des Gitters in x -Richtung
- $xGrid$ sind die x -Koordinaten des Gitters
- 7-9 dasselbe für die y -Richtung; 10 erzeugt die Gitterpunkte
- 11-16 erzeugen die Änderungsraten für Füchse bzw. Pinguine
- $h1$ sind Tangentenvektoren im Phasenraum (Pinguin-Füchse) an den Gitterpunkten
- Wir setzen das Zeit-Inkrement (so klein als nötig!?) fest
- 19 - Wir versetzen die Gitterpunkte um $\Delta t \cdot h1 \rightarrow grid2$

No.	Name	Description
1	Point lb	Corner(1)
2	Point rt	Corner(3)
3	Number xDensity	
4	Number scaleFactor	
5	Number Δx	$(x(rt) - x(lb)) / xDensity$
6	List xGrid	$Sequence(x(lb) + i \Delta x, i, 0, xDensity)$
7	Number yDensity	
8	Number Δy	$(y(rt) - y(lb)) / yDensity$
9	List yGrid	$Sequence(y(lb) + i \Delta y, i, 0, yDensity)$
10	List grid	$Join(Zip(Zip(a, b), a, xGrid), b, yGrid)$
11	Number a	
12	Number b	
13	Number c	
14	Number d	
15	Multivariable Function P	$P(x, y) = a x - b x y$
16	Multivariable Function F	$F(x, y) = c x y - d y$
17	List h1	$Zip((P(x(i), y(0)), F(x(0), y(i))), i, grid)$
18	Number Δt	

Abb.76 : Hier das Konstruktionsprotokoll

- VF bzw. VFs sind jetzt das Vektorfeld bzw. skaliertes Vektorfeld die restlichen Befehle zeichnen den Zyklus im Phasenraum *Pinguine - Füchse*

19	List grid2	$Zip(aa + \Delta t bb, aa, grid, bb, h1)$
20	List VF	$Zip(Vector(aa, bb), aa, grid, bb, grid2)$
21	List VFs	$Zip(Vector(l1, l1 + scaleFactor l2), l1, grid, Vector(l2), VF)$
22	Number DrawLength	Length of Vector(lb, rt)
23	Multivariable Function f'	$f'(t, p, f) = F(p, f)$
24	Multivariable Function p'	$p'(t, p, f) = P(p, f)$
25	Point A	
26	Locus penguins	$NSolveODE((p', f'), 0, \{x(A), y(A)\}, 9)$
26	Locus foxes	$NSolveODE((p', f'), 0, \{x(A), y(A)\}, 9)$
27	Number t_0	
28	Number t_f	
29	Function f	Freehand function on interval [0, 10]
30	Function g	Freehand function on interval [0, 10]
31	Curve e	$Curve(g(t), f(t), t, t_0, t_f)$

Abb.77 : Fortsetzung

- 23/24 sind wieder unsere Differentialgleichungen

- Punkt A ist ein frei wählbarer Anfangspunkt (p_0, f_0) für Pinguine und Füchse
- Wir lösen das Differentialgleichungssystem numerisch und bekommen die Ortskurven für Pinguine und Füchse in Abhängigkeit von t in $[0, 9]$ - das ist leider nicht das was wir brauchen und darum müssen wir uns noch kümmern!
- Anfangszeitpunkt t_0 und Endzeitpunkt t_f werden gesetzt - das wäre vorher günstiger gewesen, aber was soll's!
- Jetzt "samplen" wir die Ortslinien für Pinguine und Füchse indem wir sie "scannen" - im Protokoll wird das mit "Freehand function" bzw. "freehand(x)" bezeichnet, was die Sache nicht ganz trifft, denn die Befehle lauten

```
g(x)=Function(Join({t_0, t_f}, y(First(penguins, Length(penguins))))))
f(x)=Function(Join({t_0, t_f}, y(First(foxes, Length(foxes))))))
```

$y(\text{First}(\text{penguins}, \text{Length}(\text{penguins}))) \rightarrow y$ -Werte der Pinguinpunkte

Function - benötigt t_0 und t_f und Werte - alles in 1 Liste (wird mit *Join* erledigt)

Curve(g(t), f(t), t, t_0, t_f) zeichnet die Kurve im Phasenraum!

Durch ziehen von A können die Anfangsbedingungen gewählt werden!

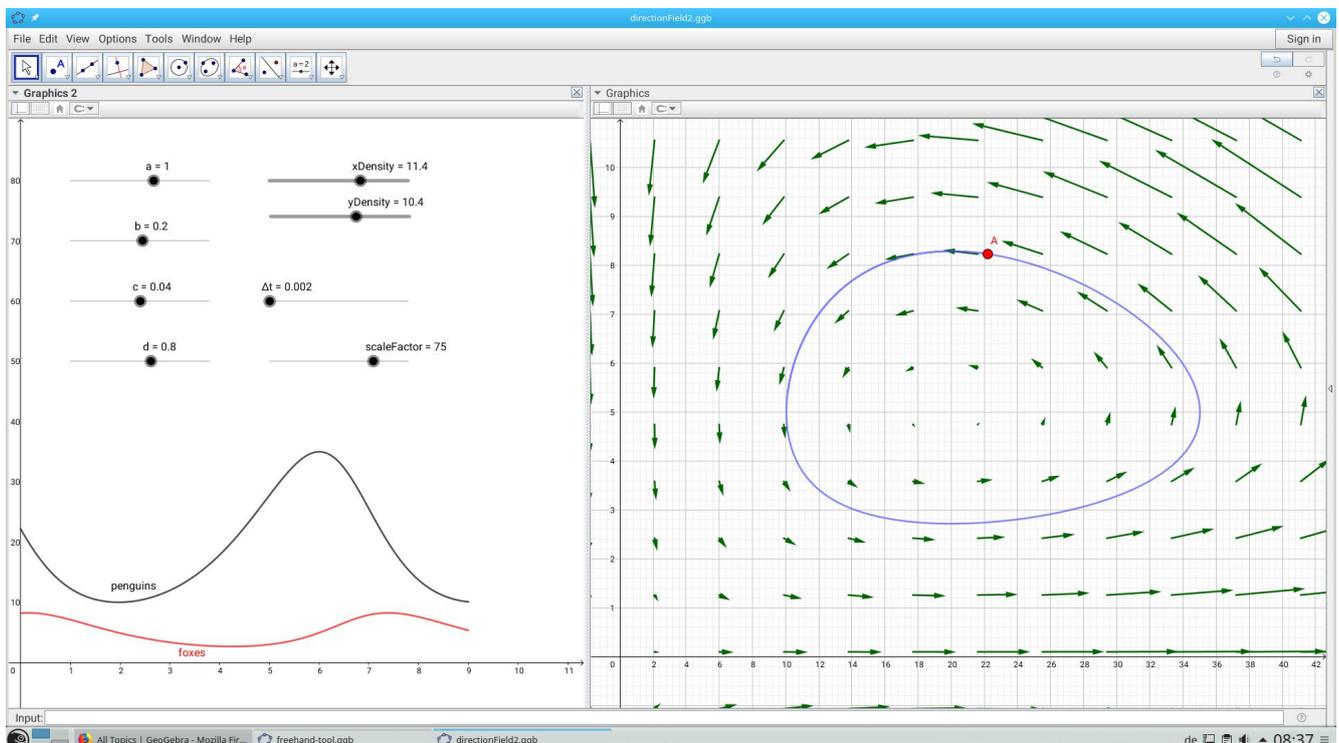


Abb.78 : Richtungsfeld und Trajektorie durch Punkt A in *Geogebra*

10. Räuber-Beute Modell

Wie man in Abb. 78 sieht, sind die Richtungsvektoren in der Nähe des stationären Punktes recht kurz, dies könnte man verbessern, indem man diese Vektoren (falls sie eine gewisse Länge unterschreiten) verlängert.

Sowohl dieser “Grenzwert”(slider *threshold*) als auch der Verlängerungsfaktor(slider *ampFactor*) sollen eingestellt werden können. Natürlich muss man diese Vektoren farblich hervorheben.

32	Number threshold	
33	Number ampFactor	
34	List grid3	<code>Zip(Point(l, Vector(k)), l, grid, Vector(k), VFs)</code>
35	List dist	<code>Zip(Distance(l, p), l, grid, p, grid3)</code>
36	List LIndex	<code>Sequence(If(dist(i) < threshold, (dist(i), i), (dist(i), 0)), i, 1, Length(dist))</code>
37	List shorts	<code>KeepIf(y(p) > 0, p, LIndex)</code>
38	List zoomed	<code>Zip(Vector(grid(y(l))), grid(y(l)) + ampFactor VFs(y(l))), l, shorts)</code>

Abb.79 : Zusätzliche Befehle für das “Zoomen”

- 34: *grid3* - man trägt von den Gitterpunkten(*grid*) die skalierten Vektoren(*VFs*) auf
- 35: Längenmessung zwischen *grid* und *grid3*
- 36: Wir konstruieren Punkte $P_i(d_i, i)$ - d_i ist die Länge vom i -ten Vektor $grid \rightarrow grid3$, falls kleiner als *threshold* - wird der Index als y-Koordinate gespeichert (Null sonst).
- 37: Welche Indices haben die “kurzen” Vektoren und diese werden bei 38 verlängert!

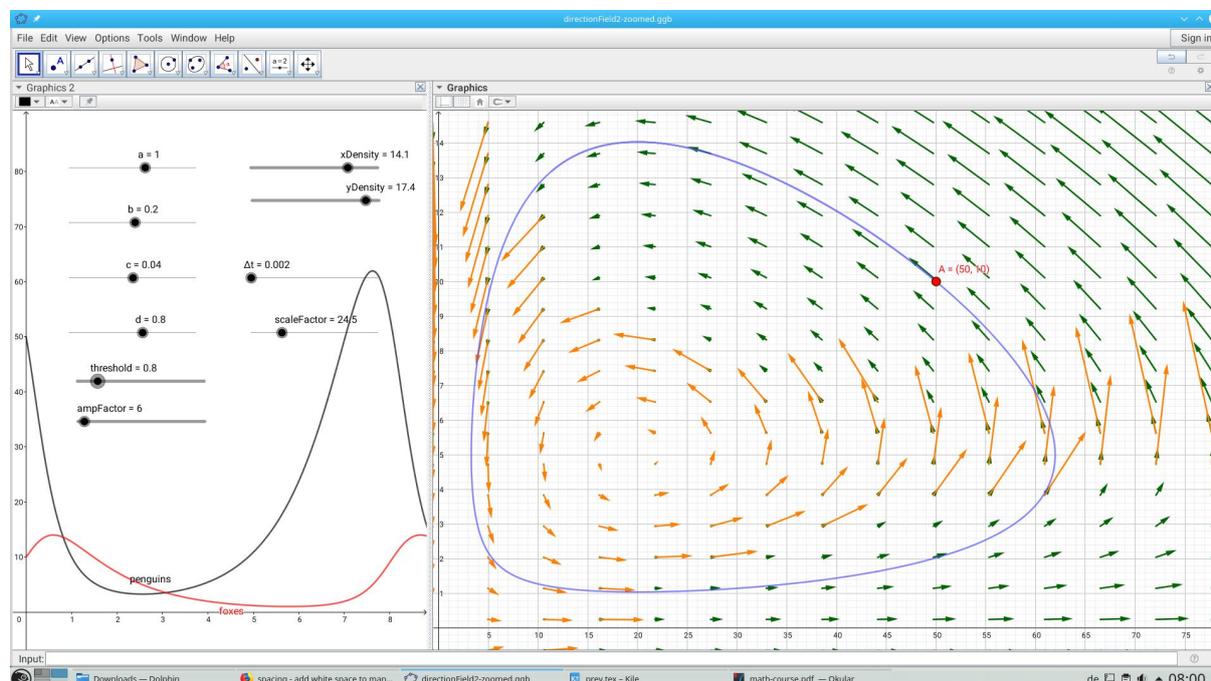


Abb.80 : Kurze Richtungsvektoren werden verlängert

10.3.2 Stützpunkte der Phasenraumkurve

Ein interessantes Zusatzproblem: Greife von der Lösungskurve im Phasenraum äquidistante Stützpunkte heraus (z.B. für eine Splinefunktion). (Anmerkung:günstiger wäre es natürlich bei starker Krümmung die Dichte der Stützpunkte zu erhöhen). Da man dabei viel über *Geogebra*-Skript lernen kann, möchte ich das hier zeigen.

Ein weiterer kritischer Punkt ist die Periodendauer festzustellen - dies wird normalerweise mit einer Fourieranalyse bzw. Autokorrelation einer Datenreihe (die bei uns hier *NSolveODE* liefert) bewerkstelligt. Weil dies sehr aufwendig ist, behelfen wir uns mit einem Slider für t_f (Ende des Lösungsintervalls) und vergrößern (ev. verkleinern) t_f solange, bis im Phasenraum ein Zyklus beendet ist - dies hängt natürlich von den Parametern ab!

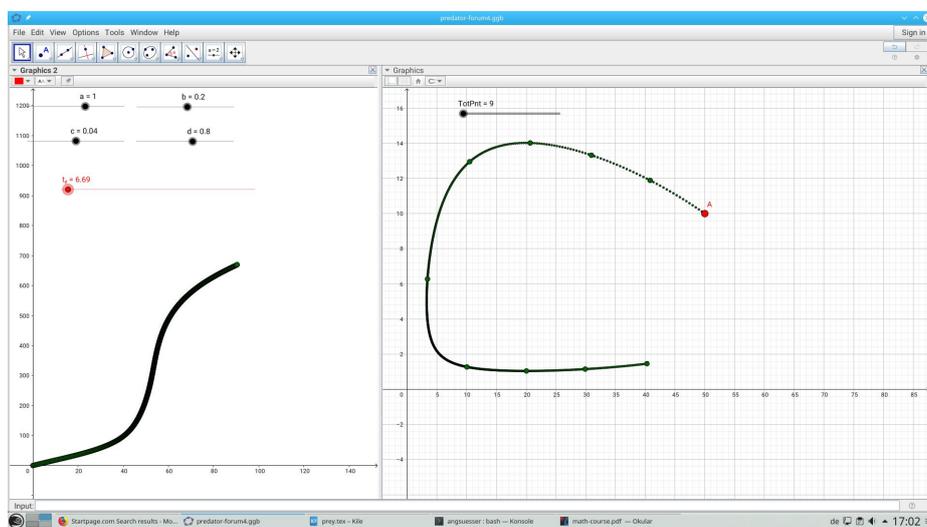


Abb.81 : t_f ist noch zu klein - Kurve im Phasenraum noch offen!

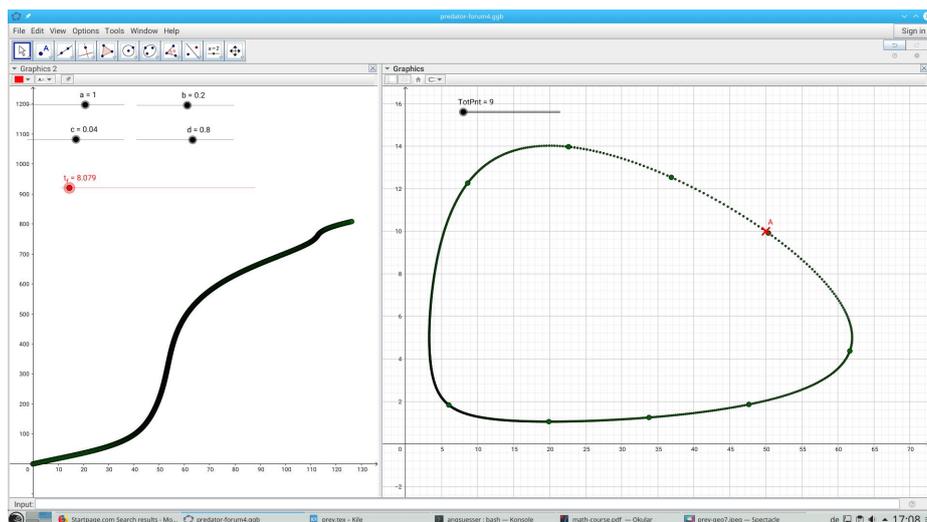


Abb.82 : Letzter Stützpunkt stimmt (fast) mit Anfangspunkt A überein

10. Räuber-Beute Modell

Um das Programm nicht noch mehr aufzublähen, lassen wir jetzt das Richtungsfeld weg. Nun zur Erklärung des Programms und wozu die seltsame Kurve in Graphics2 dient.

No.	Name	Description
1	Point A	
2	Number t_f	
3	Number t_p	
4	Number a	
5	Number b	
6	Number c	
7	Number d	
8	Number TotPnt	
9	Multivariable Function F	$F(x, y) = c \cdot x \cdot y - d \cdot y$
10	Multivariable Function f'	$f'(t, p, f) = F(p, f)$
11	Multivariable Function P	$P(x, y) = a \cdot x - b \cdot x \cdot y$
12	Multivariable Function p'	$p'(t, p, f) = P(p, f)$
13	Locus penguins	$NSolveODE(\{p', f\}, t_p, \{x(A), y(A)\}, t_p)$
13	Locus foxes	$NSolveODE(\{p', f\}, t_p, \{x(A), y(A)\}, t_p)$

Abb.83 : Lösung des ODE-Systems

Bis auf Zeile 8 (ein Slider zum Einstellen der Anzahl der Stützpunkte, die äquidistant vom Anfangspunkt entfernt sind) sind die Zeilen 1-13 sind lauter "alte Bekannte".
Hier die neuen Befehle:

14	List fL	$y(\text{First}(\text{foxes}, \text{Length}(\text{foxes})))$
15	List pL	$y(\text{First}(\text{penguins}, \text{Length}(\text{penguins})))$
16	List pf	$\text{Zip}(\{p, f\}, p, pL, f, fL)$
17	List distancesNeighbours	$\text{Sequence}(\text{Distance}(\text{pf}(n), \text{pf}(n - 1)), n, 2, \text{Length}(\text{pf}))$
18	List L2Index	$\text{IterationList}(\{x(A) + \text{distancesNeighbours}(y(A) + 1), y(A) + 1\}, A, \{(0, 0)\}, \text{Length}(\text{distancesNeighbours}))$
19	List accumDist	$x(\text{IterationList}(\{x(A) + \text{distancesNeighbours}(y(A) + 1), y(A) + 1\}, A, \{(0, 0)\}, \text{Length}(\text{distancesNeighbours})))$
20	Number L	$x(\text{Element}(\text{L2Index}, \text{Length}(\text{L2Index})))$
21	Number l_n	L / TotPnt
22	List supportPInd	$\text{Sequence}(\text{round}(y(\text{Intersect}(x = n \cdot l_n, \text{Polyline}(\text{L2Index}))))), n, 1, \text{TotPnt})$
23	List SupportPoints	$\text{Zip}(\text{pf}(), l, \text{supportPInd})$

Abb.84 : Konstruktion der äquidistanten Stützpunkte

- 14-15 Wir holen uns die Punkte von den Ortslinien - davon die y -Werte.
Es ist wichtig, dass der Slider t_f so eingestellt wird, dass genau(!) 1 Zyklus komplett ist - nicht mehr und nicht weniger!
- 16 Wir konstruieren die Punkte im Phasenraum (p, f) (penguins and foxes)
- diese liegen aber verschieden weit auseinander (zoomen!)
- 17 $d_i = |(p_i, f_i) - (p_{i-1}, f_{i-1})| \quad \forall i \in \{2, 3, \dots, N\}$ - Entfernung der Nachbarpunkte
- 18 Wir konstruieren eine Folge von Punkten: $(D_0, 0), (D_1, 1), (D_2, 2), \dots, (D_N, N)$
wobei $D_k = \sum_{i=1}^k d_i$ - also der Abstand des k -ten Punktes vom Anfangspunkt.
Es gilt natürlich die Rekursion: $D_0 = 0$ und $D_k = D_{k-1} + d_k$
Und obige Punkte (D_k, k) bilden die Kurve in Graphics2:
 x -Werte entsprechen dem Abstand vom Anfangspunkt (auf der Kurve gemessen!)
die y -Werte sind der Index des Punktes in der (p, f) -Liste.
Dass diese Kurve keine Gerade ist, zeigt die variierende Dichte der Punkte
Punkte mit gleichem Abstand sind in diesem monoton steigendem Graph Punkte, deren x -Werte sich mit gleichem Wert unterscheiden. Weitere Vorgangsweise:
- 20,21 Gesamtlänge L holen, $l_n = L/n$ n ist die Anzahl der Stützpunkte($TotPnt$)
- 22 Wir schneiden die Senkrechten $x = i \cdot l_n \quad i \in \{1, 2, \dots, n\}$ mit
 $Polyline$ von L2Index, nehmen den y -Wert und runden diesen auf eine ganze Zahl (nächster Index) und bekommen eine Liste der Stützpunkt-Indices
- 23 Diese Punkte holen wir uns aus dem (p, f) -Feld



In unserer Abb. 82 sind offensichtlich die 2 Stützpunkte, die sich am weitesten links im Zyklus befinden weiter auseinander als die anderen. Wenn Sie das Achsenverhältnis auf 1:1 schalten, klärt sich dieser Sachverhalt auf!

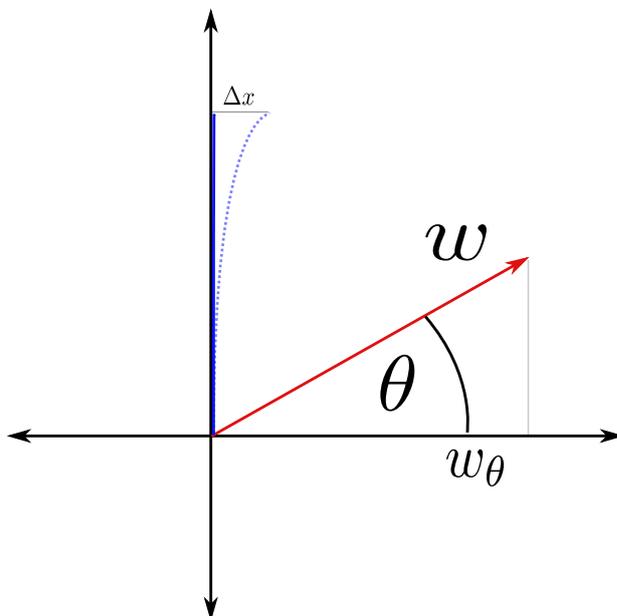
11 | Coriolis-Kraft

Haben Scharfschützen eine Ausrede

In meiner langjährigen Unterrichtspraxis behauptete einmal einer meiner Schüler - als ich im Physikunterricht auf die Corioliskraft zu sprechen kam, dass Scharfschützen sich um diese Kraft kümmern müssten, falls sie in Richtung Nord-Süd schießen. "Aus dem Bauch heraus könne ich dies kaum glauben", sagte ich, aber besser wäre natürlich eine physikalische Analyse dieser Situation - obwohl ich immer ein mulmiges Gefühl habe, wenn sich "die Physik" vor den "militärischen Karren" spannen lässt. Aber schließlich geht es ja nur um die Klärung einer Situation - so meine Entschuldigung.

Vergleichen wir die Corioliskraft gegenüber der Kraft, die ein Wind vom Betrag w ausübt und einen Winkel θ gegenüber der positiven x-Achse hat. Schussrichtung sei die positive y-Achse.

11.1 Winddrift



$$\text{Luftwiderstand: } F_{air} = \frac{1}{2} c_w \rho_{air} A v^2$$

Daten:

Schussweite: $2000 \text{ m} = W$

$c_w \approx 1,2$

Geschoß-Durchmesser: 9 mm

Geschoßlänge: 60 mm

Geschoßmasse: 10 g

$v_0 \approx 800 \text{ m/s}$

Winddrift $w_\theta = w \cos(\theta)$ (v in F_{air})

Driftzeit $t_d = \frac{W}{v_0} = 2,5 \text{ s}$

$\rho_{25} = 1,18 \text{ kg/m}^3$ (bei 25 Grad Celsius)

Abb.85 : Wind- und Schussrichtung

11. Coriolis-Kraft

Der Wind führt zu einer Beschleunigung in x-Richtung a_x :

$$a_x = \frac{F_{air}}{m} \quad \text{diese zu einer Abweichung } \Delta x = \frac{a_x}{2}(t_d)^2$$

Tippen wir das in einen Taschenrechner für $w = 18 \text{ km/h}$ und $\theta = 0$ ergibt sich ein

$$\Delta x \approx 1 \text{ m}$$

Das ist immerhin ganz schön daneben - wenn man glaubt, man braucht den Wind nicht zu berücksichtigen!

11.2 Projektilbahn auf rotierender Scheibe

In einem inertialen Koordinatensystem (KS) K befindet sich eine mit konstantem ω rotierende Scheibe. Rotationszentrum ist der Ursprung von K . Von einem Punkt $A(x_0|y_0)$ der Scheibe wird ein Geschöß mit der Geschwindigkeit v (gemessen im rotierenden KS!) abgefeuert. v legt die x -Achse von K fest. Welche Bahnkurve hat ein Geschöß in einem mit der Scheibe mitrotierendem KS K' ? (Achsen der beiden KS stimmen bei $t = 0$ überein)

$$\vec{\omega} := (0, 0, \omega) \quad \vec{v}_r = (v, 0, 0) \quad \vec{v}_t = \vec{\omega} \times \vec{r} \quad (\vec{r} := \overrightarrow{OA}, |\vec{v}_t| = \omega |\vec{r}|) \quad \vec{v}_i = \vec{v}_r + \vec{v}_t$$

\vec{v}_r Geschwindigkeit im rotierenden KS, \vec{v}_i im Inertialsystem, \vec{v}_t Tangentialgeschwindigkeit



In K bzw. K' hat das Projektil verschiedene Geschwindigkeiten

Wir fertigen uns ein *Geogebra*-Arbeitsblatt an, dies zu veranschaulichen:

- Punkte: Ursprung $0=(0,0)$, freie Punkte A, B und $\mathbf{v_r}=\mathbf{Vector}(A,B)$; $\mathbf{r_v}=\mathbf{Vector}(A)$
- Scheibe und "Tangentenkreis": $\mathbf{Disk}:x^2+y^2\leq 1.4^2$, $\mathbf{k_i}:\mathbf{Circle}(0,A)$
- Slider für ω anschl. $\mathbf{\omega_v}=\mathbf{Vector}(0,0,\omega)$, Tangentialvektor: $\mathbf{t_v}=\mathbf{Cross}(\omega,\mathbf{r_v})$
- Tangentialvektoren(displayed): $\mathbf{t_d1}=\mathbf{Vector}(A,A+\mathbf{t_v})$, $\mathbf{t_d2}=\mathbf{Vector}(B,B+\mathbf{t_v})$
- Geschwindigkeit im Inertialsystem: $\mathbf{v_i}=\mathbf{Vector}(A,A+\mathbf{v_r}+\mathbf{t_v})$
- positive Koordinatenachsen (bei $t = 0$): $\mathbf{x_A}=2*\mathbf{v_r}$, $\mathbf{y_A}=\mathbf{Cross}((0,0,1),\mathbf{x_A})$
- Anfangskoordinaten (x_0, y_0) : $\mathbf{C}=\mathbf{Intersect}(\mathbf{Line}(A, \mathbf{y_A}), \mathbf{Line}(0, \mathbf{x_A}))$
 $\mathbf{D}=\mathbf{Intersect}(\mathbf{Line}(A, \mathbf{x_A}), \mathbf{Line}(0, \mathbf{y_A}))$, Segmente von A nach C bzw. D

Mit A bzw. B lässt sich v_r einstellen, sie sind frei beweglich (rot gekennzeichnet)

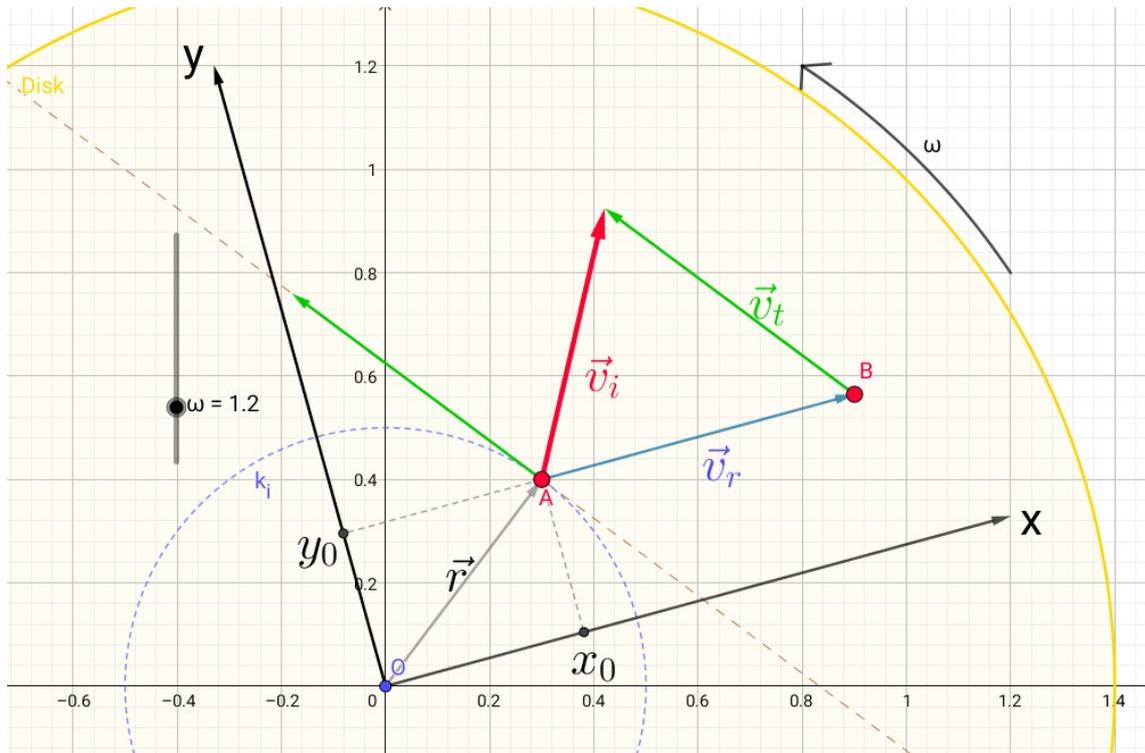


Abb.86 : rotierende Scheibe

Da die z -Koordinate immer verschwindet, lassen wir sie in Zukunft weg. Im Inertialsystem K ist die Bahn des Projekttils (Bullet) eine Gerade, da keine Kräfte wirken.

$$\vec{x}_B(t) = \vec{r}_0 + \vec{v}_i \cdot t = \vec{r}_0 + (\vec{v}_r + \vec{v}_t) t = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} + \begin{pmatrix} v - \omega y_0 \\ \omega x_0 \end{pmatrix} t \quad (11.1)$$

Aus Kap. 5 wissen wir, dass die Spaltenvektoren der Rotationsmatrix die Bilder der Standardbasisvektoren sind (hier die **aktive** Form):

$$R_\alpha = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}$$

Wir brauchen hier natürlich die **passive** Form (KS dreht sich) und $\alpha = -\omega t$

Außerdem gilt: $\begin{pmatrix} a & b \\ c & d \end{pmatrix} \cdot \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix} = \lambda_1 \begin{pmatrix} a \\ c \end{pmatrix} + \lambda_2 \begin{pmatrix} b \\ d \end{pmatrix}$

Damit wird die Bahngerade $\vec{x}_B(t)$ in K'

$$\begin{aligned} \vec{x}_B'(t) &= \begin{pmatrix} \cos\omega t & \sin\omega t \\ -\sin\omega t & \cos\omega t \end{pmatrix} \cdot \vec{x}_B(t) = \\ &= (x_0 + (v - y_0\omega)t) \begin{pmatrix} \cos\omega t \\ -\sin\omega t \end{pmatrix} + (y_0 + x_0\omega t) \begin{pmatrix} \sin\omega t \\ \cos\omega t \end{pmatrix} \end{aligned} \quad (11.2)$$

11. Coriolis-Kraft

$\vec{x}_B'(t)$ ist also die Summe von 2 orthogonalen rotierenden “Einheits-Zeigern” mit den Amplituden $A_1(t) := (x_0 + (v - y_0\omega)t)$ und $A_2(t) := (y_0 + x_0\omega t)$

Mit $e^{i\phi} = \cos \phi + i \sin \phi$ wäre auch eine Darstellung mit komplexen Zahlen möglich:

$$\vec{x}_B' \rightarrow z_B'(t) = A_1(t) e^{-i\omega t} + A_2(t) e^{i(\pi/2 - \omega t)} = e^{-i\omega t} [A_1(t) + A_2(t) e^{i\pi/2}]$$

Da in \mathbb{C} gilt: $|z_1 \cdot z_2| = |z_1| \cdot |z_2|$ kann man $|z_B'(t)|$ leicht berechnen:

$$|z_B'(t)| = |e^{-i\omega t}| |A_1(t) + i A_2(t)| = 1 \sqrt{A_1^2 + A_2^2} \quad (11.3)$$

$$\vec{e}_1'(t) = \begin{pmatrix} \cos \omega t \\ \sin \omega t \end{pmatrix} \quad \vec{e}_2'(t) = \begin{pmatrix} -\sin \omega t \\ \cos \omega t \end{pmatrix} \quad \text{sind die zeitlich variierenden}$$

Basisvektoren (natürlich vom Inertialsystem aus gesehen) des rotierenden Systems.

Eine Herleitung von Formel 11.2 wäre auch mit Unterkapitel 4.2 möglich:

Hier wurde festgestellt

$$\vec{x}' = A^T \vec{x} \quad \text{wobei } A = (\vec{e}_1', \vec{e}_2') = \left(\begin{pmatrix} \cos \omega t \\ \sin \omega t \end{pmatrix} \quad \begin{pmatrix} -\sin \omega t \\ \cos \omega t \end{pmatrix} \right) \quad \text{also}$$

$$\vec{x}' = \begin{pmatrix} \cos \omega t & \sin \omega t \\ -\sin \omega t & \cos \omega t \end{pmatrix} \begin{pmatrix} x_0 + (v - \omega y_0)t \\ y_0 + \omega x_0 t \end{pmatrix}$$

Schauen wir uns die Bahnkurve im rotierenden System (blau) gegenüber jener im Inertialsystem (rot) in einem *Geogebra*-Arbeitsblatt an:

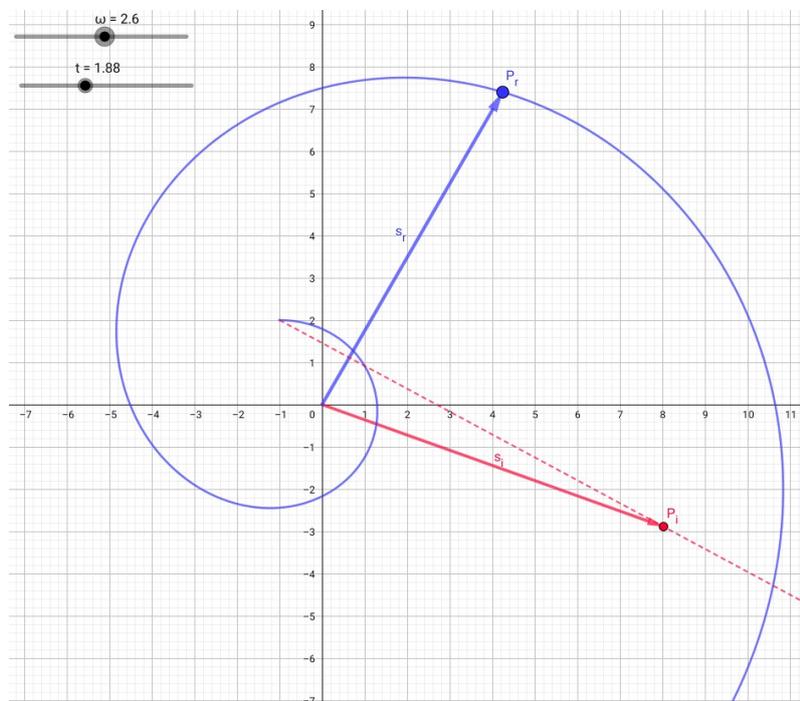


Abb.87 : Flugbahn in den Koordinaten von K' (blau, rotierend) bzw. K (rot)

Das Arbeitsblatt ist schnell erstellt:

- Zuerst die Schieberegler für x_0, y_0, v, ω und t , Ursprung $0=(0,0)$
- Die beiden "Amplituden": $A_1=x_0+t*(v-y_0*\omega)$ $A_2=y_0+t*x_0*\omega$
- Positionsvektoren im Inertialsystem \vec{s}_i bzw. rotierendem System \vec{s}_r :
 $s_i=Vector(A_1, A_2)$, $s_r=Rotate(s_i, -\omega*t)$, $P_i=s_i$, $P_r=s_r$
- Ortslinien: $Locus(P_i, t)$, $Locus(P_r, t)$,

Da die Spirale aus der Rotation der Gerade 11.1 um O hervorgeht, ist jeder Punkt der Ebene durch Anpassen der Parameter zu *irgendeinem* Zeitpunkt erreichbar!

Aus der Sicht von K' ist die blaue Kurve nur dadurch erklärbar, dass 2 Kräfte auf den Körper wirken (später folgt eine genauere Analyse):

- Eine die den Körper nach außen zieht (weg vom Rotationszentrum): Fliehkraft ($\propto \overrightarrow{OP_r}$)
- Eine die den Körper "rechts" abbiegen lässt: Corioliskraft ($\propto \vec{v}_r$)
 Beide auch $\propto |\vec{\omega}| - \vec{\omega}$ schaut in Richtung der Drehachse (Rechtschraubenregel!)
 Wechselt $\vec{\omega}$ die Richtung wird "links" abgebogen!

11.3 Näherung für die Erde an den Polen

An den Polen ist die Erde lokal eine Scheibe ($\vec{\omega} \perp \vec{v}_r$) außerdem gilt $|\omega t| \ll 1$

Man approximiert die Winkelfunktionen und nimmt $v_r(t)$ in diesem Zeitraum als konstant an

$$\cos(\omega t) \approx 1 \text{ und } \sin(\omega t) \approx \omega t \quad v_r(t) = v$$

damit wird 11.2 zu (wobei wir den Term mit $(\omega t)^2$ verschwinden lassen):

$$x - x_0 = vt + x_0(\omega t)^2 = \cancel{x_0(\omega t)^2} + vt \Rightarrow t = \frac{x - x_0}{v}$$

$$y - y_0 = -v\omega t^2 \Rightarrow \Delta y = -\frac{\omega}{v}(x - x_0)^2 = \Delta y = -\frac{\omega}{v}(\Delta x)^2$$

Näherung für die Erde
in der Nähe der Pole

$$\Delta y = -\frac{\omega}{v}(\Delta x)^2 \quad (11.4)$$

11. Coriolis-Kraft

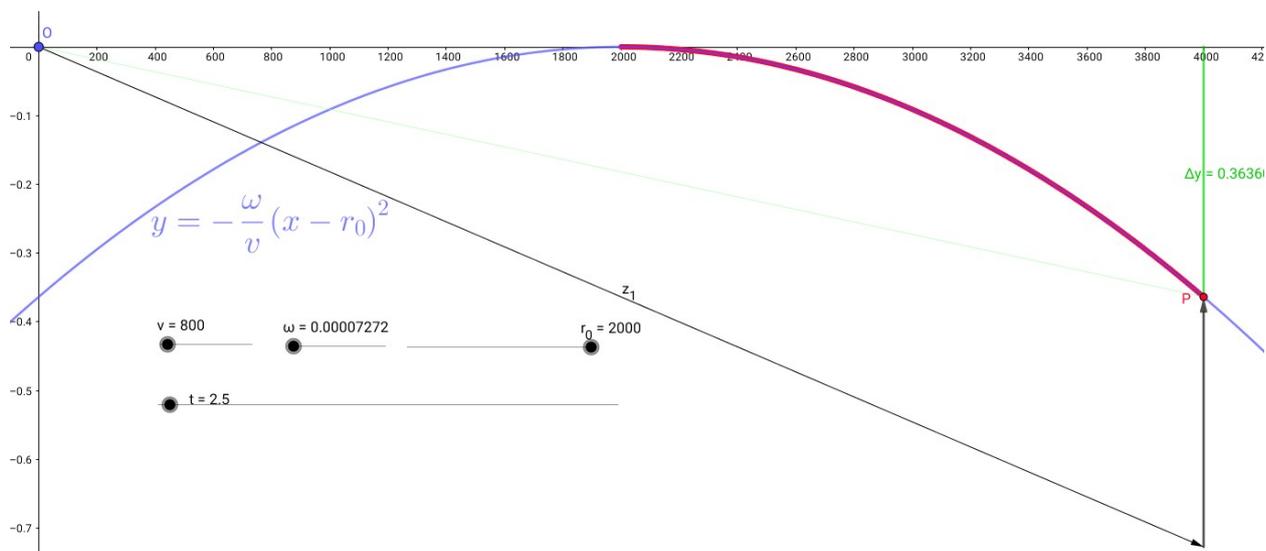


Abb.88 : Flugbahn eines Geschosses mit "Erde-Näherung" $|\omega t| \ll 1$

Im obigen *Geogebra*-Arbeitsblatt haben wir die Parameter für unsere "Schießübung" oben eingestellt. Die Näherungsbedingung $|\omega t| \ll 1$ ist gut erfüllt, denn

$$\omega_{Erde} = \frac{2\pi}{24 \cdot 3600} \text{ rad/s} \quad t = 2,5 \text{ s} \Rightarrow \omega t \approx 1,8 \cdot 10^{-4}$$

Deshalb ist auch die Bahn des Geschosses (durch *P* dargestellt) im oberen Arbeitsblatt nicht von unserer Näherungsfunktion zu unterscheiden. Arbeitsblatt als auch Näherung liefern bei einem Schuss am Pol:

$$\Delta y = \frac{7,2722 \cdot 10^{-5}}{800} (2000)^2 \approx 0,36 \text{ m}$$

Das ist mehr als ich erwartet hätte und reicht für einen Fehlschuss aus - ist aber doch weit weniger als der Einfluss des Windes. Der Einfluss von r_0 ist für die Abweichung fast unerheblich - allerdings darf man das auch nicht übertreiben, sodass man in völlig andere geografische Breiten wechselt (siehe nächstes Unterkapitel).



Die Pole haben eine Sonderstellung - nehmen wir als Beispiel den Nordpol. Es gibt nur eine einzige Richtung - nämlich nach Süden. West- oder Ostrichtung haben hier keine Bedeutung. Daher ist die Abweichung Δy für jede Richtung gleich.

Wie schaut das dann in gemäßigeren Breiten aus?

Spielt die Schussrichtung eine Rolle?

11.4 Einfluss der geografischen Breite

Die mathematische Analyse folgt etwas später. Hier wollen wir einfach die Rechenpower von *wxMaxima* einsetzen und zwar betrachten wir folgendes Modell:

Irgendwo in einem Punkt P auf der Nordhalbkugel der Erde wird ein Projektil mit Geschwindigkeit v_r und einem Winkel α zur Ostrichtung waagrecht abgeschossen.

Wir suchen jenen Ort \vec{x}_{r2} , wo das Projektil sich nach einem Zeitraum $\Delta t \approx 1$ in K_r tatsächlich befindet gegenüber dem vermeintlichen Ort \vec{x}_{r1} . Den Unterschied dieser beiden Orte werden wir näherungsweise in der Tangentialebene in P berechnen.

Für diesen kurzen Zeitraum lassen wir den Umlauf um die Sonne außer acht und betrachten nur die Eigenrotation der Erde. Wir verwenden 3 Koordinatensysteme:

1. Ein Inertialsystem K_i (mit Kugelkoordinaten r, θ, ϕ) im Mittelpunkt der Erde (in diesem Modell eine vollkommene Kugel) mit x - z -Ebene durch P (also $\phi = 0$). Die x - y -Ebene geht durch den Äquator. R sei der Erdradius.
2. Ein in K_i rotierendes Koordinatensystem K_r mit der Winkelgeschwindigkeit der Erde ω . Ursprung und z -Achse von K_r stimmen immer mit K_i überein, x - und y -Achse bei $t = 0$. Um die Koordinaten von K_i in K_r umzurechnen, braucht man lediglich $\phi (= 0$ in K_i) durch $-\omega t$ ersetzen (deswegen dürfen wir ϕ nicht **gleich** Null setzen!)
3. Ein 2-dimensionales Koordinatensystem in P mit den Basisvektoren \hat{e}_1 (Richtung Ost) und \hat{e}_2 (Richtung Nord) in den Koordinaten von K_i .

Für P in verwenden wir Kugelkoordinaten:

$$\vec{p}_0 = (\sin \theta \cos \phi, \sin \theta \sin \phi, \cos \theta) \Rightarrow \vec{p} = R \vec{p}_0$$

$$\text{In } K_i \text{ bewegt sich das Projektil kräftefrei mit } \vec{x}_i = \vec{p} + \vec{v}_i \Delta t \Big|_{\phi = 0}$$

Die Tangentialgeschwindigkeit in P beträgt $\vec{v}_t = R \sin \theta \hat{e}_1$ wobei \hat{e}_1 ein Einheitsvektor in Ostrichtung ist.

Die Äquatorkurve $\vec{k}(\phi)$ hat die Gleichung $\vec{k}(\phi) = (\cos \phi, \sin \phi, 0)$

$$\hat{e}_1 \text{ ist dann } \hat{e}_1 = \frac{d}{d\phi} \vec{k}(\phi) = (-\sin \phi, \cos \phi, 0)$$

$$\hat{e}_2 = \vec{p}_0 \times \hat{e}_1 = (-\cos \theta \cos \phi, -\cos \theta \sin \phi, \sin \theta)$$

$$\vec{v}_r = v(\cos \alpha \hat{e}_1 + \sin \alpha \hat{e}_2) \quad \vec{v}_i = \vec{v}_r + \vec{v}_t$$

Fassen wir zusammen:

$$\vec{x}_i = \vec{p} + \vec{v}_i \Delta t \Big|_{\phi = 0} \quad \text{tatsächlicher Ort in } K_i \quad (11.5)$$

$$\vec{x}_{r2} = \vec{p} + \vec{v}_i \Delta t \Big|_{\phi = -\omega t} \quad \text{derselbe Ort in } K_r$$

$$\vec{x}_{r1} = \vec{p} + \vec{v}_r \Delta t \Big|_{\phi = 0} \quad \text{berechneter Ort in } K_r, \text{ es ergibt sich ein Differenzvektor}$$

$$\vec{\delta}_v = \vec{x}_{r2} - \vec{x}_{r1} \quad \text{diesen projizieren wir in die Tangentialebene: } \vec{\delta}_{vt} = \delta_v - (\delta_v \cdot \vec{p}_0) \vec{p}_0$$

$$\text{und berechnen die Koordinaten im Tangentialsystem: } \vec{\xi}_1 = \vec{\delta}_{vt} \cdot \hat{e}_1 \Big|_{\phi = 0} \quad \vec{\xi}_2 = \vec{\delta}_{vt} \cdot \hat{e}_2 \Big|_{\phi = 0}$$

11. Coriolis-Kraft

Schauen wir uns das in *wxMaxima* an:

Einige Vereinbarungen: Vektorprodukt, keine Umwandlungswarnungen, Kommazahlenlänge

```
(% i1) (load("vect"), ratprint:false, fpprintprec:3)$
```

```
(% i2) assume(v>0, ω>0, cos(θ) >=0)$
```

Einheitsvektor zum Punkt P in Kugelkoordinaten

```
(% i3) p_0: [sin(θ)*cos(φ), sin(θ)*sin(φ), cos(θ)]$
```

```
(% i4) p:R*p_0$
```

Ostvektor - Basisvektor in der Tangentialebene

```
(% i5) e_1:diff([cos(φ),sin(φ),0], φ); [- sin (φ) , cos (φ) , 0]
```

Nordvektor - zweiter Basisvektor in der Tangentialebene

```
(% i6) e_2: trigsimp(express( p_0 ~ e_1)); [- cos (θ) cos (φ) , - cos (θ) sin (φ) , sin (θ)]
```

Winkelgeschwindigkeitsvektor - positive z -Achse

```
(% i7) ω_v:[0,0,ω]$
```

Geschwindigkeit v in der Tangentialebene in beliebiger horizontaler Richtung

```
(% i8) v_r: v*(cos(α)*e_1+sin(α)*e_2)$
```

Tangentialgeschwindigkeit des Punktes P

```
(% i9) v_t: ω*R*sin(θ)*e_1$
```

Geschwindigkeit des Projektils im Inertialsystem - nicht vergessen ϕ Null setzen!

```
(% i10) v_i:v_r+v_t$
```

Vermeintlicher Zielpunkt - gerechnet wie in einem Inertialsystem

```
(% i11) x_r1: p + v_r*Δt$
```

Tatsächlicher Zielpunkt im **Inertialsystem**

```
(% i12) x_i: p + v_i*Δt$
```

Tatsächlicher Zielpunkt im **rotierenden System** - deswegen mussten wir ϕ mitschleppen!

```
(% i13) x_r2: subst(-ω*Δt, φ, x_i)$
```

Aber jetzt ist Schluss - daher müssen wir alle vorherigen Ausdrücke auswerten $\rightarrow ev$

```
(% i14) φ:0$
```

Differenzvektor $\vec{\delta}_v$ zwischen berechnetem und tatsächlichem Zielpunkt

```
(% i15) δ_v: ev(trigsimp(x_r2 - x_r1))$
```

Komponente von $\vec{\delta}_v$ in der Tangentialebene $\vec{\delta}_{vt} = \vec{\delta}_v - (\vec{\delta}_v \cdot \vec{p}_0) \vec{p}_0$

```
(% i16) δ_p: ev( δ_v . p_0) $
```

(% i17) $\delta_vt: ev(\delta_v - \delta_p * p_0)$

Wir berechnen die Koordinaten in der Tangentialebene (ξ_1, ξ_2)

(% i18) $\xi_1: ev(e_1 . \delta_vt)$

(% i19) $\xi_2: ev(e_2 . \delta_vt)$

Jetzt setzen wir unsere Daten ein: α und θ bleiben variabel, ϕ hat sich schon erübrigt

(% i20) (R:6.3*10^6, $\Delta t:1.2$, $\omega: float(2*\%pi/(24*3600))$, v:900)

Nur der zweite Term liefert ein wesentlichen Beitrag

(% i22) $ex1: expand(ev(\xi_1)); -1.410^{-6} \sin(\theta) + 0.0942 \sin(\alpha) \cos(\theta) - 4.1110^{-6} \cos(\alpha)$

Nur der dritte Term ist der wesentliche - $\cos \theta \sin \theta$ wird maximal 0.5

(% i23) $ex2: expand(ev(\xi_2)); -0.024 \cos(\theta) \sin(\theta) - 4.1110^{-6} \sin(\alpha) \cos(\theta)^2 - 0.0942 \cos(\alpha) \cos(\theta)$

Wir konzentrieren uns auf das Wesentliche

(% i24) $\xi_1: part(ex1,2); 0.0942 \sin(\alpha) \cos(\theta)$

(% i25) $\xi_2: part(ex2,3); -0.0942 \cos(\alpha) \cos(\theta)$

Die Länge der Abweichung

(% i26) $deviation: float(sqrt(trigsimp(\xi_1^2 + \xi_2^2))); 0.0942 \cos(\theta)$

Woher kommt die Zahl 0.0942? Wir raten von der Nordpolnäherung \rightarrow Treffer!

(% i27) $\omega * \Delta t^2 * v; 0.0942$

Liegt der Vektor $[\xi_1, \xi_2, 0]$ links oder rechts von $[\cos(\alpha), \sin(\alpha), 0]$?

(% i28) $LR: float(trigsimp(express([\xi_1, \xi_2, 0] \sim [\cos(\alpha), \sin(\alpha), 0]))) [3]; 0.0942 \cos(\theta)$

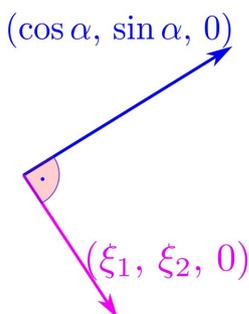


Abb.89 : Vektorenlage

Die Übereinstimmung von (% i26) mit (% i28) ist nur möglich, falls sich die Vektoren der Tangentialebene wie in nebenstehender Zeichnung verhalten:

Der Vektor $\vec{\xi}$ muss orthogonal zum Einheitsvektor der ursprünglichen Richtung $\vec{v}_{r0} = (\cos \alpha, \sin \alpha)$ sein und zwar rechts von dieser falls $\theta \in [0, \pi/2]$ (also Nordhalbkugel) sonst links.



Die Ablenkung ist unabhängig von α - es ist egal in welche Richtung man schießt (wie am Pol).

Ist φ die geografische Breite von P gilt: $\cos \theta = \sin \varphi$
Mit $\Delta x = v \Delta t$ erhalten wir für die Ablenkung Δy :

Ablenkung auf der geografischen Breite φ : $\Delta y \approx \frac{\omega}{v} \Delta x^2 \sin \varphi$

In die rechte Richtung auf der Nordhalbkugel, in die linke auf der Südhalbkugel!

11.5 Duell auf einer Scheibe

Wir simulieren folgende Aufgabe in Geogebra: 2 Männer duellieren sich auf einem Karussell, sie sind beide 20 m vom Rotationszentrum entfernt auf einem Durchmesser platziert. Wie groß muss ω sein, damit sie 50 cm daneben schießen? ($v_0 = 400$ m/s) Welche Fliehkraft wirkt auf die Männer dann?

Welche "Schwerkraft" scheint auf die Kugeln zu wirken?

Übrigens wer es weniger "kriegerisch" formulieren möchte: 2 Astronauten befinden sich auf einer rotierenden Raumstation - sie hat 100 m Durchmesser und rotiert so, dass die Schwerkraft auf der Erde simuliert wird. Astronaut A wirft dem gerade "über ihn stehenden" Astronaut B ein Werkzeug mit $v_0 = 4$ m/s zu. Wie weit verfehlt er ihn?

Das Problem ist eigentlich mit Gleichung 11.2 bereits "gelöst" - leichter gesagt als getan. Daher probieren wir es zuerst mit unserer Näherung 11.4 :

$$\Delta y = -\frac{\omega}{v}(\Delta x)^2 \Rightarrow -0.5 \approx -\frac{\omega}{400}(40)^2 \Rightarrow \underline{\omega \approx 0.125} \quad t \approx \frac{\Delta x}{v} = 0.1s \quad (11.6)$$

Dieses ω entspricht einer Winkelgeschwindigkeit von 7 Grad pro Sekunde bzw. einer Umlaufdauer des Karussells von ca. 50 Sekunden (fast einer Minute) - trotzdem sind die Auswirkungen schon relativ drastisch.

Unsere Näherungsbedingung ist ganz gut erfüllt - wie wir im Arbeitsblatt sehen (es gibt keinen sichtbaren Unterschied!). Dass wir als Abstand Geschosbahn-Ziel (bei (20,0)) einfach die Ordinate nehmen ist natürlich auch ein bisschen gefunkert, eigentlich müssten wir den Normalabstand nehmen, aber der Unterschied fällt rechnerisch nicht ins Gewicht - es sieht nur in der Zeichnung so aus, weil die y -Achse so stark "gezoomt" ist!

11.5.1 Simulation in Geogebra

Die Lösung obiger "Aufgabe" führen wir mit unserer Näherung 11.4 durch und kontrollieren mit der exakten Lösung.

- Zuerst die Schieberegler für $v_0 = 400$, $r_0 = -20$, $\omega = 0.05(?)$ und für die Zeit $t = 0$
- Zielpunkt (Target) $T=(20, -0.5)$
- Näherungsparabel $y=-\omega/v_0*(x-r_0)^2$
- Mit dem Schieberegler bestimmen wir $\omega \approx 0.125$ $\Delta t \approx \Delta x/v = 40/400 = 0.1s$

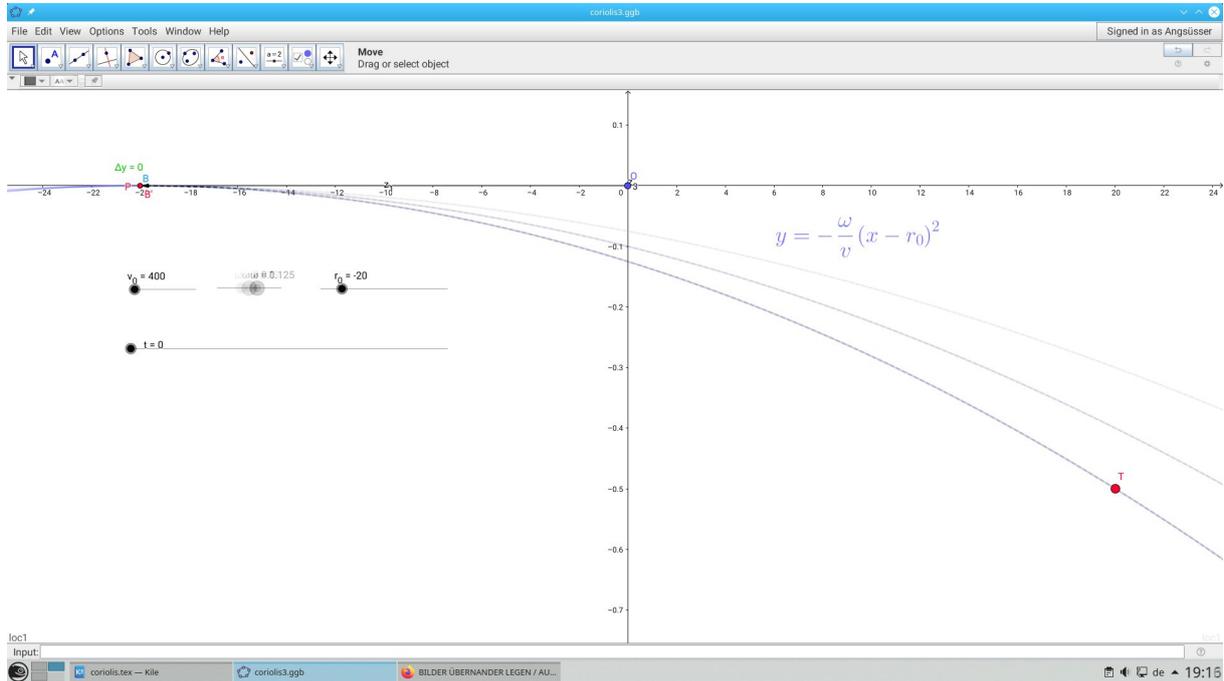


Abb.90 : Bestimmung von $\omega \approx 0.125$

Wir überprüfen nun mit der exakten Lösung wie gut unsere Näherung ist:

- Geschwindigkeit der Kugel(Bullet) im Inertialsystem: $\vec{v} = \begin{pmatrix} v_0 \\ \omega r_0 \end{pmatrix}$

Die Bahn $\vec{s} = \begin{pmatrix} r_0 \\ 0 \end{pmatrix} + t \vec{v} = \begin{pmatrix} r_0 + v_0 t \\ \omega r_0 t \end{pmatrix}$ - diese Bahn transformieren wir durch Drehung um $(-\omega t)$ ins mitrotierende Koordinatensystem (passive Transformation - invers!).

- $B = (r_0 + v_0 t, \omega r_0 t) \rightarrow B' = \text{Rotate}(B, -\omega t)$

Wer selbst Hand anlegen will - die Rotationsmatrix für einen Punkt um Winkel (ωt) lautet (siehe Kapitel 4 - Rotationen):

$$R = \begin{pmatrix} \cos(\omega t) & -\sin(\omega t) \\ \sin(\omega t) & \cos(\omega t) \end{pmatrix} \Rightarrow R^{-1} = \begin{pmatrix} \cos(\omega t) & \sin(\omega t) \\ -\sin(\omega t) & \cos(\omega t) \end{pmatrix} \quad \text{und da gilt}$$

$$\begin{pmatrix} a_1 & b_1 \\ a_2 & b_2 \end{pmatrix} \begin{pmatrix} \lambda_1 \\ \lambda_2 \end{pmatrix} = \lambda_1 \vec{a} + \lambda_2 \vec{b} \quad \text{gilt für}$$

$$R^{-1} \begin{pmatrix} r_0 + v_0 t \\ \omega r_0 t \end{pmatrix} = (r_0 + v_0 t) \begin{pmatrix} \cos(\omega t) \\ -\sin(\omega t) \end{pmatrix} + (\omega r_0 t) \begin{pmatrix} \sin(\omega t) \\ \cos(\omega t) \end{pmatrix} \quad \text{also} \quad (11.7)$$

$$z_1 = (r_0 + v_0 t) * \text{Vector}(\cos(\omega t), -\sin(\omega t))$$

$$z_2 = \omega r_0 t * \text{Vector}(\sin(\omega t), \cos(\omega t)) \quad \text{und} \quad P = z_1 + z_2 \quad \text{- wobei } P = B' \text{ gilt}$$

11. Coriolis-Kraft

Jetzt noch die Spur von B bzw. B' einschalten und Schieberegler für t betätigen:
Man sieht auch die Zeit $t = 0.1$ für die "Kollision" stimmt fast exakt.

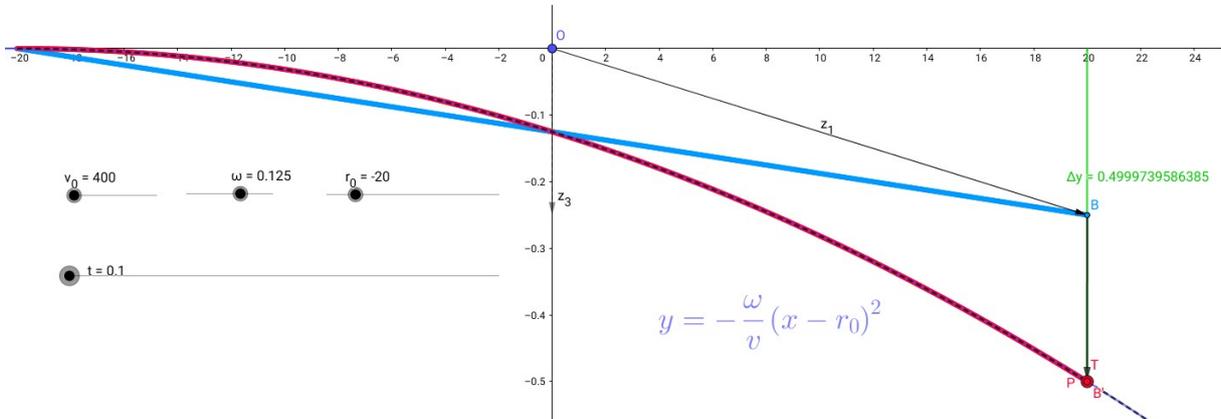


Abb.91 : Approximationsparabel und Spur der exakten Bahn sind nicht unterscheidbar

Nun zur Zentrifugalbeschleunigung (Bahn des Mannes bei r_0):

$$\vec{r}(t) = r_0 \begin{pmatrix} \cos(\omega t) \\ \sin(\omega t) \end{pmatrix} \Rightarrow \dot{\vec{r}}(t) = \vec{v}(t) = r_0 \omega \begin{pmatrix} -\sin(\omega t) \\ \cos(\omega t) \end{pmatrix} \Rightarrow \ddot{\vec{r}}(t) = \vec{a}(t) = r_0 \omega^2 \begin{pmatrix} -\cos(\omega t) \\ -\sin(\omega t) \end{pmatrix}$$

oder als Differentialgleichung $\ddot{\vec{r}}(t) = -\omega^2 \vec{r}(t)$ - wobei das Minuszeichen für das Inertialsystem gilt - vom rotierenden System aus gesehen existiert ein Kraftfeld radial nach außen welches mit r und ω^2 wächst. Bei uns hier ergibt sich:

$$a = r_0 \omega^2 = 20 \times 0.125^2 = 0.3125 \text{ m/s}^2$$

Die Flugbahn der Kugel ist eine "Wurfparabel" ("waagrechter Wurf") mit folgender Formel:

$$s = \frac{a}{2} t^2 \quad \underbrace{\Rightarrow}_{s=0.5 \quad t=0,1} \quad a = 100 \approx 10 \text{ g}$$

Auch der Vergleich der Wurfbahn (mit der "Coriolisbeschleunigung" a_C) ergibt:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} r_0 + v t \\ \frac{a_C}{2} t^2 \end{pmatrix} \quad \underbrace{\Rightarrow}_{t \text{ eliminieren}} \quad y = -\frac{a_C}{2v^2} (x - r_0)^2 = -\frac{a_C}{2v^2} (\Delta x)^2$$

Vergleich mit 11.6 liefert $\frac{a_C}{2v^2} = \frac{\omega}{v} \Rightarrow a_C = 2\omega v = 100 \approx 10 \text{ g}$

In Vektorschreibweise $\vec{a}_C = -2\vec{\omega} \times \vec{v}$ ($\vec{\omega} \perp$ zur Rotationsebene + Rechtsschraubenregel)
Im rotierenden (=beschleunigten) Bezugssystem bei unserer Rotationsrichtung scheint also eine "fiktive" Schwerkraft die Kugeln ganz ordentlich nach rechts (vom Schützen aus) zu ziehen.

Dies ist übrigens auch die Grundidee der **Allgemeinen Relativitätstheorie** die Schwerkraft durch ein beschleunigtes Bezugssystem (zumindest lokal) zu ersetzen!

11.5.2 Berechnung in *wxMaxima*

Für die Rechnung in *wxMaxima* benutzen wir eine andere Methode:

- Sei $d2(t, \omega) := \vec{x}_{TG}(t, \omega) \cdot \vec{x}_{TG}(t, \omega)$ der quadratische Abstand zwischen Ziel(Target) und Geschöß zur Zeit t bei ω .
- $d2(t, \omega)$ ist für t_m minimal, also $I. \quad \frac{d}{dt}d2(t, \omega) = 0 \quad \text{für } t = t_m$
- Außerdem gilt $II. \quad d2(t_m, \omega) = 0.5^2 \quad \text{für } \omega = \omega_m$
Also der Abstand zur minimalen Zeit ist 0.5 bei einem bestimmten ω_m

Dieses Gleichungssystem ist nichtlinear und wir benutzen ein numerischen Verfahren, das multiple Newtonverfahren (`load("mnewton")`)

Befehlssyntax:

```
mnewton([expr1, expr2], [var1, var2], [startValue1, startValue2])
```

dabei sind

- `expr1, expr2` - die Ausdrücke die verschwinden sollen
- `var1, var2` - die Variablen-Bezeichner und deren Startwerte

Zuerst die Position des Geschößes(von rechts nach links wird geschossen, Rotation positiv)

```
(%i1) vec_x_G: [r_0-v_0*t,%omega*r_0*t];
```

```
(%o1) [r_0 - t v_0, \omega r_0 t]
```

Position des Zieles(Ziel wandert nach unten)

```
(%i2) vec_x_T: [-r_0*cos(%omega*t), -r_0*sin(%omega*t) ];
```

```
(%o2) [-r_0 cos(\omega t), -r_0 sin(\omega t)]
```

Vektor zwischen Geschöß und Ziel

```
(%i3) vec_TG:vec_x_G - vec_x_T;
```

```
(%o3) [-t v_0 + r_0 cos(\omega t) + r_0, r_0 sin(\omega t) + \omega r_0 t]
```

Jetzt die Konstanten eingeben

```
(%i4) (r_0:20,v_0:400, r_min:0.5)$
```

quadratische Entfernung zwischen Projektil und Ziel und Angaben verwenden

```
(%i5) d_2: ev(vec_TG . vec_TG-r_min^2);
```

```
(%o4) (-20 sin(\omega t) - 20 \omega t)^2 + (-20 cos(\omega t) + 400 t - 20)^2 - 0.25
```

Wir starten das multiple Newtonverfahren

11. Coriolis-Kraft

```
(%i7) load("mnewton")$
```

Ableitung der quadratischen Entfernung - "s" für "Strich"

```
(%i8) d_2s:diff(d_2,t);
```

$$2(-20 \cos(\omega t) + 400t - 20)(20\omega \sin(\omega t) + 400) + 2(-20\omega \cos(\omega t) - 20\omega)(-20 \sin(\omega t) - 20\omega t)$$

Für den Startwert von t benutzen wir einfach die Entfernung Schütze-Ziel geteilt durch Projektil-geschwindigkeit, für ω probieren wir einfach 1

```
(%i9) mnewton([d_2,d_2s], [t,%omega], [2*r_0/v_0, 1]);
```

```
[[t = 0.0999804703264159, \omega = 0.1250162778093879]]
```



Die Geogebra-Simulation wird gut bestätigt

11.6 Duell mit Vorhaltewinkel

Wenn r_0 , v_0 und ω feststehen - bei welchem Vorhaltewinkel ϕ trifft ein Schütze S ein Ziel T , das denselben Abstand r_0 vom Rotationszentrum und einen "Vorsprungswinkel" von θ besitzt?

$$\text{Schütze zum Zeitpunkt } t: \quad S(t) := S' = r_0(\cos(\omega t), \sin(\omega t))$$

$$\text{Ziel zum Zeitpunkt } t: \quad T(t) := T' = r_0(\cos(\theta_T + \omega t), \sin(\theta_T + \omega t))$$

zum Abschusszeitpunkt gilt:

$$S(0) := S = r_0(1, 0) \quad T(0) := T = r_0(\cos(\theta_T), \sin(\theta_T))$$

Das Geschöß G besitzt eine Geschwindigkeitskomponente von der Scheibe \vec{v}_y und eine vom Vorhalt \vec{v}_ϕ , die sich addieren zu \vec{v}_α (α der Steigungswinkel in Inertialsystem):

$$\vec{v}_y = \begin{pmatrix} 0 \\ \omega r_0 \end{pmatrix} \quad \vec{v}_\phi = v_0 \begin{pmatrix} -\cos \phi \\ \sin \phi \end{pmatrix} \Rightarrow \vec{v}_\alpha = \vec{v}_y + \vec{v}_\phi = \begin{pmatrix} -v_0 \cos \phi \\ \omega r_0 + v_0 \sin \phi \end{pmatrix}$$

$$\text{Geschößbahn } G(t): \vec{s}_G(t) = \begin{pmatrix} r_0 \\ 0 \end{pmatrix} + t \vec{v}_\alpha = \begin{pmatrix} r_0 - t v_0 \cos \phi \\ \omega r_0 t + t v_0 \sin \phi \end{pmatrix}$$

$$\tan \alpha = \frac{\omega r_0 + v_0 \sin \phi}{v_0 \cos \phi} \Rightarrow \alpha(\phi) = \arctan \left(\frac{\omega r_0 + v_0 \sin \phi}{v_0 \cos \phi} \right) \quad \text{wobei } -\frac{\pi}{2} < \alpha < \frac{\pi}{2}$$

α ist (außer von der Zeit t) von allen Parametern abhängig, wir betonen jene von ϕ !

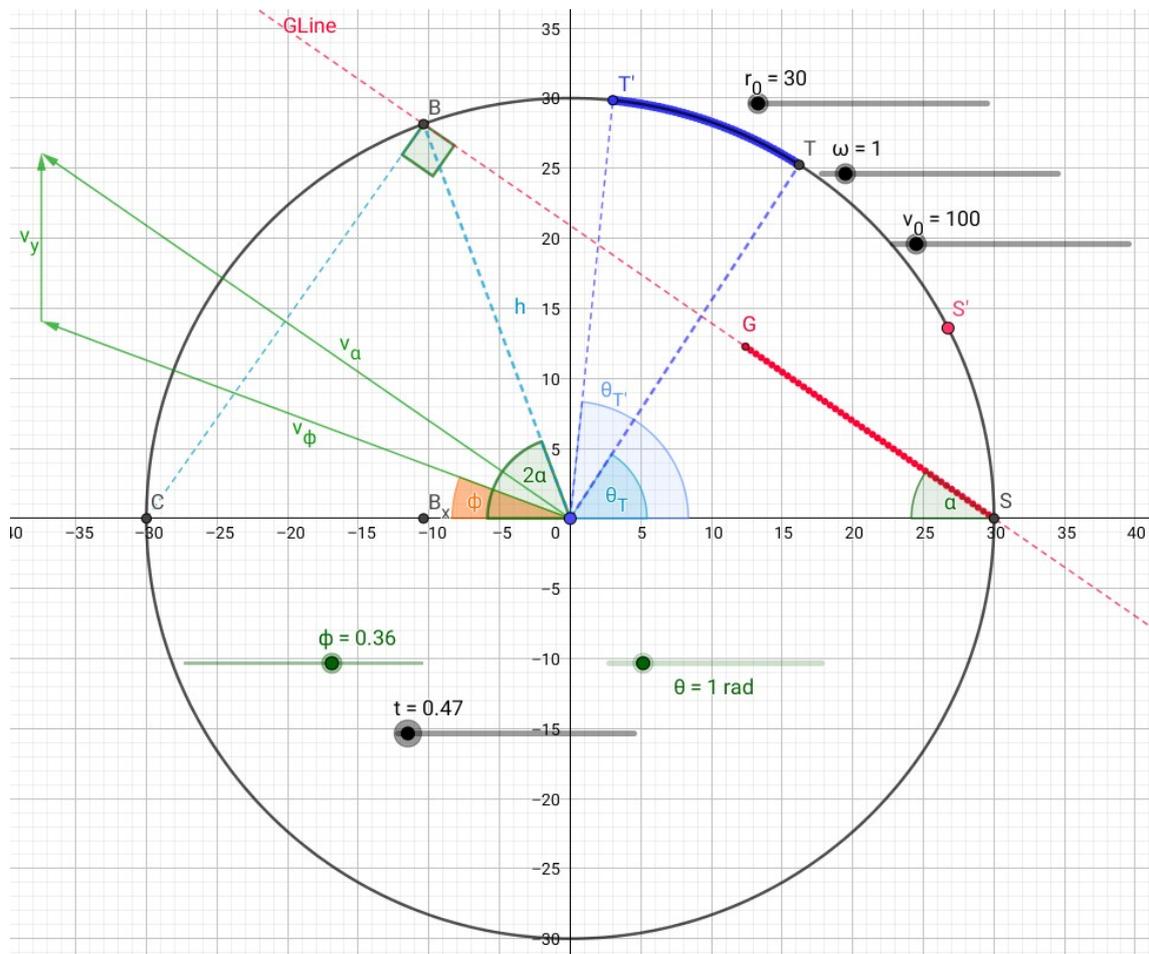


Abb.92 : Schuss mit Vorhaltewinkel

$$\angle SBC \text{ ist rechter Winkel (Thales)} \Rightarrow \overline{SB} = 2r_0 \cos \alpha$$

$$\angle BOC \text{ ist Zentriwinkel zu } \angle BSC \Rightarrow \angle BOC = 2\alpha \Rightarrow B = r_0 \begin{pmatrix} \cos(\pi - 2\alpha) \\ \sin(\pi - 2\alpha) \end{pmatrix}$$

Zu welcher Zeit t_c (c-collision) erreicht das Geschöß G den Punkt B ? $t_c(\phi) = \frac{\overline{SB}(\phi)}{|\vec{v}_\alpha(\phi)|}$

Bedingung für Treffer: $T' = B \Leftrightarrow \omega t_c(\phi) + \theta_T = \pi - 2\alpha(\phi) + k 2\pi \quad k \in \{0, 1, 2, \dots\}$

bzw.

$$[\omega t_c(\phi) + \theta_T] \pmod{2\pi} = \pi - 2\alpha(\phi) \tag{11.8}$$

11.6.1 “Brute force” mit *wxMaxima*

Wir setzen obige Gedanken in ein *wxMaxima*-Programm um - wobei keine Lösung existieren muss ($\omega r_0 \gg v_0$ - sodass die Fliehkraft das Geschöß gleich von der Scheibe “wegschleudert”) bzw. mehrere (unendlich viele?) Lösungen existieren können.

Wir geben einige Bedingungen bekannt:

```
(% i1) assume(v_0 >= 0, r_0 >= 0, phi >-%pi/2 and phi <%pi/2);
```

$$\left[v_0 \geq 0, r_0 \geq 0, \phi > -\frac{\pi}{2}, \frac{\pi}{2} > \phi \right] \quad (\% \text{ o1})$$

Geschwindigkeitsvektor des Geschößes, Geschößbahn und Vektornorm

```
(% i2) v_G: [-v_0*cos(phi), omega*r_0+v_0*sin(phi)]$
```

```
(% i3) s_G: [r_0, 0] + t*v_G; [r_0 - t*v_0*cos(phi), t*(r_0*omega + v_0*sin(phi))]
```

```
(% i4) norm(v):=sqrt(v . v)$
```

Jetzt die Parameter, Spezialfall $\omega r_0 = v_0$

```
(% i5) (r_0:12.5, omega: 1, v_0: 12.5, Theta_T:%pi/2, fpprintprec:4, ratprint: false)$
```

$\tan \alpha = -v_y/v_x$

```
(% i6) define(tg_alpha(phi), -v_G[2]/v_G[1]); tg_alpha(phi) := -frac(r_0*omega - v_0*sin(phi), v_0*cos(phi))
```

Jetzt die inverse Funktion um α zu erhalten

```
(% i7) alpha(phi):=float(atan(tg_alpha(phi)))$
```

Die Länge der Sekante, die die “Kugel” auf der Scheibe zurücklegt - siehe Fig. 92

```
(% i8) SB(phi):=2*r_0*cos(alpha(phi))$
```

Wir berechne die Zeit bis zum Ziel nur in Abhängigkeit von ϕ - deshalb die Evaluierung
Beachte, dass *wxMaxima* den Cosinus in Betragszeichen setzt, obwohl wir ganz oben für ϕ Schranken festgelegt haben, für die der Cosinus positiv ist - anscheinend ist das nicht durchgedrungen!

```
(% i9) define(t_c(phi), trigsimp(ratsimp((SB(phi))/ev(norm(v_G)))));
```

$$t_c(\phi) := \frac{50 |\cos(\phi)|}{\sqrt{2 \sin(\phi) + 2\sqrt{1250 \sin(\phi) + 1250}}} \quad (\% \text{ o9})$$

Der Winkel des Geschößes beim Verlassen der Scheibe

```
(% i10) phi_G(phi):= float(%pi - 2*alpha(phi))$
```

Der Winkel an dem sich das Ziel befindet, wenn das Geschöß die Scheibe verlässt

```
(% i11) define(theta(phi), Theta_T + omega*t_c(phi))$
```

Eine Hilfsfunktion: Gibt es keinen Sprung in den y -Werten?

```
(% i12) isNoJump(y1, y2, jmp):=abs(y1 - y2) < jmp$
```

Eingabe: Liste der x -Werte und dazugehörige y -Werte

Abgabe: die einzelnen **stetigen** Zweige der (x, y) -Tupel, wo nach Nullstellen gesucht wird

```
(% i13) partList(xL, yL):= block( [ branches:[], branch: [ [xL[1], yL[1] ] ], compEl:yL[1]],
  for i:2 thru length(xL) do block(
    if isNoJump(yL[i], compEl, 4) then block(
      branch:endcons( [ xL[i], yL[i] ], branch),
      compEl:yL[i] )
    else block(
      branches: endcons(branch, branches),
      compEl:yL[i],
      branch: [ [xL[i], yL[i] ] ]
    )
  ),
  branches: endcons(branch, branches)
)$
```

Wir benötigen die Startwerte für ein Newtonverfahren:

Eingabe: Alle Funktionszweige(branches) als Liste

Es wird das Minimum und Maximum der Funktionswerte bei einem Ast bestimmt, bei einem Vorzeichenwechsel wird das Startintervall in die Liste der Startintervalle hinzugefügt

Ausgabe: Liste der Startintervalle

```
(% i14) get_start(pL):=block([len:length(pL) , cL:[], cLx:[],interval:[],ymin,ymax],
  for i thru len do (
    cL:pL[i],
    cLy: makelist(cL[i][2], i, 1, length(cL)),
    ymin:lmin(cLy),
    ymax: lmax(cLy),
    if (ymin * ymax) <0 then interval: cons([first(cL)[1], last(cL)[1]], interval )
  ),
  interval
)$
```

Die Funktion mit den "Ästen", deren Nullstellen wir suchen!

```
(% i15) f(phi):=mod(Theta_T + omega*t_c(phi), 2*pi) - phi_G(phi)$
```

Anzahl der ϕ -Werte und Intervallgrenze - nicht $\pi/2$ sondern etwas weniger(um die Äste einzuschränken) - Erklärung folgt später!

```
(% i16) (N:2000, interval_limit: 1.48)$
```

x -Werte und y -Werte werden erzeugt und anschl. in Äste zerteilt

```
(% i17) xwerte: makelist(-interval_limit+(2*interval_limit)/N*i, i, 0, N )$
```

```
(% i18) ywerte: float(map(f,xwerte))$
```

```
(% i19) pointLists:partList(xwerte, ywerte)$
```

Eingabe: Liste der Startintervalle Ausgabe: Nullstenliste

```
(% i20) get_lead_angle(L):=block( [ len:length(L), roots:[ ] ],
  for i thru len do roots:cons(find_root(f(x), L[i][1], L[i][2]), roots), roots )$
```

11. Coriolis-Kraft

Da wir nicht wissen wieviele Funktionszweige zu zeichnen sind, erzeugen wir das Plot-Kommando aus der Punktliste mit einem Programm - zuerst Äste und Legende

```
(% i21) getPlotCmd(pL):=block( [plotCmd: [], len:length(pL)-1, leg:[legend] ],
    for i thru len do block(
        plotCmd: cons( ['discrete, pL[i]], plotCmd ),
        leg: endcons(sconcat(branch_,i), leg)
    ),
    [plotCmd,leg]
)$
```

Jetzt das eigentliche Plot-Kommando

```
(% i22) plotBranches(pL):=block([myPointLists, myLegend, combined],
    combined: getPlotCmd(pL),
    myPointLists:combined[1],
    myLegend:combined[2],
    plot2d(myPointLists,myLegend, [gnuplot_term, "qt 0"])
)$
```

```
(% i23) plotBranches(pointLists)$
```

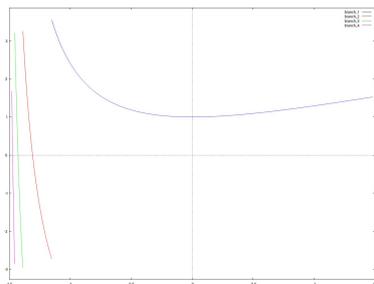


Abb.93 : Äste von 11.8

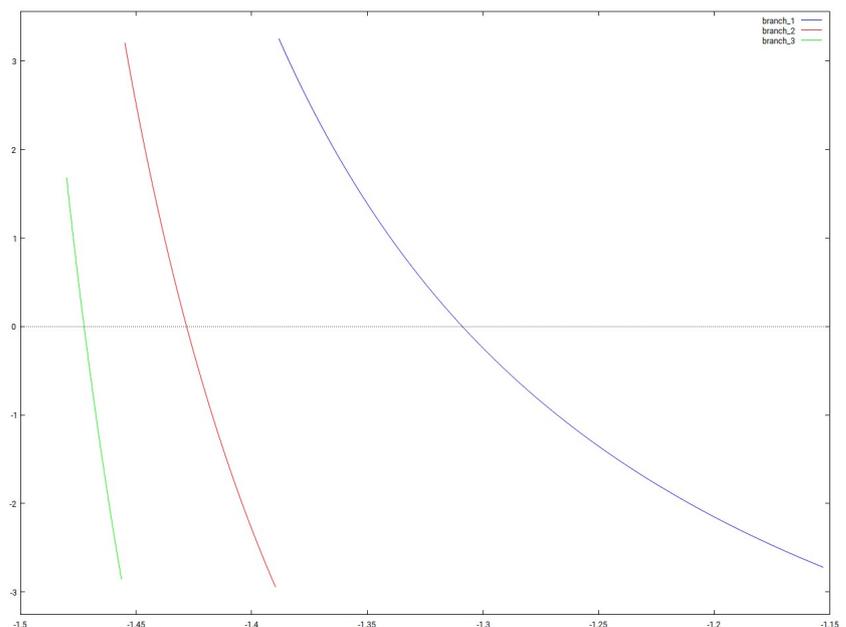


Abb.94 : Nur jene mit Nullstellen (ersten 3)

Wir berechnen die Nullstellen - sie sind gut getroffen!

Die Ausgabedaten der einzelnen Prozeduren können als Eingabedaten für die nächsten verwendet werden

```
(% i24) angleList:get_lead_angle(get_start(partList(xwerte, ywerte))); [-1.472 , -1.428 , -1.309]
```

Für die Flugbahn im rotierenden System brauchen wir die Rotationsmatrix
Da es um eine Koordinatensystem-Transformation handelt, ist diese Matrix invers zur Punkt-
transformation (passiv)

```
(% i25) R:matrix( [cos(ω*t),sin(ω*t)],
                 [-sin(ω*t), cos(ω*t)]
                )$
```

Wir berechnen die einzelnen Flugkurven im Scheibensystem und zeichnen sie

Eingabe: Liste der möglichen Vorhaltewinkel

```
(% i28) plotCurves(angles):= block( [cmdList:[], parCmd:[parametric], styleCmd:[style],
                                     opt: [gnuplot_term, "qt 1"],leg:[legend], callStr:[], plot_time, curveX, curveY ],
```

Falls es keine Lösung gibt: Abbruch

```
if (angles = [] ) then disp("no solutions found!")
```

```
else block(
```

```
    Wir basteln die Befehlszeile:
```

```
[
  [paramtric, x1(t), y1(t), [t, 0, 5]], ... [paramtric, x1(t), y1(t), [t, 0, 5]],
  [style, [linepoints, ...], ... [linepoints, ...]],
  [legend, [...], ... [...]],
  weitere Optionen wie "neues Fenster": [gnuplot_term, "qt 1"]
]
```

```
for i thru length(angles) do (
  φ:angles[i], plot_time:t_c(φ), curveX: ev(R . s_G)[1][1],
  curveY: ev(R . s_G)[2][1],
  styleCmd: endcons([linepoints,1,1,i], styleCmd),
  parCmd: endcons( curveX, parCmd),
  parCmd: endcons( curveY, parCmd),
  parCmd:endcons([t,0,plot_time], parCmd),
  leg: endcons(sconcat("φ=",angles[ length(angles)-i+1]), leg),
  cmdList:cons(parCmd, cmdList),
  parCmd:[parametric]
),
callStr: append([cmdList], [[same_xy ]]),
callStr:append(callStr, [styleCmd]),
callStr:append(callStr, [leg]),
callStr:append(callStr, [opt]),
/*disp(callStr), — for debugging only */
Hier der eigentliche Plotbefehl:
  apply(plot2d, callStr )
)
```

Jetzt wird gezeichnet:

```
(% i29) plotCurves(angleList);
```

11. Coriolis-Kraft

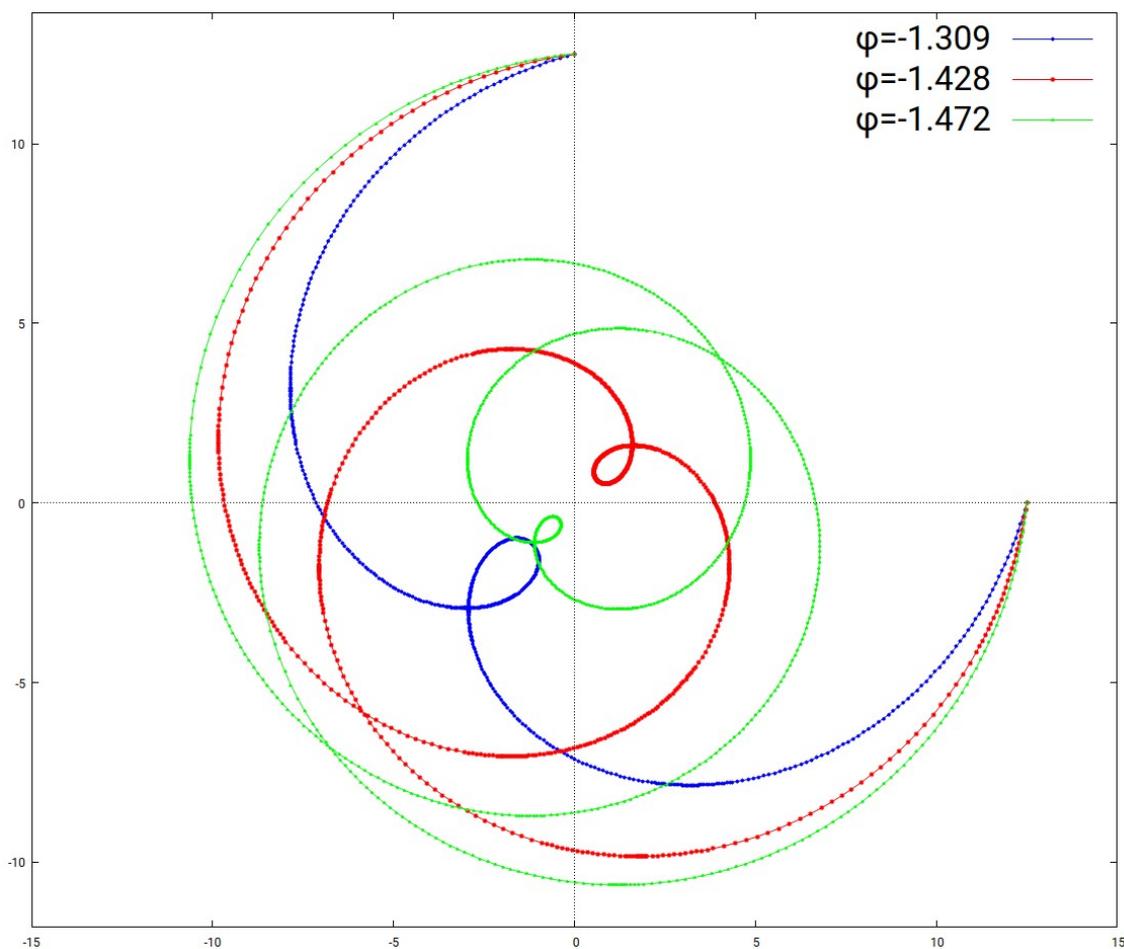


Abb.95 : Schussbahnen für obige Funktionsäste (blau, rot, grün - 0,1,2 Umdrehungen)

Jetzt zu (% i16) (interval_limit: 1.48) – warum hier nicht $\frac{\pi}{2}$ sondern etwas weniger?

Dem aufmerksamen Leser ist sicher aufgefallen, dass wir bei unserem Beispiel $\omega r_0 = v_0$ gewählt haben. Dies bedeutet aber, dass bei einem Vorhaltewinkel $\phi = -\pi/2$ das Geschöß “stillsteht” - man trifft das Ziel (und sich selbst) beliebig oft! Ist allerdings der Vorhaltewinkel $\phi = -\pi/2 + \varepsilon$ dann ist das Geschöß extrem langsam und man trifft das Ziel erst nach vielen (N) Umdrehungen.

Kleinste Änderungen von ε würden dieses N bereits wieder verändern und eine neue Kurve ergeben - wir bräuchten hier eine extreme Auflösung in den Werten für ϕ um diese Lösungen zu finden (im Programm *xwerte*).

Deshalb haben wir uns auf den Wert 1.48 statt $\pi/2$ beschränkt, um obiges Schaubild übersichtlich zu halten (nur die “ersten drei” - vergleiche Abb. 94).



Außerdem sieht man, dass die “erste” Lösung bei $\phi_1 = -1.309$ liegt - man könnte also in einem 2-ten Durchlauf gleich die Suche auf Vorhaltewinkel kleiner ϕ_1 beschränken und so viel Rechenzeit einsparen!

11.6.2 Analytischer Ansatz

In 11.8 ergibt sich mit $\cos \alpha = \frac{1}{\sqrt{1 + \tan^2 \alpha}} = \frac{1}{\sqrt{1 + \left(\frac{v_{\alpha,y}}{v_{\alpha,x}}\right)^2}} = \frac{v_{\alpha,x}}{\sqrt{v_{\alpha,x}^2 + v_{\alpha,y}^2}}$

$$t_c(\phi) = \frac{2r_0 \cos \alpha}{\sqrt{v_{\alpha,x}^2 + v_{\alpha,y}^2}} = \frac{2r_0}{\sqrt{v_{\alpha,x}^2 + v_{\alpha,y}^2}} \frac{v_{\alpha,x}}{\sqrt{v_{\alpha,x}^2 + v_{\alpha,y}^2}} = \frac{2r_0 v_{\alpha,x}}{v_{\alpha,x}^2 + v_{\alpha,y}^2}$$

$$\omega \frac{\overbrace{2r_0 v_0 \cos(\phi)}^{t_c(\phi)}}{(\omega r_0)^2 + 2\omega r_0 v_0 \sin(\phi) + v_0^2} + \theta_T \pmod{2\pi} = \pi - 2 \arctan \left(\overbrace{\frac{\omega r_0 + v_0 \sin \phi}{v_0 \cos \phi}}^{\alpha(\phi)} \right)$$

Wir führen jetzt die dimensionlose Größe $k := \frac{\omega r_0}{v_0}$ ein und schreiben um zu

$$\frac{2k \cos(\phi)}{k^2 + 2k \sin(\phi) + 1} + \theta_T \pmod{2\pi} = \pi - 2 \arctan \left(\frac{k + \sin(\phi)}{\cos(\phi)} \right) \quad (11.9)$$

1) Sonderfall $k \ll 1 \Leftrightarrow v_0 \gg \omega r_0$

Gleichung 11.9 wird dann - wenn wir alle Terme mit k vernachlässigen - zu

$$\theta_T = \pi - 2\phi \Rightarrow \phi = \frac{\pi - \theta_T}{2}$$

Unser Schusswinkel ist also die momentane Position des Zieles - kein Vorhalt. Wir wissen aus der Praxis, das ist zwar nah dran, aber daneben.

Wir versuchen 11.9 zu “linearisieren” an der Stelle $\frac{\pi - \theta_T}{2}$ - alle Potenzen der Ordnung “größer” gleich k^2 werden vernachlässigt!

Bevor wir 11.9 als Ganzes angehen, einige Vereinfachungen der Teilterme:

Wir entwickeln f nach Taylor an der Stelle x_0 und Abweichung $h := (x - x_0)$, Funktionsvariable wird mit *var* bezeichnet - *lin* wie “linearisieren”

(% i1) `lin(f,var,x_0,h):=at(f,[var=x_0])+at(diff(f,var), [var=x_0])*h$`

$$\frac{1}{1 + 2k \sin(\phi)} \approx 1 - 2k \sin(\phi) \quad k^2 \text{ wurde vernachlässigt}$$

(% i2) `t1:lin(1/x,x,1,2*k*sin(phi));` $1 - 2k \sin(\phi)$

$$\arctan \left(\frac{k}{\cos \phi} + \tan \phi \right) \approx k \cos(\phi) + \phi$$

(% i3) `t2:trigsimp(subst(phi, atan(tan(phi)), lin(atan(phi),phi,tan(phi),k/cos(phi))));` $k \cos(\phi) + \phi$

11. Coriolis-Kraft

Später brauchen wir: $\sin\left(2\left(\frac{\pi-\theta}{2}\right)-h\right) \approx \sin(\theta) + 2h \cos(\theta)$

(% i4) `t3:lin(sin(2*x),x,(%pi-θ)/2,-h);` $\sin(\theta) + 2h \cos(\theta)$

Außerdem benötigen wir: $\cos\left(\left(\frac{\pi-\theta}{2}\right)-h\right)$

(% i5) `t4:expand(lin(cos(x),x,(%pi-θ)/2,-h));` $\sin\left(\frac{\theta}{2}\right) + h \cos\left(\frac{\theta}{2}\right)$

Jetzt die Gleichung 11.9 mit eingesetzten Vereinfachungen:

(% i6) `eq1:k*cos(φ)*t1+θ/2=%pi/2-t2;` $k \cos(\phi) (1 - 2k \sin(\phi)) + \frac{\theta}{2} = -k \cos(\phi) - \phi + \frac{\pi}{2}$

Wir fassen die Terme mit $k \cos(\phi)$ zusammen und multiplizieren aus

(% i7) `eq2:expand(k*cos(φ)*(1+t1))+φ=(%pi-θ)/2;`

$$-2k^2 \cos(\phi) \sin(\phi) + 2k \cos(\phi) + \phi = \frac{\pi - \theta}{2} \quad (\text{eq2})$$

Wir ersetzen ϕ durch $(\phi - \theta)/2 - h$ und $\cos(\phi) \sin(\phi)$ durch $1/2 \sin(2\phi)$

(% i8) `eq3: trigsimp(-k^2*sin(2*φ)+(2*k*cos(φ))=h);` $2k \cos(\phi) - k^2 \sin(2\phi) = h$

Wir entwickeln $\cos \phi$ und $\sin(2\phi)$ - wie eingangs berechnet in `t3` und `t4`

(% i9) `eq4:expand(subst(t4,cos(φ),subst(t3,sin(2*φ),eq3)));`

$$-k^2 \sin(\theta) - 2hk^2 \cos(\theta) + 2k \sin\left(\frac{\theta}{2}\right) + 2hk \cos\left(\frac{\theta}{2}\right) = h \quad (\text{eq4})$$

Um in `linsolve` die Variable h zu setzen:

(% i10) `globalsolve:true;`

Wir lösen Gleichung `eq4` nach h :

(% i11) `linsolve(eq4,h);` $\left[h : -\frac{k^2 \sin(\theta) - 2k \sin\left(\frac{\theta}{2}\right)}{2k^2 \cos(\theta) - 2k \cos\left(\frac{\theta}{2}\right) + 1} \right]$

Die Näherungsformel für ϕ

(% i12) `define(φ(θ,k), (%pi-θ)/2-h);`
$$\phi(\theta, k) := \frac{k^2 \sin(\theta) - 2k \sin\left(\frac{\theta}{2}\right)}{2k^2 \cos(\theta) - 2k \cos\left(\frac{\theta}{2}\right) + 1} + \frac{\pi - \theta}{2}$$

Wir überprüfen die Güte unserer Formel (exakt: 0.638)

(% i13) `check_solution(float(φ(%pi/2,0.1));` 0.6323296504638397

Die Güte der linearen Lösungsformel ist nicht berauschend, aber sie funktioniert für $v_0 \gg \omega r_0$ (große Geschwindigkeit, kleines sich langsam drehendes Karussell). Wenn man bis h^2 entwickelt und die entsprechende quadratische Gleichung löst, füllt die Lösungsformel bereits eine A4-Seite!

2) **Sonderfall** $k \gg 1 \Leftrightarrow v_0 \ll \omega r_0$ (“Geschoß” wird fast senkrecht “nach oben” geschleudert)
 Gleichung 11.9 wird dann zu

$$\frac{2k \cos(\phi)}{k^2} + \theta_T \approx \pi - 2\pi/2 = 0 \Rightarrow 2 \frac{\cos(\phi)}{k} + \theta_T = 0$$

Da der Cosinus für $\phi \in [-\pi/2, \pi/2]$ immer positiv ist, kann nur getroffen werden, wenn sich das Ziel am eigenen Standort befindet: $\theta_T = 0 \wedge (\phi = \pi/2 \vee \phi = -\pi/2)$



Die Geschosßbahn ist von einer Sekante zur Tangente mutiert, sodass auch zu einem späteren Zeitpunkt kein Treffer möglich ist!

3) **Sonderfall** $k = 1 \Leftrightarrow v_0 = \omega r_0$: Formel 11.9 wird zu

$$\frac{\cos\phi}{\sin\phi + 1} + \theta_T \pmod{2\pi} = \pi - 2 \arctan\left(\frac{1 + \sin\phi}{\cos\phi}\right) \quad (11.10)$$

mit $a(\phi) := \frac{\cos\phi}{\sin\phi + 1}$ ergibt sich

$$a + \theta_T \pmod{2\pi} = \pi - 2 \arctan\left(\frac{1}{a}\right) \quad (11.11)$$

wobei wenn $\phi : \frac{\pi}{2} \rightarrow -\frac{\pi}{2}$ dann geht $a : 0 \rightarrow \infty$

Das Produkt aus a und ω ist die Kollisionszeit: $\omega a(\phi) = t_c(\phi)$

Je näher ϕ an $-\pi/2$ heranrückt umso länger ($\omega \neq 0$ vorausgesetzt) ist das Geschoß unterwegs - bei endlicher Entfernung bedeutet dies, dass dessen Geschwindigkeit (in Bezug auf das Inertialsystem) immer kleiner wird. Es bewegen sich “Abschusspunkt” und “Ziel-punkt” mit ωr_0 und das Geschoß ist fast in Ruhe.



Für große ωr_0 bedeutet dies, dass im Inertialsystem “Geschoß” und “Werfer/Ziel” ihre Rollen vertauscht haben!

Wenn sich der Vorhaltewinkel ϕ in der Nähe von $-\pi/2$ befindet, führen bereits winzige “Fehler” ($\phi = \phi_0 + \varepsilon$) zu beachtlichen Abweichungen - dies erhöht die Gefahr, dass sich der “Werfer” selbst trifft!

Da wir an großen a interessiert sind, vereinfachen wir Formel 11.11 weiter:

Für $a \gg 1$: $\tan \frac{1}{a} \approx \frac{1}{a} \Rightarrow \arctan \frac{1}{a} \approx \frac{1}{a}$ damit wird 11.11 zu

$$\underbrace{a + \theta_T \pmod{2\pi}}_{f(a)} = \underbrace{\pi - \frac{2}{a}}_{h(a)} \quad \text{für } a > 1 \quad (11.12)$$

11. Coriolis-Kraft

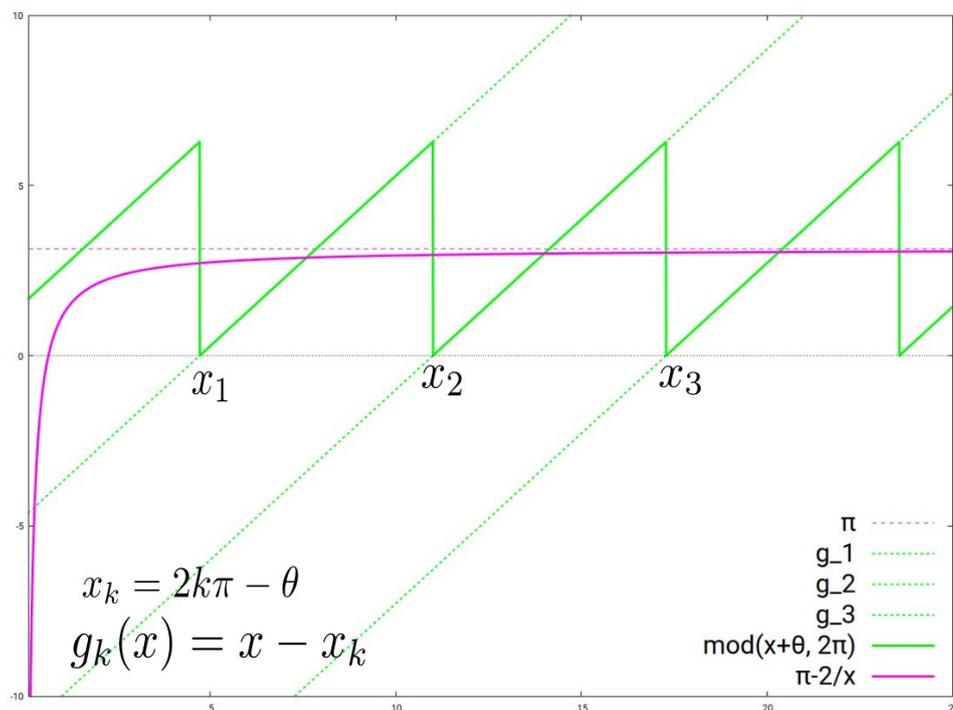


Abb.96 : $f(x) \cap h(x) = \{x | x \in g_k(x) \cap h(x) \wedge x > x_1 \wedge k \in \mathbb{N}\}$

Die Schnittpunkte von f mit dem Hyperbelast von h sind dieselben wie die der Geradenschar $g_k(x) := x - 2k\pi - \theta$ mit h wenn man nur den größeren x -Wert nimmt, also von :

$$g_k(a) = \pi - \frac{2}{a} \quad \text{nur den größeren Lösungswert nehmen!}$$

Damit haben wir die Lösungen a_k von Gleichung 11.12

Bleibt nur das Zurückrechnen von a_k auf ϕ_k :

$$\frac{\cos \phi_k}{1 + \sin \phi_k} = a_k \Big| \cdot^2 \Leftrightarrow \frac{(1 + \sin \phi_k)(1 - \sin \phi_k)}{(1 + \sin \phi_k)(1 + \sin \phi_k)} = (a_k)^2 \Leftrightarrow \sin \phi_k = -\frac{a_k^2 - 1}{a_k^2 + 1}$$

Wir setzen das mit *wxMaxima* um:

```
(% i1) (theta:%pi/2, fpprintprec:5)$
```

Unsere Funktion f der linken Seite von 11.12

```
(% i2) define(f(x), mod(x+theta,2*%pi))$
```

Die horizontal um $2k\pi - \theta$ verschobenen 1. Medianen – f ist Teil davon

```
(% i3) g[k](x):=x-(2*k*%pi-theta)$
```

Formatierung für das Plotten von Abb. 96 – Beachte: KEINE Beistriche!

```
(% i4) p: "set linetype 1 dashtype '-'
      set linetype 2 dashtype ''
      set linetype 3 dashtype ''
      set linetype 4 dashtype ''
      set key font \", 25\"
      set key bottom right "$
```

In `gnuplot_preamble` wird obige Formatierung verwendet

```
(% i5) plot2d([%pi,g[1](x),g[2](x),g[3](x),f(x),%pi-2/x], [x,0.1,25],
             [gnuplot_preamble, p ],[y,-10,10],
             [style, [lines, 1, 4], [lines, 2, 3], [lines, 2, 3], [lines, 2, 3], [lines, 3, 3], [lines, 3, 4] ],
             [legend, "π", "g_1", "g_2", "g_3", "mod(x+θ, 2π)", "π-2/x" ]
             )$
```

Unsere Gleichung 11.12 – quadratisch, nur die größere wird benötigt!

```
(% i6) eq:g[k](x)=%pi-2/x;           $x - 2\pi k + \frac{\pi}{2} = \pi - \frac{2}{x}$  (eq)
```

Die größere Lösung ist eindeutig die zweite

```
(% i7) solutions:(solve(eq,x));
```

$$\left[x = -\frac{\sqrt{16\pi^2 k^2 + 8\pi^2 k + \pi^2 - 32} - 4\pi k - \pi}{4}, x = \frac{\sqrt{16\pi^2 k^2 + 8\pi^2 k + \pi^2 - 32} + 4\pi k + \pi}{4} \right]$$

Wir definieren a_k wie im Text

```
(% i8) define(a[k],rhs(solutions[2]))$
```

```
(% i9) φ(a):=-float(asin((a^2 - 1)/(a^2 + 1)))$
```

```
(% i10) angles:makelist(φ(a[i]),i,1,5);      [-1.3088, -1.4281, -1.4725, -1.4957, -1.5101]
```

Die Ergebnisse sind mit Abb. 95 ident - unsere Näherung hat sich also ganz gut bewährt!

Auch mit *Geogebra* verschafft man sich schnell einen Überblick:

■ $\theta = \pi/2$ (Schieberegler)

■ Liste der verschobenen Medianstücke:

```
g_k= Sequence(If( k * 2π - θ ≤ x ≤ (k + 1) * 2π - θ, x - (k * 2π - θ), k, 1, 10)
```

11. Coriolis-Kraft

- Die Hyperbel und ihre Asymptote : $h(x)=\pi-2/x$ $h_a(x)=\pi$
- Wir schneiden die Liste der Funktionen g_k mit der Hyperbel h :
 $g_kh = \text{Sequence}(\text{Intersect}(h(x), \text{Element}(g_k, k), k*2\pi - \theta, (k+1)*2\pi - \theta), k, 1, 10)$
- Unsere Lösungen von Gleichung 11.12 sind die x -Werte der Liste g_kh : $a_k = x(g_kh)$
- Jetzt die "Sinusse": $s_x = - (a_k*a_{k-1})/(a_k*a_{k+1})$ - die Konvergenz gegen (-1) ist offensichtlich, daher der Winkel gegen $(-\pi/2)$
- $\phi_k = \text{asin}(a_k)$ \rightarrow $\{-1.30882, -1.42812, -1.47246, -1.49572, -1.51007, \dots\}$
 Gute Übereinstimmung mit den Ergebnissen von *wxMaxima*

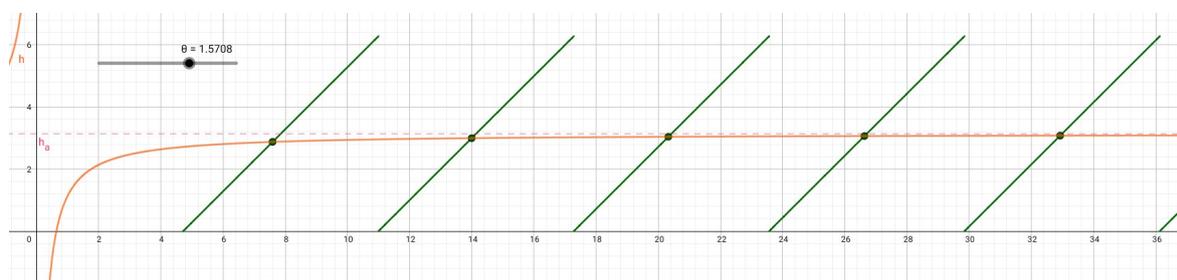


Abb.97 : Schnitt der Sägezahnfunktion f mit der Hyperbel h in *Geogebra*

11.6.3 Fiktive Kräfte im rotierenden Bezugssystem

Sei \hat{n} der Einheitsvektor in Richtung Rotationsachse, $\vec{\omega} := \omega \hat{n}$. Nach Rodriguez (Formel 4.7) mit $\theta := \omega t$ gilt dann (mit der dortigen Nomenklatur und ω konstant)

$$\begin{aligned} \vec{v}_{rot}(t) &= R(\hat{n}, \omega t) \vec{v} = (I + \sin \omega t [\hat{n}]_{\times} + (1 - \cos \omega t) [\hat{n}]_{\times}^2) \vec{v} \\ &= \vec{v} + \sin \omega t \vec{v}_{\times} + (1 - \cos \omega t) \vec{v}_{\times \times} \end{aligned} \quad (11.13)$$

$$\lim_{t \rightarrow 0} \left[\frac{d}{dt} \vec{v}_{rot}(t) \right] = \lim_{t \rightarrow 0} [\omega \cos \omega t \vec{v}_{\times} + \omega \sin \omega t \vec{v}_{\times \times}] = \omega \vec{v}_{\times} = \vec{\omega} \times \vec{v} \quad (11.14)$$

Ich gebe hier die Herleitung wider, wie sie auf der website www.psiquadrat.de von Oliver Passon zu finden ist:

Ein Beobachter in einem rotierenden Bezugssystem drückt Ort, Geschwindigkeit und Beschleunigung mit Hilfe seines Koordinatensystems \hat{e}_i ($i \in \{1, 2, 3\}$) aus (Summenkonvention):

$$\vec{r}(t) = r_i \cdot \hat{e}_i(t) \quad \vec{v}_r = \dot{r}_i \cdot \hat{e}_i(t) \quad \vec{a}_r = \ddot{r}_i \cdot \hat{e}_i(t)$$



Index r bedeutet: "im rotierenden Koordinatensystem"; bei $\vec{r}(t)$ wurde auf den Index verzichtet, das es sich ja um den eindeutigen physikalisch Ort handelt - nur seine Koordinaten variieren je nach Koordinatensystem!

Die Zeitabhängigkeit der Basisvektoren \hat{e}_i gilt natürlich nur im Inertialsystem – im rotierenden System werden sie als konstant wahrgenommen!

In einem Inertialsystem ("außerhalb" der rotierenden Scheibe) ergibt sich

$$\frac{d}{dt}\vec{r}(t) = \dot{r}_i \hat{e}_i(t) + r_i \frac{d}{dt} \hat{e}_i(t) \stackrel{11.14}{=} \vec{v}_r + \vec{\omega} \times r_i \hat{e}_i(t) = \vec{v}_r + \vec{\omega} \times \vec{r}(t) \quad (11.15)$$

$$\frac{d^2}{dt^2}\vec{r}(t) = \ddot{r}_i \hat{e}_i(t) + 2 \dot{r}_i \frac{d}{dt} \hat{e}_i(t) + r_i \frac{d^2}{dt^2} \hat{e}_i(t) = \vec{a}_r + 2 \dot{r}_i (\vec{\omega} \times \hat{e}_i(t)) + r_i \frac{d}{dt} (\vec{\omega} \times \hat{e}_i(t))$$

da $\vec{\omega}$ konstant ist, können wir den letzten Term vereinfachen und es ergibt sich

$$\frac{d^2}{dt^2}\vec{r}(t) = \vec{a}_r + 2\vec{\omega} \times \vec{v}_r + \vec{\omega} \times (\vec{\omega} \times \vec{r}(t))$$

Besteht der Beobachter im rotierenden System auf die Beschreibung seiner Koordinaten und wendet das Newton'sche Kraftgesetz $m \vec{a} = \vec{F}$ an (wir lösen obige Gleichung nach \vec{a}_r auf), ergibt sich

$$m \vec{a}_r = \underbrace{m \frac{d^2}{dt^2} \vec{r}(t)}_{\text{Kraft im Inertialsystem}} \underbrace{- 2m \vec{\omega} \times \vec{v}_r}_{\text{Corioliskraft}} \underbrace{- m \vec{\omega} \times (\vec{\omega} \times \vec{r}(t))}_{\text{Fliehkraft}} \quad (11.16)$$

Mit der Graßmann Identität (Theorem 4.13: $\vec{a} \times (\vec{b} \times \vec{c}) = \vec{b}(\vec{a} \cdot \vec{c}) - \vec{c}(\vec{a} \cdot \vec{b})$)

Merkregel: $BAC - CAB$ - nicht mit Theorem 11.1 verwechseln (Klammersetzung beachten!) lässt sich die Fliehkraft umschreiben

$$-m[\vec{\omega} \times (\vec{\omega} \times \vec{r})] = -m[\vec{\omega}(\vec{\omega} \cdot \vec{r}) - \vec{r}(\vec{\omega} \cdot \vec{\omega})] \stackrel{\vec{\omega} \perp \vec{r}}{=} m \omega^2 \vec{r}$$

Auf einer Scheibe ($\vec{\omega} \perp \vec{r}$) ist also die Fliehkraft radial "nach außen" gerichtet und proportional zur Masse und zur quadratischen Winkelgeschwindigkeit. Auch im Raum können wir die "Scheibenregel" verwenden - siehe Abb. 98

Sei $\vec{\omega} = \omega \vec{\omega}_0 \wedge |\vec{\omega}_0| = 1$ (Beschreibung der Rotation mit Rechtsschraubenregel), dann gilt:

$$\omega^2 \vec{r}_\perp = \omega^2(\vec{r} - \vec{r}_\omega) = \omega^2(\vec{r} - (\vec{r} \cdot \vec{\omega}_0) \vec{\omega}_0) = \vec{r}(\vec{\omega} \cdot \vec{\omega}) - \vec{\omega}(\vec{\omega} \cdot \vec{r})$$

Mit der Graßmann Identität können wir den letzten Term umschreiben in

$$\omega^2 \vec{r}_\perp = -\vec{\omega} \times (\vec{\omega} \times \vec{r})$$

11. Coriolis-Kraft

Auch bei der Corioliskraft können wir \vec{v}_r in Komponenten parallel und senkrecht zur Rotationsachse $\vec{\omega}$ zerlegen:

$$\vec{v}_r = \vec{v}_\omega + \vec{v}_\perp \Rightarrow \vec{\omega} \times \vec{v}_r = \vec{\omega} \times (\vec{v}_\omega + \vec{v}_\perp) \stackrel{\vec{\omega} \parallel \vec{v}_\omega}{=} \vec{\omega} \times \vec{v}_\perp$$



Also nur die radiale Geschwindigkeitskomponente führt zu einer Ablenkung!

Darstellung der Fliehkraft in *Geogebra*

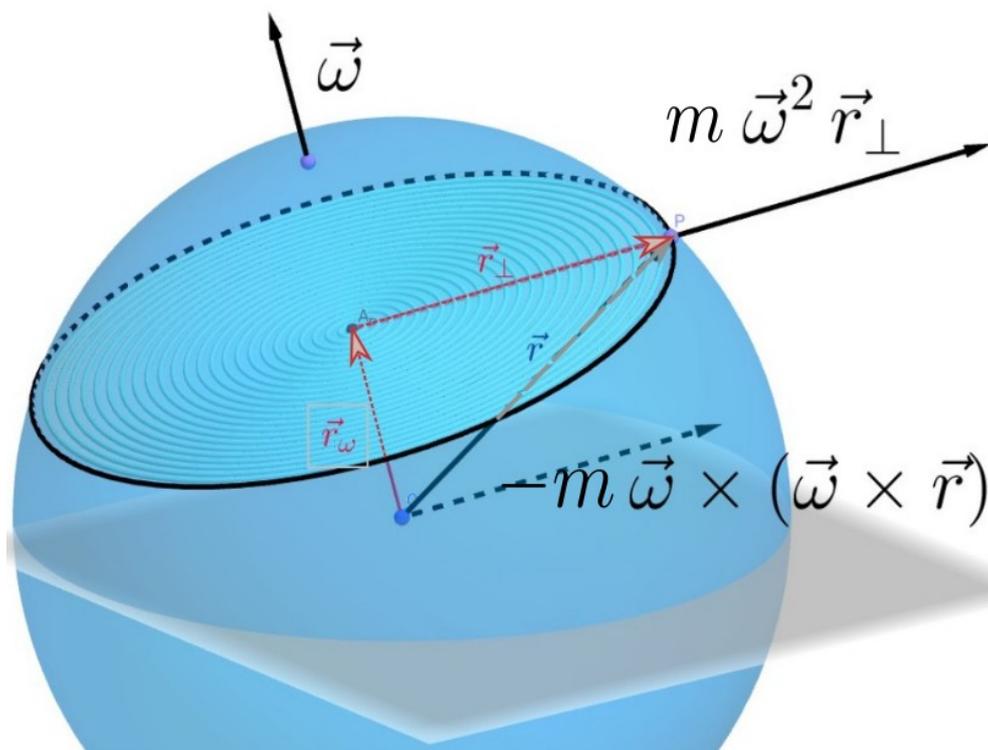


Abb.98 : Fliehkraft im Raum

- Ursprung und Einheitskugel: $O=(0,0,0)$ und $s:x^2+y^2+z^2=1$
- Frei bewegliche Punkte auf s : $P=PointIn(s)$ und $\Omega=PointIn(s)$
- Drehachse und Breitenkreis: $h_1=Line(\Omega,0)$ und $c:Circle(h_1,P)$
- Mittelpunkt des Breitenkreises: $A_P:Midpoint(c)$ anschl. Vektoren $\vec{r}, \vec{r}_\omega, \vec{r}_\perp$
- Vektor $\vec{\omega}$: Slider für Zahl ω und Vektor $\omega_v:\omega*Vector(0,\Omega)$
- Slider für Masse $m < 1$ - dient als Verkürzungsfaktor für die Kraftvektoren!
- Jetzt die Berechnung der Fliehkraftvektoren mit 2 Methoden:

- (a) `f_1=Vector(m*ω^2*r_⊥)` und für die Darstellung bei P ansetzen
 (b) Berechnung mit unserer Formel: `f_2=-m*Cross(ω_v,Cross(ω_v,r))`

■ Um die Scheibe etwas sichtbarer zu machen füllen wir sie mit konzentrischen Kreisen:

- ☛ Punkte von A_P bis zum Kreisrand: `P_i=Sequence(A_P+0.04*i*r_⊥,i,1,25)`
 ☛ Jetzt die Kreise: `Pattern:Sequence(Circle(h_1, Element(P_i, i)), i, 1, 24)`

Bleibt nur die Feinarbeit mit Farben und Stilen(Sichtbarkeit) der einzelnen Elemente.

Darstellung und Berechnung der Corioliskraft in der Tangentialebene

Zuerst einige prinzipielle Überlegungen::

- Ohne Beschränkung der Allgemeinheit legen wir die Rotationsachse auf die z -Achse im mathematisch positiven Sinn. (Bei der Erde ist das der Blick auf die Nordhalbkugel). Die Tangentialebene ε_p sei in einem beliebigen Punkt P der Kugel "verankert".

$$\vec{\omega} = \begin{pmatrix} 0 \\ 0 \\ \omega \end{pmatrix}, \quad \vec{p} = R \begin{pmatrix} \sin \theta \cos \phi \\ \sin \theta \sin \phi \\ \cos \theta \end{pmatrix}$$

Wir verwenden Kugelkoordinaten für die Festlegung von P . θ , ϕ , R und ω werden wir mit Slider festlegen. $\vec{p} = R\vec{p}_0$

- ε_p stattdessen wir mit 2 Basisvektoren \hat{e}_1 (Ostrichtung) und \hat{e}_2 (Nordrichtung) aus:

$$\begin{aligned} \text{Äquator: } eq = R(\cos \phi, \sin \phi, 0) &\Rightarrow \hat{e}_1 = \frac{d}{R d\phi} eq = (-\sin \phi, \cos \phi, 0) \\ \hat{e}_2 = \vec{p}_0 \times \hat{e}_1 &= (-\cos \theta \cos \phi, -\cos \theta \sin \phi, \sin \theta) \end{aligned}$$

- Beliebige Geschwindigkeit in ε_p in 2- bzw. 3 dimensionalen Koordinaten:

$$\begin{aligned} \vec{v}_2 = v(\cos \alpha, \sin \alpha) &\quad v, \alpha \text{ als Slider, } \alpha = 0 \rightarrow \text{Richtung Ost} \\ \vec{v}_3 = v \cos \alpha \hat{e}_1 + v \sin \alpha \hat{e}_2 &\quad \alpha = \pi/2 \rightarrow \text{Richtung Nord} \end{aligned}$$

- Coriolisbeschleunigung \vec{a}_c :

$$\begin{aligned} \vec{a}_c &= -2\vec{\omega} \times \vec{v}_3 = -2\vec{\omega} \times (v \cos \alpha \hat{e}_1 + v \sin \alpha \hat{e}_2) = \\ &= -2v \cos \alpha (\vec{\omega} \times \hat{e}_1) + (-2v \sin \alpha (\vec{\omega} \times \hat{e}_2)) \\ \vec{a}_c &= \lambda_1 (\vec{\omega} \times \hat{e}_1) + \lambda_2 (\vec{\omega} \times \hat{e}_2) \quad \text{mit } \lambda_1 = -2v \cos \alpha, \lambda_2 = -2v \sin \alpha \end{aligned}$$

- Sei \vec{a}_t die Projektion von \vec{a}_c in die Tangentialebene ε_p : $\vec{a}_t = \vec{a}_c - \underbrace{(\vec{a}_c \cdot \vec{p}_0)}_{\nu} \vec{p}_0$

11. Coriolis-Kraft

- Koordinaten (λ_t, μ_t) von \vec{a}_t bezüglich der Basisvektoren \hat{e}_1 und \hat{e}_2 :

$$\begin{aligned} \lambda_t &= \vec{a}_t \cdot \hat{e}_1 = (\vec{a}_c - \nu \vec{p}_0) \cdot \hat{e}_1 \stackrel{\hat{e}_1 \perp \vec{p}_0}{=} \vec{a}_c \cdot \hat{e}_1 = [\lambda_1 (\vec{\omega} \times \hat{e}_1) + \lambda_2 (\vec{\omega} \times \hat{e}_2)] \cdot \hat{e}_1 \\ \lambda_t &= \lambda_2 (\vec{\omega} \times \hat{e}_2) \cdot \hat{e}_1 = \lambda_2 \vec{\omega} \cdot (\hat{e}_2 \times \hat{e}_1) = -\lambda_2 \vec{\omega} \cdot \vec{p}_0 = 2v\omega \sin \alpha \cos \theta \quad (11.17) \\ \mu_t &= \lambda_2 (\vec{\omega} \times \hat{e}_1) \cdot \hat{e}_2 = \lambda_2 \vec{\omega} \cdot \vec{p}_0 = -2v\omega \cos \alpha \cos \theta \Rightarrow \end{aligned}$$

$$\vec{a}_t = 2v\omega \cos \theta \begin{pmatrix} \sin \alpha \\ -\cos \alpha \end{pmatrix} = 2v\omega \sin \varphi \begin{pmatrix} \sin \alpha \\ -\cos \alpha \end{pmatrix} \quad (11.18)$$

geografische Breite φ ersetzt Kugelkoordinatenwinkel θ im zweiten Ausdruck

Wenn sich also der Körper mit $\vec{v}_2 = v(\cos \alpha, \sin \alpha)$ bewegt, dann ist $\vec{a}_t \perp \vec{v}_2$ und zwar liegt \vec{a}_t für $\theta \in [0, \pi/2]$ (Nordhalbkugel, $\varphi \geq 0$) rechts von \vec{v}_2 d.h. die Koordinaten (x, y) von \vec{v}_2 werden durch $(y, -x)$ ersetzt, auf der



Südhalkugel ($\varphi \leq 0$) liegt \vec{a}_t links von \vec{v}_2

$|\vec{a}_t| = \sqrt{\lambda_t^2 + \mu_t^2} = 2v\omega \cos \theta \rightarrow$ keine Abhängigkeit von α und ϕ (geograf. Länge); dies stimmt auch mit der Näherung ?? überein:

$$\Delta y = \frac{|\vec{a}_t|}{2} (\Delta t)^2 \quad \text{mit} \quad \Delta x = v \cdot \Delta t \quad \Rightarrow \quad \Delta y = \frac{\omega}{v} (\Delta x)^2 \sin \varphi$$

Jetzt zum *Geogebra*-Arbeitsblatt

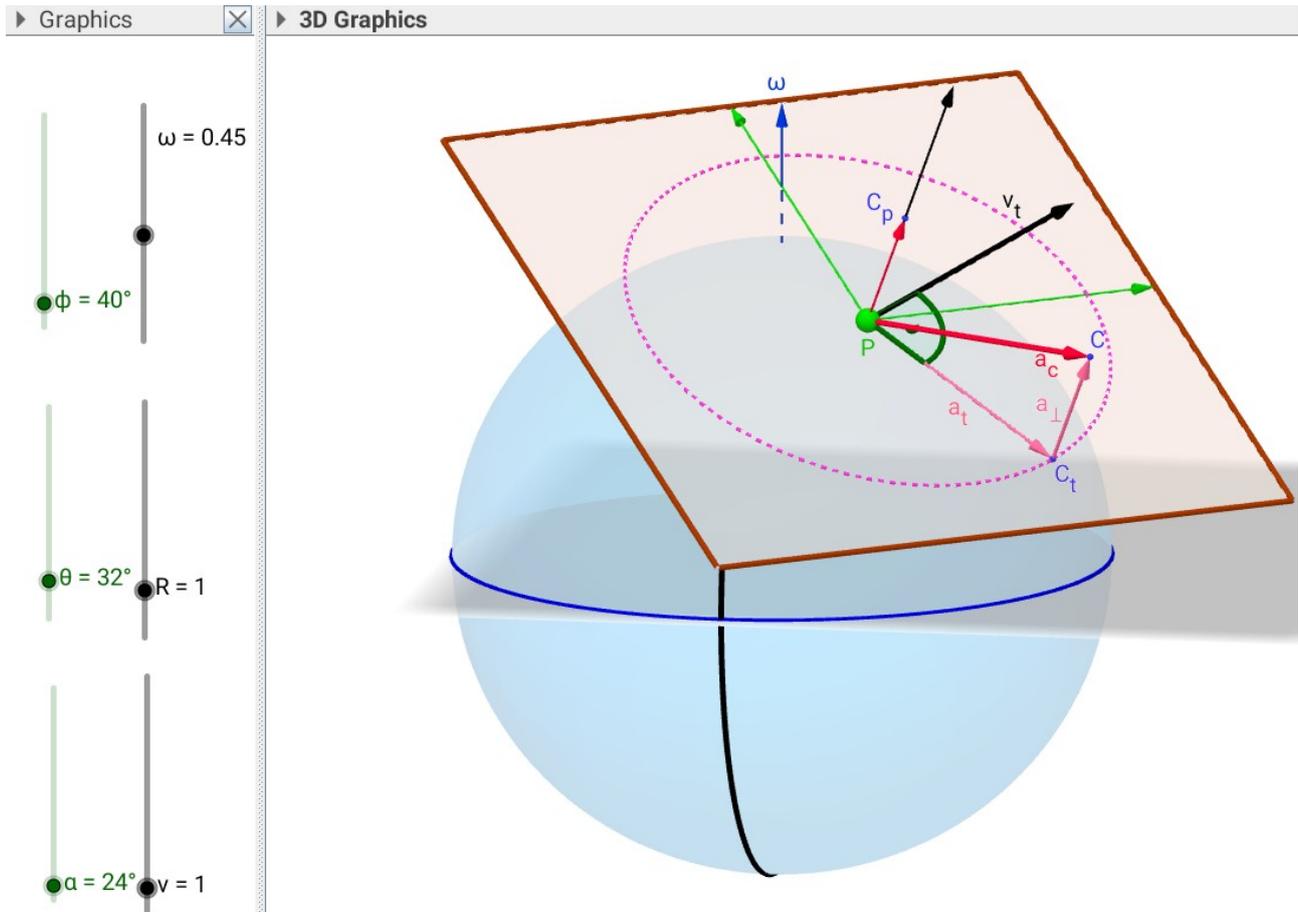


Abb.99 : Corioliskraft in Geogebra

Hier die wichtigsten Konstruktionsschritte:

- Slider für R , v , ω und die Winkel θ , ϕ und α ; Kugel: $x^2+y^2+z^2=R^2$
- Punkte O (Ursprung) und P in Kugelkoordinaten $P = R*(\sin(\theta)*\cos(\phi) \dots)$
- Vektoren: $p_0 = \text{UnitVector}(P)$, $e_1 = \text{Vector}((- \sin(\phi), \cos(\phi), 0))$, $\text{east} = \text{Vector}(P, P+e_1)$, $e_2 = \text{Cross}(p_0, e_1)$, $\omega_v = \text{Vector}((0, 0, \omega))$, $\text{north} = \text{Vector}(P, P+e_2)$
- Geschwindigkeit: $A = P + v * \cos(\alpha) * e_1 + v * \sin(\alpha) * e_2$, $v_t = \text{Vector}(P, A)$
- Coriolisvektor: $a_c = -2 * \text{Cross}(\omega, v_t)$, $C = P + a_c$, $a_{\{PC\}} = \text{Vector}(P, C)$
- Komponenten der Coriolisbeschleunigung:
 $C_p = P + \text{Dot}(a_c, p_0) * p_0$, $a_t = \text{Vector}(P, P + \text{Vector}(C_p, C))$, $a_{\perp} = \text{Vector}(P + a_t, C)$
- Tangentialebene: $\text{Polygon}(P \pm e_1 \pm e_2)$ – 4 Eckpunkte
- Kreis um a_t (Längenmessung) $\text{Circle}(P, \text{Length}(a_t), p_0)$

11. Coriolis-Kraft

Einge Beobachtungen zu den Reglern(Slider)?

- ϕ (geograf. Länge) und R : kein Einfluss auf \vec{a}_t bzw. \vec{a}_\perp
- v : Sowohl \vec{a}_t als auch \vec{a}_\perp sind proportional zu v
- α : $|\vec{a}_t|$ bleibt konstant, weiterhin gilt $\vec{a}_t \perp \vec{v}_t$, $\vec{a}_\perp \propto \cos \alpha$
- ω : Sowohl \vec{a}_t als auch \vec{a}_\perp sind proportional zu ω
- θ : $|\vec{a}_t| \propto \cos \theta$, $|\vec{a}_\perp| \propto \sin \theta$

Übrigens auf die überraschende Beobachtung, dass die Corioliskraft auch eine Auftriebskomponente \vec{a}_\perp besitzt, sind wir bis jetzt kaum eingegangen. Bevor wir diese berechnen noch der Hinweis, dass wir beim Spatprodukt von Vektoren die Operationzeichen vertauschen dürfen (wir haben dies bereits in 11.17 verwendet - Klammern müssen angepasst werden):

$$(\vec{a} \times \vec{b}) \cdot \vec{c} = \vec{a} \cdot (\vec{b} \times \vec{c})$$

Sei a_p die Projektion von \vec{a}_c auf \vec{p}_0 , dann gilt $|a_p| = |\vec{a}_\perp|$. Ist a_p positiv, dann hat \vec{a}_\perp dieselbe Richtung wie \vec{p}_0 , sonst entgegengesetzt:

$$\begin{aligned} a_p &= \vec{a}_c \cdot \vec{p}_0 = [\lambda_1 (\vec{\omega} \times \hat{e}_1) + \lambda_2 (\vec{\omega} \times \hat{e}_2)] \cdot \vec{p}_0 = \lambda_1 (\vec{\omega} \times \hat{e}_1) \cdot \vec{p}_0 + \lambda_2 (\vec{\omega} \times \hat{e}_2) \cdot \vec{p}_0 \\ &= \lambda_1 \vec{\omega} \cdot (\hat{e}_1 \times \vec{p}_0) + \lambda_2 \vec{\omega} \cdot (\hat{e}_2 \times \vec{p}_0) = -\lambda_1 \vec{\omega} \cdot \hat{e}_2 + \lambda_2 \vec{\omega} \cdot \hat{e}_1 \end{aligned}$$

da ω nur eine z -Komponente hat verschwindet der letzte Ausdruck und beim ersten "überlebt" nur die z -Komponente:

$$a_p = 2v\omega \cos \alpha \sin \theta \tag{11.19}$$

Das ist genau was wir am *Geogebra*-Arbeitsblatt sehen, wenn wir mit α über Nord hinausgehen, wird die senkrechte Komponente negativ!

11.6.4 Foucault'sches Pendel

Bei einem Pendel der Länge L sei die gesamte Masse m im Pendelkopf (der Haltefaden sei "nahezu" masselos).

Das Trägheitsmoment ($I = \sum_i^N m_i \cdot r_i^2$) bezüglich der

Drehachse beträgt dann einfach $I = m \cdot L^2$

Die Gravitationskraft $\vec{F}_G = m \vec{g}$ können wir zerlegen:

$$\vec{F}_G = \vec{F}_R + \vec{F}_T \quad \text{wobei} \quad \vec{F}_R = m \vec{g} \sin \theta$$

Für das Drehmoment \vec{M} gilt: $\vec{M} = I \cdot \vec{\alpha}$

(\vec{M} = Trägheitsmoment \times Winkelbeschleunigung)

$\vec{\theta}$ (aus der Zeichenblattebene heraus) ist entgegengesetzt zu \vec{M} ($= \vec{r} \times \vec{F}_R$) - wenn wir dem Pendel nach der Auslenkung keinen Impuls geben, können wir die Vektoren vergessen:

$$m L^2 \ddot{\theta} = -m g \sin \theta \cdot L \quad (11.20)$$

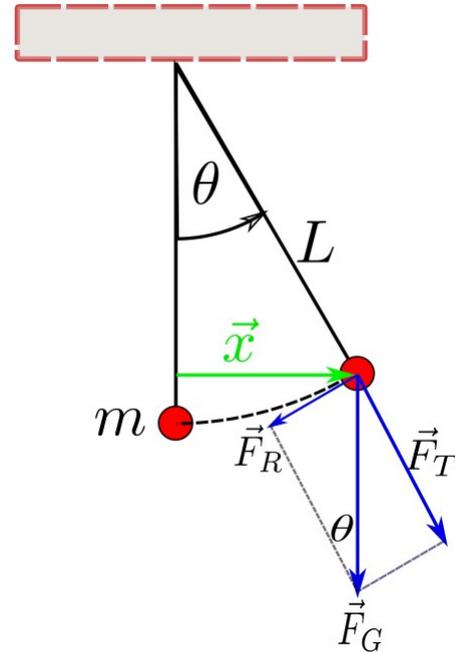


Abb.100 : Kräfte am Pendel durch Gravitation

Für kleine Auslenkungen gilt: $\sin \theta \approx \theta$, damit wird Gleichung 11.20 zur

Bewegungsgleichung des Fadenpendels $\ddot{\theta} + \omega_p^2 \theta = 0$ wobei $\omega_p := \sqrt{\frac{g}{L}}$ (11.21)

Unsere spezielle Lösung lautet

$$\theta(t) = \theta_0 \cos(\omega_p t) \quad \Rightarrow \quad \theta(0) = \theta_0 \wedge \dot{\theta}(0) = 0 \quad (11.22)$$

Bei großer Pendellänge L und kleinem θ (- wie oben vorausgesetzt) gilt:

$$\theta(t) \cdot L \approx |\vec{x}(t)| \quad \text{siehe Abb. 100}$$

Also können wir die Lösung 11.22 umschreiben zu

$$\vec{x}(t) = \vec{x}_0 \cos(\omega_p t) \quad \Rightarrow \quad \ddot{\vec{x}} = -\omega_p^2 \vec{x} \quad \text{Bewegungsgleichung **nur** mit Gravitation}$$

Bewegungsgleichung mit der Coriolisbeschleunigung \vec{a}_t (siehe 11.18) für die Nordhalbkugel:

$$\ddot{\vec{x}} = -\omega_p^2 \vec{x} + 2\omega_E \sin \varphi [\dot{\vec{x}}]_r \quad [u]_r := \vec{u} \text{ nach rechts gedreht um } \pi/2 \quad (11.23)$$

11. Coriolis-Kraft

Wir dröseln obige Vektorgleichung in Koordinaten auf, wobei wir wissen

$$\left[\begin{pmatrix} x \\ y \end{pmatrix} \right]_r = \begin{pmatrix} y \\ -x \end{pmatrix}$$

dann wird aus 11.23 folgendes System von ODE's zweiter Ordnung:

$$\begin{aligned} I \quad \ddot{x} &= -\omega_p^2 x + 2\omega_E \sin \varphi \dot{y} \\ II \quad \ddot{y} &= -\omega_p^2 y - 2\omega_E \sin \varphi \dot{x} \end{aligned}$$

Mit $x_1 := x$ und $x_2 := \dot{x}_1$ (analog für y) verwandeln wir in ein System erster Ordnung und wir starten mit $(x, y) := (1, 0)$ und $(\dot{x}, \dot{y}) := (0, 0)$

$$\begin{aligned} I \quad \dot{x}_1 &= x_2 & x_1(0) &:= 1, x_2(0) := 0 \\ II \quad \dot{x}_2 &= -\omega_p^2 x_1 + (2\omega_E \sin \varphi) y_2 \\ III \quad \dot{y}_1 &= y_2 & y_1(0) &:= 0, y_2(0) := 0 \\ IV \quad \dot{y}_2 &= -\omega_p^2 y_1 - (2\omega_E \sin \varphi) x_2 \end{aligned}$$

Die Änderungsraten unserer Variablen zusammen mit den Anfangsbedingungen reichen für ein numerisches Verfahren aus. Wie auch in anderen Kapiteln (z.B. Räuber-Beute) greifen wir auf das eingebaute Runge-Kutta Verfahren von *wxMaxima* bzw. *lsode* von *GNU-Octave* zurück.

Die Umlaufzeit des Pendels T_U ist am Nordpol gleich der Umlaufzeit der Erde T_E . Am Äquator gibt es keine Corioliskraft also gilt: $T_U \rightarrow \infty$. Ein Ansatz, der beide Bedingungen erfüllt wäre

$$T_U(\varphi) = \frac{T_E}{\sin \varphi} \quad (11.24)$$

Wir überprüfen dies in unserer Rechensimulation für $\varphi = 30^\circ$ nördlicher Breite.

Für Pendellänge $L \approx 40 \text{ m}$ ergibt sich eine Periodendauer $T_p = 4\pi \text{ s} \Rightarrow \omega_p = 1/2 \text{ rad/s}$

Für die Rotationsdauer der Erde nehmen wir als fiktiven Wert das 10-fache der Pendelperiode um die Zeichnung nicht vor lauter Linien unübersichtlich zu machen (ohne natürlich die entstehenden Fliehkräfte zu korrigieren!): $T_E = 40\pi \Rightarrow \omega_E = \omega_p/10$. Bei einer Simulationsdauer von 40π sollte sich nach 11.24 ein halber Pendelumlauf ergeben! Jetzt zur Rechnung mit *wxMaxima*:

Die Angaben wie im Text besprochen

```
(% i1) (\omega_p:1/2, \omega_E:\omega_p/10,\phi:%pi/6)$
```

Nun die 4 Differentialterme für Runge-Kutta

```
(% i2) (x1_dot:x_2, x2_dot: -\omega_p^2*x_1+2*\omega_E*y_2*sin(\phi))$
```

```
(% i3) (y1_dot:y_2, y2_dot: -\omega_p^2*y_1-2*\omega_E*x_2*sin(\phi))$
```

Runge-Kutta: Input→Gleichungen, Vars, Inits, Simulationsintervall bzw. -schrittweite

```
(% i4) points:rk([x1_dot, x2_dot, y1_dot, y2_dot], [x_1,x_2,y_1,y_2],
[1,0,0,0],[t,0,40*%pi,0.1])$
```

Für die Bahn ist nur x_1 und y_1 interessant - außerdem markieren wir Start- und Endpunkt

```
(% i5) trajectory:map(lambda([x], [second(x),fourth(x)]), points)$
```

```
(% i6) (startPoint:trajectory[1], endPoint:last(trajectory))$
```

```
(% i7) plot2d([
  [discrete,trajectory], [discrete,[startPoint] ], [discrete,[endPoint] ]
  ],
  [style, [lines,2,1], 'points, 'points ], same_xy,
  [gnuplot_preamble, "set key bottom right; set xtics font \", 15\";
  set ytics font \", 15\"; set key font \", 15\";
  set title font \", 20\" "],
  [legend,"Foucault-Path","start-point","end-point"])$
```

11. Coriolis-Kraft

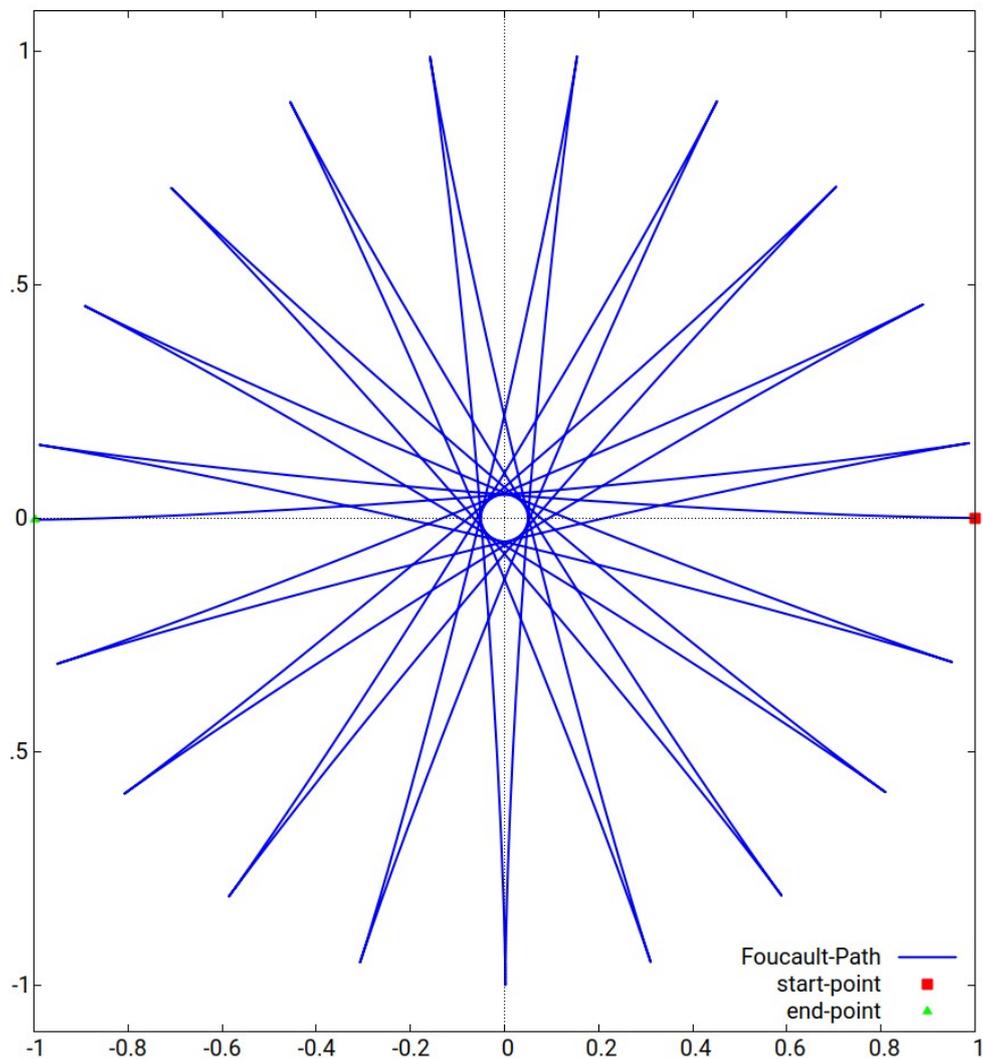


Abb.101 : Bahn des Pendels für $\omega_E = \omega_p/10$ bei $\varphi = 30^\circ$, $t_{sim} = [0, T_E] \Rightarrow T_U = 2T_E$

Hier das das *GNU-Octave* Programm dazu

```

1 # think of: the first keyword should not be *function* in the main program
2 # so we start with some definitions
3
4 # initial values for x_1,x_2,y_1 und y_2 as vector
5 initVec=[1;0;0;0];
6 t=0:0.1:40*pi; # time vector for evaluation
7
8 # called by **lsode(x,t)** — must have
9 # input: x is vector of present state, t is simulation-time-vector
10 # output: changing-rate of x-vector
11 function xdot=nextStep(x,t)
12 # Parameter of model equations
13 o_p=1/2; # omega pendulum
14 o_E=o_p/10; # omega of earth

```

```

16  lat=pi/6; # geographic latitude
# ----- the equations right side -----
18  xdot(1)=x(2); #die Gleichungen
    xdot(2)=-o_p*o_p*x(1) + 2*o_E*sin(lat)*x(4);
18  xdot(3)=x(4); #die Gleichungen
20  xdot(4)=-o_p*o_p*x(3) - 2*o_E*sin(lat)*x(2);
endfunction

22
##### — WORKING HORSE —#####
24 solMat=lsode("nextStep", initVec, t);
# first column of solutionMatrix: x_1,
26 # second column: (dot x_1), third column:y_1, fourth column: (dot y_1)

28 # we extract columns x_1, y_1 for trajectory plot
xp=solMat(1:end,1);
30 yp=solMat(1:end,3);
# axis should have same unit-length, so a circle is drawn as a circle
32 axis equal;
#we connect the points with lines
34 line(xp,yp, 'LineWidth', 2, "color", 'blue');
hold on;
36 # start-point is plotted
s=scatter(xp(1),yp(1),150,"r","filled", 'MarkerEdgeColor',[0 .1 .9],...
38         'MarkerFaceColor',[1 .1 .1],...
         'LineWidth',1.5 );
40
# plot end
42 scatter(xp(end),yp(end),100,"g","filled");
# make the plot more readable
44 title ({ "Trajectory of Pendulum with Coriolis-Force" }, "fontsize",20);
legend("trajectory","start-point","end-point");
46 set(legend,"fontsize",15);
xlabel('x axis', 'FontSize', 20);
48 ylabel('y axis', 'FontSize', 20);
set(gca, 'FontSize',18);

```

Auf der nächsten Seite sehen Sie die Ausgabe des Programms:

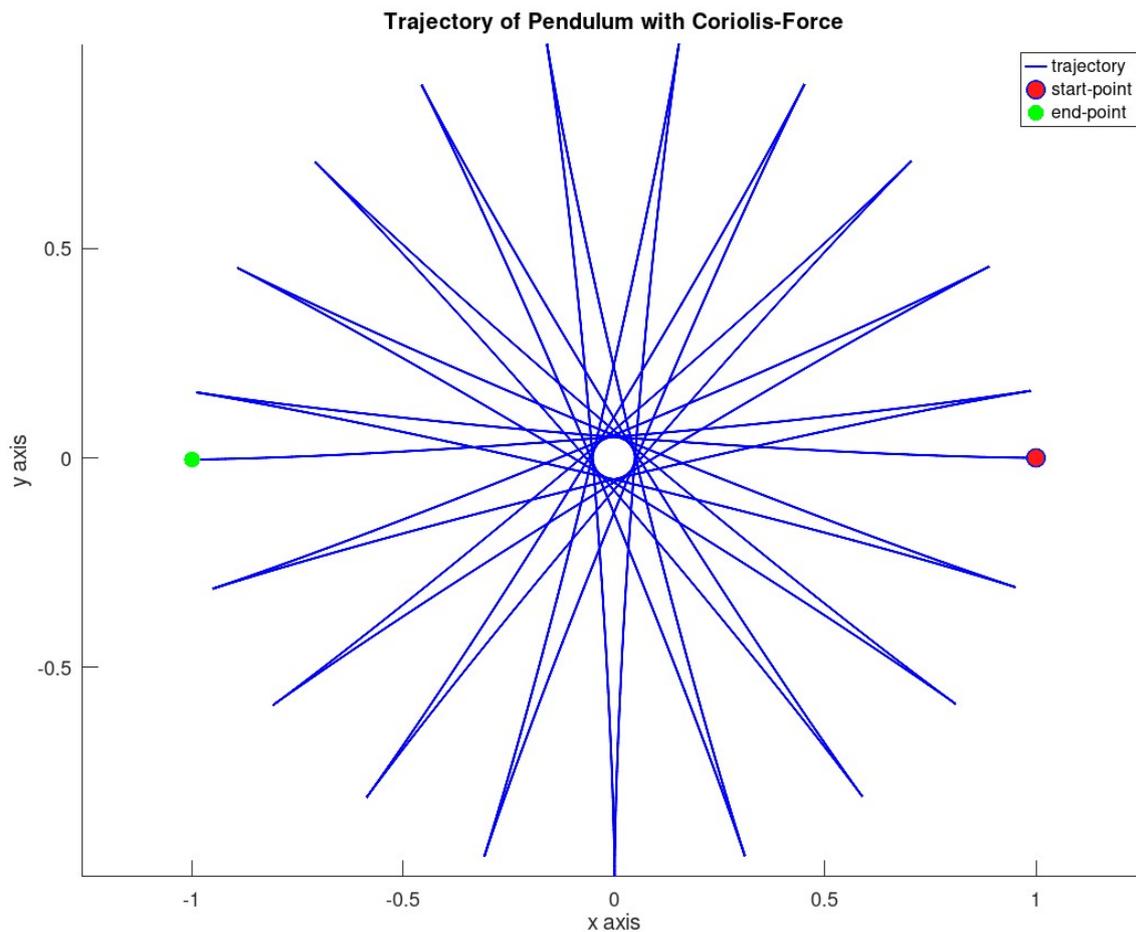


Abb.102 : Bahn des Pendels für $\omega_E = \omega_p/10$ bei $\varphi = 30^\circ$, $t_{sim} = [0, T_E] \Rightarrow T_U = 2T_E$

11.7 Rollende Kugel auf einer Scheibe

In diesem Abschnitt wollen wir uns mit der Bewegung einer rollenden Kugel auf einer rotierenden Scheibe beschäftigen. Dieses Experiment ist auf <https://www.youtube.com/watch?v=4luexfFQ5JQ> zu sehen. Im Wesentlichen spiralt die Kugel um ein Zentrum - bis sie schließlich über den Rand der Scheibe fällt. Wir wollen versuchen, dieses ungewöhnliche Verhalten (vergleiche mit einer liegenden Münze) zu berechnen.

Bevor wir uns der Dynamik dieser Bewegung zuwenden. Ein Theorem aus der Vektoralgebra, welches wir später benötigen.

Theorem 11.1

$$(\vec{a} \times \vec{b}) \times \vec{c} = (\vec{a} \cdot \vec{c}) \vec{b} - (\vec{b} \cdot \vec{c}) \vec{a}$$

Beweis: Mit der 3 dimensionalen ε - δ -Beziehung (siehe Wikipedia). Es gilt die Einstein'sche Summenkonvention:

$$\varepsilon_{ijk} \varepsilon_{ilm} = \delta_{jl} \delta_{km} - \delta_{jm} \delta_{kl}$$

damit ist die l -te Komponente des obigen Doppelvektorprodukts

$$\varepsilon_{lim} \underbrace{(\varepsilon_{ijk} a_j b_k)}_{(\vec{a} \times \vec{b})_i} c_m = -\varepsilon_{ilm} \varepsilon_{ijk} a_j b_k c_m = (\delta_{jm} \delta_{kl} - \delta_{jl} \delta_{km}) a_j b_k c_m = a_m c_m b_l - a_l b_m c_m$$

Die letzte Zeile in Vektorschreibweise ergibt die Behauptung! \square

Jetzt zur Dynamik des Balls:

Unser KS liegt im Laborsystem, die rotierende Scheibe in der $x - y$ -Ebene, folgende Größen benötigen wir:

Scheibe(Turntable): Winkelgeschwindigkeit $\vec{\omega} = \vec{k}\omega = \mathbf{const}$; Trägheitsmoment $I_T = \infty$

Ball: Position \vec{r} ($r_z = 0$), $\dot{\vec{r}} = \vec{v}$, $\vec{\Omega}$, m , $\vec{R}_B = \vec{k} R_B$, I_B

- Zuerst die Rollbedingung (Non slipping condition):

$$\vec{v} = \vec{\omega} \times \vec{r} + \vec{\Omega} \times \vec{R}_B \quad \text{wobei gilt } v_z = 0 \quad (11.25)$$

- Wir leiten 11.25 ab und benutzen, dass $\dot{\vec{\omega}} = 0$ und $\dot{\vec{R}}_B = 0$

$$\dot{\vec{v}} = \vec{\omega} \times \vec{v} + \dot{\vec{\Omega}} \times \vec{R}_B \quad (11.26)$$

- Summe der Drehmomente führt zur Winkelbeschleunigung; die Reibungskraft setzen wir proportional zu \vec{v} und zum Gewicht (reine Rollreibung (rolling friction) wäre zwar vom Betrag der Geschwindigkeit unabhängig und man müsste \vec{v} durch \vec{v}_0 ersetzen, dass entstehende Differentialgleichungssystem ist dann aber schwer zu knacken. Außerdem wenn sich bei der Lösung der Betrag der Geschwindigkeit kaum ändert, läuft das nur auf eine Anpassung von C_F hinaus):

$$\vec{M} = (m\dot{\vec{v}} - C_F mg\vec{v}) \times \vec{R}_B = I_B \dot{\vec{\Omega}} \quad (11.27)$$

Da sowohl $\dot{\vec{v}}$ als auch \vec{v} Vektoren der x-y-Ebene sind, ist dies auch $\dot{\vec{v}} - \vec{v}$, das Kreuzprodukt mit \vec{R}_B ergibt wieder einen Vektor der x-y-Ebene $\Rightarrow \dot{\Omega}_z = 0$
 $\Rightarrow \Omega_z$ ist eine Erhaltungsgröße!

- Jetzt wird 11.27 mit $\dots \times \vec{R}_B$ multipliziert, mit Theorem 1.1, $\dot{\vec{v}} \perp \vec{R}_B$ und $\vec{v} \perp \vec{R}_B$ bekommen wir

$$\begin{aligned} & \left((m\dot{\vec{v}} - C_F mg\vec{v}) \times \vec{R}_B \right) \times \vec{R}_B = I_B \dot{\vec{\Omega}} \times \vec{R}_B \\ & - \left(\dot{\vec{v}} - C_F g\vec{v} \right) \frac{mR_B^2}{I_B} = \dot{\vec{\Omega}} \times \vec{R}_B \end{aligned} \quad (11.28)$$

11. Coriolis-Kraft

- Obiges Ergebnis für $\dot{\vec{\Omega}} \times \vec{R}_B$ eingesetzt in 11.25 und $\dot{\vec{v}}$ explizit gemacht, ergibt das Differentialgleichungssystem

$$\dot{\vec{v}} = G \vec{\omega} \times \vec{v} + H \vec{v} \quad (11.29)$$

wobei gilt:

$$0 < G = \frac{I_B}{I_B + mR_B^2} < 1 \quad 0 < H = \frac{C_F g}{1 + \frac{I_B}{mR_B^2}} \ll 1$$

Da bei H die Rollreibungszahl C_F sehr klein (ca. bei $C_F \approx 0.002$, $g \approx 10$) und der Nenner größer als 1 ist, könnte in erster Näherung der H -Term weggelassen werden.

Wir schreiben 11.29 in Komponenten auf, wobei $v_3 = 0 \wedge \omega_G = G\omega_3$ gilt:

$$\begin{aligned} I \quad \dot{v}_1 &= H v_1 - \omega_G v_2 \\ II \quad \dot{v}_2 &= H v_2 + \omega_G v_1 \end{aligned} \quad (11.30)$$

Wir lösen dieses System mit Ableiten von I und einsetzen von II und anschl. I :

$$\ddot{v}_1 = H\dot{v}_1 - \omega_G \dot{v}_2 = H\dot{v}_1 - \omega_G [H v_2 + \omega_G v_1] = H\dot{v}_1 - \omega_G \left[H \frac{Hv_1 - \dot{v}_1}{\omega_G} + \omega_G v_1 \right]$$

das ergibt eine gewöhnliche Differentialgleichung 2.-ter Ordnung:

$$\ddot{v}_1 - 2H\dot{v}_1 + (\omega_G^2 + H^2) v_1 = 0 \quad (11.31)$$

Anfangsbedingungen:

$$v_1(0), \dot{v}_1(0) = H v_1(0) - \omega_G v_2(0)$$

Die nichttriviale Lösung von Gleichung 11.31 ist

$$v_1(t) = e^{Ht} (C_1 \cos \omega_G t + C_2 \sin \omega_G t)$$

Wir überprüfen unsere Lösung in *wxMaxima*:

(%i1) `v_1(t):=exp(H*t)*(C_1*cos(omega*t)+C_2*sin(omega *t));`

$$v_1(t) := \exp(Ht) (C_1 \cos(\omega t) + C_2 \sin(\omega t)) \quad (\%o1)$$

(%i2) `define(dv_1(t),diff('v_1(t),t));`

$$dv_1(t) := \frac{d}{dt} v_1(t) \quad (\%o2)$$

(%i3) `define(ddv_1(t),diff(dv_1(t),t));`

$$ddv_1(t) := \frac{d^2}{dt^2} v_1(t) \quad (\%o3)$$

(%i4) `diffEq:ddv_1(t)-2*H*dv_1(t)+(omega^2+H^2)*v_1(t);`

$$\frac{d^2}{dt^2} v_1(t) - 2H \left(\frac{d}{dt} v_1(t) \right) + (\omega^2 + H^2) v_1(t) \quad (\text{diffEq})$$

Die Diff-Gleichung wird mit ihren "noun"-Formen (Ableitungen) evaluiert - und verschwindet

(%i5) `diffEqEv:trigsimp(ev(diffEq,nouns));`

$$0 \quad (\text{diffEqEv})$$

Über 11.30 Gleichung I wird $v_2(t)$ bestimmt

(%i6) `define(v_2(t),ratsimp(ev((H*v_1(t)-dv_1(t))/omega,nouns)));`

$$v_2(t) := C_1 e^{Ht} \sin(\omega t) - C_2 e^{Ht} \cos(\omega t) \quad (\%o6)$$

In der letzten Zeile haben wir gleich $v_2(t)$ berechnet, sodass jetzt die vollständige nichttriviale Lösung lautet:

$$\begin{aligned} v_1(t) &= e^{Ht} (C_1 \cos(\omega_G t) + C_2 \sin(\omega_G t)) \\ v_2(t) &= e^{Ht} (C_1 \sin(\omega_G t) - C_2 \cos(\omega_G t)) \end{aligned}$$

mit den Anfangswerten $v_1(0) = \dot{x}_0 = C_1$ und $v_2(0) = \dot{y}_0 = -C_2$ ergibt sich

$$\begin{aligned} v_1(t) &= e^{Ht} (\dot{x}_0 \cos(\omega_G t) - \dot{y}_0 \sin(\omega_G t)) \\ v_2(t) &= e^{Ht} (\dot{x}_0 \sin(\omega_G t) + \dot{y}_0 \cos(\omega_G t)) \end{aligned} \quad (11.32)$$

Integrieren liefert dann schließlich die Bahn des Balls:

11. Coriolis-Kraft

$$\begin{aligned}
 x_1(t) &= \frac{(C_1 \omega_G + C_2 H) e^{Ht} \sin(\omega_G t) + (C_1 H - C_2 \omega_G) e^{Ht} \cos(\omega_G t)}{\omega_G^2 + H^2} + x_0 \\
 x_2(t) &= -\frac{(C_2 \omega_G - C_1 H) e^{Ht} \sin(\omega_G t) + (C_2 H + C_1 \omega_G) e^{Ht} \cos(\omega_G t)}{\omega_G^2 + H^2} + y_0 \quad (11.33)
 \end{aligned}$$

Für $C_1 = C_2 = 0$ würde sich wieder die triviale Lösung ergeben!

Hier beispielsweise 2 Bahnen:

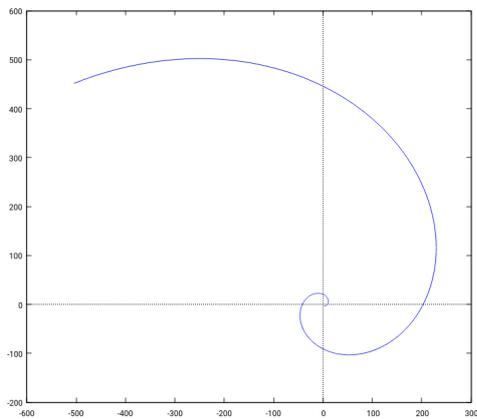


Abb.103 : Friction is high: $H = 0.05$

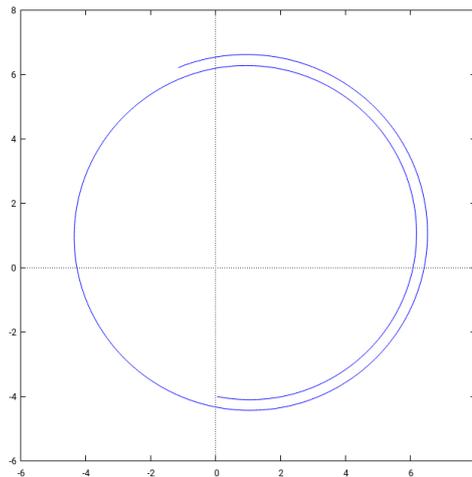


Abb.104 : Friction is low: $H = 0.001$

12 | Sonnenscheindauer

Natürlich ist hier mit “Sonnenscheindauer” nicht die meteorologische Sonnenscheindauer, sondern die astronomische Sonnenscheindauer gemeint - also wenn keine atmosphärische Hindernisse existieren (Wolken, Nebel, Staub, ...). Außerdem treffen wir einige Vereinfachungen in unserem Modell gegenüber der Realität.

12.1 Vereinfachungen

- Die *Erde* (bzw. Schwerpunkt Erde-Mond) bewegt sich mit gleichförmiger Geschwindigkeit auf einer Kreisbahn (Koordinatensystem wie in Abb. 1) - wir ignorieren also die Erkenntnisse von Kepler völlig.
- Die Sonne wird als Punkt angenommen (Schätze die Auswirkung auf Sonnenauf- und Sonnenuntergang ab!)
- Wir berücksichtigen nicht die atmosphärische Lichtbrechung (man sieht die Sonne schon “bevor sie aufgegangen” bzw. sieht sie noch “obwohl sie schon untergegangen” ist - ca. 5 Minuten)
- Wir berücksichtigen nicht, dass die Erdachse gegenüber dem Fixsternhimmel nicht ganz stabil ist (Präzessionsbewegung der Rotationsachse!)
- Auch die Lage des Perihels ändert sich im Lauf der Jahrhunderte
- Wir berücksichtigen erst recht nicht den Einfluss der großen Planeten Jupiter und Saturn auf die Erdbahn

Trotz dieser vielen Vereinfachungen wird unsere Näherung ganz gut sein. Da wir übrigens eine *Zeitdauer* angeben, spielt die Synchronisation der Uhrzeit (was unsere Uhr anzeigt, z.B. Sommerzeit oder überhaupt andere Zeitzone) mit der “lokalen Sonnenzeit” keine Rolle!



Recherchiere im Internet: große und kleine Halbachse der Erdbahnellipse: a , b
Berechne den mittleren Abstand $(a + b)/2 = r_{av}$; $(a - r_{av})/a$ in Prozent

12.2 Terminologie und Rechnung

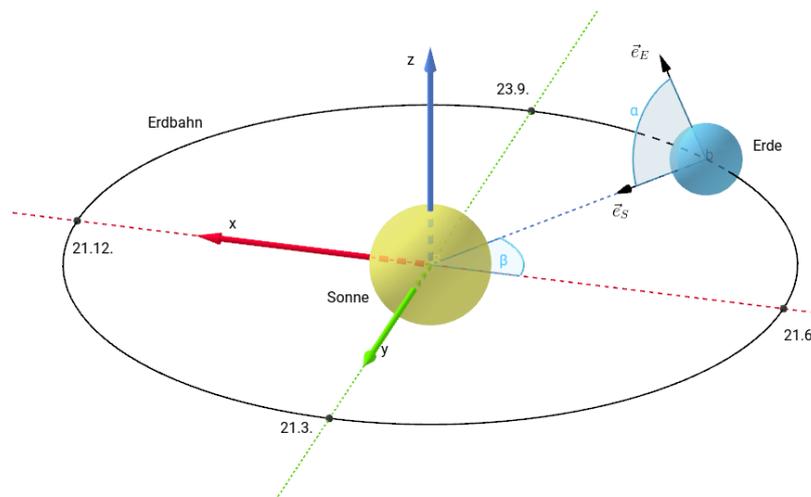


Abb.105 : Koordinatensystem - Bahn der Erde

1. Sei n der n -te Tag im Jahr. Berechne β in Abhängigkeit von n !

$$\beta = (n - 172) \cdot \frac{360^\circ}{365}$$

2. Die Erdachse bildet mit der Ekliptik einen Winkel von $\alpha_0 = 66,5^\circ$! Bestimmen Sie die Koordinaten des Vektors \vec{e}_E im angegebenen Koordinatensystem !

$$\vec{e}_E = (\cos \alpha_0 | 0 | \sin \alpha_0)$$

3. Bestimmen Sie die Koordinaten des Vektors \vec{e}_S im angegebenen Koordinatensystem !

$$\vec{e}_S = (\cos \beta | \sin \beta | 0)$$

4. Berechnen Sie den Winkel α zwischen den Vektoren \vec{e}_E und \vec{e}_S !

$$\alpha(n) = \arccos(\cos \alpha_0 \cdot \cos \beta(n)) \quad (12.1)$$

5. Wir betrachten nun ein neues Koordinatensystem, in dem die Sonne in $\pi_3(xz\text{-Ebene})$ steht und die Erdachse in z -Richtung!(Ist dies immer möglich? Siehe Kapitelanhang!)

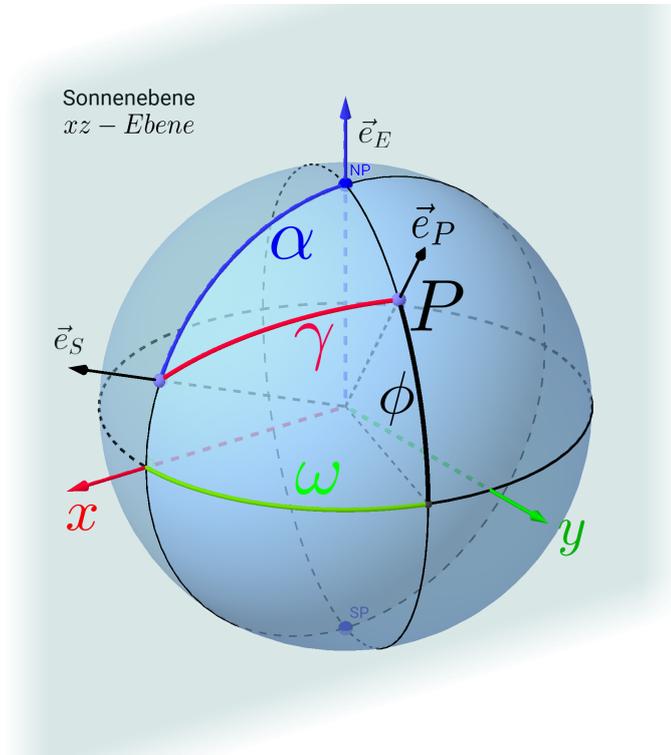


Abb.106 : Neues KS mit $\vec{e}_E = z\text{-Achse}$

6. Jeder Punkt der Erdoberfläche dreht sich in diesem Koordinatensystem täglich einmal auf einem Kreis um die z -Achse herum. Genau um 12 Uhr Mittag (Ortszeit) durchquert er π_3 . Die Koordinatendarstellung von \vec{e}_S in diesem KS ist :

$$\vec{e}_S = (\sin \alpha | 0 | \cos \alpha)$$

7. Wir betrachten nun einen beliebigen Punkt P (mit geografischer Breite ϕ und einem Stundenwinkel ω vom Mittagsmeridian entfernt) - wie lautet der Einheitsvektor vom Erdmittelpunkt durch P:

$$\vec{e}_P = (\cos \phi \cos \omega | \cos \phi \sin \omega | \sin \phi)$$

12. Sonnenscheindauer

8. Falls der Winkel $\gamma = 90^\circ$ geht die Sonne in P unter. ω ist daher die Zeit zwischen Mittag und Sonnenuntergang, wobei $2\pi = 24$ Stunden entsprechen. Die Zeit zwischen Sonnenauf- und Sonnenuntergang ist daher $2 \cdot \omega$! Also:

$$\vec{e}_P \cdot \vec{e}_S = 0 \Rightarrow \omega$$

9. Geben Sie nun die astronomische Sonnenscheindauer T_A als Funktion von ϕ und n an:

$$T_A(\phi, n) = \frac{24}{\pi} \arccos(-\tan \phi \cot \alpha(n)) \quad (12.2)$$

Für $\alpha(n)$ ist natürlich obige Formel einzusetzen. Wir können nun diese Formel in *Geogebra* (mit Schieberegler für ϕ und n und Spur von $T_A(\phi, n)$) veranschaulichen!

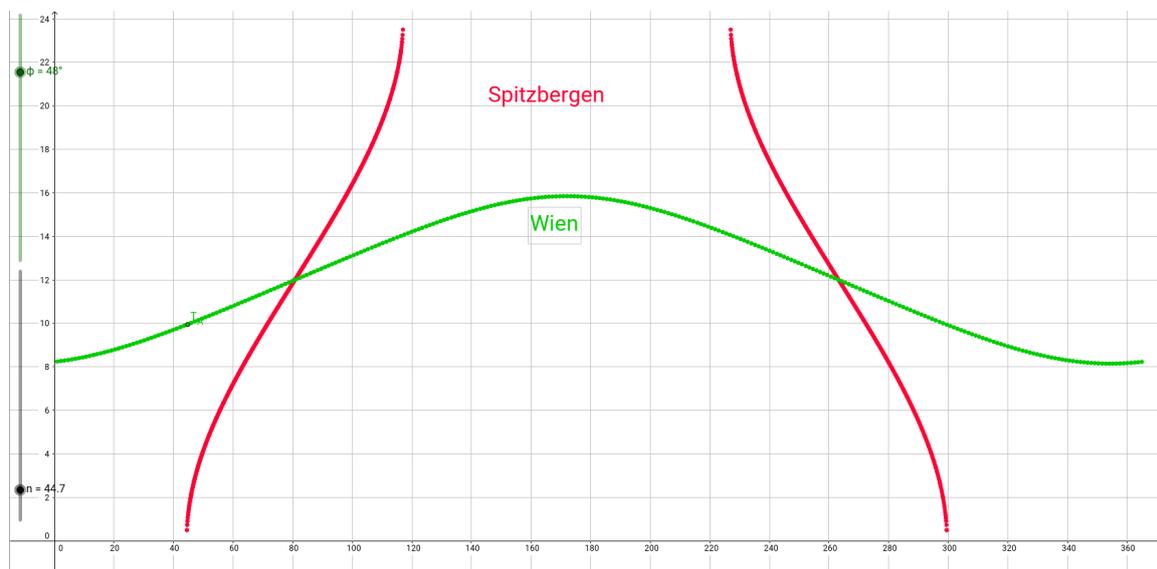


Abb.107 : Graph von $T_A(\phi, n)$

Für Wien (48 Grad Nord) bzw. Spitzbergen (ca. 76 Grad Nord - nördlich des Polarkreises) ist die Spur von $T_A = (n, T_A(\phi, n))$ gezeichnet (wobei natürlich $T_A(\phi, n)$ durch den Ausdruck in 12.2 zu ersetzen ist). Die beiden Graphen schneiden sich in den Äquinoken, für Spitzbergen erkennt man die Polarnacht bzw. den Polartag, Symmetrie des Graphen um den 21. Juni, usw.



Die Stadt Hammerfest liegt nördlich des Polarkreises. Recherchiere aus Wikipedia seine geografische Breite. Mit obiger Formel 12.2 bestimme Beginn und Ende des Polartages bzw. der Polarnacht - wie lange dauern diese Abschnitte?

Wie lange dauert die Periode, in der die Sonnenscheindauer unter 6 Stunden liegt?

Wie lange ist dieser Zeitabschnitt für die chilenische Stadt Punta Arenas?

12.3 Anhang

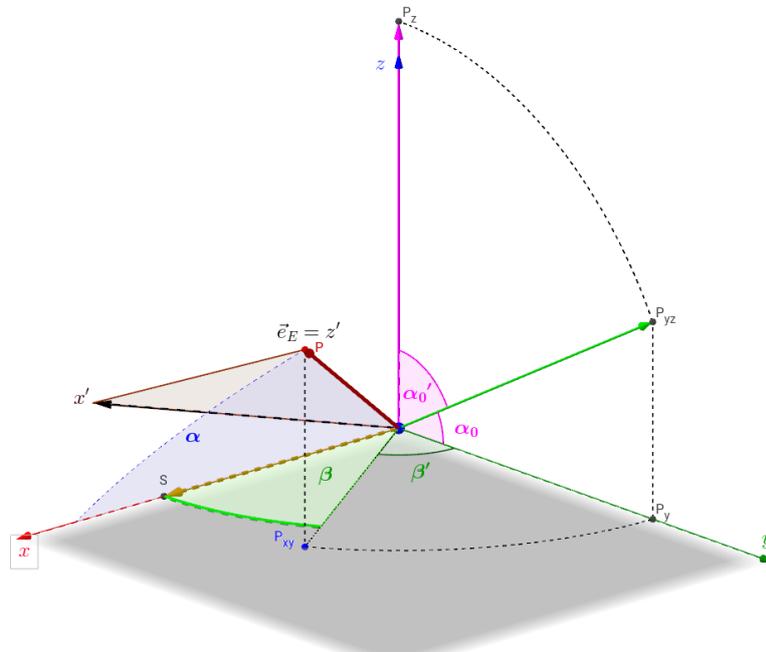


Abb.108 : Yes - we can!

KS in der Abb. 2 ist möglich! Sei die xy -Ebene die Ekliptikebene, die x -Achse in Richtung Sonne(S) und \vec{e}_E der Erdachsenvektor (er vollführt in diesem KS in einem Jahr eine Drehung mit β).

Mit den Euler-Winkeln $\beta' = \pi/2 - \beta$ und $\alpha_0' = \pi/2 - \alpha_0$ können wir $\vec{e}_E (= \vec{e}_3')$ in die z -Achse (\vec{e}_3) drehen:

$$R_x(\alpha_0') (R_z(\beta') \vec{e}_E) = \vec{e}_3 \quad \Rightarrow \quad \vec{e}_E (= \vec{e}_3') = \underbrace{R_z(-\beta') R_x(-\alpha_0')}_T \vec{e}_3 \quad \Rightarrow \quad \vec{e}_i' = T \vec{e}_i$$

Jetzt bleibt nur mehr eine Drehung um \vec{e}_3' bis S in der $x'z'$ -Ebene "einrastet"!

13 | Zeitgleichung

Grundsätzlich wäre es möglich, dass wir alle eine "Uhr" tragen, die eine gleichmäßig(?) größer werdende Zahl zeigt. Als Nullpunkt nehmen wir ein Ereignis z. B.: das erste Erscheinen der Supernova 1054 (Krebsnebel) - ja und das war's. Dies ist der Gedanke der UTC - gemessen wird mit Atomuhren.

Aber durch Rotationen der Erde sollte die Uhr gewisse Periodizitäten widerspiegeln - Tage und Jahreszeiten! Immer wenn unsere Uhr 12 (warum ausgerechnet diese Zahl) anzeigt, sollte die Sonne möglichst hoch am Himmel stehen. Wenn noch dazu Oktober ist, sollten die Laubbäume in gemäßigten nördlichen Breiten eine bunte Farbe haben.

Wir werden sehen, dass wir das "bei 12 Uhr Mittags(Lokalzeit) steht die Sonne am höchsten" nicht ganz hinkriegen, denn der Sonnenhöchststand (12 Uhr bei einer Sonnenuhr) schwankt wegen der Schiefstellung der Erdachse (Tilt) und der Keplerbewegung der Erde(keine glm. Umlaufgeschwindigkeit). Die Korrektur zwischen Sonnenuhrzeit (tatsächlicher Stand der Sonne) und unserer Uhrzeit (Lokalzeit) wird "**Zeitgleichung**" genannt. Dazu kommt noch, dass unsere Lokalzeit von einem bestimmten Meridian stammt (alle ca. 15 Grad sind eine Zeitzone mit territorialer Berücksichtigung) und daher der Abstand von diesem "zeitgebenden" Meridian zu berücksichtigen ist!

Bevor wir uns genauer mit dem "Lauf der Erde um die Sonne" (oder ist's doch umgekehrt?) beschäftigen, einige Dinge, die wir herausgefunden haben:

$$a = 152.1 \cdot 10^6 \text{ km} \quad b = 147.1 \cdot 10^6 \text{ km} \quad \kappa = 0,016722, \quad \alpha = 78,5^\circ, \quad \epsilon = 23,45^\circ \quad (13.1)$$

- a große Halbachse der Ellipse
- b kleine Halbachse der Ellipse
- e lineare Exzentrizität (Abstand Zentrum zu Brennpunkt)
- ϵ Abweichung der Erdachse von der Senkrechten zur Ekliptik
- κ Exzentrizität (dimensionslos) = $\frac{e}{a}$
- α Winkel zwischen Vernal Equinox und Perihel (schwankt zw. 79 und 76)
meist ist hier in den Tabellen ca. 282° angegeben - von Vernal Equinox zu Perihel im Umlaufsinn gemessen (Longitude of Perihelion ω)

Die schlechte Nachricht zuerst: Leider ist auch die Keplernäherung nicht exakt. Unser Sonnensystem ist kein 2 Körperproblem (Erdmond, Jupiter, Saturn & Co sorgen für allerlei Störungen

13. Zeitgleichung

- stellen Sie sich vor, Erdmond, Mars, Jupiter und Saturn bilden eine Linie auf derselben Seite der Erde - da trägt es die Erde ein wenig aus der Kurve!). Übrigens “actio = reactio” führt auch dazu, dass die Sonne auch “wackelt” - sich der Brennpunkt der Ellipse also auch ein wenig verschiebt! Diese Störungen bzw. Allgemeine Relativitätstheorie sorgen für Periheldrehung (keine geschlossene Ellipse), außerdem ist die Erde kein Massenpunkt - dies führt zu Gezeitenkräften wie Präzession der Erdachse (“Taumeln wie ein Kreisel”) und Verlangsamung der Erdrotation durch Reibung der Gezeiten. Also genau genommen ist die Sache ein wenig komplex (understatement).

Die gute Nachricht: Die Keplernäherung ist nicht so schlecht. Die Fehler liegen im einstelligen Sekundenbereich - damit kann man leben.

Wer es ganz genau haben will (Originalton):

There are certainly far better models than the 6-parameter elliptical one. Your best bet for very accurate positioning is an ephemeris like the JPL DE or VSOP. These models provide very long series with literally thousands of terms, which you must compute at a given point in time to get the value of a parameter. They supply such series for a wide range of orbital parameters in various coordinate systems and reference frames. In particular, VSOP87 claims an accuracy of around ± 4 km for Earth – and this is one of the older ones. A basic elliptic approximation cannot come close to this, no matter what parameters you choose.

Here are computed distances of Sun and Earth in the years 2001-2010 - notice no simple periodicity can be seen!

Distance to sun, km			
Min	Date	Max	Date
147,097,497	2001-01-04 09:15	152,087,573	2001-07-04 14:00
147,098,060	2002-01-02 14:30	152,094,389	2002-07-06 04:00
147,102,639	2003-01-04 05:15	152,100,377	2003-07-04 06:00
147,094,327	2004-01-04 18:00	152,095,209	2004-07-05 11:15
147,099,112	2005-01-02 00:45	152,102,378	2005-07-05 05:15
147,103,625	2006-01-04 15:45	152,095,745	2006-07-03 23:30
147,093,631	2007-01-03 20:00	152,097,044	2007-07-07 00:15
147,096,603	2008-01-02 23:59	152,104,160	2008-07-04 07:59
147,095,552	2009-01-04 15:44	152,091,131	2009-07-04 01:59
147,098,040	2010-01-03 00:29	152,096,447	2010-07-06 11:44
147,097,909	average	152,096,445	average

Abb.109 : “exakte” Daten der Entfernungen Erde-Sonne

Bevor wir uns allerdings in die Tiefen der Berechnungen stürzen, einige “Basic-Facts” über die Erdumlaufbahn.

13.1 Sterntag vs. Sonnentag

Zeigt ein Planet immer mit demselben Punkt zur umlaufenden Sonne (kein Sonnentag), so legt er doch während eines Umlaufs bereits einen Sterntag zurück.

Anzahl der Sterntage n_{st} ist um 1 größer als die Anzahl der Sonntage n_{sun} :

$$n_{st} = n_{sun} + 1$$

Folgende Zeichnung demonstriert diesen Sachverhalt für $n_{st} = 5$ also 4 Sonntage bei gleichmäßigen Umlauf und senkrechter Umlaufachse (zur Umlaufebene):

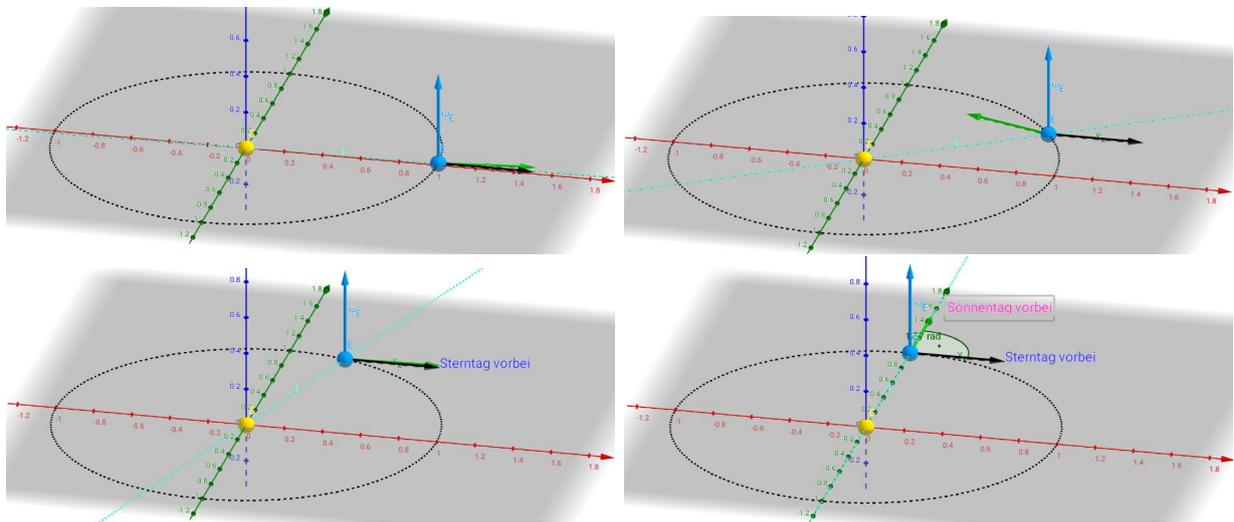


Abb.110 : Sonnentag vs. Sterntag $\Delta T = \frac{1}{4} T_{st}$

Eine entsprechende Animation für *Geogebra* ist auf meinem Server

<https://www.angsuesser.at/docs/math/geogebra/sunVSstarDay1.ggb> zum Herunterladen!

Sei T_{st} die Sternaumlauftzeit (Sterntag) und T_{sun} die Dauer des Sonntages, so gilt:

$$n_{st} \cdot T_{st} = n_{sun} \cdot T_{sun} = 1 \text{ year} \Rightarrow T_{st} = \left(1 - \frac{1}{n_{st}}\right) T_{sun} \quad (13.2)$$

$$T_{sun} - T_{st} = \Delta T = \frac{T_{sun}}{n_{st}} \stackrel{\text{Erde}}{\downarrow} \frac{24 \cdot 60}{366.25} \text{ min} \approx 3.93 \text{ min} \quad (13.3)$$

Für die obere Zeichnung ($n_{st} = 5$) gilt:

$$\frac{T_{sun} - T_{st}}{T_{st}} \stackrel{13.3}{\downarrow} \frac{T_{sun}}{n_{st} T_{st}} \stackrel{13.2}{\downarrow} \frac{T_{sun}}{n_{sun} T_{sun}} = \frac{1}{4}$$

13. Zeitgleichung

13.2 6 wichtige Punkte in der Erdumlaufbahn

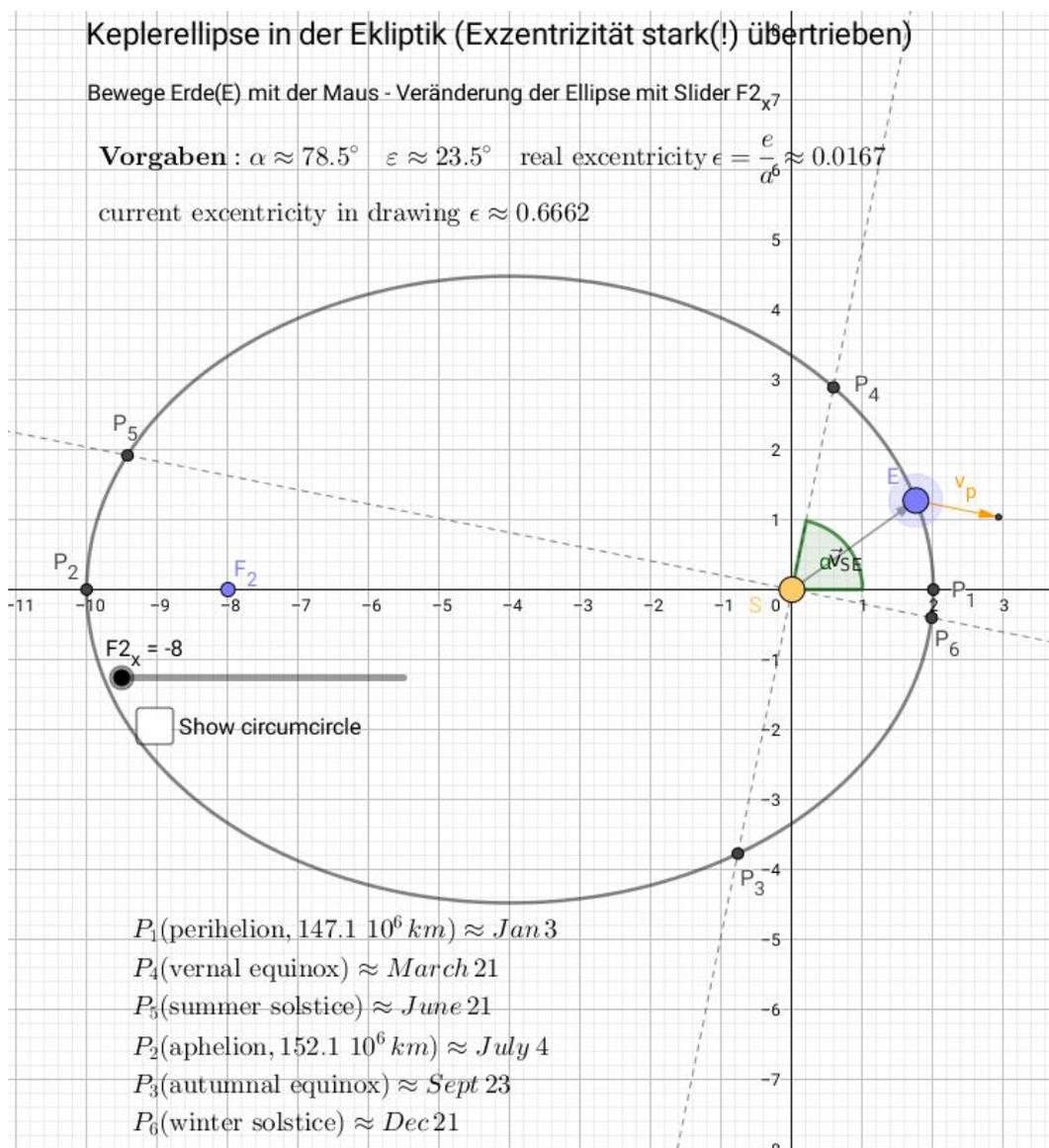


Abb.111 : 6 wichtige Punkte in Geogebra 2D

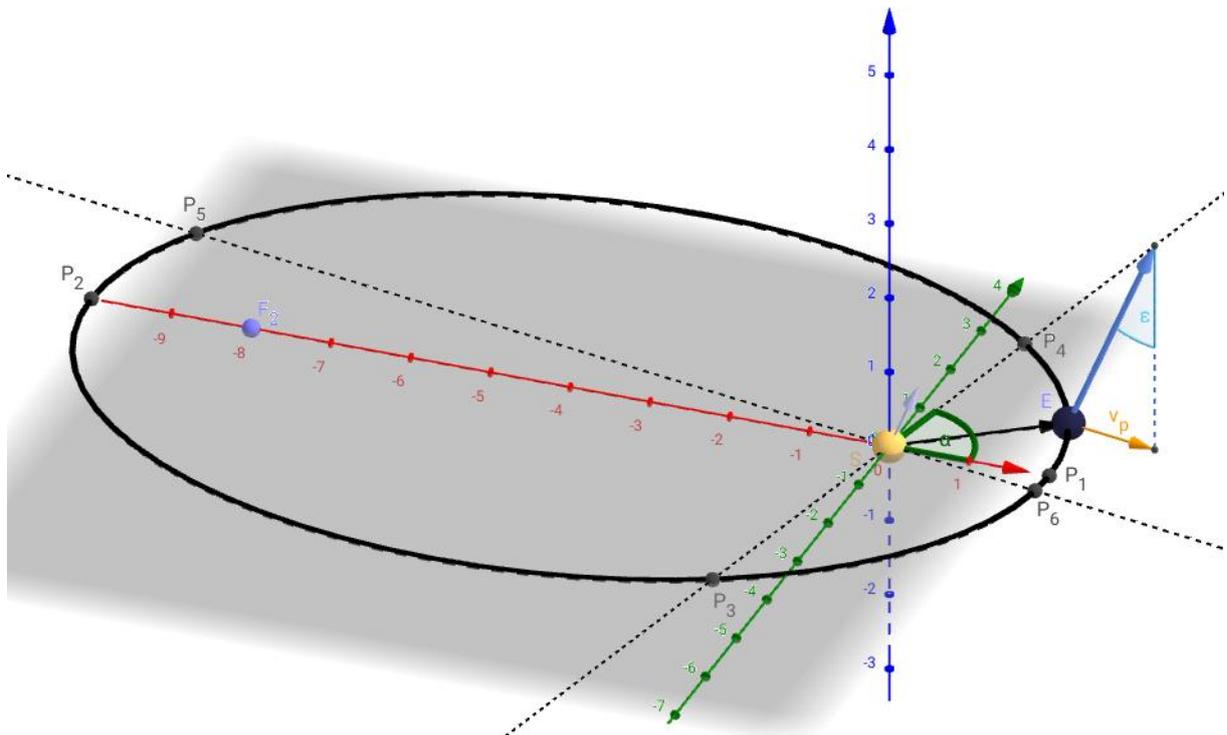
Das vollständige Arbeitsblatt ist unter

<https://www.angsuesser.at/docs/math/geogebra/sixImportantPoints2.ggb>

zum Herunterladen. Eine Animation kommt der Realität nicht nahe, weil ja laut Kepler sich die Geschwindigkeit der Erde ändert.



Beachte, dass die Exzentrizität der Ellipse stark übertrieben ist - sie kann man mit dem Schieberegler $F2_x$ an die Realität heranführen. Zum Vergleich lässt sich auch der Umkreis sichtbar machen!

Abb.112 : 6 wichtige Punkte in *Geogebra* 3D

13.2.1 Erstellung des Arbeitsblattes

- Zuerst wird mit Slider $F2_x$ für Punkt F_2 , Punkt F_1 mit Caption "S" (sun) und einem Punkt P_{ell} die Ellipse mit dem Ellipsenwerkzeug festgelegt.
- α und ε werden als Zahlen eingegeben
- Mit $g_1 = \tan(\alpha) \cdot x$ und $g_2 \perp g_1$ bekommen wir die Equinoxen und Sonnenwenden
- Die Kugelkoordinaten der Erdachse \vec{a} sind $(r, \phi, \theta) = (1, \alpha, \varepsilon)$ also im kartesischen System: $(\sin(\varepsilon) \sin(\alpha), \sin(\varepsilon) \cos(\alpha), \cos(\varepsilon))$ als Einheitsvektor
- Für die Projektion in die x-y-Ebene \vec{a}_{xy} verschwindet die z-Koordinate (mit Faktor 3 skaliert wegen Sichtbarkeit)
- Diese Vektoren werden jetzt bei der Erde eingezeichnet und beschriftet
- Momentane Exzentrizität wird berechnet *eps*
- x_{min} - Minimalwert für den Schieberegler (damit Exzentrizität realen Wert annimmt) wird festgelegt
- Diverse "Verzierungen"



Wichtig ist es zu erkennen, dass bei den Equinoxen der Projektionsvektor der Erdachse \vec{v}_p senkrecht zur Richtung Erde-Sonne ist und bei den Sonnenwenden parallel. (Daher bilden diese Achsen ein kartesisches Koordinatensystem!)

13.3 Die “mittlere Sonne” (mean sun)

Sehen wir von der Schiefstellung (Tilt) der Erdachse ab und davon, dass sich (laut Kepler) die Erde verschieden schnell bewegt, ergibt sich - von der Erde aus gesehen (wir wechseln vom heliozentrischen Ekliptiksystem zum geozentrischen Ekliptiksystem) - eine Sonne, die wie ein Uhrwerk gleichmäßig um die Erde läuft. Diese mittlere Sonne wird für die Festsetzung der lokalen Uhrzeit herangezogen. Die mittlere Sonne hat selbstverständlich dieselbe Umlaufzeit (1 Jahr) wie die reale Sonne, aber zwischendrin gibt es Abweichungen - für diese Abweichungen ist der Name “Zeitgleichung” gebräuchlich. Diese Abweichung zwischen 12 Uhr mittags auf der Uhr (Lokalzeit) und Sonnenhöchststand (auf der Sonnenuhr) zu ermitteln ist unser Ziel.

Unser Ziel ist also die *Zeitgleichung* zu finden!

Wir werden dieses Ziel im geozentrischen Ekliptiksystem angehen - also brauchen wir die Transformationsmatrix für die Umrechnung der Koordinaten.

Das *Geogebra* - Arbeitsblatt auf <https://www.angsuesser.at/docs/math/geogebra/meanSun.ggb> veranschaulicht dies “step by step”. (Rotationen Kap. 5)

Man benötigt

- Eine Translation zur Erdposition und anschl.
- eine Rotation um die z -Achse mit π (Umlaufsinn spielt keine Rolle!)



Führt man zuerst die Rotation durch, bekommt die Erde **neue** Koordinaten und die Translation ist zu diesen neuen Koordinaten durchzuführen - also keine “einfache” Vertauschung möglich!

Orbit of the Mean Sun

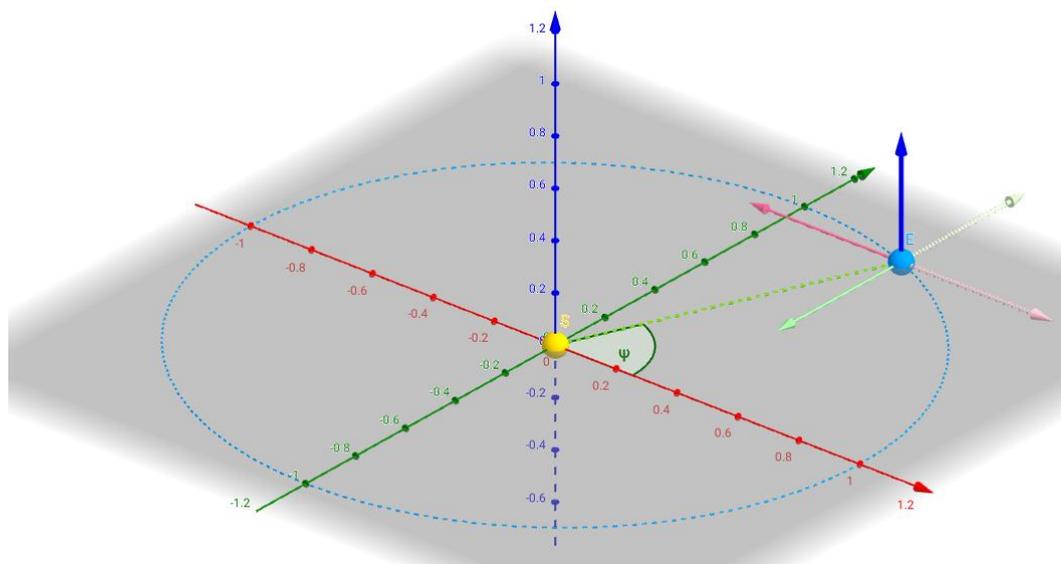


Abb.113 : Translation und Rotation ausgeführt

Das Arbeitsblatt selbst ist einfach "gestrickt" und sollte vom Konstruktionsprotokoll leicht verständlich sein. Mit der Sichtbarkeit der einzelnen Elemente wurde etwas "gespielt". Die Animation selbst wird mit einem Befehl im **OnKlick**-Reiter des Buttons **Start** ausgelöst.

Wir bestimmen jetzt mit *wxMaxima* die Koordinaten im geozentrischen Ekliptiksystem der "Referenzsonne" (sprich Uhrzeit):

Rotation um die z -Achse und Translation für homogene Koordinaten

```
(% i1) R_z(%alpha):=matrix([cos(%alpha),-sin(%alpha),0,0],
[sin(%alpha),cos(%alpha),0,0],
[0,0,1,0],
[0,0,0,1])$
```

```
(% i2) T(x_T,y_T,z_T):=matrix([1,0,0,x_T],
[0,1,0,y_T],
[0,0,1,z_T],
[0,0,0,1])$
```

Wegen *passiver* Transformation die inversen Matrizen nehmen; Reihenfolge von rechts nach links

```
(% i3) Tr: R_z(-%pi) . T(-cos(%psi),-sin(%psi),0) ;
```

$$\begin{pmatrix} -1 & 0 & 0 & \cos(\psi) \\ 0 & -1 & 0 & \sin(\psi) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (\text{Tr})$$

```
(% i4) S:Tr . [0,0,0,1]$
```

Das Ergebnis (dieselben Koordinaten wie im anderen KS die der Erde) hätten wir natürlich schneller haben können

```
(% i5) display(S)$
```

$$S = \begin{pmatrix} \cos(\psi) \\ \sin(\psi) \\ 0 \\ 1 \end{pmatrix}$$

Nachdem $\psi_s = \omega_s \cdot t$ mit $\omega_s = \text{const} \in \mathbb{R}$ - bewegt sich diese Sonne wie die Zeiger einer gleichmäßig laufenden Uhr!



Beachte außerdem, dass im ekliptischen KS die x -Achse mit der Richtung zum Frühlingspunkt (vernal equinox) übereinstimmt - **nicht** mit dem Perihel!

13. Zeitgleichung

13.3.1 Nachbetrachtung zur “mittleren” Sonne

Dadurch dass die mittlere Sonne und die Erde **parallele Rotationsachsen** besitzen, ist die Zeitdauer (wie immer man sie misst!) zwischen 2 Sonnentagen immer gleich lang - gleichgültig an welcher Stelle der “Umlaufbahn” sich die Sonne befindet, gleichgültig wie schnell (Hauptsache gleichmäßig!) die Erde rotiert.

Es ist genau diese Eigenschaft, die die mittlere Sonne zu einer perfekten Uhr macht.

Diesen Sachverhalt kann man sich leicht in *Geogebra* veranschaulichen - wobei hier beim Bild der Verlust der Dynamik besonders schmerzlich ist (Download des Arbeitsblattes mit <https://www.angsuesser.at/docs/math/geogebra/sunDays.ggb>):

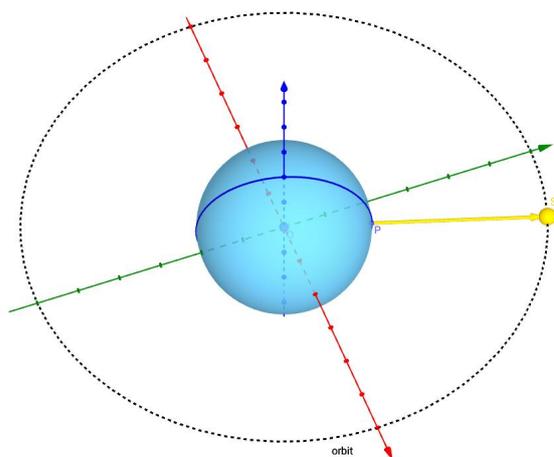


Abb.114 : mittlere Sonne als Uhr

Das Arbeitsblatt ist so einfach gestrickt, dass man mit Hilfe des Arbeitsprotokolls leicht nachvollziehen kann, wie es aufgebaut ist.

Zu beachten ist nur (wie schon vorher erwähnt), dass bei 1 Umdrehung der Sonne bei d_s Sonnentagen die Erde $d_s + 1$ -mal (Anzahl der Sterntage) rotiert!

Wir überprüfen kurz durch Rechnung, dass die mittlere Sonne am Ende des k -ten Sonnentages genau wieder senkrecht über dem rotierenden Punkt P der Erde steht:

Als Zeiteinheit nehmen wir 1 Jahr (Umlaufzeit $T = 1$). Außerdem wissen wir, dass die Sterntage um 1 höher sind als die Sonnentage: $d_{st} = d_s + 1$ - dann gilt:

$$t_k = \frac{k}{d_s} \quad k \in \{0, 1, 2 \dots [d_s]\} \quad \text{Zeit für Ende des } k\text{-ten Sonnentages}$$

$$\omega_s \cdot \overbrace{1}^T = 2\pi \quad \alpha_s(t) = \omega_s \cdot t$$

$$\omega_P \cdot 1 = \underbrace{(d_s + 1)}_{d_{st}} 2\pi \quad \alpha_P(t) = \omega_P \cdot t$$

Es sollte gelten:

$$\alpha_P(t_k) - \alpha_s(t_k) \stackrel{!}{=} k \cdot 2\pi$$

Durch Einsetzen kann man sich von der Richtigkeit überzeugen!

13.4 Die fiktive Sonne

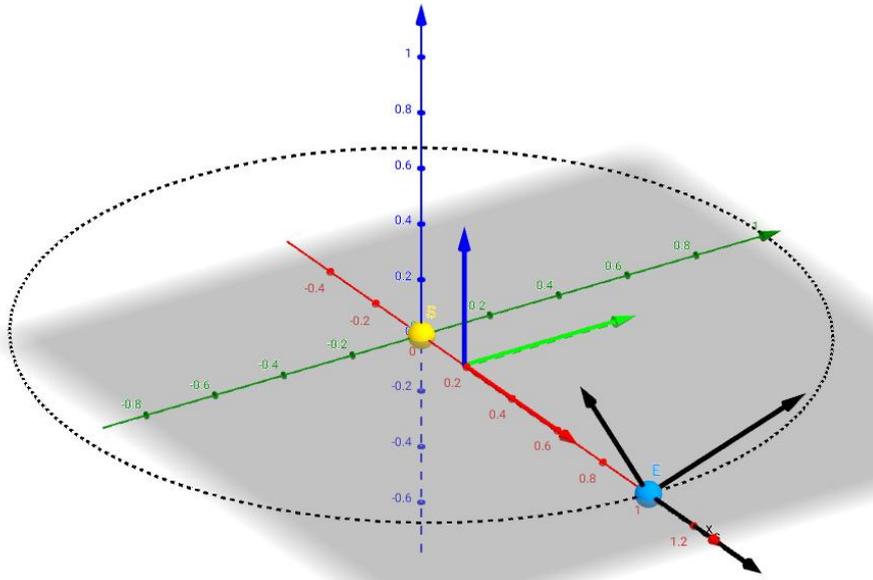


Abb.115 : Translation dann eine Rotation um die x -Achse um ε - Blick auf Nordhalbkugel

So es wird Zeit, dass wir die Erdachse schieflegen und zwar um den Winkel $\varepsilon = 23.4^\circ$ (momentan abnehmend um 47 Bogensekunden pro Jahrhundert). Für die Erstellung der Transformationsmatrix ins "Erdsystem" kommt jetzt eine Drehung um ε um die x -Achse dazu (siehe Abb. 115), wenn man Sonne-Frühlingspunkt als x -Achse wählt, Senkrechte zur Ekliptikebene z -Achse. Erdachse hat im Frühlingspunkt die Koordinaten $(0, -\sin \varepsilon, \cos \varepsilon)$.

Im beginnenden Sommerhalbjahr der nördlichen Hemisphäre neigt sich die Erdachse der Sonne zu!

Will man es dynamisch haben, dann

<https://www.angsuesser.at/docs/math/geogebra/fictitiousSun.ggb>
besuchen!

Wir berechnen die Bahn der fiktiven Sonne in *wxmaxima*:

Abkürzung für ε und Definition der Rotationsmatrizen

(% i1) eps:%epsilon\$

(% i2) R_z(%alpha):=matrix([cos(%alpha),-sin(%alpha),0,0], [sin(%alpha),cos(%alpha),0,0], [0,0,1,0], [0,0,0,1])\$

(% i3) R_x(%alpha):=matrix([1,0,0,0], [0,cos(%alpha),-sin(%alpha),0], [0,sin(%alpha),cos(%alpha),0], [0,0,0,1])\$

(% i4) T(x_T,y_T,z_T):=matrix([1,0,0,x_T], [0,1,0,y_T], [0,0,1,z_T], [0,0,0,1])\$

Gesamt-Transformationsmatrix: Beachte die negativen Vorzeichen (passive Transformation!)

(% i5) Tr: R_z(-%pi) . R_x(-eps) . T(-cos(%phi),-sin(%phi),0) ;

13. Zeitgleichung

Die Koordinaten der Sonne (ursprünglich im Ursprung) im Erdsystem

(% i7) (Sun:Tr . [0,0,0,1], display(Sun))\$

$$Sun = \begin{pmatrix} \cos(\phi) \\ \sin(\phi) \cos(\epsilon) \\ \sin(\phi) \sin(\epsilon) \\ 1 \end{pmatrix} \quad (13.4)$$

Im Grenzfall $\epsilon \rightarrow 0$ geht natürlich die fiktive Sonne in die mittlere Sonne über - wie man sich durch Einsetzen überzeugt. Man sieht übrigens auch am *Geogebra*-Arbeitsblatt, dass für spitze ϕ die Sonne unterhalb des Äquators abtaucht, daher die negative z -Koordinate!

Obwohl sich auch die fiktive Sonne gleichmäßig auf ihrer Bahn bewegt, tut es ihre Projektion auf die Äquatorebene (wo die mittlere Sonne = Uhr umläuft) nicht mehr - wir haben einen Unterschied Sonnenzeit-Uhrzeit auf Grund der Schrägstellung der Erdachse!

Sehen wir und die Veranschaulichung der beiden Bahnen (mittlere Sonne und fiktive Sonne) an:

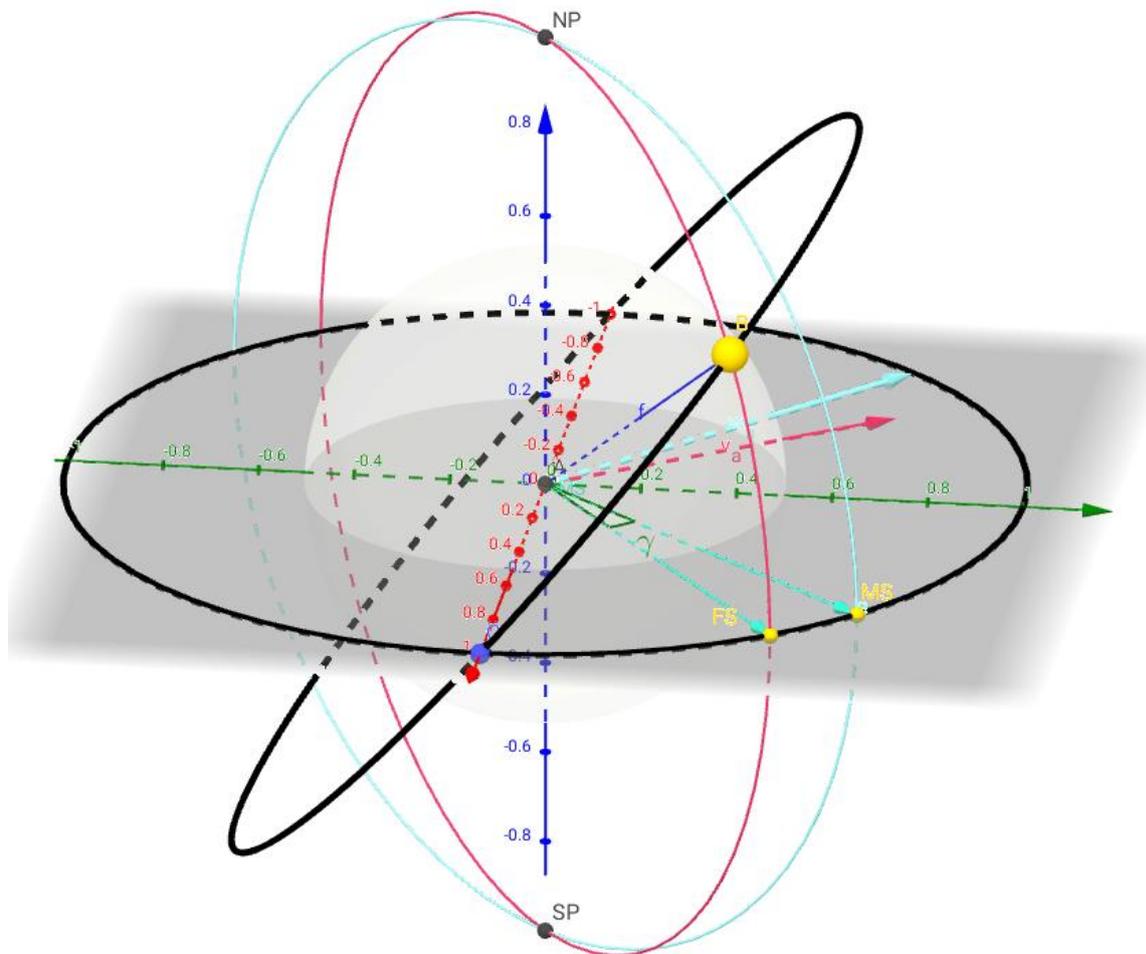


Abb.116 : Sonnenbahnen während eines Jahres - Zeitunterschied als Stundenwinkel γ ($\epsilon = 50^\circ$!)

Die mittlere Sonne MS bewegt sich gleichförmig in der xy -Ebene mit $(\cos(\phi), \sin(\phi), 0)$
 Die fiktive Sonne B bewegt sich ebenfalls gleichförmig nach der Formel 13.4 mit $\dot{\phi} = 1$

Natürlich hätten wir auch die Bahn der fiktiven Sonne herleiten können, indem wir die Bahn der mittleren Sonne um ε um die x -Achse (**aktiv**) gedreht hätten:



$$R_x(\varepsilon) \cdot \begin{pmatrix} \cos \phi \\ \sin \phi \\ 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \varepsilon & -\sin \varepsilon \\ 0 & \sin \varepsilon & \cos \varepsilon \end{pmatrix} \cdot \begin{pmatrix} \cos \phi \\ \sin \phi \\ 0 \end{pmatrix} = \begin{pmatrix} \cos \phi \\ \cos \varepsilon \sin \phi \\ \sin \varepsilon \sin \phi \end{pmatrix}$$

Nehmen wir an es sei 12 Uhr Mittag (Uhrzeit!) - die fiktive Sonne FS sollte eigentlich mit ihrem roten Sonnenhöchststandmeridian (rot) beim blauen Meridian der mittleren Sonne MS sein - aber FS hinkt nach um den Stundenwinkel γ .

Ist z.B. $\gamma = 2.5^\circ$ dann gilt, da 24 Stunden 360° entsprechen:

$$\gamma = \Delta\phi = \frac{24 \cdot 60}{360} \times 2.5 \text{ min} = 10 \text{ min}$$

Den Sonnenhöchststand der fiktiven Sonne bekommen wir

1. als Projektion von B auf die xy -Ebene oder
2. als Schnitt des Großkreises durch B mit Achse $\vec{k} \times \overrightarrow{AB}$ mit dem "Äquator" (\vec{k} Basiseinheitsvektor in Richtung z -Achse)

Wir werden beide Methoden in unser Arbeitsblatt implementieren (jeweils die eine als Gegencheck für die andere).

Quintessenz des Konstruktionsprotokolls:

1. 2 Schieberegler für ε (Tiltangle) und ϕ (Winkelposition von MS)
2. Ursprung Punkt A
3. Bahn von MS : `Circle with center A and radius 1`
4. Bahn von FS : `Curve(cos(f), sin(f) cos(ε), sin(f) sin(ε), f, 0, 2π)`
5. Position der fiktiven Sonne B :
`(cos(φ), sin(φ) cos(ε), sin(φ) sin(ε))`
6. Position der mittleren Sonne MS : `(cos(φ), sin(φ))`
7. Projektion der fiktiven Sonne B in die xy -Ebene B_p : `(x(B), y(B))` (unsichtbar)
8. Einheitsvektor vom Ursprung Richtung B_p : `u=Unit vector of Vector(A, B_p)`
9. Fiktive Sonne in die xy -Ebene FS : `A + u`

13. Zeitgleichung

10. Jetzt zur Winkelmessung γ zwischen FS und MS :

$$v_{\{MS\}} = \text{Vector}(A, MS)$$

11. $\gamma_1 = \text{Angle}(u, v_{\{MS\}})$ (Sichtbarkeit: $\gamma_1 < 180^\circ$)

$\gamma_2 = \text{Angle}(v_{\{MS\}}, u)$ (Sichtbarkeit: $\gamma_2 < 180^\circ$)

12. Abweichung der beiden Zeiten FS und MS in Minuten wäre $\left(1^\circ \cong \frac{24 \cdot 60}{360} = 4 \text{ min}\right)$

$$\underbrace{\text{Abweichung in 10 min Einheiten}}_{\text{Abweichung in rad}} \quad \underbrace{\text{Min}(\gamma_1, \gamma_2) \cdot \frac{180}{\pi}}_{0.4} \Rightarrow \begin{aligned} \text{deviation} &= \text{Min}(\gamma_1, \gamma_2) * 72 / \pi \\ \text{DeltaT} &= \text{If}(\gamma_1 < 180^\circ, \text{deviation}, -\text{deviation}) \end{aligned}$$

Falls FS nachhinkt ($\gamma_1 < 180^\circ$) muss die Abweichung addiert werden sonst subtrahiert!

13. Zum krönenden Abschluss noch der Graph der Funktion $\Delta T(\phi)$ (jedem ϕ wird die vorzeichenbefahete Zeitabweichung zugeordnet - wobei die tatsächliche lokale Sonnenzeit durch Addition entsteht!):

$$G = (\phi, \text{DeltaT})$$

$$\text{delay} = \text{Locus}(G, \phi)$$

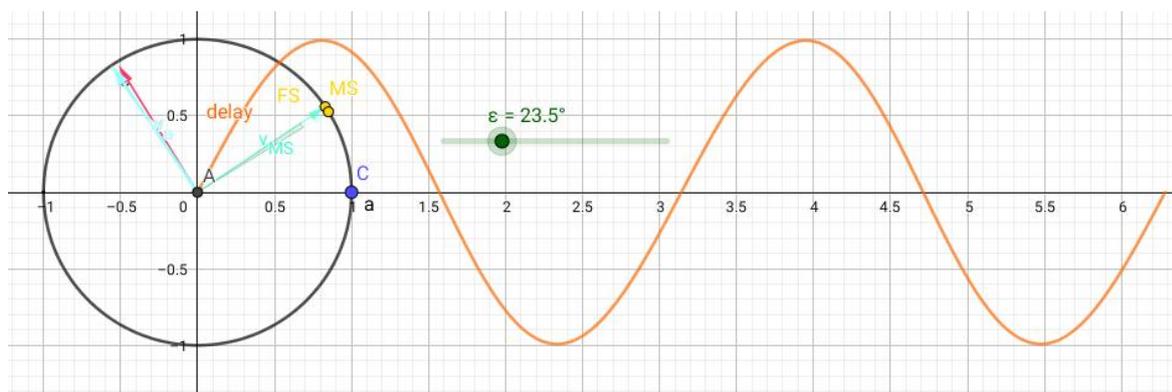


Abb.117 : Zeitunterschied bei ϕ in 10 Min Einheiten bei $\varepsilon = 23.5^\circ$ in 2D Graphik

Deutlich erkennt man die Periodendauer $T = \pi = (1/2)$, da ja 2π 1 Jahr entsprechen. Die Amplitude beträgt ziemlich genau 10 Minuten!

14. Jetzt noch die “Kontrollkonstruktion” über die Meridiankreise:

Die Achse für den Meridian der fiktiven Sonne liegt in der xy -Ebene und ist senkrecht zu \overrightarrow{AB} :

$$\vec{v}_a = \vec{k} \times \overrightarrow{AB} \quad \text{wobei } \vec{k} \text{ Einheitsvektor der z-Achse}$$

```

v_a = Cross(Vector( (0,0,1) ), Vector(A,B) )
g = Line(A,v_a)
d = Circle(g,B)

```

Man sieht der Meridian geht durch unseren Punkt FS - was unserer vorige Konstruktion bestätigt!

Meridian durch MS analog. Alles andere in der Konstruktion ist "Zuckerguss".

Da wir die Erdachsenneigung in unserem *Geogebra*-Blatt leicht durch einen Schieberegler verändern können, untersuchen wir wie "sinusartig" sich unser Tilt-Fehler verhält, wenn wir die Achsenneigung ordentlich erhöhen:

Man sieht: Den Tilt-Fehler durch eine einfache Sinusfunktion anzunähern mag bei der Erde noch durchgehen (sieht man allerdings in vielen Publikationen), bei größeren Achsenneigungen bricht die Symmetrie um $x = (2k + 1)\pi/4$, $k \in \mathbb{N}$ ein! Hier würde vermutlich eine Fouriertransformation für eine Näherung helfen!

Außerdem erkennt man in Abb. 118 dass bei großen Achsenneigungen der Sonnenhöchststand mit 12 Uhr ± 3 Stunden schwanken kann - die Uhr und die Sonne kommen ganz schön aus dem "Takt"!

Das *Geogebra*-Arbeitsblatt kann heruntergeladen werden von

<https://www.angsuesser.at/docs/math/geogebra/fictitiousSun-orbit.ggb>

Am besten man schaltet die Animation für den Schieberegler ϕ und die Spur von G ein. Zwischen 3D und 2D Graphik kann man wechseln - je nach Fokus.

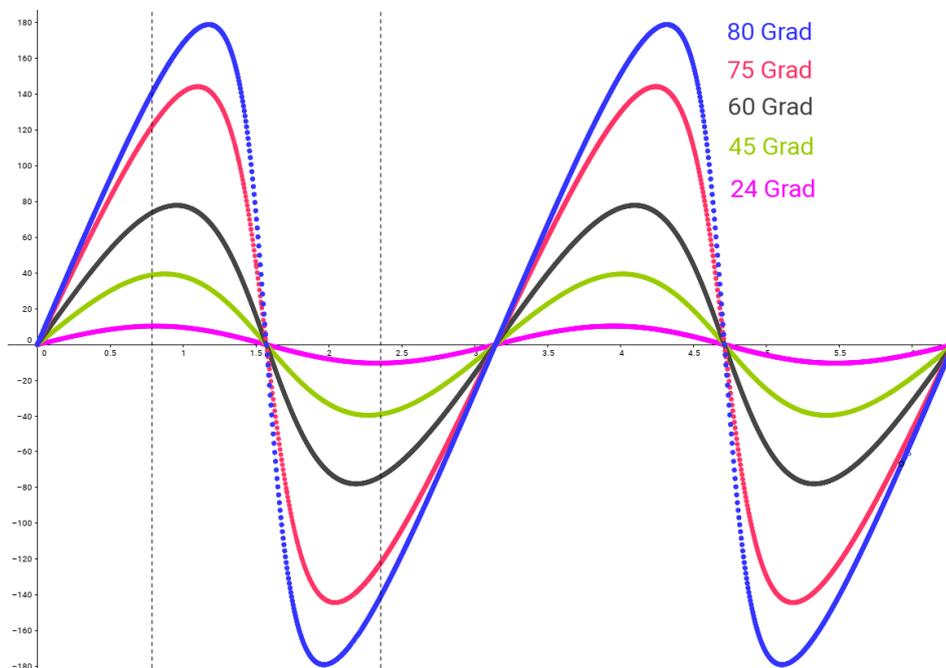


Abb.118 : verschiedene Achsenneigungen

13. Zeitgleichung

13.4.1 Berechnung der Tilt-Korrektur

Haben die Funktionsgraphen in Abb. 118 auch eine formelmäßige Darstellung? Ja - sie haben, aber wie man den Graphen entnehmen kann, wird es keine ganz einfache sein!

Wir haben mit 13.4 eine Funktionsdarstellung der fiktiven Sonne.

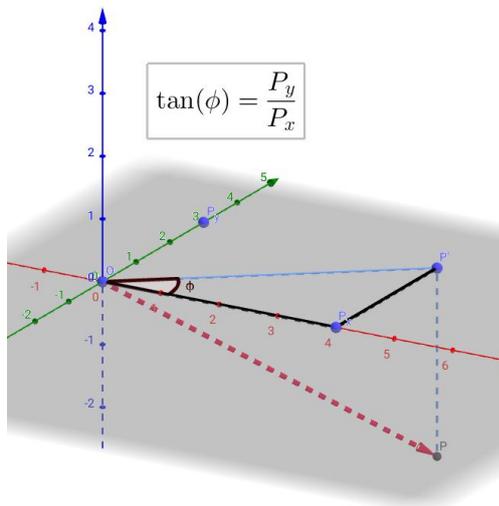


Abb.119 : Ermittlung des Projektionswinkels

Wie man nebenstehender Zeichnung entnehmen kann, ist der Tangens des Projektionswinkels ϕ in der xy -Ebene das Verhältnis von y -Koordinate zur x -Koordinate. Wir benennen diesen Winkel der fiktiven Sonne mit ϕ' , da ϕ schon für die mittlere Sonne vergeben ist. Dann ergibt sich mit 13.4:

$$\tan \phi' = \frac{\sin \phi \cos \varepsilon}{\cos \phi} = \tan \phi \cos \varepsilon$$

$$\tan \phi' = \tan \phi \cos \varepsilon \quad \phi \in [0, 2\pi] \quad (13.5)$$

daraus kann man ϕ' wenigstens in $[0, \pi/2[$ berechnen:

$$\phi' = \arctan(\tan \phi \cos \varepsilon) \quad (13.6)$$

Das Problem mit Formel 13.6 ist, dass der Arcustangens nur Werte zwischen $-\pi/2$ und $+\pi/2$ zurückliefert - wir müssen also von Hand anpassen!

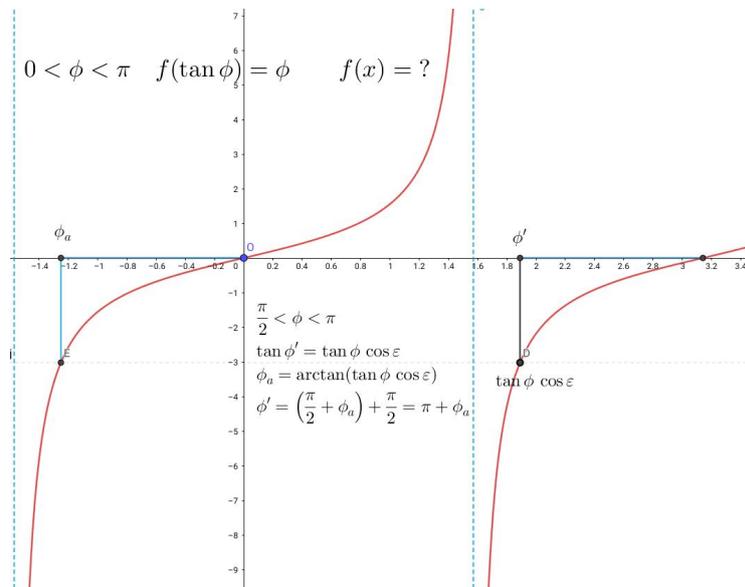


Abb.120 : Erweiterung des Arcustangens

Ist $\phi \in [0, \pi/2[$ können wir die “normale” Arcustangens-Fkt. verwenden.

Für $\phi \in]\pi/2, \pi[$ gibt \arctan den Wert ϕ_a zurück - wie man in Abb.120 ablesen kann. Unser gesuchter Wert $\phi' = \phi_a + \pi$.

Da der Tangens eine Periode von π hat, wiederholen sich dann die Funktionswerte:

$$f(x) := \left\{ \begin{array}{ll} \arctan(\tan x \cos \varepsilon) & \phi \in [0, \pi/2[\\ \arctan(\tan x \cos \varepsilon) + \pi & \phi \in]\pi/2, \pi[\\ f(x - \pi) & \phi \in]\pi, 2\pi[\end{array} \right\} \pi/2 \text{ als Problemstelle!}$$

Da wir am Unterschied $\phi - \phi'$ interessiert sind, legen wir eine Differenzfkt. fest

$$t_1(x) := x - \arctan(\tan x \cos \varepsilon) = x - f(x) \quad \text{wobei jetzt } \lim_{x \rightarrow \frac{\pi}{2}} t(x) = 0$$

Es ergibt sich

$$t(x) := \left\{ \begin{array}{ll} t_1(x) & \phi \in [0, \pi/2[\\ t_1(x) - \pi & \phi \in]\pi/2, \pi[\\ t(x - \pi) & \phi \in]\pi, 2\pi[\end{array} \right.$$

Es gibt mehrere Möglichkeiten in *wxMaxima* eine Funktion mit “Ästen” zu implementieren. 3 habe ich hier zur Veranschaulichung ausgewählt:

- mit `unit_step(<Sprungstelle>)`
- mit der charakteristischen Funktion `charfun(<Bedingung für 1>)`
- mit indizierten Funktionen (hier mit Plotliste `[[x1,y_1],[x_2,y_2] ...]`)

Hier das Session-Protokoll mit dem Plot:

```
(% i1) (epsilon:23.4*%pi/180,display(epsilon))$ ε = 0.13π
```

Differenzfunktion für den 1. Quadranten

```
(% i2) t(x):=x-atan(tan(x)*cos(epsilon));
```

$$t(x) := x - \operatorname{atan}(\tan(x) \cos(\varepsilon))$$

Abkürzung für die Sprungfunktion

```
(% i3) s(t):=unit_step(t)$
```

$t \in]t_1, t_t[\Rightarrow s1(t) = 1$ sonst 0

```
(% i4) s1(t_1,t_2,t):=s(t-t_1)*s(t_2-t)$
```

Festlegung auf 1. und 2.-ten Quadranten

```
(% i5) f(x):=t(x)*s1(0,%pi/2,x) + (t(x)-%pi)*s1(%pi/2,%pi,x)$
```

$f_1(x)$ ist in allen 4 Quadranten festgelegt

```
(% i6) f_1(x):=f(x)*s1(0,%pi,x)+f(x-%pi)*s1(0,2*%pi,x)$
```

13. Zeitgleichung

Dasselbe Problem mit der charakteristischen Funktion gelöst

```
(% i7) g(x):=if x<=%pi then (t(x)*charfun(x>=0 and x<%pi/2)+
      +(t(x)-%pi)*charfun(x>%pi/2 and x<=%pi)) else g(x-%pi)$
```

Jetzt eine Folge von indizierten Funktionen

```
(% i8) (h[1](x):=t(x), h[2](x):=t(x)-%pi, h[3](x):=t(x-%pi), h[4](x):=t(x-%pi)-%pi, h[5](x):=0)$
```

Wir suchen den Funktionsindex in Abhängigkeit vom Argument

```
(% i9) getFuncIndex(x):= block([r:5], if not(mod(x,%pi/2)=0) then r:ceiling(x/(%pi/2)), r )$
```

Wir kürzen weiter ab!

```
(%      branchedFunc(x):=h[getFuncIndex(x)](x)$
i10)
```

Eine kleine Hilfsfunktion: vom Bogenmaß zu den Minuten

```
(%      rad2min(rad):= 24*60/(2*%pi)*rad$
i11)
```

Wir erstellen eine Plottabelle

```
(%      table:makelist([float(i*2*%pi/100),rad2min(float(branchedFunc(i*2*%pi/100))) ] ,i,0,100)$
i12)
```

Zeichnen der 3 Funktionen

```
(%      plot2d([rad2min(f_1(x)), rad2min(g(x)) , [discrete,table] ],[x,0,0,2*%pi], [legend, "with
i14) unit\\_step", "with charfun", "with indexed functions"], [style,[lines,10,1,1], [lines,6,2,2], [li-
      nes,2,3,2] ])$
```

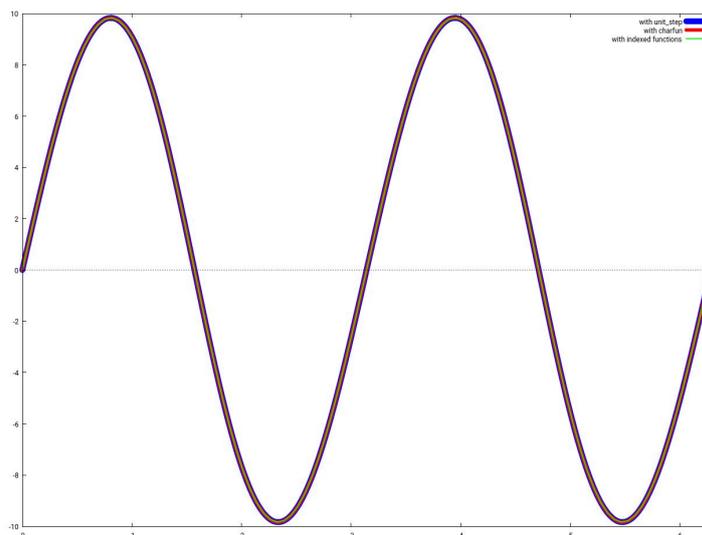


Abb.121 : Zeitunterschied auf Grund des Tilts - berechnet

13.4.2 Berechnung mit numerischer Integration

Wir differenzieren Gleichung 13.5 nach der Zeit ($\dot{\phi} = 1$):

$$\tan \phi' = \tan \phi \cos \varepsilon \left| \frac{d}{dt} \right. \quad \text{wobei} \quad \frac{d}{dt} \tan x = 1 + \tan^2 x$$

$$(1 + \tan^2 \phi') \dot{\phi}' = (1 + \tan^2 \phi) \cos \varepsilon \quad \xrightarrow{\tan = \sin / \cos} \quad \dot{\phi}' = \frac{\cos \varepsilon}{\cos^2 \phi + \sin^2 \phi \cos^2 \varepsilon}$$

$$\dot{\phi}' = \frac{\cos \varepsilon}{\cos^2 \phi + \sin^2 \phi (1 - \sin^2 \varepsilon)}$$

$$\dot{\phi}' = \frac{\cos \varepsilon}{1 - \sin^2 \phi \sin^2 \varepsilon} \quad \Rightarrow \quad \phi'(t_1) = \int_0^{t_1} \frac{\cos \varepsilon}{1 - \sin^2 \phi \sin^2 \varepsilon} d\phi \quad (13.7)$$

13.7 hat den Vorteil, dass der Integrand nur bei $\phi = \pi/2$ **und** $\varepsilon = \pi/2$ undefiniert ist - wir uns also die mühseligen Fallunterscheidungen ersparen!

Ein kurzer Check mit $\varepsilon = 0$ ergibt "die mittlere Sonne".

Da wir nur an der Zeitdifferenz interessiert sind ergibt sich

$$\phi - \phi' = t - \int_0^t \frac{\cos \varepsilon}{1 - \sin^2 \phi \sin^2 \varepsilon} d\phi = \int_0^t \left(1 + \frac{\cos \varepsilon}{\sin^2 \phi \sin^2 \varepsilon - 1} \right) d\phi \quad (13.8)$$

Schauen wir uns das kurz in *Geogebra* an - wir brauchen

- einen Schieberegler ε für die Erdachsenneigung ($0 \leq \varepsilon \leq \pi/2$)
- einen Schieberegler a für die obere Integralgrenze ($0 \leq a \leq 2\pi$)
- die Integrandenfunktion $f(x) = 1 + \frac{\cos \varepsilon}{\sin^2 x \sin^2 \varepsilon - 1}$
- das Integral `myInt = Integral(f,0,a)`
- den Integralwert in Minuten zum Vergleich `inMin = myInt*720/pi`
- einen Punkt `P = (a,inMin)`
- jetzt noch die Ortslinie - das war's: `tiltError = Locus(P,a)`
bei den Eigenschaften "Erweitert" auf Grafik2 - damit wir verschieden skalieren können!

13. Zeitgleichung

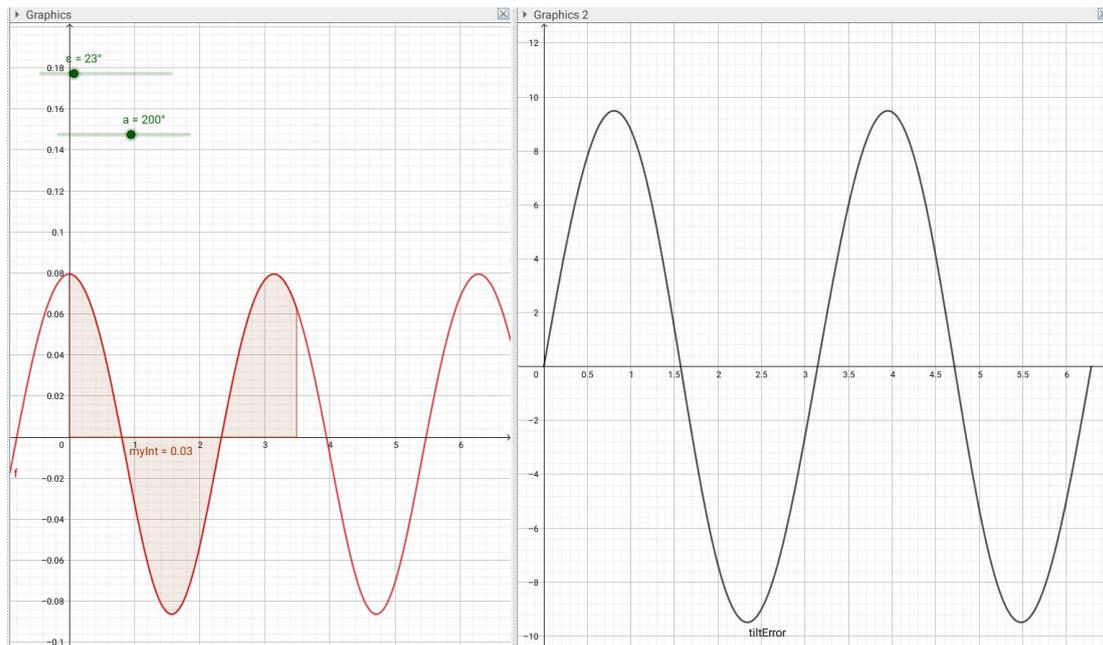


Abb.122 : Zeitunterschied auf Grund des Tilts - mit Integral

Mit *wxMaxima* ist es auch nicht schwieriger:

```
(% i1) rad2min(rad):= 24*60/(2*%pi)*rad$
```

```
(% i2) f(x,e):=cos(e)/(1-sin(e)^2*(sin(x))^2) ;
```

$$f(x, e) := \frac{\cos(e)}{1 - \sin(e)^2 \sin(x)^2} \quad (\% o2)$$

Uns interessiert vom numerischen Integral(`quad_qag`) nur das Ergebnis (erster Listeneintrag)

```
(% i3) numInt(ul,e):=first(quad_qag (f(x,e), x, 0, ul, 3))$
```

```
(% i4) g(t,e):=rad2min(t - numInt(t,e))$
```

Wir erstellen Plotlisten $[x, g(x, e)]$ mit 101 Stützpunkten in $[0, 2\pi]$

```
(% i5) iList(e):=makelist([i*2*%pi/100,g(i*2*%pi/100,e) ],i,0,100) $
```

9 Listeneinträge für $\varepsilon \in \{i\pi/20 | i \in 1 \dots 9\}$

```
(% i6) plotLists:makelist(['discrete,iList(i*%pi/20)],i,1,9)$
```

```
(% i7) plot2d(plotLists)
```

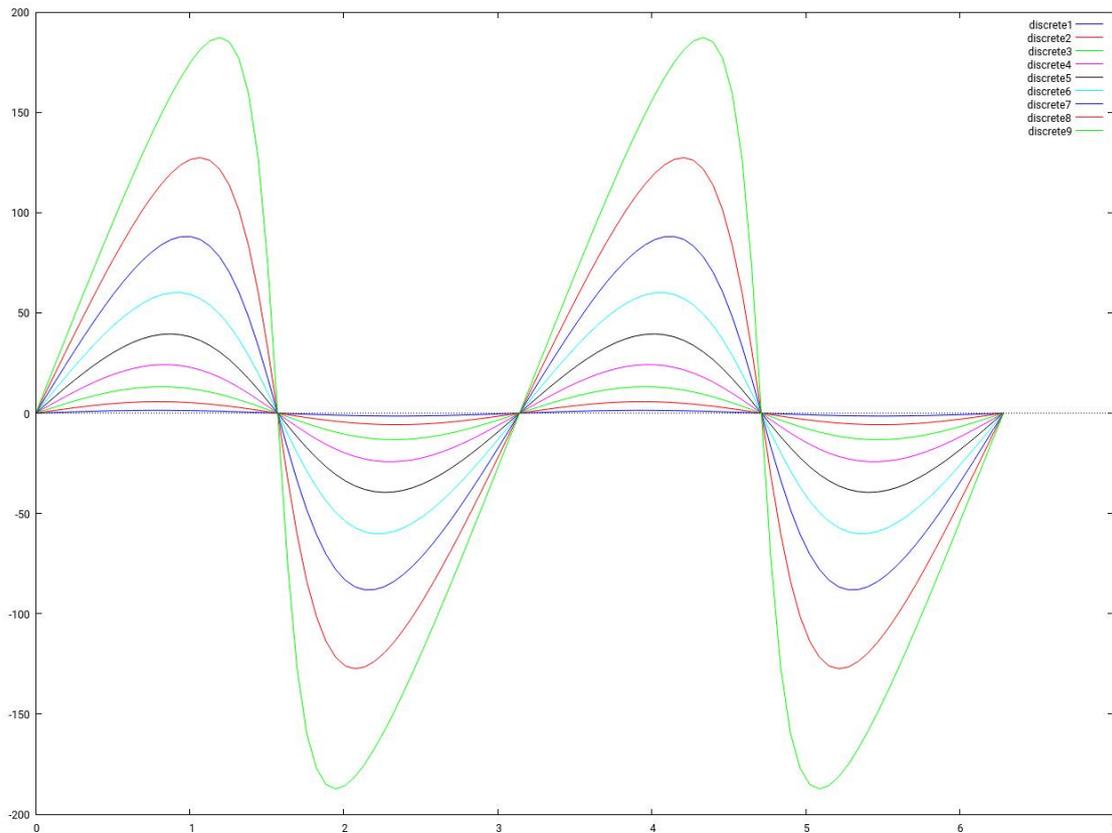


Abb.123 : Zeitunterschiede auf Grund des Tilts - mit Integral

13.4.3 Berechnung mit Fourierreihe

Wie wir gesehen haben, ist die Zeitdifferenz $\delta_T(t)$ auf Grund der Schiefstellung der Erdachse

$$\delta_T(t) = \int_0^t \underbrace{\left(1 + \frac{\cos \varepsilon}{\sin^2 x \sin^2 \varepsilon - 1}\right)}_{f(x)} dx$$

Die Idee ist nun $f(x)$ in eine Fourierreihe zu entwickeln, dann können wir das Integral praktisch "zu Fuß" berechnen - die Fourierreihe für eine Funktion f , die im Intervall $[-p, p]$ periodisch ist, lautet

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} \left[a_n \cos\left(\frac{n\pi x}{p}\right) + b_n \sin\left(\frac{n\pi x}{p}\right) \right] \quad \text{wobei} \quad (13.9)$$

$$\underbrace{a_n = \frac{1}{p} \int_{-p}^p f(x) \cos\left(\frac{n\pi x}{p}\right) dx}_{n \in \{0,1,2,\dots\}}, \quad \underbrace{b_n = \frac{1}{p} \int_{-p}^p f(x) \sin\left(\frac{n\pi x}{p}\right) dx}_{n \in \{1,2,3,\dots\}} \quad (13.10)$$

13. Zeitgleichung

Wir wissen über $f(x)$

- $p = \frac{\pi}{2}$, die Periode beträgt π
- $f(x) = f(-x)$ f ist *gerade*, d.h. alle b_n verschwinden
- Der Funktionsmittelwert über eine Periode verschwindet $\Rightarrow a_0$ verschwindet
- Für $\varepsilon = 0$ ist f eine Konstante, daher eine Fourierreihe nicht sinnvoll

Es gilt also

$$f(x) = \sum_{n=1}^{\infty} a_n \cos\left(\frac{n\pi x}{p}\right) = \sum_{n=1}^{\infty} a_n \cos(2nx)$$

die Integralfunktion $F(t)$ ergibt sich dann

$$F(t) := \int_0^t f(x) dx = \sum_{n=1}^{\infty} \frac{a_n}{2n} \sin(2nt) \approx \sum_{n=1}^N \frac{a_n}{2n} \sin(2nt) \frac{720}{\pi} [min]$$

Der letzte Faktor stammt von der Umrechnung von *rad* in Minuten, wobei $2\pi \approx 24 \cdot 60$ Minuten entsprechen.

Wir schauen uns das jetzt mit *wxMaxima* an:

Erdexzentrizität, Fließkommazahlpräzision, Periodenintervall, Anzahl der Fourierreihenglieder

```
(% i1) (e:float(%pi/180*23.4), fpprintprec:4, p:%pi/2,N:5)$
```

Integrand der Fehlerfunktion $\delta_T(t)$ - "T" für Tilt

```
(% i2) define(f(x),1+cos(e)/(sin(x)^2*sin(e)^2-1)) ;
```

$$f(x) := \frac{0.9178}{0.1577 \sin(x)^2 - 1} + 1$$

Syntax of `quad_qawo`: $\int_0^1 x^3 \cdot \cos(4x) dx \rightarrow \text{quad_qawo}(x^3, x, 0, 1, 4, \cos)$

Koeffizienten der Fourierreihe

```
(% i3) a(n):=first(quad_qawo(1/p*f(x),x,-p,p,2*n,cos))$
```

```
(% i4) a_n:=map(a,makelist(i,i,1,N));
```

```
[0.08577, -0.003678, 1.57810^-4, -6.76610^-6, 2.90110^-7]
```

Die abgebrochene Fourierreihenfunktion wird erstellt - zum Vergleich mit der Originalfunktion

(% i5) `fourier(x):=sum(a_n[n]*cos(2*n*x),n,1,N);`

$$\text{fourier}(x) := \sum_{n=1}^N a_{nn} \cos(2nx)$$

Die Koeffizienten der Integralfunktion mit Minutenskalierung

(% i6) `coeff:makelist(a_n[n]/(2*n)*float(720/%pi),n,1,N);`

$$[9.829, -0.2108, 0.006026, -1.93810^{-4}, 6.6510^{-6}]$$

Wir speichern für später zum Vergleich die Koeffizientenliste in CSV

`write_data(coeff,"coeff-fourier5.data",comma)$`

Hier die ausgeschriebene Integralfunktion

(% i7) `define(h(x),sum(coeff[n]*sin(2*n*x),n,1,N));`

$$h(x) := 6.6510^{-6} \sin(10x) - 1.93810^{-4} \sin(8x) + 0.006026 \sin(6x) - \\ - 0.2108 \sin(4x) + 9.829 \sin(2x)$$

Hier der Plot der 3 Funktionen - zwischen Fourier-Entwicklung und Original ist bei $N = 5$ keine Unterschied erkennbar (grün $\delta_T(t)$ $t \in [0, 2\pi]$, $t = 0$ ist Frühlingspunkt)

(% i8) `plot2d([100*f(x),100*fourier(x),h],[x,-%pi/2,%pi/2]);`

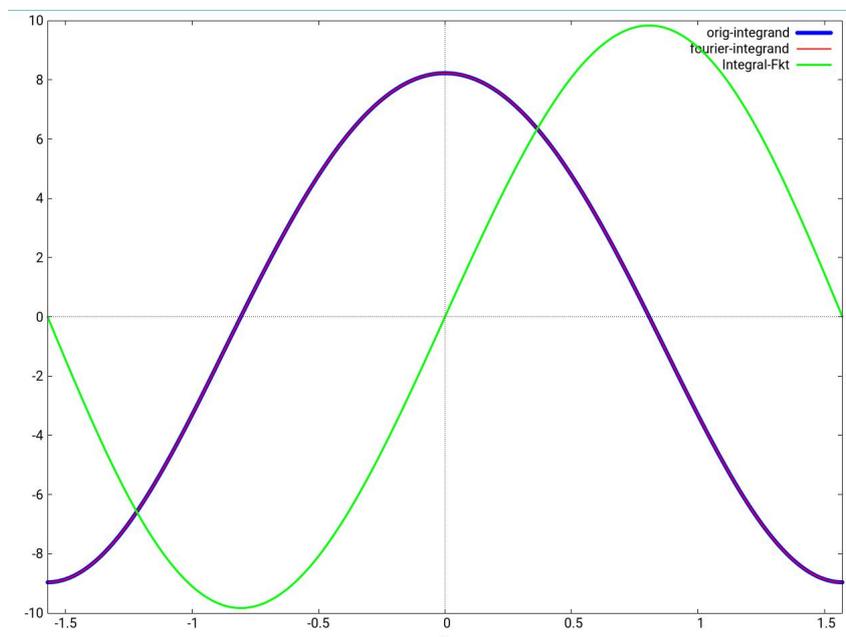


Abb.124 : f und $fourier$ (100-fach vergrößert!) und Integralfunktion $h = \delta_T(t)$

13. Zeitgleichung

Auch bei Achsenschieftellung 63 Grad ist noch kein Unterschied bei $N = 5$ erkennbar, obwohl der Zeitunterschied bereits mehr als ± 1.5 Stunden beträgt

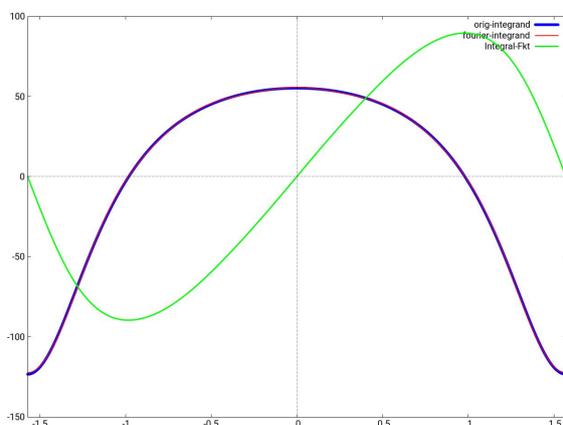


Abb.125 : f , $fourier$ und h , $\varepsilon = 63^\circ$

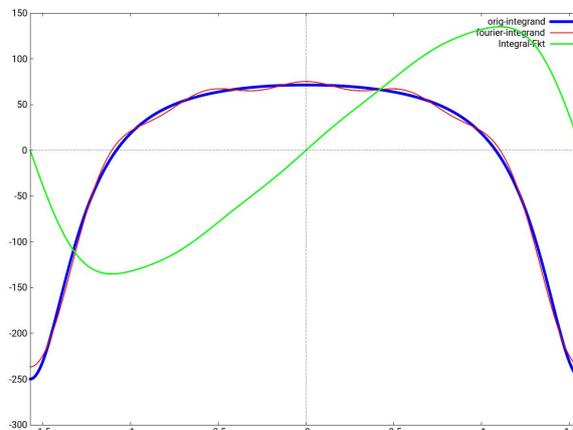


Abb.126 : f , $fourier$ und h , $\varepsilon = 73^\circ$

Bei ca. 70° und $N = 5$ wird die Fourierentwicklung sichtbar (bei 100-facher Vergrößerung!). Jetzt müßte man N vergrößern. Ein günstiger Augenblick um sich mit *a priori* und *a posteriori* Fehlerabschätzungen zu beschäftigen - aber das ist wieder eine andere Geschichte.

Wir aber gehen weiter und verfolgen einen anderen Ansatz

13.4.4 Berechnung mit Potenzreihe

$$\delta_T(t) = \int_0^t \left(1 + \frac{\cos \varepsilon}{\sin^2 x \sin^2 \varepsilon - 1} \right) dx = t - \cos \varepsilon \int_0^t \left(\frac{1}{1 - \underbrace{\sin^2 x \sin^2 \varepsilon}_{0 < q < 1}} \right) dx$$

Der Integrand ist also die Summe einer geometrischen Reihe. Die Potenzreihe ist also hier ohne Kenntnis von Taylor zugänglich! Damit gilt

$$\delta_T(t, \varepsilon) = t - \cos \varepsilon \int_0^t \sum_{n=0}^{\infty} (\sin^2 x \sin^2 \varepsilon)^n \approx t - \cos \varepsilon \sum_{n=0}^N \sin^{2n} \varepsilon \int_0^t \sin^{2n} x dx \quad (13.11)$$

Der Vorteil dieser Näherung ist, dass die Integrale von ε entkoppelt sind - man braucht also ε nicht zu kennen, um eine Reihenentwicklung zu bekommen. Bei der Fourier-Entwicklung war eine Integration ohne Kenntnis von ε nicht möglich!

Man überzeugt sich leicht, dass Formel 13.11 für $\varepsilon \rightarrow 0$ das richtige Resultat $\delta_T(t,0) \rightarrow 0$ liefert.

Je kleiner ε (und daher $\sin \varepsilon$) desto schneller konvergiert die Reihe. Das Problem sind die Integrale der Sinus-Potenzen, aber dafür gibt es eine Rekursionsformel (siehe 13.11):

$$\int \sin^n x dx = -\frac{\sin^{n-1} x \cos x}{n} + \frac{n-1}{n} \int \sin^{n-2} x dx$$

Damit können wir eine Funktion $f(n, x)$ mit 2 Argumenten n und x aufbauen

- n ist der nächste Exponent der Sinuspotenz
- x ist das letzte berechnete Integral, wobei $\int \sin^0 x dx = x$

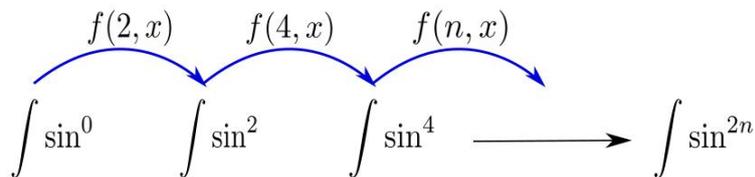


Abb.127 : Aufbau der rekursiven \sin^{2n} -Liste vom Anfang her

Dieser Aufbau erinnert an das Falten einer Liste in funktionalen Programmiersprachen:

```
FoldList([x], [a,b,c,d],f)
--> [x, f(x,a), f(f(x,a),b), f(f(f(x,a),b),c), f(f(f(f(x,a),b),c),d)]
```

In der *funktionalen Programmierung* wird **FoldList** als *Funktion höherer Ordnung* bezeichnet, da sie eine Funktion als Argument besitzt (dies gilt auch für Funktionen, deren Rückgabewert eine Funktion ist). Mit solchen Konstrukten lassen sich "einfache" Rekursionen in den Griff kriegen.

Zum Beispiel im nächsten Code-Snippet hängt *fold* solange $g(\text{last}(L1), \text{first}(L2))$ an $L1$ an bis $L2$ "erschöpft" ist, zum Schluss wird $L1$ zurückgegeben. Nimmt man als Startwert $\int \sin^0 = x$ (also $n = 0$), dann bekommt man mit `next_integral(x,2)` das Integral $\int \sin^2$ und dies wird an $L1$ angehängt und wird so zum Argument der Funktion beim nächsten Durchlauf, n wird in 2-er Schritten erhöht:

```
fold(L1,L2,g):= if length(L2)>0 then
  fold( endcons(g(last(L1), first(L2)), L1), rest(L2,1),g)
else L1$

next_integral(last_sine_integral,n):=-sin(x)^(n-1)*cos(x)/n+(n-1)/n*last_sine_integral$

integrated_sine_list:fold([x],makelist(i,i,2,2*N,2),next_integral)$
```

13. Zeitgleichung

Interessant ist (% i7) wo eine Summe als skalares Produkt codiert wird.

OK - jetzt das Programm in *wxMaxima*:

Parameter: Tilt 63.4°, Floatingpoint-precision output, no warnings, 10 Potenzienglieder

```
(% i1) [e:float(%pi/180*63.4),fpprintprec:4, ratprint:false, N:10]$
```

Wie im Text erklärt

```
(% i2) fold(L1,L2,g):= if length(L2)>0 then
      fold( endcons(g(last(L1), first(L2)), L1), rest(L2,1),g) else L1$
```

```
(% i3) next_integral(last_sine_integral,n):=-sin(x)^(n-1)*cos(x)/n+(n-1)/n*last_sine_integral$
```

Wir bauen die Liste der Sinus-Integrale

```
(% i4) integrated_sine_list:fold([x],makelist(i,i,2,2*N,2),next_integral)$
```

n ist hier nur eine dummy-Variable - wir brauchen sie nicht. Dafür können wir unsere Funktion höherer Ordnung *fold* wieder verwenden!

```
(% i5) multiply_by_sine2(x,n):=x*sin(e)^2$
```

Wir bauen die Liste $\sin(\varepsilon)^{2n}$ - Faktoren

```
(% i6) sine_tilt_list:float(fold([1],makelist(i,i,1,N),multiply_by_sine2));
```

```
[1.0, 0.7995, 0.6392, 0.5111, 0.4086, 0.3267, 0.2612, 0.2088, 0.167, 0.1335, 0.1067]
```

Die Funktion $\delta_T(t)$ von 13.11 wird erstellt

```
(% i7) power_series(x):=(x-(integrated_sine_list . sine_tilt_list) * cos(e))*720/%pi$
```

Die Koeffizientenliste der früher erstellten Fourier-Approximation wird eingelesen - zu Vergleichszwecken!

```
(% i8) fourier_coeff:read_list("coeff-fourier5.data",comma);
```

```
[87.42, -16.67, 4.24, -1.213, 0.3701]
```

Die Fourier-Approximations-Funktion wird erstellt - 5 Koeffizienten!

```
(% i9) define(fourier(x),sum(fourier_coeff[n]*sin(2*n*x),n,1,5));
```

```
fourier(x) := 0.3701 sin(10x) - 1.213 sin(8x) + 4.24 sin(6x) - 16.67 sin(4x) + 87.42 sin(2x)
```

Numerische Berechnung von 13.8 mit 100 Punkten - das ist wohl von der Genauigkeit die Referenz an der sich die anderen Näherungen messen lassen müssen!

```
(% i10) points: makelist([i*%pi/100,first(quad_qag(1+cos(e)/(sin(x)^2*sin(e)^2-1),
      0,i*%pi/100,3))*720/%pi],i,-50,50)$
```

```
(% plot2d([fourier(x),power\_series(x),[discrete,points] ],[x,-\%pi/2,\%pi/2],
i11) [legend, " Fourier-Series"," Power-Series"," Numeric-Integral"],
[style,[lines,10,1,2], [lines,5,2,2],[lines,2,3,2]],
[gnuplot_preamble, "set key bottom right; set xtics font \", 15\";
set ytics font \", 15\"; set key font \", 15\";
set title font \", 20\" "]
)$
```



Wie man aus der Ausgabe des Plotbefehls erkennt, besitzt die Näherung mit der Potenzreihe ein wesentlich schlechteres Konvergenzverhalten als die Fourierreihe - kein Wunder, die ist auf periodische Funktionen spezialisiert!

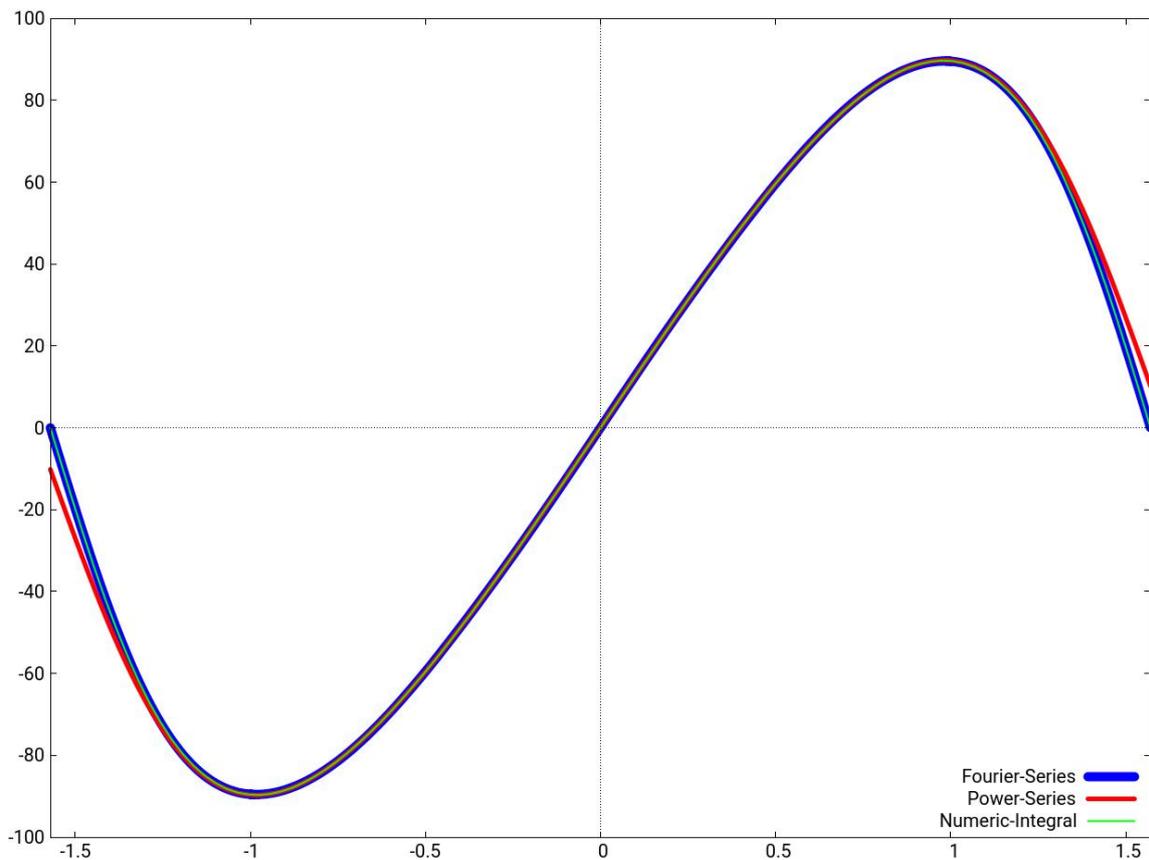


Abb.128 : Vergleich von Fourier ($N = 5$), Potenziereihe ($N = 10$) und numerisches Integral bei $\varepsilon = 63^\circ$

13.4.5 Simulation mit *wxMaxima*

Bevor wir uns der Kepler-Korrektur der Zeitdifferenz Sonnenhöchststand - 12 Uhr Uhrzeit zuwenden, zum Abschluss noch eine relativ aufwendige Simulation in *wxMaxima* - eine "brute force" Attacke:

13. Zeitgleichung

Ein beliebiger Punkt am Äquator der Erde - nehmen wir der Einfachheit $P = (1, 0, 0)^T$ - rotiert um die Erdachse (z -Achse) $n = 366.25$ -mal im Jahr, das ergibt - wie wir wissen - 365.25 Sonnentage.

Wann sind die Sonnenhöchststände (lokale Mittage)?

Gießen wir diese Fragestellung in Mathematik ($\phi \in [0, 2\pi]$):

Position P' während der Drehungen: $P' = R_z(n\phi) \cdot P = (\cos(n\phi), \sin(n\phi), 0)^T$

Position der Sonne (siehe 13.4): $S = (\cos \phi, \cos \varepsilon \sin \phi, \sin \varepsilon \sin \phi)^T$

Winkel zwischen P' und Sonne: $\cos[\angle(P', S)] = \cos(n\phi) \cdot \cos \phi + \sin(n\phi) \cdot \cos \varepsilon \sin \phi$

Der Winkel $\angle(P', S)$ wird am kleinsten (Sonnenhöchststand), wenn der Cosinus sein Maximum erreicht, wo also besitzt die Funktion

$$f(\phi) = \cos(n\phi) \cos \phi + a \sin(n\phi) \sin \phi \quad \text{mit } a = \cos \varepsilon \in [-1, 1]$$

ihre Maxima?

Wenn die Schiefstellung der Erdachse (ε) nicht allzu groß, sollte das in der Nähe sein, wo die mittlere Sonne ihren Sonnenhöchststand hat, also in der Nähe von $2\pi/365.25 \times \text{Tagesanzahl}$ - mit dieser Näherung können wir ein Newtonverfahren starten!

Vorerst schreiben wir allerdings $f(\phi)$ noch um (die Produkte in additive Terme) - damit wird die Ableitung um einiges "einfacher":

$$\begin{aligned} f(\phi) &= \cos(n\phi) \cos \phi + a \sin(n\phi) \sin \phi = \\ &= \frac{\cos(n \cdot \phi - \phi) + \cos(n \cdot \phi + \phi)}{2} - \frac{a \cdot (\cos(n \cdot \phi + \phi) - \cos(n \cdot \phi - \phi))}{2} \end{aligned} \quad (13.12)$$

Die Ableitung $f'(\phi)$ ergibt sich dann zu (konstanter Faktor wurde weggelassen):

$$f'(\phi) = -(1 - a) (1 + n) \sin((n + 1) \phi) - (1 + a) (n - 1) \sin((n - 1) \phi)$$

Hier jetzt die Implementation in *wxMaxima*:

Zuerst überprüfen wir unsere obige Gleichung 13.12

$$\begin{aligned} &\cos(n\phi) \cos \phi + a \sin(n\phi) \sin \phi = \\ &= 1/2 (\cos((n + 1)\phi) + \cos((n - 1)\phi)) - a/2 (\cos((n + 1)\phi) - \cos((n - 1)\phi)) \end{aligned}$$

```
(%i2) term:1/2*(expand(cos((n+1)*t)+cos((n-1)*t))
      -a/2*expand(cos((n+1)*t)-cos((n-1)*t));
```

```
(%i3) trigexpand(term);
```

$$(%o3) \quad a \cdot \sin(t) \cdot \sin(n \cdot t) + \cos(t) \cdot \cos(n \cdot t)$$

Wir eliminieren den für die Lage der Extremwerte unwesentlichen Faktor 1/2

```
(%i4) f(t):=ratsimp(2*term)$
```

Wir berechnen die Ableitung als Funktion der Zeit t und Achsenneigung a

```
(%i5) define(f_s(t,a),diff(f(t),t));
```

```
(%o5) f_s(t,a) := -(1-a)·(1+n)·sin((n+1)·t) - (1+a)·(n-1)·sin((n-1)·t)
```

Festlegung der Umdrehungszahl der Erde (Beachte: $n = N$ bedeutet (N-1)-Tage im Jahr

```
(%i6) n:366.249$
```

Wir programmieren ein Newton-Verfahren, θ wird dabei auf 23.5° festgelegt, die Abbruchgenauigkeit wird auf 10^{-10} gesetzt

```
(%i7) newton(x0):=block([x_alt:x0,d:1,%theta:cos(%pi/180*23.5),accuracy:10^(-10)],
  define(g_s(t), f_s(t,%theta)), define(g_ss(t), diff(g_s(t),t)),
  for i: 1 unless d<accuracy do (
    x_neu: float(x_alt - g_s(x_alt)/g_ss(x_alt)), d:abs(x_neu - x_alt),
    x_alt: x_neu,
  x_alt)$
```

Wir bestimmen nun für jeden Tag des Jahres (0 entspricht dem Frühlingspunkt (21. März) den Unterschied zwischen Sonnenhöchststand und 12 Uhr (Uhrzeit) herauszufinden; als Startpunkt für das Newton-Verfahren verwenden wir die Uhrzeit(=mittlere Sonne); day =Winkel pro Jahrestag Alternativ könnte man auch die in *wxMaxima* eingebaute Funktion *find_root* benutzen!

```
(%i8) findTiltError(days):=block([mins:[0.0],noons:[0.0],
  day:2*%pi/(n-1),root:0.0, %theta:cos(%pi/180*23.5)],
  for d thru days do (
    /*root:find_root(f_s(t,%theta),t,d*day-day/20,d*day+day/20),*/
    root:newton(d*day), mins:cons(float(root),mins), /* local noon */
    noons:cons(float(d*day),noons) /* actual noon on clock = 12h */
  ),
  reverse(noons - mins) )$
```

Funktion: Winkel(Einheiten rad) \rightarrow in Minuten

```
(%i9) rad2min(rad):=block(
  float(365.25*24*60/(2*%pi)*rad)
)$
```

Wir rechnen die Winkeldifferenzen für 365 Tage in Minuten um - diese Werte bilden eine Liste

```
(%i10) diffsInMin:=map(rad2min,findTiltError(365))$
```

Wir suchen die ersten 11 Tage und hängen sie "hinten" an - 376 Tage

```
(%i11) tail:makelist(diffsInMin[i],i,2,11)$
```

Wir erstellen die Argumenteliste für das Plotten: 0 bis 375 (376 Tage)

```
(%i12) args:makelist(i,i,0,375)$
```

Anhängen - wie oben erwähnt

```
(%i13) diffsInMin:=append(diffsInMin,tail)$
```

Als Zugabe eine Funktion: Tagenummer(0 bis 365) \rightarrow Datum

13. Zeitgleichung

```
(%i14) determineDate(dayNr):=block([d:[10,30,31,30,31,31,30,31,30,31,31,28,21],
    size:13, accList:[10], month:1, date:0,
    monthNames:['März','April','Mai','Juni','Juli','Aug','Sep','Okt,
                'Nov','Dez','Jan','Feb','März]],
    if (dayNr<0 or dayNr>365) then
        return (disp("Tag nicht zwischen 0 und 365 --> Fehler!")),
    for i:2 thru size do accList:cons(accList[1]+d[i],accList),
    accList:reverse(accList),
    for i:1 while accList[i] < dayNr do month:i+1,
    if (month = 1) then date:21+dayNr else date:dayNr - accList[month-1],
    print(date,". ",monthNames[month])
    )$
```

```
(%i15) determineDate(182);           19. Sep
```

Jetzt sind wir soweit - wir erstellen den Plot

```
(%i16) plot2d([discrete, args, diffsInMin], [x,0,375])$
```

An welchen Tagen ist der Fehler ungefähr 0?

```
(%i17) searchForRoots(L):=block([size:length(L), pred:0, succ:0, indexList:[]],
    for i thru size do (
        if (abs(L[i]) < 0.3) then indexList:cons(i-1,indexList)
    ), reverse(indexList))$
```

```
(%i18) searchForRoots(diffsInMin);   (%o18) [0, 91, 92, 182, 183, 274, 365]
```

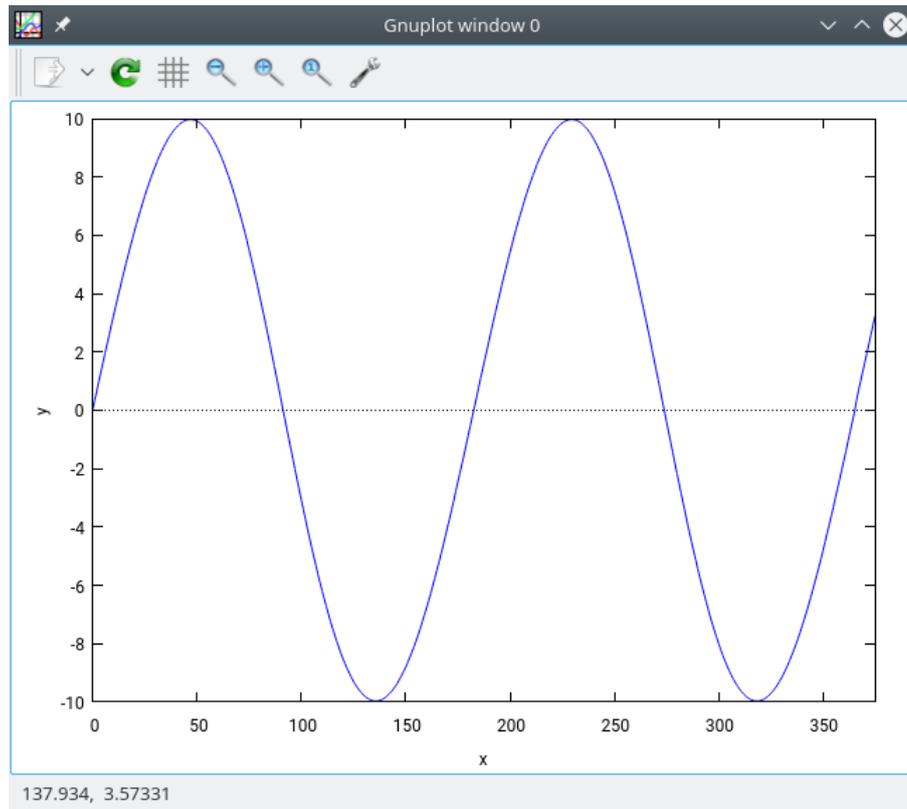


Abb.129 : Plot aus *wxMaxima*

Hier nun das Ergebnis des Plots - wie man sieht ist es eine Bestätigung unserer *Geogebra*-Simulation. Aber bei genauerer Untersuchung kommt man darauf, dass die Herbst-Tag-und-Nachtgleiche auf den 19. September fällt!

Wir haben ja die Keplerbewegung nicht brücksichtigt und nach dem Frühlingspunkt bewegt sich die reale Sonne langsamer als die mittlere Sonne (unsere Uhr) - daher tritt die Herbst-Tag-und-Nachtgleiche später ein als geplant!

13.5 Position der wahren Sonne $\psi(t)$ - Keplerkorrektur

Polarform einer Ellipse mit den Halbachsen a und b (wobei $a > b$) mit Koordinatenursprung im Brennpunkt (Sonne). Für den Polarwinkel ψ gilt die Entsprechung $2\pi \hat{=} 365,249 \cdot 24$ Stunden:

$$r(\psi) = \frac{a^2 - e^2}{a + e \cos(\psi)} \text{ wobei } e = \sqrt{a^2 - b^2} \text{ die lineare Exzentrizität ist}$$

Für die Erdbahn (mit obigen Vorbehalten) und der Längeneinheit $a = 1$ gilt:

$$e = \varepsilon (= e/a) = \kappa = 0,016722, \alpha = 78,5^\circ, \epsilon = 23,45^\circ \quad (13.13)$$

($\alpha = \angle(\text{Perihel}, \text{Frühlingspunkt}), \epsilon = \angle(\text{Ekliptiknormale}, \text{Erdachse})$)

13. Zeitgleichung

Die Fläche A ergibt sich mit obigen Einheiten

$$\kappa = \sqrt{1 - b^2} \Rightarrow b = \sqrt{1 - \kappa^2} \Rightarrow A = ab\pi = \pi\sqrt{1 - \kappa^2}$$

Übrigens der kleine Wert von κ zeigt, dass unsere obige Näherung so schlecht nicht war - aber wie die obere Tabelle zeigt, eine Viertelstunde kann man schon daneben liegen, dazu kommt natürlich noch der Abstand vom "zeitgebenden Längengrad".

Wenden wir zuerst Kepler an: Der Flächenzuwachs vom Radiusvektor pro Zeit soll konstant sein! Also (wenn wir die Zeit in rad messen $2\pi \hat{=} 1$ Jahr)

$\dot{A}(t) = c \Rightarrow A(t) = c \cdot t + c_1$ mit den Randbedingungen $A(0) = 0$ und $A(2\pi) = ab\pi$ ergibt sich

$$c_1 = 0 \text{ und } c = \frac{1}{2}\sqrt{1 - \kappa^2}$$

also mit Differentialen geschrieben: $\frac{dA}{dt} = c = \frac{1}{2}\sqrt{1 - \kappa^2}$ (13.14)

Erinnern wir uns

$$\int_a^b f(x) dx \Leftrightarrow \lim_{n \rightarrow \infty} \sum_{i=1}^n f(\xi_i) \Delta x \quad \text{mit } \Delta x = \frac{b-a}{n}, x_i = a + i * \Delta x \text{ und } \xi_i \in [x_{i-1}, x_i]$$

genauso lässt sich zeigen, wenn $r(\psi)$ die Polarform einer Kurve darstellt, dass sich der Flächeninhalt ergibt zu

$$A(\psi_1, \psi_2) = \lim_{n \rightarrow \infty} \sum_{i=1}^n \frac{1}{2} [r(\xi_i)]^2 \Delta\psi = \frac{1}{2} \int_{\psi_1}^{\psi_2} [r(\psi)]^2 d\psi$$

also mit Differentialen geschrieben: $\frac{dA}{d\psi} = \frac{1}{2} [r(\psi)]^2$ (13.15)

Aus (1) und (2) lässt sich über die Kettenregel $\dot{\psi}(t)$ gewinnen:

$$\dot{\psi}(t) = \frac{d\psi}{dt} = \frac{d\psi}{dA} \frac{dA}{dt} = \left(\frac{dA}{d\psi} \right)^{-1} c = \frac{2}{r^2} c = (1 + \kappa \cos \psi)^2 (1 - \kappa^2)^{-\frac{3}{2}}$$

diese Differentialgleichung für ψ gilt es zu lösen:

$\dot{\psi}(t) = (1 - \kappa^2)^{-\frac{3}{2}} (1 + \kappa \cos \psi)^2$ (13.16)

Damit die Argumentationslinie nicht durchbrochen wird, habe ich die Diskussion über deren Lösung auf ein anderes Kapitel (siehe 14) verschoben. Im Folgenden werden wir mit dieser Näherung der Lösung weiterrechnen:

$$\psi(t) = t + \underbrace{2\kappa \sin(t) + \frac{5}{4}\kappa^2 \sin(2t)}_{\delta(t)} = t + \delta(t) \quad \text{wobei } \delta(0) = \delta(\pi) = \delta(2\pi) = 0$$

13.5.1 Berechnung mit *wxMaxima*

Es handelt sich also um eine gleichförmige Bewegung (uniform motion) mit einer “Störung”.

Berechnen wir diese sich akkumulierende “Störung” mit *wxMaxima*:

Exzentrizität und Anzahl der Tage im Jahr werden festgelegt

```
(%i1) k:0.016722$ year:365$
```

Jetzt obige Näherungsformel für $\psi(t)$

```
(%i3) %psi(t):= t+ 2*k*sin(t)+ 5/4*k^2*sin(2*t);
```

```
(%o3)  $\psi(t) := t + 2k \sin(t) + \frac{5}{4}k^2 \sin(2t)$ 
```

Wie groß ist der Unterschied zur gleichförmigen Bewegung? Input: day-number; (0 entspricht 3. Jänner(Perihel))

```
(%i4) define(errorkepler(day),2*%pi/year*day-%psi(2*%pi/year*day))$
```

Umrechnung in Minuten und Werteliste (values) erstellen; $v[1] = \psi(21.Dez)$

```
(%i5) v:makelist(24*60*errorkepler(i)/(2*%pi),i,-13, year),numer$
```

Punktliste p erstellen: dayNr -13 \rightarrow 21. Dez.(Sonnenwende), dayNr 0 \rightarrow 3. Jan. (Perihel)

```
(%i6) p:makelist([i,v[i+14]],i,-13,year)$
```

Jetzt plotten wir die Argument-Werteliste $p(\text{points})$

```
(%i7) plot2d([[discrete,p ]],[y,-10,10], [style,[lines]], [color, magenta],
[legend,"Fehler in Minuten"]);
```

13. Zeitgleichung

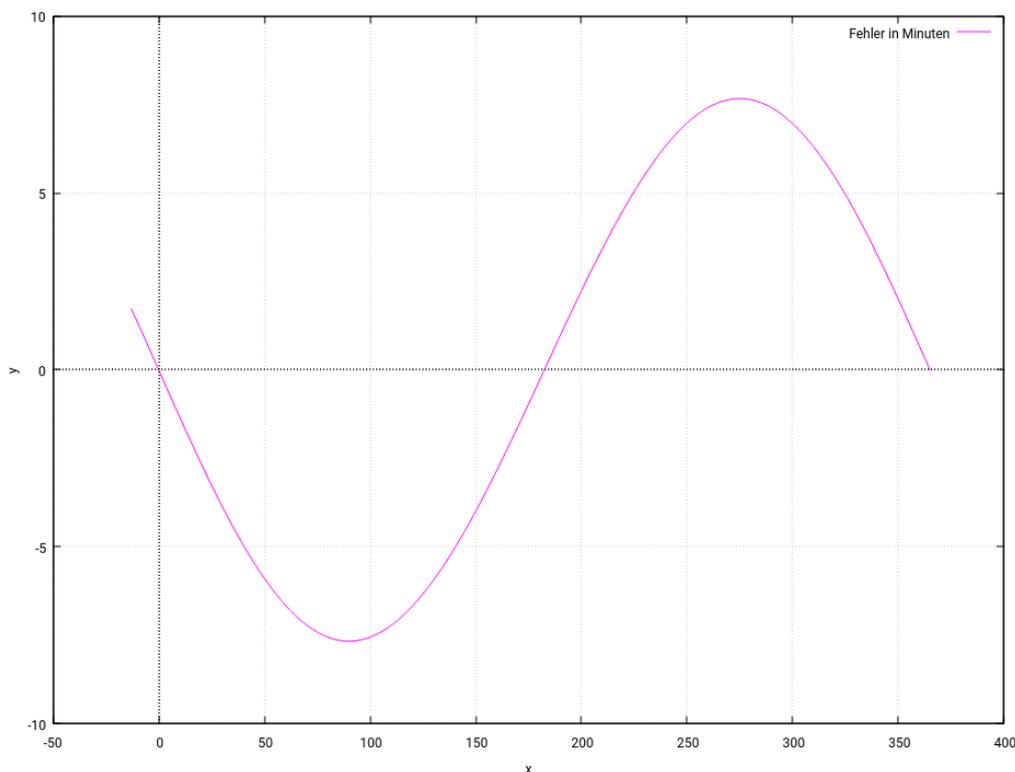


Abb.130 : Fehler auf Grund der “Keplerbewegung”

Der Graph der Zeitgleichung (**nur die Keplerbewegung betreffend**) ist natürlich

$$t - \psi(t) = - \left[2\kappa \sin(t) + \frac{5}{4}\kappa^2 \sin(2t) \right] = \delta_K(t) \quad \text{wobei } t \hat{=} \frac{2\pi}{365}n \quad (13.17)$$

Der Wert $\Delta\psi = \text{errorkepler}(\text{day})$ ist eine Korrektur des Stundenwinkels $1 \text{ rad} \hat{=} \frac{60 \times 24}{2\pi} \text{ min}$. Der Fehler auf Grund der Keplerbewegung fällt also etwas geringer aus als jener auf Grund der Erdachsenneigung (ca. 10 Minuten). Wir halten fest:

Der Fehler schwankt zwischen $-7\frac{1}{2}$ und $+7\frac{1}{2}$ Minuten



Bei der folgenden Simulation mit *Geogebra* ist der Schieberegler für k für die Exzentrizität auf den Wert der Erde eingestellt. Man kann ihn natürlich “in die Höhe fahren”, aber bedenken Sie, das wir für $\psi(t)$ eine Reihenentwicklung für kleine k vorgenommen haben - die wird dann immer fehlerhafter!

Zum Herunterladen:

<https://www.angsuesser.at/docs/math/geogebra/kepler-error.ggb>

Die Ortskurve für P bestätigt unser Ergebnis von *wxMaxima*!

13.5.2 Simulation mit Geogebra

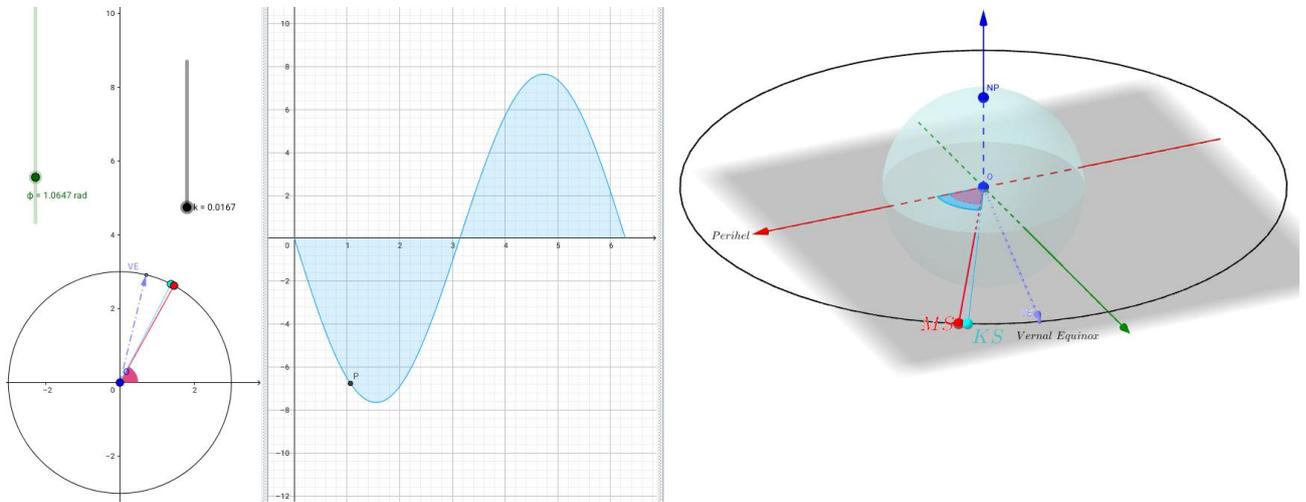


Abb.131 : Keplerfehler alleine in Geogebra

No.	Name	Description	Value
1	Point X		$X = (1, 0, 0)$
2	Point SP		$SP = (0, 0, -1)$
3	Point O		$O = (0, 0, 0)$
4	Point NP		$NP = (0, 0, 1)$
5	Number k		k
6	Function ψ	$\psi(t) = t + 2k \sin(t) + 5/4 k^2 \sin(2t)$	$\psi(t) = t + 2 * 0.0167 \sin(t) + 5 / 4 * \dots$
7	Angle ϕ		$\phi = 1.0647 \text{ rad}$
8	Point KS	$3(\cos(\psi(\phi)), \sin(\psi(\phi)), 0)$	$KS = (1.3764, 2.6656, 0)$
9	Sphere e	Sphere with center O and radius 1	$e: x^2 + y^2 + z^2 = 1$
10	Number α		$76.5\pi / 180$
11	Vector vequ	$3(\cos(\alpha), \sin(\alpha), 0)$	$vequ = (0.7003, 2.9171, 0)$
12	Circle c	Circle with center O and radius 3, axis parallel to zAxis	$c: X = (0, 0, 0) + (3 \cos(t), 3 \sin(t), 0)$
13	Point MS	$3(\cos(\phi), \sin(\phi), 0)$	$MS = (1.4544, 2.6239, 0)$
14	Segment g	Segment O, MS	$g = 3$
15	Segment f	Segment KS, O	$f = 3$
16	Angle δ_K	Angle between X, O, KS	$\delta_K = 1.0942 \text{ rad}$
17	Angle δ_M	Angle between X, O, MS	$\delta_M = 1.0647 \text{ rad}$
18	Angle δ	$\delta_M - \delta_K$	$\delta = -0.0295 \text{ rad}$
19	Point P	$(\phi, \delta \cdot 720 / \pi)$	$P = (1.0647, -6.7627)$
20	Locus loc1	Locus(P, ϕ)	$loc1 = \text{Locus}(P, \phi)$

- 1-4 einige Orientierungspunkte
- 5 Regler für κ (default 0.0167)
- 6 Näherung laut Text für $\psi(t)$
- 7 Schieberegler für Zeit (ϕ)
- 8 KS "Keplersonne"
- 9 Andeutung der Erdkugel
- 10 Näherungswert für Vernal Equinox VE (vom Perihel)
- 11 Zeiger zur VE
- 12 Sonnenorbit
- 13 Mittlere Sonne
- 14,15 Strecken zum Ursprung
- 16,17 Winkel zu MS bzw. KS
- 18 Differenz in rad
- 19 Ortspunkt $(\phi, \delta \frac{720}{\pi})$ y in Minuten
- 20 Ortskurve für P

Abb.132 : Konstruktionsprotokoll für den Keplerfehler

13.6 Kepler- und Schiefstellungskorrektur

13.6.1 Mit Geogebra als Simulation

Wir stellen die Keplersonne “schief” entlang der Achse Sonne-Frühlingpunkt (dort schneidet ja die “Sonnenbahn” (im geozentrischen System) die xy -Ebene). Wo dieser liegt (gegenüber dem Perihel), wie lang das Perihelintervall dauert (von Perihel zu Perihel) ist vom Jahr abhängig - da kommt man um eine Ephemeriden-Tafel nicht umhin, eine mögliche wäre <http://astropixels.com/ephemeris/ephemeris.html>

Bei der ersten Zeichnung geht's uns noch nicht ums Datum - einmal rundherum ist 2π (unabhängig vom Perihelintervall). Die Parameter ε (Achsenneigung), κ (Bahnexzentrizität) und α (Winkel zwischen Perihel und Frühlingpunkt) sind einstellbar. Sowohl mit κ als auch mit ε lässt sich der Stundenwinkel zwischen MS und R' (der Meridian der realen Sonne) steigern. Um den Effekt deutlich zu machen bin ich ziemlich hochgefahren:

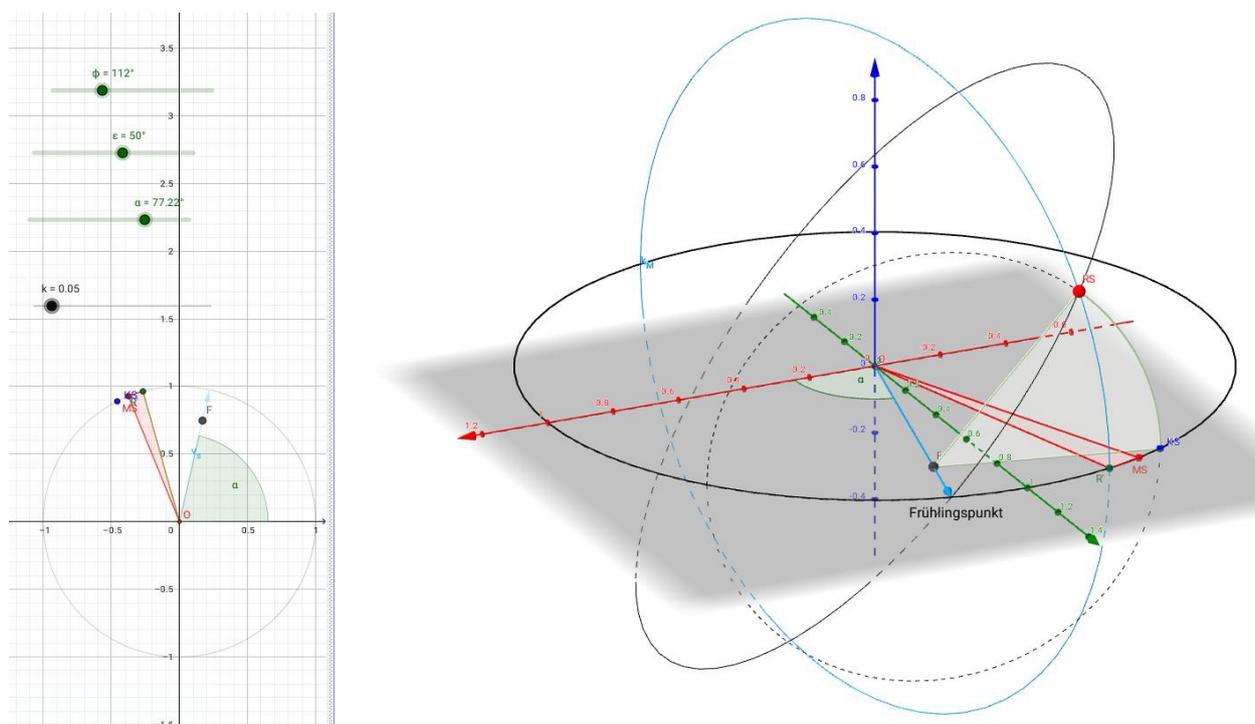


Abb.133 : Fehler mittlere Sonne reale Sonne

Die Keplersonne KS wird um die Frühlingpunktachse um ε gedreht zur realen Sonne RS , durch diese wird der (Mittags)Meridian gelegt um R' zu erhalten - der Winkel $\angle(MS, R')$ - hier mit roter Farbe hervorgehoben - ist dann der Zeitunterschied!

Zum Herunterladen

<https://www.angsuesser.at/docs/math/geogebra/realSun-error.ggb>

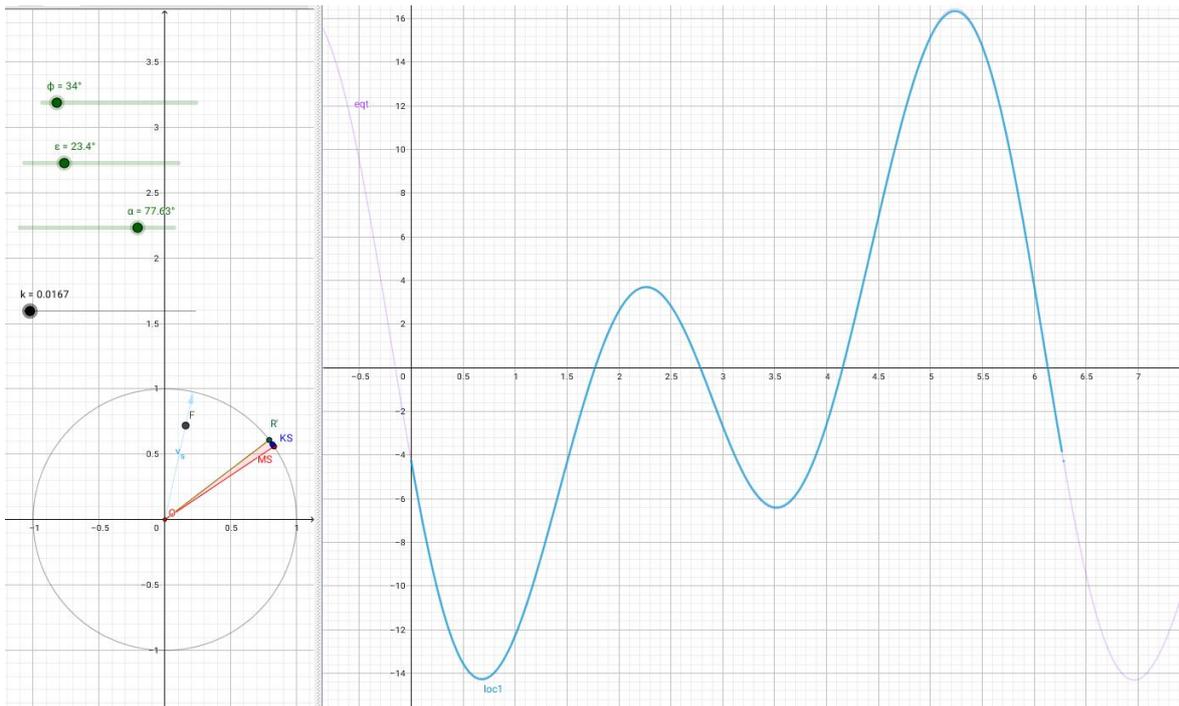


Abb.134 : Einstellung für die Erde - Ortslinie von $P(\phi, \delta[\text{min}]) = \text{Zeitgleichung}$ und Näherungsformel(Text)

Konstruktion obiger Zeichnung

- Slider für $\phi, \epsilon, \alpha, \kappa$ ✓ Orbit von $MS \rightarrow k_1 : x^2 + y^2 = 1$ ✓ Ursprung O ✓
- $MS : (\cos(\phi), \sin(\phi))$ ✓ Keplernäherung $\rightarrow \psi(t) = t + 2k \sin(t) + \dots$ ✓
- Keplersonne $KS : (\cos(\psi(\phi)), \sin(\psi(\phi)))$; ✓
- Vektor zum Frühlingspunkt: v_s : Vector $((\cos(\alpha), \sin(\alpha)))$ ✓ g_s : Line $(0, v_s)$ ✓
- Orbit reale Sonne: Rotate($k_1, -\epsilon, g_s$) ✓ RS (reale Sonne): Rotate($KS, -\epsilon, g_s$) ✓
- e_3 : Vector $((0, 0, 1))$ ✓ v_R : Vector(RS) ✓
- Richtungsvektor für Meridiankreis durch RS : $\vec{u}_R = \vec{e}_3 \times \vec{v}_R \rightarrow u_R = \text{Cross}(e_3, v_R)$ ✓
- Meridiankreis durch KS : $k_M = \text{Circle}(0, 1, u_R)$ ✓ R' : Intersection($k_M, k_1, 1$) ✓
- Fußpunkt F des Rotationskreises: $F = (\vec{v}_R \cdot \vec{v}_s) \vec{v}_s \rightarrow F = 0 + \text{Dot}(v_R, v_s) * v_s$ ✓
- Rotationskreis $c = \text{Circle}(F, \text{Distance}(RS, F), g_s)$ ✓ Hilfspunkt $X = (0, 0, 1)$ ✓
- Winkel-Messen: $\delta_M = \text{Angle}(X, 0, MS)$ ✓ $\delta_R = \text{Angle}(X, 0, R')$ ✓ $\delta = \delta_M - \delta_R$ ✓
- Punkt für die Ortslinie: $P(\phi, \delta \frac{720}{\pi})$ ✓ Locus(P, ϕ) ✓

Einige Befehle, die nur der Ästhetik dienen, wurden weggelassen. Zur Kontrolle wurde eine Näherungsformel aus den "Weiten des Internet" genommen:

$$\text{EQT (in hours)} = 3600 \cdot (-1) \cdot (-104.7 \cdot \sin(F) + 596.2 \cdot \sin(2F) + 4.3 \cdot \sin(3F) - 12.7 \cdot \sin(4F) - 429.3 \cdot \cos(F) - 2 \cdot \cos(2F) + 19.3 \cdot \cos(3F)) \quad \text{where } F = \pi / 180 \cdot (279.5 + 360 / 365 \cdot n);$$

13. Zeitgleichung

Obige Formel liefert zu jeder Jahrestagsnummer die Zeitabweichung in Stunden - wir müssen also umrechnen:

Das Ergebnis in Minuten zu erhalten ist leicht - wir multiplizieren mit 60. Für unsere Zeit gilt $\phi \in [0, 2\pi]$ wobei wir beim Perihel - das ist meist der 3. Jänner - starten, also

$$\phi = \frac{2\pi}{365} (n - 3) \Rightarrow n = \frac{365}{2\pi} \phi + 3 \Rightarrow F(\phi) = \frac{\pi}{180} 279.5 + \phi + \frac{6\pi}{365}$$

Diese Funktion wurde zur Kontrolle in *Geogebra* gezeichnet und stimmt bei einem gewissen α tadellos überein - wobei wir inzwischen wissen, das Jahr hat keine 365 Tage, das Perihel liegt nicht unbedingt am 3. Jänner, ...

13.6.2 Zeitgleichung mit *wxMaxima*

Der Konstruktionsvorgang von *Geogebra* lässt sich auch mit *wxMaxima* leicht nachvollziehen. Für die Rotation um eine beliebige Achse benutzen wir die Matrixformel von Rodrigues 4.7. Hier das Programm mit den entsprechenden Erklärungen:

Wir definieren die Kreuzproduktmatrix in der Rodriguez-Formel: Anfangszeichen - Endzeichen

```
(% i1) matchfix("|", "|_x")$
```

Eine Funktion, die aus einem Listenvektor die entsprechende Kreuzproduktmatrix bildet!

```
(% i2) f(x):=matrix([0,-x[3],x[2]], [x[3],0,-x[1]],[-x[2],x[1],0]);
```

$$f(x) := \begin{pmatrix} 0 & -x_3 & x_2 \\ x_3 & 0 & -x_1 \\ -x_2 & x_1 & 0 \end{pmatrix}$$

Funktion und Zeichenkette werden verbunden, um eine Schreibweise wie in der Rodriguez-Formel nachzukommen

```
(% i3) |n|_x:=f(n)$
```

Wir basteln die Rotationsmatrix um Einheitsvektor \vec{n} um Winkel θ

```
(% i4) Rod(n,theta):=diagmatrix(3,1)+sin(theta)*|n|_x+(1-cos(theta))*(|n|_x . |n|_x);
```

$$\text{Rod}(n, \theta) := \neq \{\text{Lispfunction}\} (3, 1) + \sin(\theta)|n|_x + (1 - \cos(\theta)) (|n|_x \cdot |n|_x)$$

Dieses Monster müssen wir nicht sehen!

```
(% i5) R:Rod([cos(alpha),sin(alpha),0], epsilon)$
```

Um die folgende Funktion zu verstehen muss man etwas ausholen:

Sie konvertiert kartesische Koordinaten in Kugelkoordinaten, gibt aber nur den Winkel ϕ zurück - jener Winkel von der x -Achse bis zum Meridian der realen Sonne, deren Koordinaten eben (x, y, z) sind.

Die kartesischen Koordinaten (x, y, z) verhalten sich zu den Kugelkoordinaten (r, θ, ϕ) wie

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = r \begin{pmatrix} \sin \theta \cos \phi \\ \sin \theta \sin \phi \\ \cos \theta \end{pmatrix} \quad \text{bei unseren Einheitsvektoren gilt } r = 1$$

$$z = \cos \theta \Rightarrow \sqrt{1 - z^2} = \sin \theta \Rightarrow \frac{x}{\sqrt{1 - z^2}} = \cos \phi \quad \wedge \quad \frac{y}{\sqrt{1 - z^2}} = \sin \phi$$

aus $\cos \phi$ und $\sin \phi$ basteln wir eine komplexe Zahl auf dem Einheitskreis und holen uns mit der *wxMaxima*-Funktion *carg* den dazugehörigen Winkel ϕ . Da *carg* negative Winkel liefert, müssen wir händisch korrigieren!

```
(% i6) meridianAngle(coords):=block([x:first(coords),y:second(coords),
                                     z:third(coords), angle:0],
    angle:carg(x/sqrt(1-z^2)+%i*y/sqrt(1-z^2)),
    if angle < 0 then angle:2*%pi+angle,
    float(angle)
)$
```

Daten der Erde, N ist Anzahl der Datenpunkte, Ψ - Keplernäherung

```
(% i7) (k:0.0167,alpha:78.5*%pi/180,epsilon:23.4*%pi/180, N:100, fpprintprec:4)$
```

```
(% i8) psi(t):=t+2*k*sin(t)+5/4*k^2*sin(2*t);   (t) := t + 2k sin(t) + 5/4 k^2 sin(2t)
```

Ermittlung der Position der realen Sonne - die brauchen wir für das *ANALEMMA*

Der Parameter t wird natürlich noch durch $\psi(t)$ ersetzt - siehe (% i10)

```
(% i9) define(posXYZ(alpha,epsilon,t),Rod([cos(alpha), sin(alpha), 0],epsilon)
    . apply('matrix,[[cos(t)],[sin(t)],[0]]));
```

```
posXYZ( $\alpha, \epsilon, t$ ) :=
```

$$\begin{pmatrix} \cos(\alpha) \sin(\alpha) (1 - \cos(\epsilon)) \sin(t) + (1 - \sin(\alpha)^2 (1 - \cos(\epsilon))) \cos(t) \\ (1 - \cos(\alpha)^2 (1 - \cos(\epsilon))) \sin(t) + \cos(\alpha) \sin(\alpha) (1 - \cos(\epsilon)) \cos(t) \\ \cos(\alpha) \sin(\epsilon) \sin(t) - \sin(\alpha) \sin(\epsilon) \cos(t) \end{pmatrix} \quad (13.18)$$

Liste der Sonnenpositionen, dann Stundenwinkel der realen und mittleren Sonne

```
(% i10) L1:makelist(posXYZ(alpha,epsilon,psi(2*%pi/N*i) ), i,0,N-1)$
```

```
(% i11) RS:map(meridianAngle,map(lambda([x], flatten(args(x) ) ,L1))$
```

```
(% i12) MS:makelist(float(2*%pi/N*i) , i,0,N-1)$
```

```
(% i13) DeltaInRad:MS-RS$
```

```
(% i14) DeltaInMin:float(720/%pi)*DeltaInRad$
```

```
(% i19) plot2d([discrete,MS,DeltaInMin],[x,0,6.28], [style,[lines,4,4,4]],grid2d,
    [xtics,0,1,6], [ytics,-14,2,16],[ylabel,"Abweichung in Min"],
    [legend,"Equation of Time"],[title,"Equation of Time"],
    [gnuplot\_preamble,"set key bottom right; set xtics font \" , 15\";
    set ytics font \" , 15\"; set key font \" , 15\";
    set title font \" , 20\ensuremath{\backslash} " ]
);
```

13. Zeitgleichung

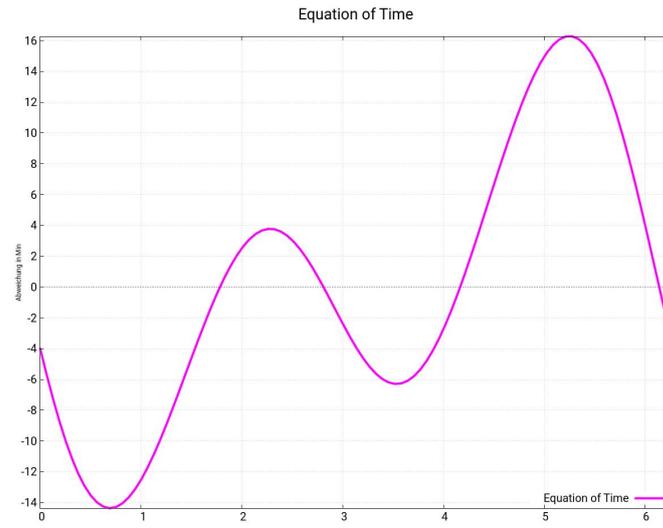


Abb.135 : Berechneter Gesamtfehler jetzt mit *wxMaxima* (Rodrigues-Formel)

13.6.3 Addition der Fehler - eine Analyse

In der astronomischen Literatur (in deren Nomenklatur ich mich nicht eingearbeitet habe) ist immer wieder von einer “Zerlegung der Abweichung auf Grund der Schiefstellung der Erdachse und auf Grund der Keplerbewegung” zu lesen - wobei diese Zerlegung eine Summe suggeriert. Im Allgemeinen stimmt das nach meiner Ansicht nicht, wie folgende Zeichnung nahelegt:

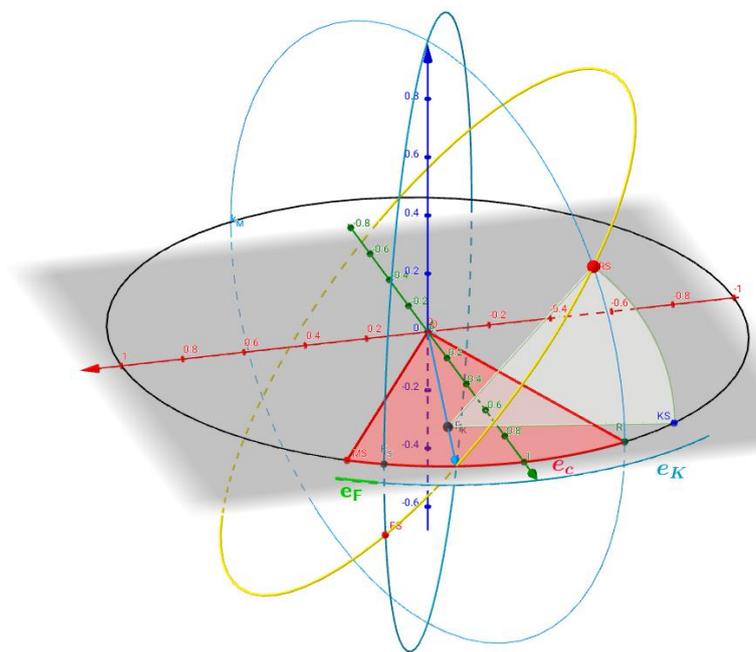
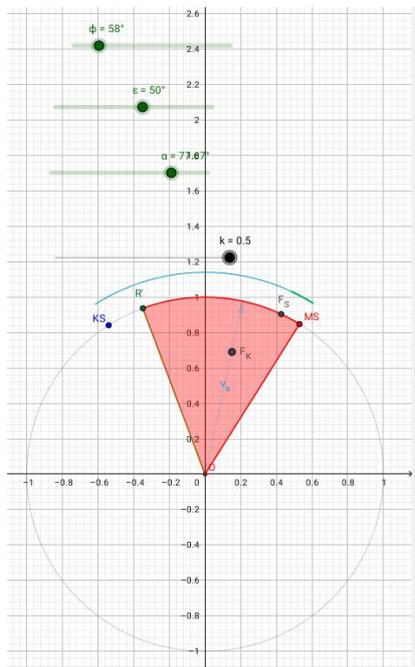


Abb.136 : Addition der Fehler auf Grund “Tilt/Kepler”

e_c ... error combined
 $e_c \neq e_F + e_K$

e_F ... error fictitious sun (tilt)

e_K ... error Kepler

Jedenfalls bei “extremen” Umlaufdaten scheint dies nicht zu stimmen - siehe die Einstellung der Parameter!

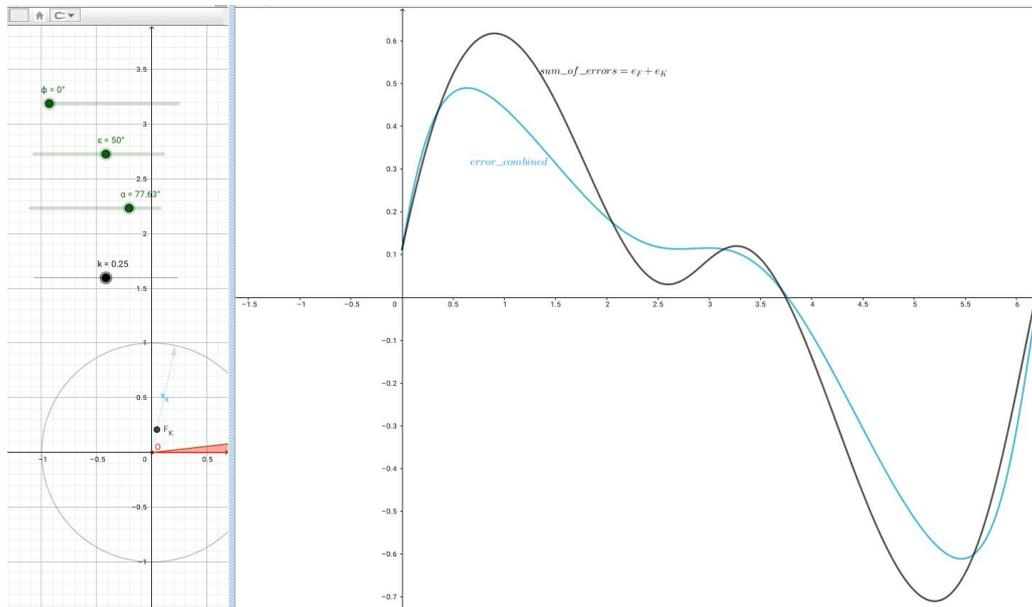


Abb.137 : Graph von Fehler *fictitious sun+ kepler sun vs. combined*

Wenn wir allerdings die Erdparameter zu Grunde legen ist der Unterschied praktisch vernachlässigbar:

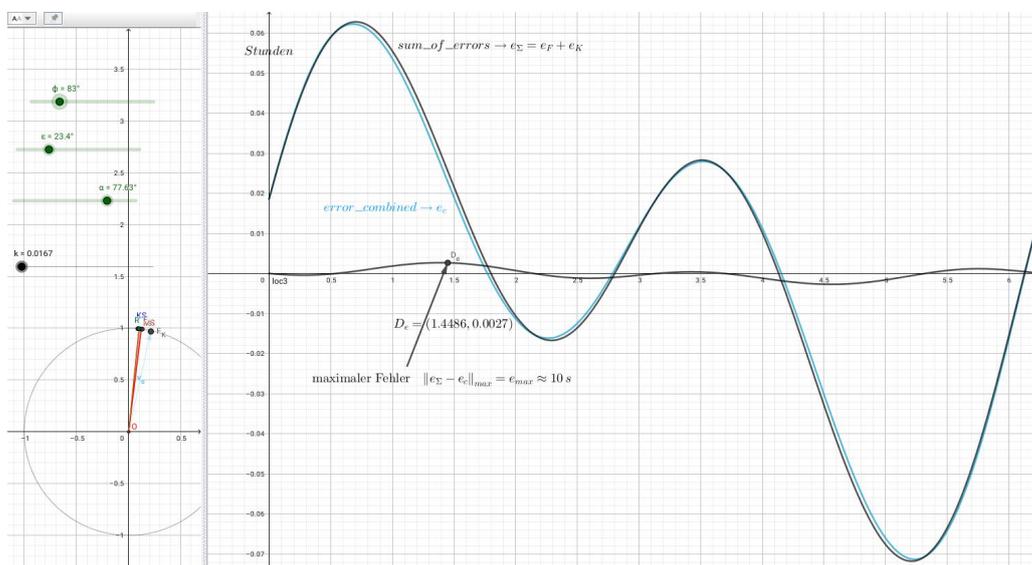


Abb.138 : Graphen bei Erdparameter und Differenzgraph



Bei der Erde ist es zulässig die Fehlerursachen zu addieren, da die anderen Einflüsse (Mond, Planeten) größere Ungenauigkeiten verursachen!

13. Zeitgleichung

Das Arbeitsblatt zum Download

<https://www.angsuesser.at/docs/math/geogebra/errorAddition.ggb>

Konstruktion der Zeichnung wie bei "Tilt- und Keplerfehler" zusammen, aber zusätzlich

- fictitious sun FS : `Rotate(MS, -ε, g_s)` ✓
- MS : $(\cos(\phi), \sin(\phi))$ ✓ Keplernäherung $\rightarrow \psi(t) = t + 2k \sin(t) + \dots$ ✓
- Keplersonne KS : $(\cos(\psi(\phi)), \sin(\psi(\phi)))$; ✓
- Vektor zum Frühlingspunkt: v_s : `Vector((cos(α), sin(α)))` ✓ g_s : `Line(0, v_s)` ✓
- Orbit reale Sonne: `Rotate(k_1, -ε, g_s)` ✓ RS (reale Sonne): `Rotate(KS, -ε, g_s)` ✓
- e_3 : `Vector((0,0,1))` ✓ v_R : `Vector(RS)` ✓
- Richtungsvektor für Meridiankreis durch RS : $\vec{u}_R = \vec{e}_3 \times \vec{v}_R \rightarrow u_R = \text{Cross}(e_3, v_R)$ ✓
- Meridiankreis durch KS : `k_M=Circle(0,1,u_R)` ✓ R' : `Intersection(k_M, k_1,1)` ✓
- Fußpunkt F des Rotationskreises: $F = (\vec{v}_R \cdot \vec{v}_s)\vec{v}_s \rightarrow F = 0 + \text{Dot}(v_R, v_s)*v_s$ ✓
- Rotationskreis $c = \text{Circle}(F, \text{Distance}(RS, F), g_s)$ ✓ Hilfspunkt $X = (0, 0, 1)$ ✓
- Winkel-Messen: $\delta_M = \text{Angle}(X, 0, MS)$ ✓ $\delta_R = \text{Angle}(X, 0, R')$ ✓ $\delta = \delta_M - \delta_R$ ✓
- Punkt für die Ortslinie: $P(\phi, \delta \frac{720}{\pi})$ ✓ `Locus(P, φ)` ✓

13.6.4 Mit *Geogebra* als Rechnung mit Datum

Wie wir gesehen haben ist es bei der Erde zulässig die Fehler aus Schiefstellung der Erdachse (Tilt) und ihrer Keplerbewegung einfach zu addieren.

Allerdings haben wir 2 Probleme

- ☞ Als Zeitachse benutzen wir $\phi \in [0, 2\pi]$. Wobei der Startpunkt für den Keplerfehler im Perihel liegt, der des Tiltfehlers im Frühlingspunkt
- ☞ "Unsere Zeitrechnung" beginnt am 1. Jänner und dauert 365 bzw. 366 Tage

Für führen eine Variable d_y ein, die die verstrichenen Tage (Dezimalzahl) seit dem 1. Jänner 0 Uhr bezeichnet. $d_y = 2.5$ ist als der 3. Jänner 12 Uhr.

Da sich die Position der realen Sonne durch die Hintereinanderausführung von mittlerer Sonne zur Keplersonne ($MS \rightarrow KS$) und anschließender Schiefstellung also Keplersonne zur realen Sonne ($KS \rightarrow RS$) näherungsweise bei der Erde darstellen lässt, brauchen wir auch nur die einzelnen Fehler addieren - natürlich zum selben Zeitpunkt!

Bisher haben wir ja 2 verschiedene Nullpunkte der Zeit verwendet:

- Für die Schiefstellung den Frühlingspunkt(im Mittel $\alpha = 77.1^\circ$ vom Perihel entfernt) t_V
- Für die Keplerkorrektur das Perihel (meist um den 3. Jan.) $\phi = t_V + \alpha$

Sei p die Periode (in Tagen) von Perihel zu Perihel (näherungsweise auch von Frühlingspunkt zu Frühlingspunkt), p_0 der Tag des Perihels, p_1 der Tag des Frühlingspunkts, dann gilt

damit ergibt sich, da ja $d_y \propto \phi$

ϕ	d_y
0	p_i
2π	$p_i + p$

$$d_y = \frac{p}{2\pi} \phi + p_i \quad \text{und} \quad \phi(d_y) = (d_y - p_i) \cdot \frac{2\pi}{p} \quad (13.19)$$

Für den Keplerfehler ist $i = 0$, für den Tiltfehler ist $i = 1$

Damit ist die Vorgangsweise klar, wir ersetzen $\delta_T(\phi)$ durch $\delta_T(\phi(d_y))$ und analog ergeht es dem Keplerfehler $\delta_K(\phi)$ - es bleibt d_y in ein Datum umzuwandeln!

Für die Berechnung von $\delta_T(\phi)$ im Arbeitsblatt benutzen wir die Fourierkoeffizienten, die wir abgespeichert haben und die wir in der spreadsheet-Ansicht von *Geogebra* mit dem Kontextmenü \rightarrow "Import Data File" einlesen können (genaue Erklärung bei 13.10). Wir haben sie dann in den Zellen A1 bis E1 zur Verfügung.

Für $\delta_K(\phi)$ benutzen wir die Näherung 13.17

Download mit

https://www.angsuesser.at/docs/math/geogebra/eqt_by_day4.ggb

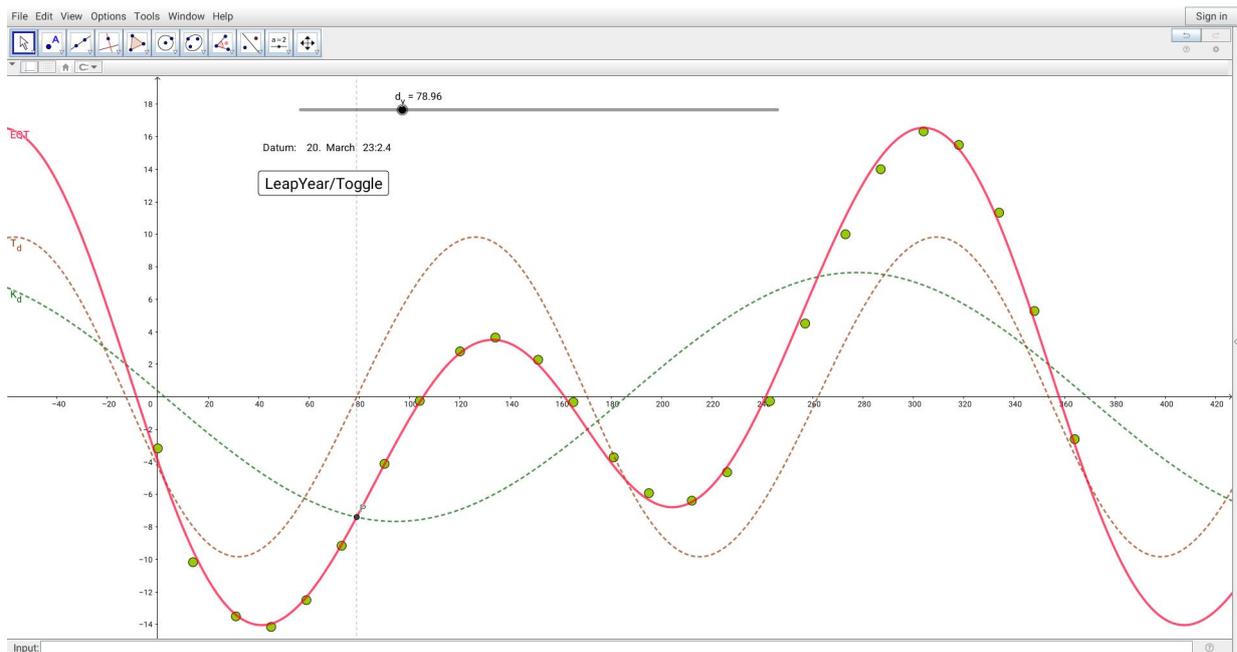


Abb.139 : Berechneter Gesamtfehler(rot) vs. Ephemeriden-Daten(grüne Punkte) für 2011

Die Ephemeridenpunkte haben einen Durchmesser von ca. 30 Sekunden - die Näherung ist nicht schlecht, aber nicht so gut wie ich mir das vorgestellt habe. Ob das nur am Jahr 2011 liegt oder ob System dahinter steckt - das sollte ein anderer herausfinden.

Nun zur Erstellung des Arbeitsblattes:

13. Zeitgleichung

- Die boole'sche Variable `leap=false` - Schaltjahr oder nicht
- Liste der Fourierkoeffizienten `LF=A1:E1`
- Liste der Sinus: `LSin=Sequence(sin(2n x), n, 1, 5)`
- Multiplikation der beiden Listen `l1=LF * LSin` und anschl. `fourier(x)=sum(l1)`
- `k=0.0167` und $\psi(x) = -(2k \sin(x) + 5/4 k^2 \sin(2k)) / 720 / \pi$
- Eingabe von p , p_0 und p_1
- Umwandlung von Kepler und Tilt auf d_y :
`g_K=(x - p_0) * 2π / p` bzw. `g_T=(x - p_1) * 2π / p`
- Jetzt Kepler- und Tiltfehler für Jahrestage: `K_d(x)=ψ(g_K(x))` bzw. `T_d(x)=fourier(g_T(x))`
- Jetzt die Zeitgleichung `EQT(x)=T_d(x) + K_d(x)`
- Jetzt kommt die Umwandlung von d_y in ein Datum:
Einen Slider d_y als Dezimalzahl (Schrittweite überlasse ich dem Leser)
- Tagesliste der Monate:
`{0, 31, If(leap, 29, 28), 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}`
Beachte Schalttag und dummy-Monat 0!
- Liste der Monatsnamen:
`mL={"Jan", "Feb", "March", "April", "May", "June", "July", "Aug", "Sep", "Oct", "Nov", "Dec", "Jan"}`
- Die akkumulierte Tagesliste: `accDL=Sequence(Sum(dL, i), i, 1, 13)`
- Eine Hilfsliste: `hL=accDL - (floor(d_y) + 1)`
Das letzte negative Element ist der Monatstag, dessen Index ist der Monat - also
- wir holen die negativen Elemente `L1=KeepIf(k < 0, k, hL)`
- Wir holen den Tag des Monats:`d_m=abs(Element(L1, Length(L1)))`
- Wir holen den Monat: `month=Element(mL, Length(L1))`
- Wieviel Sekunden sind noch übrig? `seconds=(d_y - floor(d_y)) 24 * 3600`
- Umwandlung in Stunden und Minuten:
`hours=floor(seconds / 3600)` und `minutes=(seconds - hours 3600) / 60`
- Ausgabe des Datums (`d_m`, `month`, `hours`, `minutes` in der Objektliste anklicken!)
- Zum Vergleich habe ich noch die Ephemeriden-Daten für das Jahr 2011 eingegeben - dazu habe ich extra ein *wxMaxima* - Programm geschrieben (im Anhang)

Die Zeitgleichung (Graph) wird nun so angewendet, dass ihr Wert zur “Armbanduhrzeit” addiert wird, um die lokale wahre Sonnenzeit zu erhalten:

Z.B.: Wie groß ist Abweichung am 4. März - 12 Uhr?

Der Wert der Zeitgleichung ist dort ca. -11.5 Minuten, d.h. um 11 Uhr 48.5 Minuten ist Mittag!

13.7 Analemma

Die Verschiedenheit von lokaler Uhrzeit und Sonnenzeit ergibt 2 Fragestellungen:

1. Die Sonne zeigt eine bestimmte Zeit (z.B.: 12 Uhr Mittag), wie spät ist es auf der Uhr?
Die Antwort darauf liefert die Zeitgleichung.
2. Die Uhr zeigt eine bestimmte Zeit (z.B.: 12 Uhr Mittag), wo befindet sich die Sonne?
Die Figur, die die Sonne am Himmel "zeichnet", wenn man sie am selben Ort zur selben Uhrzeit fotografiert und diese Bilder dann übereinander legt, heißt *Analemma*.
Klarerweise spielt auch hier die Zeitgleichung eine Rolle - wir werden gleich sehen welche.

13.7.1 Positionsbestimmung eines Himmelsobjekts auf der Erde

Man kann in jedem Punkt P der Erde mit einem Lot die Normale \vec{n}_P bestimmen und mit einem (Kreis)Kompass die Nordrichtung, \vec{n}_P und Richtung \vec{d} zum Himmelsobjekt bilden die sog. *Vertikalebene* - sie schließt mit der Nordrichtung über Osten einen Winkel $\alpha \in [0, 360^\circ]$ (*Azimut*) ein. Der Höhenwinkel $\beta \in [0, 90^\circ]$ heißt *Elevation* (engl. Altitude). Sowohl Azimut als auch Elevation ändern sich mit der geografischen Breite bzw. Uhrzeit.

Der Schlüssel zur Berechnung beider Winkel liegt in Zerlegung von \vec{d} in Komponenten in Richtung $\vec{n}_P \rightarrow \vec{d}_{\parallel}$ und senkrecht zu $\vec{n}_P \rightarrow \vec{d}_{\perp}$ (die Projektion von \vec{d} auf die Horizontalebene!) $\vec{n}_P \times \vec{d}$ dient dazu zu bestimmen, ob $\alpha > 180^\circ$ ist

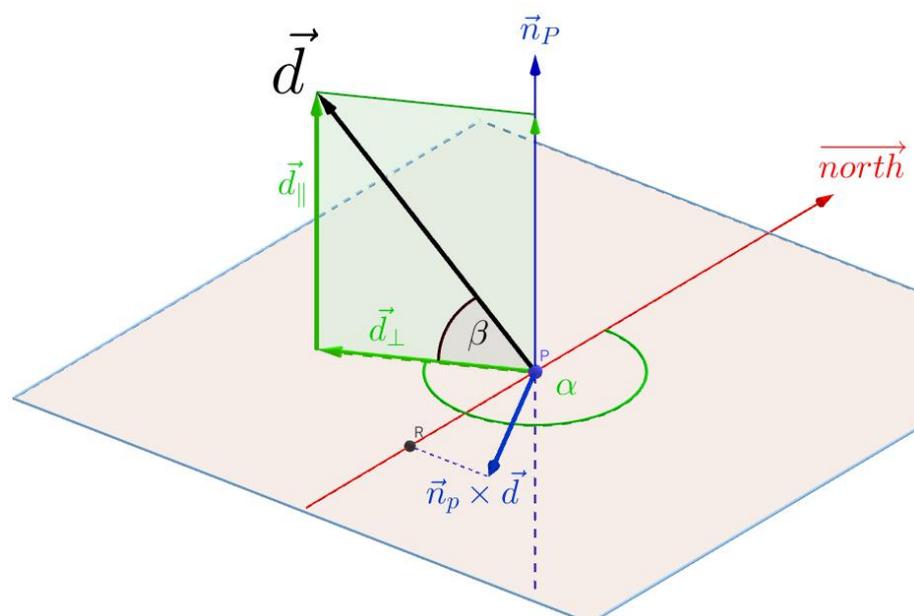
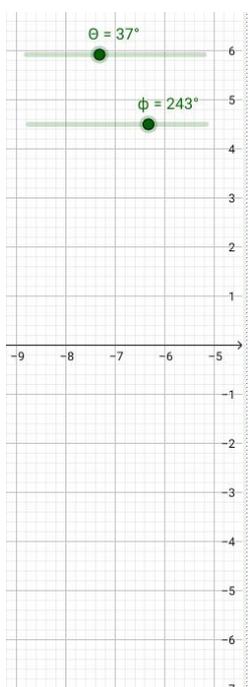


Abb.140 : Azimut α und Elevation β eines Objekts

Download des Arbeitsblattes

<https://www.angsuesser.at/docs/math/geogebra/azimut.ggb>

Wenn man \vec{d} mit den Schieberegler verändert, dann blendet man den "Bezeichnungstext" am besten aus - er stört dann, weil er sinnlos am Bildschirm herumhängt!

Obiges Arbeitsblatt bedient sich folgender Berechnung:

\vec{d} , \vec{n}_P und $\vec{north} =: \vec{N}$ seien normiert (Einheitsvektoren), dann gilt

- 1) $\vec{d}_{\parallel} = (\vec{n}_P \cdot \vec{d}) \vec{n}_P$
- 2) $\vec{d}_{\perp} = \vec{d} - \vec{d}_{\parallel}$ oder gleich $\vec{n}_P \times [\vec{d} \times \vec{n}_P] \stackrel{4.13}{=} (\vec{n}_P \cdot \vec{n}_P) \vec{d} - (\vec{n}_P \cdot \vec{d}) \vec{n}_P = \vec{d} - \vec{d}_{\parallel}$
- 3) $\vec{n}_P \cdot \vec{d} = \sin \beta \geq 0$ sonst Sonne unsichtbar $\Rightarrow \cos \beta = \sqrt{1 - \sin^2 \beta}$
- 4) $|\vec{d}_{\perp}| = \cos \beta \Rightarrow (\vec{d}_{\perp})_0 = \frac{1}{\cos \beta} \vec{d}_{\perp}$ Einheitsvektor
- 5) $\cos \alpha_1 = (\vec{d}_{\perp})_0 \cdot \vec{N} = \frac{1}{\cos \beta} \vec{d}_{\perp} \cdot \vec{N} = \frac{1}{\cos \beta} (\vec{d} - \vec{d}_{\parallel}) \cdot \vec{N} = \frac{1}{\cos \beta} [\vec{d} - (\vec{n}_P \cdot \vec{d}) \vec{n}_P] \cdot \vec{N}$
- 6) $\vec{N} \perp \vec{n}_P \Rightarrow \cos \alpha_1 = \frac{1}{\cos \beta} \vec{d} \cdot \vec{N}$
- 7) $\vec{n}_P \times \vec{d}$ hat eine positive Nordkomponente falls $0 \leq \alpha \leq \pi \Rightarrow$
- 8) $(\vec{n}_P \times \vec{d}) \cdot \vec{N} \geq 0 \quad \alpha = \alpha_1 \quad \text{sonst} \quad \alpha = 360^\circ - \alpha_1$

So - damit hätten wir einen Algorithmus, der aus \vec{d} , \vec{n}_P und \vec{N} Azimut und Elevation berechnet!

Jetzt ist bekanntlich die Erde nur lokal eine Scheibe, wesentlich besser eignet sich die Kugel als Modell. Wir führen ein kartesisches Koordinatensystem ein und ohne die Allgemeinheit zu beschränken, liege P in der xz -Ebene und habe die geografische Breite φ , dann gilt

$$\vec{n}_P = \overrightarrow{OP} = \begin{pmatrix} \cos \varphi \\ 0 \\ \sin \varphi \end{pmatrix}$$

der Meridiankreis auf dem P liegt hat dann die Gleichung

$$k_M : \begin{pmatrix} \cos t \\ 0 \\ \sin t \end{pmatrix} \quad t \in [0, 2\pi]$$

die Nordrichtung ist die Tangente an k_M in φ :

$$\vec{t}_P(\varphi) = \frac{d}{dt} \begin{pmatrix} \cos t \\ 0 \\ \sin t \end{pmatrix} (\varphi) = \begin{pmatrix} -\sin \varphi \\ 0 \\ \cos \varphi \end{pmatrix} = \vec{N} \quad \text{Beachte: } |\vec{t}_P(\varphi)| = 1$$

13. Zeitgleichung

Eine andere Methode (ohne Differentialrechnung) den Nordvektor zu bestimmen, geht über den Ostvektor \overrightarrow{east} :

$$\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \times \vec{n}_P = \overrightarrow{east} \quad \text{und} \quad \vec{n}_P \times \overrightarrow{east} = \overrightarrow{north}$$

mit der Graßmann-Identität 4.13 folgt

$$\vec{n}_P \times [\vec{e}_3 \times \vec{n}_P] = \underbrace{(\vec{n}_P \cdot \vec{n}_P)}_1 \vec{e}_3 - \underbrace{(\vec{n}_P \cdot \vec{e}_3)}_{\sin \varphi} \vec{n}_P = \begin{pmatrix} -\sin \varphi \cos \varphi \\ 0 \\ \cos^2 \varphi \end{pmatrix} = \cos \varphi \vec{t}_P(\varphi)$$

Für die Konstruktion des Analemmas müssen wir die Kugelkoordinaten-Winkel θ und ϕ kennen - aber um 12 Uhr Mittag sollte die Sonne über dem Meridian durch P stehen - also $\phi = 0$ - die Zeigleichung sagt, dass es nicht so ist. Abb. 141 zeigt die Situation (stark übertrieben) für einen negativen Wert der Zeitgleichung - die Uhr zeigt 12 aber die Sonne steht erst auf 11 - Mittag kommt erst noch (Blick auf Nordpol, Osten geht Sonne auf).

- ϕ folgt aus der Zeitgleichung $\phi(t) = eqt(t)$ für ein bestimmtes $t \in [0, 2\pi]$, wobei wir den Umrechnungsfaktor für Minuten weglassen!
- θ bekommen wir aus der momentanen Position der realen Sonne aus 13.18:

$$\vec{x}_{RS} = \begin{pmatrix} \cos(\alpha) \sin(\alpha) (1 - \cos(\varepsilon)) \sin \psi(t) + (1 - \sin(\alpha)^2 (1 - \cos(\varepsilon))) \cos \psi(t) \\ (1 - \cos(\alpha)^2 (1 - \cos(\varepsilon))) \sin \psi(t) + \cos(\alpha) \sin(\alpha) (1 - \cos(\varepsilon)) \cos \psi(t) \\ \cos(\alpha) \sin(\varepsilon) \sin \psi(t) - \sin(\alpha) \sin(\varepsilon) \cos \psi(t) \end{pmatrix}$$

da $|\vec{x}_{RS}| = 1$ gilt:

$$\cos \theta(t) = \vec{e}_3 \cdot \vec{x}_{RS} = \cos(\alpha) \sin(\varepsilon) \sin \psi(t) - \sin(\alpha) \sin(\varepsilon) \cos \psi(t) = \sin(\varepsilon) \sin(\psi(t) - \alpha) \quad (13.20)$$

Die Position der Sonne um 12 Uhr ist also

$$\vec{d} = \begin{pmatrix} \sin \theta(t) \cos \phi(t) \\ \sin \theta(t) \sin \phi(t) \\ \cos \theta(t) \end{pmatrix} = \begin{pmatrix} \sqrt{1 - [\sin(\varepsilon) \sin(\psi(t) - \alpha)]^2 \cos^2(eqt(t))} \\ \sqrt{1 - [\sin(\varepsilon) \sin(\psi(t) - \alpha)]^2 \cos^2(eqt(t))} \\ \sin(\varepsilon) \sin(\psi(t) - \alpha) \end{pmatrix} \quad (13.21)$$

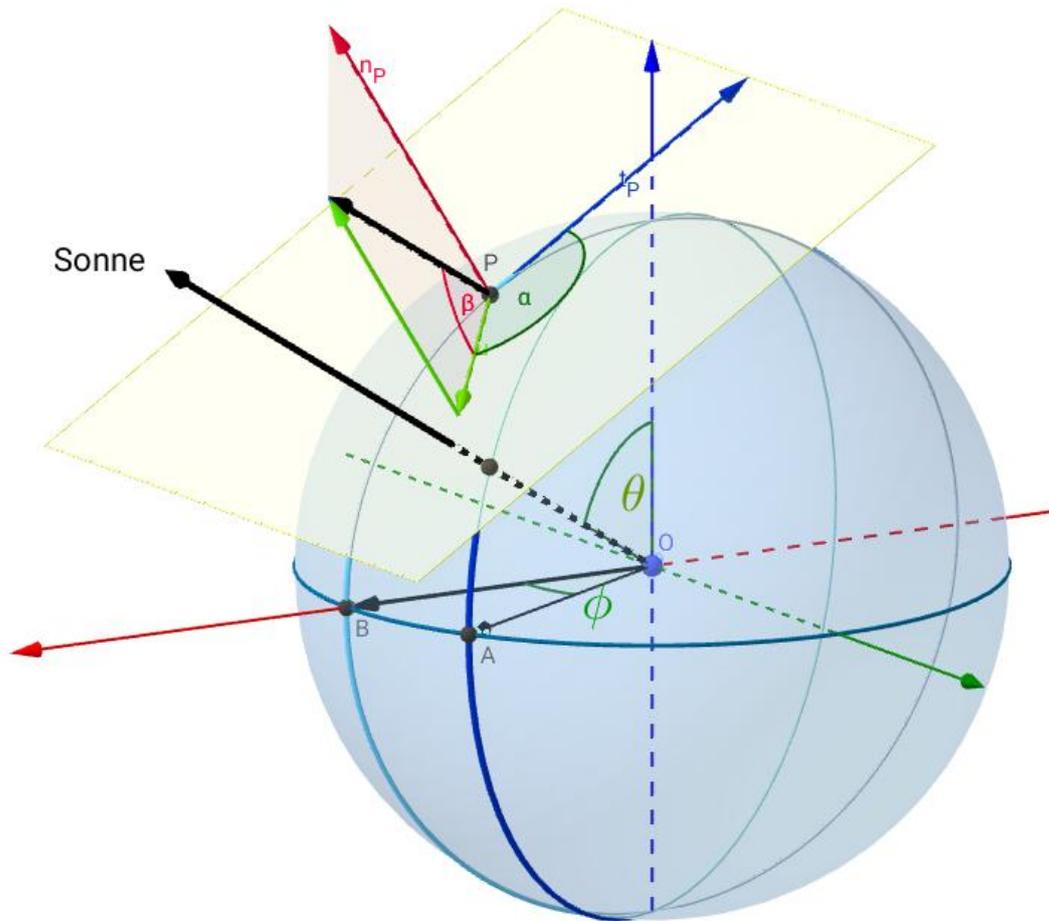


Abb.141 : Konstruktion des Analemmas

Nun zur Erstellung des Arbeitsblattes:

- Slider ϕ , θ (Kugelkoordinaten) und φ (geograf. Breite), Ursprung O und Einheitskugel
- Nullmeridian `Curve((cos(t),0,sin(t),t,0,2*pi)` und P darauf mit geograf. Breite φ : `P=(cos(phi),0,sin(phi))`
- Vector \vec{d} in Richtung Sonne: `d:Vector((sin(theta)cos(phi), sin(theta)sin(phi), cos(theta))`
- Normalenvektor in P ist \vec{n} : `n:Vector(O,P)` und Nordvektor in P : `t_P:Vector((-sin(phi),0,cos(phi))`
- Jetzt die Zerlegung von \vec{d} in Komponenten wie im Text beschrieben:
`d_||: Dot(d,n)*n` und `d_⊥: d - d_||`
- Nun die Hilfspunkte: `P_1=P+d` `P_2=P+n` `P_3=P+t_P` `P_4=P+v_⊥`
- β (Elevation) ist jetzt der Winkel zwischen $P_4.P$ und $P_1.P$, α (Azimut) jener zwischen $P_4.P$ und P_3

13. Zeitgleichung

Das Arbeitsblatt

<https://www.angsuesser.at/docs/math/geogebra/analemma1.ggb>
wurde etwas aufwendiger gestaltet, es beinhaltet

- die Vertikalebene und
- Tangentialebene in P
- Äquator und Meridiane
- Für $\vec{n}_P \cdot \vec{d} < 0$ wird β auf unsichtbar geschaltet.

Zu tun bleibt u.a. noch, dass das Arbeitsblatt auch für die Südhalbkugel sauber funktioniert.

13.7.2 Analemma

Während im vorigen Abschnitt die Position der Sonne (des Himmelskörpers) mit Schiebereglern bestimmt wurde, legen wir sie jetzt aus ihrer Umlaufbahn fest und ermitteln dazu Elevation und Azimut - fertig ist das Analemma!

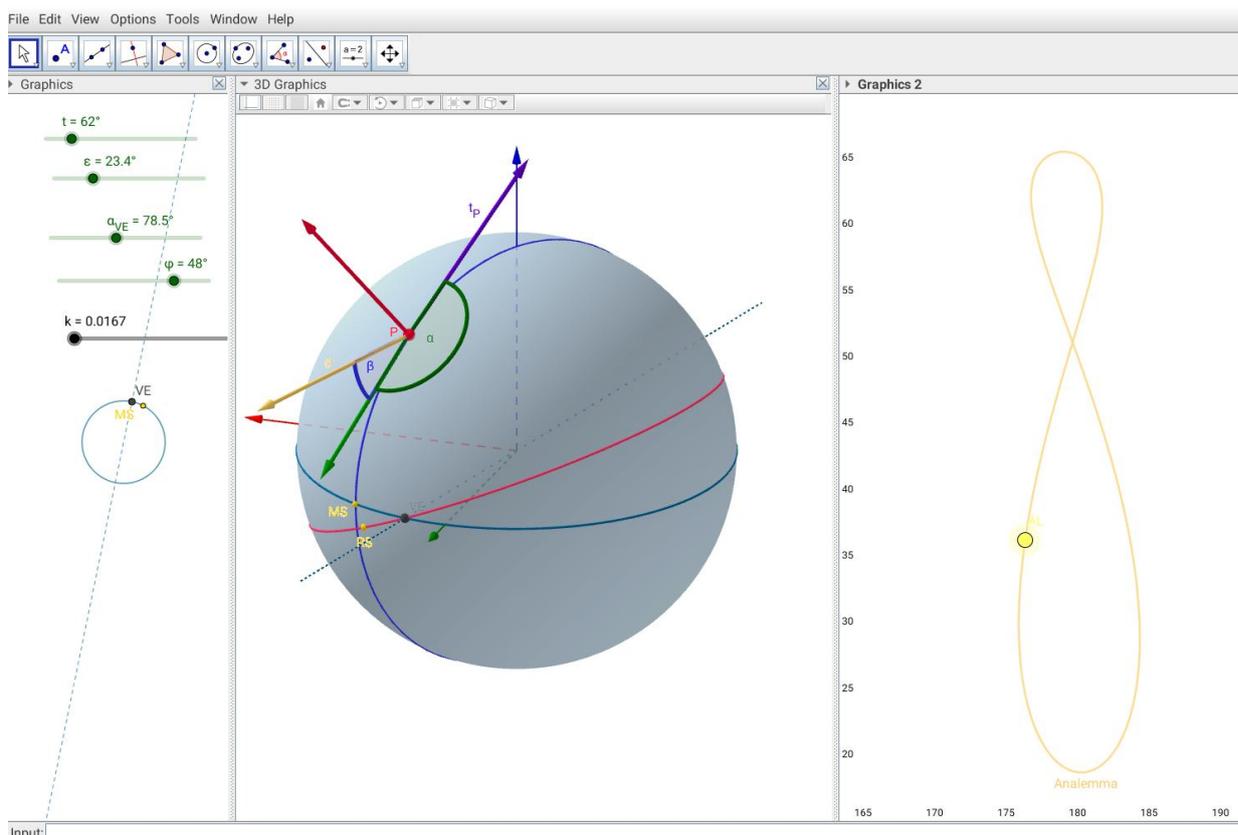


Abb.142 : Analemma bei 48° Nord auf der Erde für 12 h Uhrzeit (mittlere Sonne im Zenit)

Erstellung des Arbeitsblattes(<https://www.angsuesser.at/docs/math/geogebra/analemma7.ggb>):

- Slider für $t \in [0, 2\pi]$ (Zeit), ε (Erdachsenneigung), α_{VE} (Winkel für Vernal Equinox), φ (geografische Breite des Beobachters) und k (Bahnexzentrizität)
- Erde als Einheitskugel `earth: x^2+y^2+z^2=1`
- Punkte: Ursprung O , Nordpol NP , Südpol SP , Vernal Equinox $VE: (\cos(\alpha_{VE}), \sin(\alpha_{VE}))$
mittlere Sonne $MS: (\cos(t), \sin(t))$ und $X_1: (1, 0, 0)$
- Erdäquator `eq: x^2+y^2=1` (vorher auf 2D-Grafik klicken!)
- Rotationsgerade durch den Frühlingspunkt festlegen `g_s: y=tan(alpha_VE)*x`
- Kepler-Näherung $\psi(t)=t+2*k*\sin(t)+5/4*k^2*\sin(2*t)$
- Kepler-Sonne $KS: (\cos(\psi(t)), \sin(\psi(t)), 0)$
- Orbit der realen Sonne `orbitRS: Rotate(eq, -epsilon, g_s)`
- reale Sonne `RS: Rotate(KS, -epsilon, g_s)`
- Meridian durch reale/mittlere Sonne `k_R: Circle(RS, NP, SP)` bzw. `k_M: Circle(MS, NP, SP)`
- Stundenwinkel von RS : $R': \text{Intersect}(eq, k_R, 1)$ und $\delta_R: \text{Angle}(X_1, 0, R', zAxis)$
- Zeitgleichung ist jetzt: `eqt=t-delta_R`
- Die Achse von k_M ist $\overrightarrow{MS} \times \overrightarrow{NP}$ - um den Beobachtungspunkt P zu erhalten, rotieren wir MS um diese Achse: `P: Rotate(MS, phi, Line(0, Cross(Vector(MS), Vector(NP))))`
- Vektor zur Sonne `d: Vector(P, P+RS)`
- Nordrichtung in P : `t_1: Tangent(P, k_M)` und `t_P: Direction(t_1)`
- Normalvektor in P : `n_P: Vector(P, P+P)`
- Komponente von \vec{d} senkrecht zu \vec{n}_P : `d_perp: Cross(n_P, Cross(d, n_P))`
- Vorläufiger Azimut: $\alpha: \text{Angle}(P + t_P, P, P + d_perp, \text{Vector}(-n_P))$
- Vorläufige Elevation: $\beta: \text{Angle}(P + d, P, P + d_perp)$
- `Elevation: If(Dot(n_P, d) >= 0, beta * 180/pi, undefined)`
`Azimut: If(Elevation < 89.95, alpha * 180/pi, undefined)`
- Ziel `AL: (Azimut, Elevation)` und `Analemma: Locus(AL, t)`

13. Zeitgleichung

Für P (Beobachtungspunkt) auf der Südhalbkugel lässt sich das Arbeitsblatt nur bedingt verwenden, weil der Azimut beim Winkel 361 deg natürlich auf 1 deg springt - der Graph "springt" daher - ist also nicht mehr zusammenhängend. Statt dem Nordvektor muss man also einen Südvektor als Referenz nehmen!

Wie man oben sieht, ist die Elevation nur sinnvoll, wenn die Sonne über dem Horizont steigt - nördlich des Polarkreises ist das im Nordwinter nicht der Fall.

Wenn die Sonne außerdem im Zenit steht, hat es keinen Sinn von einem Azimut zu sprechen - auch hier werden keine Punkte mehr gezeichnet!

Interessant ist auch wenn man das Analemma südlich des nördlichen Wendekreises betrachtet. Man kann das Analemma nicht nur bei verschiedenen Breitengraden betrachten, sondern sieht auch den Einfluss von Bahnexzentrizität, Erdachsenneigung und Verlegung des Frühlingspunkts.

13.8 Lichter Tag - "Wiederaufnahme des Verfahrens"

Eine vereinfachte Untersuchung der Tageslänge haben wir schon in Kapitel 12 unternommen. Wir untersuchen diesen Sachverhalt jetzt wieder und schauen uns den Unterschied an:

Wir nehmen das Koordinatensystem von Abb. 141. Der Punkt P befinde sich auf dem Mittagmeridian der Sonne, Kugelkoordinate ϕ sei hier Null - jetzt eine Vereinfachung - die Sonne bewegt sich nicht weiter. Der Punkt P bewegt sich auf dem Breitenkreis mit der Zeit H (in Stunden) innerhalb von 24 h runderherum.

Also sind die Kugelkoordinaten (θ, ϕ) von Sonne und Beobachtungspunkt P (Breite φ):

$$\left\{ \begin{array}{l} S(\theta, 0) \\ P(\pi/2 - \varphi, \underbrace{H[h] \cdot 15 \cdot \pi/180}_{\tau}) \end{array} \right\} \begin{array}{l} \text{kartesisch} \\ \Downarrow \\ \end{array} S = \begin{pmatrix} \sin \theta \\ 0 \\ \cos \theta \end{pmatrix}, P = \begin{pmatrix} \cos \varphi \cos \tau \\ \dots \\ \sin \varphi \end{pmatrix}$$

Da sowohl S und P Einheitsvektoren sind, ist ihr skalares Produkt der Sinus der Elevation β :

$$\sin \beta(t, \tau) = \overrightarrow{OS} \cdot \overrightarrow{OP} = \sin \theta \cos \varphi \cos \tau + \cos \theta \sin \varphi \quad (13.22)$$

Da sowohl $\sin \theta \geq 0$ und $\cos \varphi \geq 0$ hat 13.22 bei $\tau = 0$ sein Maximum - die Sonne steht zu Mittag am höchsten!

Die erste Nullstelle τ_1 von 13.22 ist also der Zeitpunkt des Sonnenuntergangs (falls β verschwindet, macht dies auch $\sin \beta$).

Die Länge des lichten Tages (ohne atmosphärische Lichtbrechung) ist dann also doppelt so lang: $T = 2\tau_1$, wobei aus 13.22 und 13.20 folgt

$$\tau_1 = \arccos \left(-\tan \varphi \frac{\cos \theta}{\sin \theta} \right) = \arccos \left(-\tan \varphi \underbrace{\frac{\sin \varepsilon \sin(\psi(t) - \alpha)}{\sqrt{1 - (\sin \varepsilon \sin(\psi(t) - \alpha))^2}}}_{\cot \theta} \right) \quad (13.23)$$

Da

$$T(t, \varepsilon, \varphi, \kappa, \alpha) = \frac{24}{\pi} \arccos(-\tan \varphi \cot \theta(t)) \quad (13.24)$$

Wenn wir dies mit der früheren Formel 12.2 vergleichen

$$T(\varphi, n) = \frac{24}{\pi} \arccos(-\tan \varphi \cot \alpha_n)$$

dann erkennt man, dass die Grundstruktur der Formel gleich ist, nur unserer jetzigen $\theta(t)$ entspricht unserem früheren α_n .

Schauen wir uns das genauer an: In 12.1 wird festgelegt

$$\cos \alpha_n = \cos \alpha_0 \cdot \cos \beta_n = \sin \varepsilon \cos \beta_n \quad (13.25)$$

wobei β_n bei der Sommersonnenwende beginnt - mit unser jetzigen Zeitskala gilt also

$$t = \beta_n + \alpha + \pi/2 \quad \text{zwischen Equinoxen und Sonnenwenden ist ein rechter Winkel!}$$

Wenn wir diese Beziehung in 13.25 einsetzen ergibt sich

$$\cos \alpha_n = \sin \varepsilon \cos(t - \alpha - \pi/2) = \sin \varepsilon \sin(t - \alpha) = \cos \alpha_n$$

andererseits gilt

$$\cos \theta(t) = \sin \varepsilon \sin(\psi(t) - \alpha)$$



Wir hätten also in der alten Formel nur t durch die Keplernäherung $\psi(t)$ ersetzen müssen (Startpunkt anpassen) und das Ziel wäre erreicht gewesen!

Hier jetzt der minimale Unterschied (≤ 7 min) zwischen beiden Näherungen:

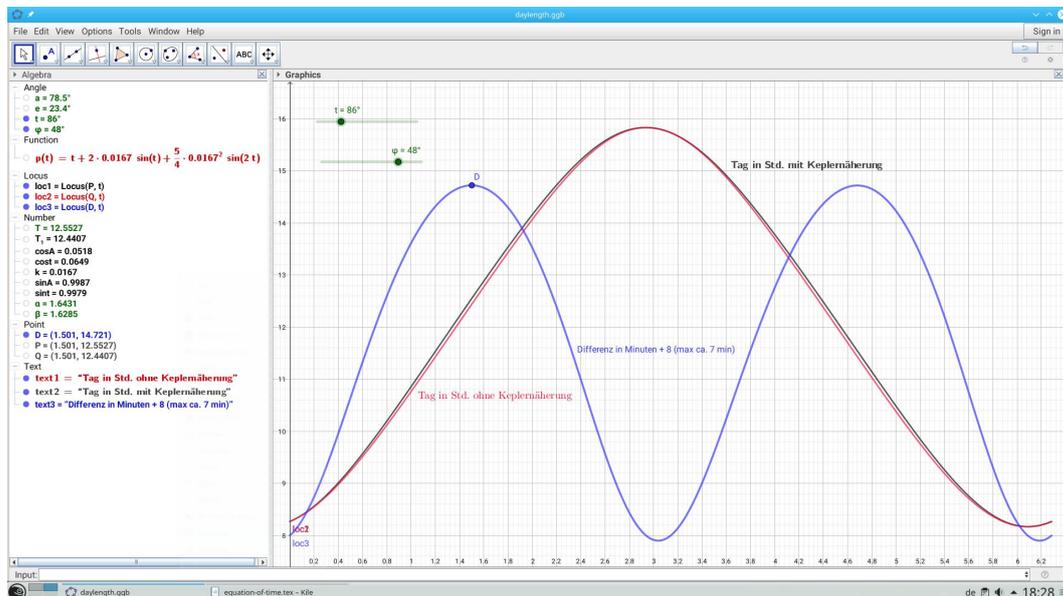


Abb.143 : Vergleich der beiden Näherungen für lichten Tag - max. Differenz ≈ 7 min

13. Zeitgleichung

Download: <https://www.angsuesser.at/docs/math/geogebra/daylength.ggb>
Konstruktionsprotokoll selbsterklärend!

13.9 Sonnenauf- bzw. Sonnenuntergang

Durch das Aufkommen der Eisenbahn (große Distanzen!) ergab sich das Problem:
Nach welcher Zeit wird der Fahrplan erstellt?

Man schlug vor, die Erde in 24 Zeitzonen (je 1 Stunde Differenz) einzuteilen und die Bezugsmeridiane sollten die Vielfachen von 15 (Grad) sein.

Die meisten Länder schlossen sich dieser Empfehlung an - wobei oft länderspezifische Interessen Vorrang vor dem Bezugsmeridian bekamen. Zum Beispiel liegen Spanien und Frankreich um den Nullmeridian, benutzen aber die mitteleuropäische Zeit mit dem Bezugsmeridian 15° Ost. Unsere Uhren (mittlere Sonne) würden bei diesen Bezugsmeridianen - wenn die mittlere Sonne dort im Zenit steht - genau 12 Uhr anzeigen! Pro Grad Abstand zum Bezugsmeridian haben wir einen Fehler von 4 Minuten. Ist die Abweichung d_M nach Osten geht unsere Uhr nach, ist sie nach Westen, zeigt unsere Uhr schon 12, obwohl die mittlere Sonne noch nicht ihren Zenit erreicht hat. Für die reale Sonne kommt dann noch der Fehler der Zeitgleichung $eqt(t)$ dazu! Sowohl Tageslänge $T(t)$ und Fehler aus der Zeitgleichung $eqt(t)$ sind von der "Jahreszeit" t abhängig, d_M ist es nicht.

Damit ergibt sich für Sonnenauf- bzw. Sonnenuntergang folgender Sachverhalt:

Um die Uhrzeit für den Zenit der realen Sonne $Z(t)$ zu bekommen, addieren wir zu 12 die Zeitgleichung und d_M die Differenz von Zeitzonenbezugsmeridian d_{TZ} und lokaler Länge d_{loc}

$$Z(t) = 12 + eqt(t) + \underbrace{d_{TZ} - d_{loc}}_{d_M} \Rightarrow S_{\pm}(t, \varphi) = Z(t) \pm \frac{T(t, \varphi)}{2} \quad (13.26)$$

$S_-(t, \varphi)$ ist die Uhrzeit des Sonnenaufgangs, $S_+(t, \varphi)$ die Uhrzeit des Sonnenuntergangs.

Sonnenauf- bzw. -untergang in Santiago de Compostela am 1. November

Ortsdaten

42.88 Breite, -8.54 Länge, Zeitzone CET $\Rightarrow d_{TZ} = 15^\circ$, $d_M = 23.84^\circ = 95.36 \text{ min}$

Mit *eqt_by_day4.ggb* stellen wir fest, dass der Wert der Zeitgleichung da 16.62 Minuten (positiv) beträgt und der 1. November der 304-te Tag im Jahr ist.

Mit Formel 13.19 (Annahmen $p_0 = 2.75$ und $p = 365.25$) ergibt sich ein Bogenwinkel $t_1 \approx 5.179 \text{ rad}$

$eqt(t_1) + d_M \approx 1.87 \text{ h}$ - die reale Sonne hinkt der mittleren Sonne(Uhr) also beträchtlich hinterher - sie steht erst $\approx 1.87 \text{ h}$ später im Zenit.

Die Tageslänge $T(t_1, \varphi)$ ist dort 10.27 h (sie bekommen wir mit *daylength.ggb*), damit ergibt sich

$$S_{\pm}(t_1, \varphi) = 12 + 1.87 \pm 5.135 \Rightarrow \begin{cases} S_+(t_1) & = 19 \\ S_-(t_1) & = 8.73 \approx 8 \text{ h } 44 \text{ min} \end{cases}$$





Sonnenauf- und -untergangszeiten können beachtlich unsymmetrisch um 12 Uhr liegen

13.10 ANHANG1: Datenlisten von *wxMaxima* nach *Geogebra*

13.10.1 Mit dem Clipboard und einem Editor

Eine Möglichkeit, die immer funktioniert (aber eben umständlich ist) ist über das Clipboard und über einen Editor (der darf keine Formatierungen einfügen!) als Zwischenstufe:

1. Kopieren der Daten von *wxMaxima* in das Clipboard
2. Vom Clipboard in den Editor und dort *Suchen und Ersetzen*, Listenklammern: $[\rightarrow \{$,
Punkteklammern: $[\rightarrow ($, usw.
Anschl. wieder alles markieren und ins Clipboard
3. Jetzt *Paste* in *Geogebra*

Aber mit CSV-Dateien geht's auch anders:

13.10.2 Export von *wxMaxima*

Wir demonstrieren das an Hand der Ephemeriden-Daten.
Diese Daten haben in meiner benutzten Tabelle die Form

Monat	Tag	Uhrzeit	Zeitgleichung
Feb	06	15.5	-8.3

Dabei bedeutet $15.5 \rightarrow 15\text{ h }50\text{ min}$ und $-8.3 \rightarrow -8\text{ min }30\text{ sec}$. (In der Originaltabelle ist natürlich statt des Dezimalpunktes eine Doppelpunkt - aber so kann man keine Zahl schreiben.) Wir verwandeln dann das korrekt. Die Uhrzeit ist bei meiner Tabelle sowieso 0.0 - das braucht keine Aufbereitung!

Hier das Programm zur Erstellung einiger Datenpunkte:

Schaltjahr? Dezimalzahlausgabe

```
(% i1) [leap:false, fpprintprec:5]$
```

Monatsliste

```
(% i2) months:[jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dec,jan2]$
```

Monatstage - beachte Schalttag!

```
(% i3) day_list:[31,if leap then 29 else 28,31,30,31,30,31,31,30,31,30,31,31]$
```

Falten der Liste mit der Funktion *f* - Zwischenergebnisse in Liste *L1*

```
(% i4) fold(L1,L2,f):= if length(L2)>0 then fold( endcons(f(last(L1), first(L2)), L1), rest(L2,1),f)
else L1$
```

13. Zeitgleichung

Falten der Liste mit der Funktion f - nur Endergebnis (wird hier nicht gebraucht!)

```
(% i5) fold1(L1,L2,f):= if length(L2)>0 then fold1(endcons(f(last(L1), first(L2)), L1), rest(L2,1),f)
      else last(L1)$
```

Akkumulierte Monatstage im Jahr

```
(% i6) accumL:=fold([0], day_list, "+"); [0,31,59,90,120,151,181,212,243,273,304,334,365,396]
```

Welchen Index hat ein Monat in der Monatsliste?

```
(% i7) get_index_of_month(x):=first(sublist_indices(months, lambda([x1], x1 = x)))$
```

Wir konvertieren 12 *h30 min* → Eingabe: 12.3 in Dezimaltage

```
(% i8) hours2days(t):=block([h:0,m:0], h:floor(t), m:(t-h)*100, 1/24.0*h+1/(24.0*60)*m ) $
```

-12 min 35 sec Eingabe: -12.30 werden in *echte* Dezimalminuten konvertiert

```
(% i9) toMinutes(m):= (floor(abs(m)) + (abs(m)-floor(abs(m)) )*100/60 )*signum(m) $
```

Aus Listenlementen wie [jan,15,0.0,-9.07] wird ein Punkt erzeugt: [14.0,-9.1167]

```
(% i10) create_point(L):=block([m:first(L), d:second(L),t:third(L), eqt:fourth(L),days:0],
      days: accumL[get_index_of_month(m)]+(d-1),
      days:days + hours2days(t),[days,toMinutes(eqt)]) $
```

Eingabe der Daten; das Wort "aus" hält die Eingabe an!

```
(% i11) inp():=(dL: [],
      block([m,d,t:0.0,minutes],
      m:read("monat (aus->schluss): "),
      while m # aus do (
      d:read("Tag des Monats: "),
      minutes:read("Zeitgleichung (-8:30 -> -8.30): "),
      dL:endcons([m,d,t,minutes],dL),
      m:read("monat (aus->schluss): ")
      )))$
```

```
(% i12) inp()$
```

Das Abspeichern ist auskommentiert, damit die Daten nicht zerstört werden

```
(% i13) /*dL_2011:*/dL$
```

```
(% i14) dL_2011; [[jan,1,0.0,-3.1],[jan,15,0.0,-9.07],...]
```

```
(% i15) points:=map(create_point, dL_2011); [[0.0,-3.1667],[14.0,-10.167],...]
```

Für den Import in *Geogebra* zerlegen wir die Punkteliste in x-Koordinaten- und y-Koordinatenlisten

```
(% i16) getXY(L, orderFunc):=map(lambda([x], orderFunc(x)), L) $
```

CSV-Dateien(Komma separiert mit Punkt Dezimalpunkt) für den Import in *Geogebra* schreiben

```
(% i17) (write_data(getXY(points, first),"eqt_ephem_2011_x.dat",comma),
      write_data(getXY(points, second),"eqt_ephem_2011_y.dat",comma))
```

Eine Alternative: 1 Datei mit 2 Zeilen; Filehandle *OUT* wird auf *append* geschaltet!

```
(% i18) (write_data(getXY(points, first), "eqt_ephem_2011.dat", comma),
        OUT : opena("eqt_ephem_2011.dat"),
        write_data(getXY(points, second), OUT, comma))$
```

13.10.3 Import in Geogebra

So die Daten sind auf der Festplatte. Bis jetzt (Jänner 2019) kann bei *Geogebra* nur die Tabellenansicht (View → Spreadsheet) mit dem Kontextmenü Daten im CSV Format einlesen - dabei werden alle bisherigen Daten gelöscht. Man muss sie also mit *Copy and Paste* sichern!

Wir demonstrieren hier die Möglichkeit mit 1 File - nach dem Import (Kontextmenü *Import Data File*), stehen die x-Koordinate in der ersten Zeile, die y-Koordinaten in der zweiten Zeile.

- Wir blenden Befehlszeile und Algebra-Fenster ein
- Wir erzeugen die Punktliste: `points=(A1:Y1,A2:Y2)` oder Zellen markieren, Werkzeug *List of Points* auswählen - schon sollte es passen!
- Laden das Tool *cubic-spline-with-LU-decomp2.ggt* aus dem Kapitel *Splines*
- Menü *Tools* → *Customize Toolbar* → Insert csplineLU → *Apply*
- Klicken auf das Spline-Icon und anschl. auf die Liste *points*
- Oder einfach den `Spline(<Punktliste>)`-Befehl von *Geogebra*

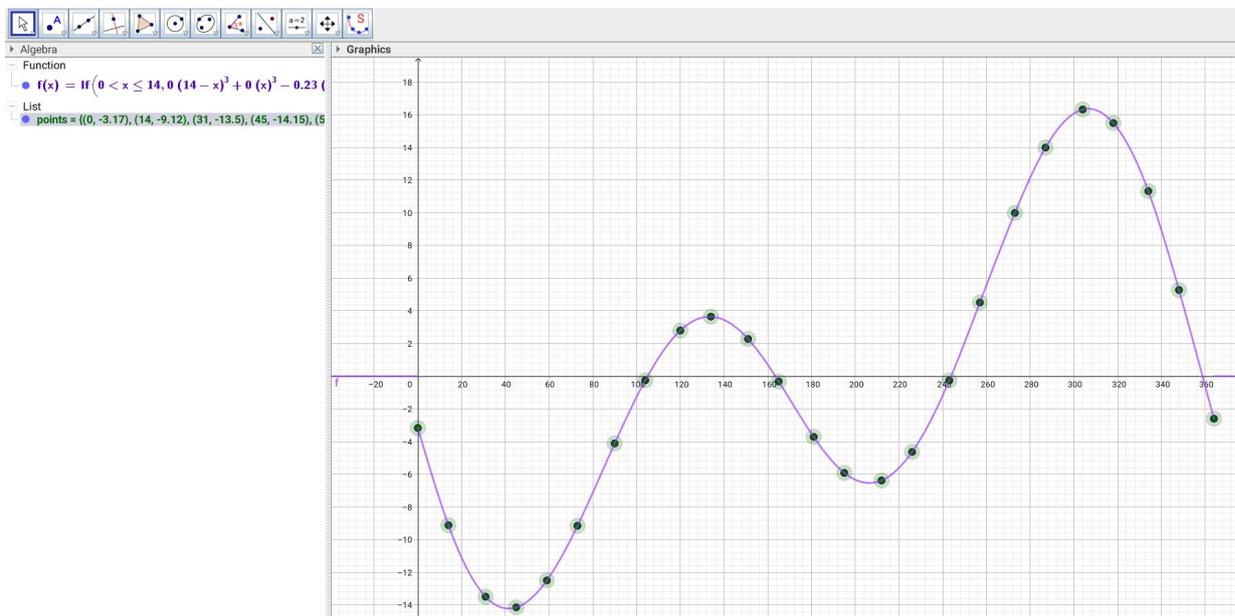


Abb.144 : Daten von wxMaxima nach Geogebra mit anschl. Spline-Funktion

13. Zeitgleichung

Die Methode mit den 2 Files für x- bzw. y-Koordinaten ist etwas aufwendiger, denn vor dem Import des zweiten Files muss man die gerade importierte Zeile markieren (Zeilennummer anklicken und *Copy*), anschl. zweiten File importieren und mit *Paste* Daten des ersten Files zurückholen.

13.11 ANHANG2: Integrale von Sinus-Potenzen

Ausgangspunkt ist die Produktregel der Differentialrechnung und die sich daraus ergebende Regel für das partielle Integral - alle Buchstaben bedeuten Funktionen, außerdem verwenden wir eine "abgekürzte" Nomenklatur:

$$(u \cdot v)' = u' \cdot v + u \cdot v' \quad \Big| \int \Rightarrow \quad u \cdot v = \int (u' \cdot v) + \int (u \cdot v')$$

Theorem 13.1 Regel der partiellen Integration

$$\int (u' \cdot v) = u \cdot v - \int (u \cdot v') \quad (13.27)$$

$$\begin{aligned} \int \sin^n x &= \int \sin x \sin^{n-1} x \stackrel{13.27}{\underset{\downarrow}{=}} -\cos x \sin^{n-1} x - \int (-\cos x)(n-1) \sin^{n-2} x \cos x = \\ &= -\cos x \sin^{n-1} x + (n-1) \int (1 - \sin^2 x) \sin^{n-2} x = \\ &= -\cos x \sin^{n-1} x + (n-1) \left[\int \sin^{n-2} x - \int \sin^n x \right] \end{aligned}$$

Wir fassen die n -te Potenz auf der linken Seite zusammen

$$\begin{aligned} \int \sin^n x + (n-1) \int \sin^n x &= -\cos x \sin^{n-1} x + (n-1) \int \sin^{n-2} x \\ \int \sin^n x [1 + (n-1)] &= -\cos x \sin^{n-1} x + (n-1) \int \sin^{n-2} x \quad | : n \end{aligned}$$

$$\int \sin^n x = -\frac{\cos x \sin^{n-1} x}{n} + \frac{n-1}{n} \int \sin^{n-2} x$$

14 | Erdbahngleichung

Hier die versprochene Beschäftigung mit der “Erdbahngleichung” 13.16:

$$\dot{\psi}(t) = (1 - \kappa^2)^{-\frac{3}{2}} (1 + \kappa \cos \psi)^2 \quad \psi(0) = 0 \quad \kappa \ll 1 \quad (14.1)$$

14.1 Qualitative Überlegungen

Taxonomie: Es handelt sich um eine gewöhnliche Differentialgleichung 1. Ordnung (ODE first order) und zwar um eine autonome, d.h. die rechte Seite ist nicht von t abhängig. Normalerweise gilt:

$$\dot{y} = F(x, y) \quad \text{hier} \quad \dot{y} = F(y)$$

Zeichnen wir den Graph $F(y)$ der Steigungen:

Bei einer autonomen ODE 1. Ordnung kümmert man sich um die “kritischen Punkte”, wo gilt

$$F(y) = 0$$

es ist leicht einzusehen, dass es hier keine solchen gibt!

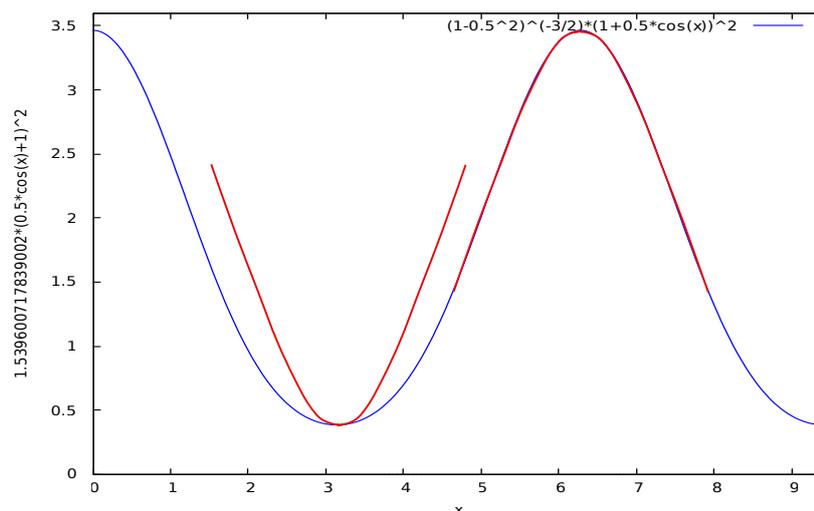


Abb.145 : Rechte Seite der Diffglg. für $\kappa = 0.5$

14. Erdbahngleichung

Wir schauen uns erst einmal die rechte Seite von 14.1 an: Für kleine κ ist der erste Faktor ungefähr 1, der zweite eine periodische "Störung" um 1. Die einfachste Näherung ist also

$$\dot{\psi}(t) \approx 1 \Rightarrow \psi(t) = t$$

- das entspricht einer gleichförmigen Bewegung! Selbst bei $\kappa = 0.5$ wie in Abb. 145 ist dies noch gut zu sehen!

Erzeugt wurde dieser Graph mit folgendem *wxMaxima-Code*;
Die "Nachbehandlung" erfolgte mit *Inkscape*.

```
f(x):=0.75^(-3/2)*(1+0.5*cos(x))^2$  
  
plot2d([f(x)], [x,0,3*pi], [y,0,3.6],  
[legend,"(1-0.5^2)^(-3/2)*(1+0.5*cos(x))^2"])$
```

Wie aus dem Term ersichtlich ist (und auch aus dem Graph) ist die Funktion periodisch mit 2π und symmetrisch um π . Die Steigungen in den "Bögen" sind allerdings verschieden - wie der "rote Graph" deutlich macht - der Bogen links von 2π ist ein anderer als rechts von π (auch wenn in der nächsten Zeichnung dies kaum auffällt!). In π ist außerdem das Minimum (also Wendepunkt von ψ).

Als nächstes schauen wir uns das Richtungsfeld von 14.1 an:

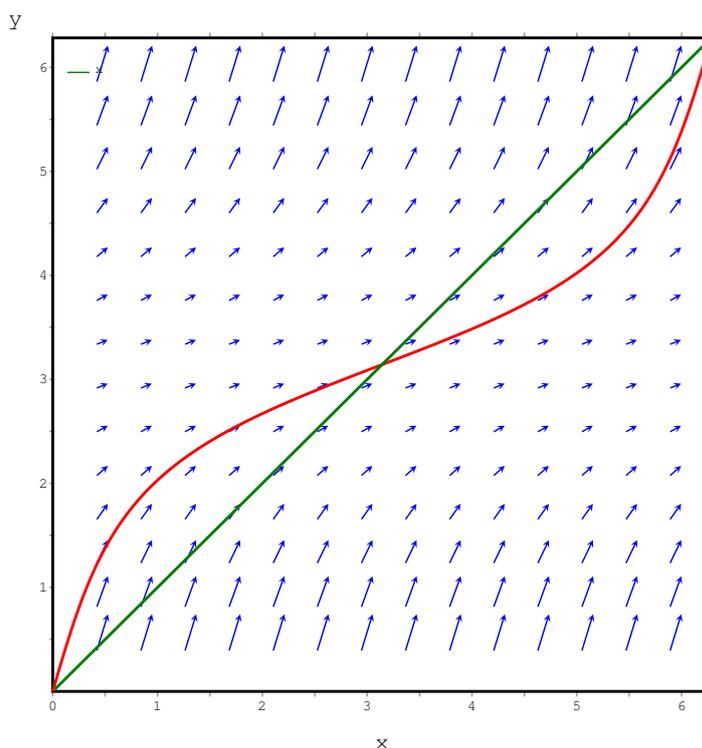


Abb.146 : Richtungsfeld der Diffgl.

Abb. 146 zeigt das Richtungsfeld für $\kappa = 0.5$. Außerdem wurde die spätere Näherung für unser Anfangswertproblem eingezeichnet und zur Orientierung die Funktion für die gleichförmige Bewegung (Gerade).

Erzeugt wurde dieser Graph mit folgendem *wxMaxima-Code* (damit der Befehl `plotdf` (plot direction field) funktioniert, muss das Paket `”xmaxima“` installiert sein!):

```
plotdf(0.75^(-3/2)*(1+0.5*cos(y))^2, [xfun, "x"],
      [trajectory_at,0,0], [y,0,2*%pi], [x,0,2*%pi])$
```

14.2 Numerisch: Runge-Kutta

Wir verschaffen uns einen Überblick über die Lösung mit dem Runge-Kutta-Verfahren

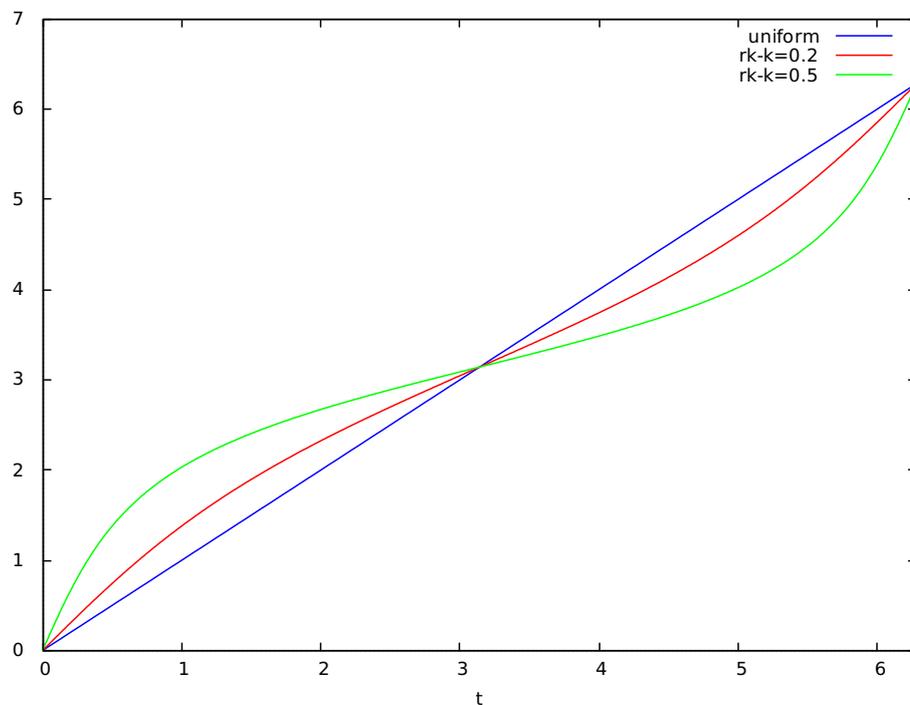


Abb.147 : Runge-Kutta-Verfahren

Auch wenn wir keinen Term für $\psi(t)$ bekommen, so sehen wir doch am Graph einige Eigenschaften:

- Je größer κ , umso größer die Abweichung von der gleichförmigen Kreisbahn
- Im Perihel ($t = 0$ bzw. 2π) und Aphel ($t = \pi$) stimmen die Durchgangspunkte mit der gleichförmigen Bahn (1. Mediane) überein

14. Erdbahngleichung

- $\psi(t)$ ist gegenüber dem Punkt (π, π) zentrisch symmetrisch! (Dies ist leicht an Hand der Eigenschaften von $\dot{\psi}(t)$ einzusehen: Minimum bei π (Wendepunkt für $\psi(t)$), symmetrisch um π)
- Leider sind die “Bögen” selbst nicht symmetrisch - wie man bei $\kappa = 0.5$ erkennen kann

Erzeugt wurde dieser Graph mit folgendem *wxMaxima*-Code:

```
k:0.2$a:(1-k^2)^(-3/2)$
points1:rk(a*(1+k*cos(u))^2,u,0,[t,0,2*%pi,0.01])$
k:0.5$a:(1-k^2)^(-3/2)$
points2:rk(a*(1+k*cos(u))^2,u,0,[t,0,2*%pi,0.01])$
plot2d([t,[discrete,points1],[discrete,points2]],
[t,0,2*%pi],[legend,"uniform","rk-k=0.2","rk-k=0.5"])$
```

rk steht natürlich für das Runge-Kutta-Verfahren, angegeben wird der Term für die Ableitungsfunktion, die abhängige Variable, ihr Anfangswert, dann in einer Liste die unabhängige Variable, Startwert und Endwert und schließlich die Schrittweite des Verfahrens.

14.3 Numerisch: Integrieren mit Taylor-Näherung

Als nächstes verschaffen wir uns eine Näherung für $\psi(t)^{-1}$ indem wir das Integral mit einer Taylorreihenentwicklung nach κ lösen:

$$\int \underbrace{(1 + \kappa \cos \psi)^{-2}}_{f(\kappa) = \sum_{i=0}^n \frac{f^{(i)}(0)}{i!} \kappa^i} d\psi(t) = (1 - \kappa^2)^{-\frac{3}{2}} t$$

Wir führen das in *wxMaxima* durch:

Wir entwickeln den Integranden nach κ bis zum Grad 3 an der Stelle 0

```
(%i1) define(f(x),taylor((1+k*cos(x))^(-2),k,0,3));
```

```
(%o1) f(x) := 1 - 2 cos(x) k + 3 cos(x)^2 k^2 - 4 cos(x)^3 k^3 + ...
```

Wir integrieren

```
(%i2) ratsimp(define(F(x),integrate(f(x),x)));
```

```
(%o2) F(x) := 
$$\frac{9 k^2 \sin(2 x) + 16 k^3 \sin(x)^3 + (-48 k^3 - 24 k) \sin(x) + (18 k^2 + 12) x}{12}$$

```

Setzen jetzt κ und den Koeffizienten

```
(%i3) k:0.2;
```

```
(%o3) 0.2
```

```
(%i4) a:(1-k^2)^(-3/2);
```

```
(%o4) 1.063146589749643
```

Hier die Gleichung, die es für t aufzulösen gilt

```
(%i5) eq1:F(%psi)=a*t, numer;
```

```
(%o5) .06 (0.5 sin(2 psi) + psi) - .032 (sin(psi) - .333 sin(psi)^3) - 0.4 sin(psi) + psi = 1.063146589749643 t
```

```
(%i6) define(U(%psi), F(%psi)/a), numer;
```

```
(%o6) U(psi) := .940604061 (.06 (0.5 sin(2 psi) + psi) - .032 (sin(psi) - .333 sin(psi)^3) - 0.4 sin(psi) + psi)
```

Hier der Term für die Umkehrfunktion $\psi(t)^{-1}$

```
(%i7) expand(U(%psi));
```

```
(%o7) .0282 sin(2 psi) + 0.01 sin(psi)^3 - .4063 sin(psi) + .99704 psi
```

Wir basteln eine Argument und Werteliste der Umkehrfunktion und geben diese Listen im Plot-Befehl weiter unter in umgekehrter Reihenfolge an

```
(%i8) args:makelist(i*0.1, i, 0, 62)$
```

```
(%i9) vals:map(lambda([x], U(x)), args), numer$
```

Hier wieder das Runge-Kutta-Verfahren zum Vergleich

```
(%i10) points1:rk(a*(1+k*cos(u))^2, u, 0, [t, 0, 2*pi, 0.01])$
```

Jetzt schauen wir uns das an:

```
(%i11) plot2d([x, U(x)], [discrete, vals, args], [discrete, points1], [x, 0, %pi],
  [style, lines, [lines, 0, 0], [points, 1, 5, 6], lines],
  [gnuplot_preamble, "set key bottom"],
  [legend, "uniform", "Umkehrfkt", "Taylor-Numerisch", "Runge-Kutta"]);
```

14. Erdbahngleichung

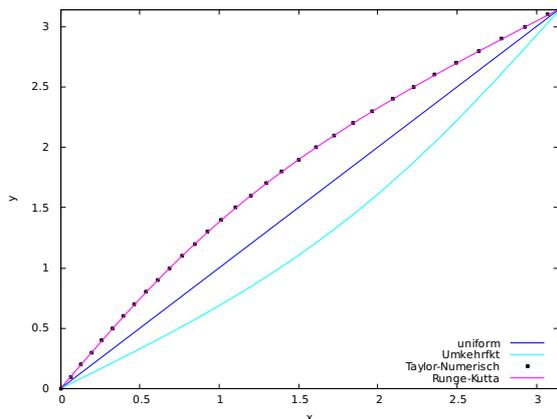


Abb.148 : Taylor Näherung

Selbst für $\kappa = 0.2$ ist unsere Lösung vom Runge-Kutta-Verfahren nicht mehr zu unterscheiden. Dies muss erst recht für kleinere κ gelten. Die Fortsetzung des Graphen können wir über die zentrische Symmetrie bekommen.

Nachteil: Wir haben nur einen Näherungsterm für die Umkehrfunktion statt für $\psi(t)$

14.4 Versuchs doch mal mit Reihen!

Im vorigen Abschnitt haben wir numerisch für jedes ψ den Zeitpunkt t bestimmt. Haben uns eine Tabelle mit Punkten $(\psi_i | t_i)$ aufgestellt und anschl. die Koordinaten vertauscht. Nachteil: Man mußte die Exzentrizität κ kennen.

Ausgangspunkt ist wieder unsere Gleichung

$$\underbrace{(1 - \kappa^2)^{\frac{3}{2}}}_a \int \underbrace{(1 + \kappa \cos \psi)^{-2}}_B d\psi(t) = t \quad \Leftrightarrow \quad f(\psi) = t \quad \Leftrightarrow \quad \psi = f^{-1}(t) = g(t)$$

Unser jetziger Ansatz ist sehr ambitioniert:

- Eine Reihenentwicklung für den Integrand b und dem Faktor a vorm Integral
- Das Integral auf die Summanden angewendet
- Das Cauchy-Produkt der beiden Reihen ist dann eine Reihenentwicklung für f
- Mit dem Lagrange Inversion Theorem $f^{-1}(t) = g(t)$ zumindest lokal bestimmen!

So das Programm liegt vor uns - nun zur Durchführung (Einzelheiten in den Anhängen!)

1.

$$a = (1 - \kappa^2)^{\frac{3}{2}} = 1 - \frac{3}{2}\kappa^2 + \frac{3}{8}\kappa^4 + 3 \sum_{n \geq 3} \binom{2n-4}{n-2} \frac{\kappa^{2n}}{2^{2n-2}(n-1)n} \quad \text{laut 14.12}$$

$$B = (1 + \kappa \cos \psi)^{-2} = \sum_{n \geq 0} (n+1)(-\kappa \cos \psi)^n \quad \text{laut 14.13}$$

$$b = \sum_{n \geq 0} (n+1)(-\kappa)^n \int (\cos \psi)^n d\psi(t)$$

Für die Integrale der Cosinus-Potenzen gibt es eine Rekursionformel!

2. Jetzt das Cauchy-Produkt der beiden Reihen:

$$c_n := \sum_{k=0}^n a_k \cdot b_{n-k} \quad \Rightarrow \quad f(\psi, \kappa) = \sum_{n=0}^{\infty} c_n, \quad f_N(\psi, \kappa) = \sum_{n=0}^N c_n, \quad f_N \approx f$$

Die letzte Behauptung müsste man genauer untersuchen.

Wir überprüfen, ob eine Konvergenz vorliegt, einfach experimentell mit *wxMaxima* - wir unterteilen 2π in 20 Teile, ermitteln an den Grenzen die Werte, davon das Maximum des Absolutwerts und schauen, ob die Werte mit wachsendem n kleiner werden - dabei haben wir $\kappa = 0.8$ angenommen:

```
(% i1) a[n]:=3*binomial(2*n-4,n-2) / ( (n-1)*n*2^(2*n-2) ) *k^(2*n)$
```

```
(% i2) (a[0]:1,a[1]:-3/2*k^2,a[2]:3/8*k^4)$
```

```
(% i3) b[n](x):=xthru((n+1)*(-k)^n * integrate( (cos(x))^n,x))$
```

```
(% i4) c[m](x):=sum(a[j]*b[m-j](x),j,0,m) $
```

```
(% i5) k:0.8$
```

Maximum der ersten 30 Reihenglieder - sieht ganz gut aus!

```
(% i6) pointList:makelist([j,lmax(makelist(float(abs(c[j](i*2*%pi/20))),i,0,20))] , j ,1,30)$
```

```
(% i7) plot2d([discrete,pointList],[x,0,31])$
```

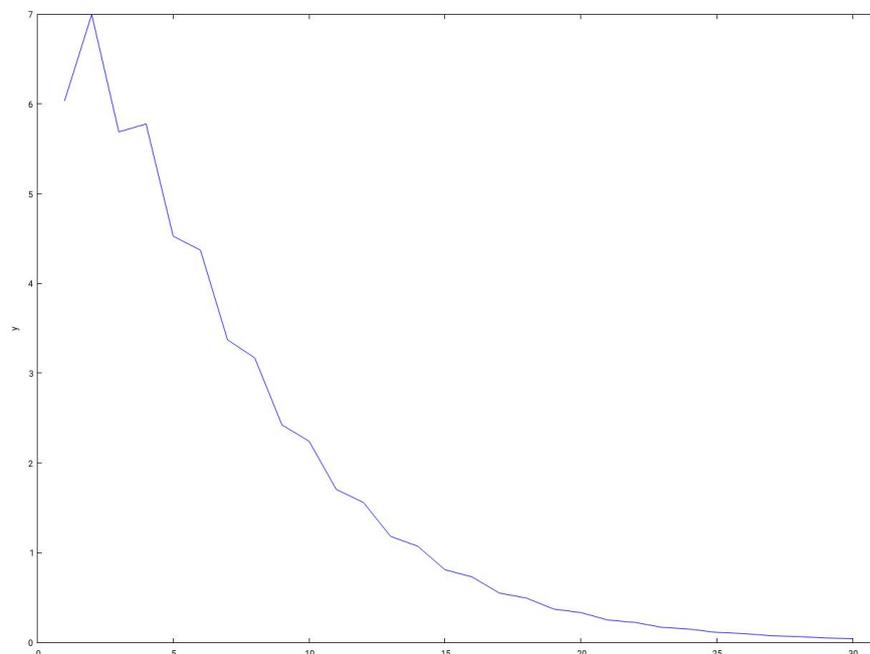


Abb.149 : Konvergenz-Check für $\kappa = 0.8$

14. Erdbahngleichung

3. Die Lagrange Inversion (siehe Wikipedia):

$$\psi(t) = \psi_0 + \sum_{n \geq 1} \frac{(t - f(\psi_0))^n}{n!} \underbrace{\lim_{\psi \rightarrow \psi_0} \left[\frac{d^{n-1}}{d\psi^{n-1}} \left(\frac{\psi - \psi_0}{f(\psi) - f(\psi_0)} \right)^n \right]}_{g_n} \quad (14.2)$$

Da in erster Näherung $\psi(t) \approx t$ gilt, nehmen wir für den Startwert $\psi_0 = t$. Außerdem ersetzen wir f durch f_N - damit ergibt sich

$$\psi(t) \approx t + \sum_{n \geq 1} \frac{(t - f_N(t))^n}{n!} \underbrace{\lim_{\psi \rightarrow t} \left[\frac{d^{n-1}}{d\psi^{n-1}} \left(\frac{\psi - t}{f_N(\psi) - f_N(t)} \right)^n \right]}_{g_n} \quad (14.3)$$

Am schwierigsten sind die Faktoren g_n zu bestimmen - die Ableitungen werden bei größerem n bereits sehr komplizierte Bruchterme auf die wiederholt die Regel von L'Hospital anzuwenden sind, um den Grenzwert zu erhalten!

Hier das "Code-Snippet" - das die Arbeit leistet

```

1  getG(L, nr):=block([ZOld,NOld,ZNew, NNew, Zlim,Nlim:0, G],
3  G: xthru(ev(L[nr],diff)),
   ZOld:part(G,1), NOld:part(G,2),
5  Zlim: subst(0, f(x)-f(x_0), subst(0,x-x_0,ZOld)),
   Nlim: subst(0, f(x)-f(x_0), subst(0,x-x_0,NOld)),
7  for i:1 while Nlim=0 do (
   NNew:diff(NOld,x), ZNew:diff(ZOld,x),
9  NOld:NNew, ZOld:ZNew,
   Zlim: subst(0, f(x)-f(x_0), subst(0,x-x_0,ZOld)),
11  Nlim: subst(0, f(x)-f(x_0), subst(0,x-x_0,NOld))
   ),
13  define(G[nr](x_0),at(ratsimp(Zlim/Nlim),x=x_0))
15  )

```

- Zeile 2 Übergabe-Parameter: $L \rightarrow$ Liste der Ableitungen,
 $nr \rightarrow$ Nummer des zu bearbeitenden Listenelements
 lokale Vars: $ZOld \rightarrow$ alter Zähler, $NNew \rightarrow$ neuer Nenner
- Zeile 3 wir evaluieren die "Diff's" und bringen auf gemeinsamen Nenner (*xthru*)
- Zeile 4 wir holen uns Zähler bzw. Nenner für L'Hospital
- Zeile 5/6 wir ersetzen $x - x_0$ und $f(x) - f(x_0)$ durch 0 (Grenzwert ausführen)
- Zeile 7 solange der Nenner verschwindet, werden Zähler und Nenner abgeleitet
- Zeile 13 wir legen eine Array-Funktion $G[nr](x_0)$ fest, die unserem g_n entspricht -
 im Bruch wird x durch x_0 ersetzt
 Vorsicht! Hier handelt es sich um einen Seiteneffekt!

Schauen wir uns das *wxMaxima*- Programm dazu an!

Bis g_4 , Keine Umwandlungswarnungen, 4 Stellen der Dezimaldarstellung und Mitteilung $\kappa > 0$

(% i1) (max_order:4, ratprint:false, fpprintprec:4, assume(k>0))\$

Definition der g_n -Faktoren in der Lagrange-Inversions-Reihe

(% i2) g[n]:=diff((g(x)) ^n, x, n-1) ; $g_n := \frac{d^{n-1}}{dx^{n-1}} g(x)^n$

Liste der g_i mit $i \in \{1, 2, 3, 4\}$

(% i3) gDerivatives:makelist(g[i],i,1,max_order)\$

Wir schauen uns einfach mal g_3 an:

(% i4) gDerivatives[3]; $3g(x)^2 \left(\frac{d^2}{dx^2} g(x) \right) + 6g(x) \left(\frac{d}{dx} g(x) \right)^2$

Wir setzen für $g(x)$ ein:

(% i5) g(x):=(x-x_0)/(f(x)-f(x_0)); $g(x) := \frac{x - x_0}{f(x) - f(x_0)}$

Ersetzen von $g(x)$ und auf gemeinsamen Nenner bringen

(% i6) gList:xthru(ev(gDerivatives))\$

g_3 besichtigen:

(% i7) gList[3];
$$\frac{3(x - x_0)^2 \left(\frac{d^2}{dx^2} \frac{x - x_0}{f(x) - f(x_0)} \right) + 6(x - x_0) (f(x) - f(x_0)) \left(\frac{d}{dx} \frac{x - x_0}{f(x) - f(x_0)} \right)^2}{(f(x) - f(x_0))^2}$$

Erklärung im Text oben!

```
(% i8) getG(L, nr):=block([ZOld,NOld,ZNew, NNew, Zlim,Nlim:0, G],
  G:xthru(ev(L[nr],diff)), ZOld:part(G,1), NOld:part(G,2),
  Zlim: subst(0, f(x)-f(x_0), subst(0,x-x_0,ZOld)),
  Nlim:subst(0,f(x)-f(x_0), subst(0,x-x_0,NOld)),
  for i:1 while Nlim=0 do (
    NNew:diff(NOld,x), ZNew:diff(ZOld,x),
    NOld:NNew, ZOld:ZNew,
    Zlim: subst(0, f(x)-f(x_0), subst(0,x-x_0,ZOld)),
    Nlim:subst(0,f(x)-f(x_0), subst(0,x-x_0,NOld))
  ),
  define(G[nr](x_0),at(ratsimp(Zlim/Nlim),x=x_0))
)$
```

$G[n](x_0)$ über Seiteneffekt festlegen, Liste wird nicht benötigt!

(% i9) makelist(getG(gList,i),i,1,max_order) \$

Reihe des Faktors wird festgelegt, die ersten 3 müssen "extra" festgelegt werden!

14. Erdbahngleichung

(% i10) $a[n]:=3*\text{binomial}(2*n-4,n-2)/((n-1)*n*2^{(2*n-2)}*k^{(2*n)}; a_n := \frac{3 \binom{2n-4}{n-2}}{(n-1)n2^{2n-2}} k^{2n}$

(% i11) $(a[0]:1,a[1]:-3/2*k^2,a[2]:3/8*k^4)\$$

Reihe des Integranden

(% i12) $b[n](x):=\text{xthru}((n+1)*(-k)^n * \text{integrate}(\cos(x)^n,x));$

$$b_n(x) := \# \{ \text{Lispfunction} \} \left((n+1) (-k)^n \int \cos(x)^n dx \right)$$

Cauchy-Produkt Summanden

(% i13) $c[m](x):=\text{sum}(a[j]*b[m-j](x),j,0,m) ; \quad c_m(x) := \sum_{j=0}^m a_j b_{m-j}(x)$

$f_N(x)$ wird festgelegt, wobei N willkürlich auf 2 *max_order* gesetzt wurde

(% i14) $f(x):=\text{sum}(c[j](x),i,0,2*max_order)\$$

Wir schauen uns g_2 bei $x_0 = \pi/2$ an: die Faktoren bei den k -Potenzen sind groß - Konvergenz?

(% i15) $\text{ratsimp}(\text{ev}(G[2](\%pi/2),\text{nouns})); - (309237645312k^{15} + 481036337152k^{13} + \dots$

Festlegung unserer ψ -Funktion in Abhängigkeit vom Entwicklungspunkt und κ

(% i16) $\text{psi}(t,x_0,k):= x_0 + \text{sum}((t-f(x_0))^n/n! * (\text{ev}(G[n](x_0),\text{nouns})), n,1,max_order);$

$$(t, x_0, k) := x_0 + \sum_{n=1}^{max_order} \frac{(t - f(x_0))^n}{n!} \text{ev}(G_n(x_0), nouns)$$

Funktion für Runge-Kutta wird festgelegt

(% i18) $h(k,u):=(1-k^2)^{-3/2}*(1+k*\cos(u))^2; \quad h(k,u) := (1 - k^2)^{-\frac{3}{2}} (1 + k \cos(u))^2$

Punktlisten für Runge-Kutta werden erstellt

(% i19) $(\text{pointsRK35:rk}(h(0.35,u),u,0,[t,0,2*\%pi,0.01]), \text{pointsRK45:rk}(h(0.45,u),u,0,[t,0,2*\%pi,0.01]))\$$

Plot-Programme für $\kappa = 0.35$ bzw. $\kappa = 0.45$ für qt-Fenster 0 bzw. 1

(% i21) $\text{plot35}() := \text{plot2d}([\text{psi}(x,x,0.35),x, [\text{discrete}, \text{pointsRK35}], [x,0,2*\%pi],$
 $[\text{style}, [\text{lines},5,4,1], [\text{lines},1,5,2], [\text{lines},2,3,2]],$
 $[\text{legend}, " \text{Reihe für } k=0.35", " \text{gleichförmig}", " \text{RUNGE-KUTTA}],$
 $[\text{gnuplot_preamble}, "set key bottom right"], [\text{gnuplot_term}, "qt 0"]$
 $)\$$

```
(% i22) plot45():=plot2d([psi(x,x,0.45),x,[discrete, pointsRK45]], [x,0,2*%pi],
[style,[lines,5,4,1], [lines,1,5,2], [lines,2,3,2] ],
[legend, " Reihe für k=0.45"," gleichförmig", " RUNGE-KUTTA"],
[gnuplot\_preamble, "set key bottom right"], [gnuplot\_term, "qt 1"]
)$
```

Sequentielle Ausführung - dadurch wird nach dem ersten Plot Ausführung nicht unterbrochen!

```
(% i23) (plot35(),plot45())$
```

Wir sehen, dass bei $\kappa = 0.45$ - wenn x_0 von der Identität stark abweicht - die Reihe "davonläuft"!

Außerdem wenn man `max_order` über 6 hinaus vergrößert, werden die Terme so rechenintensiv, dass *wxMaxima* sehr lange braucht, um die Graphen zu produzieren. Hier die Graphen für `max_order=6`

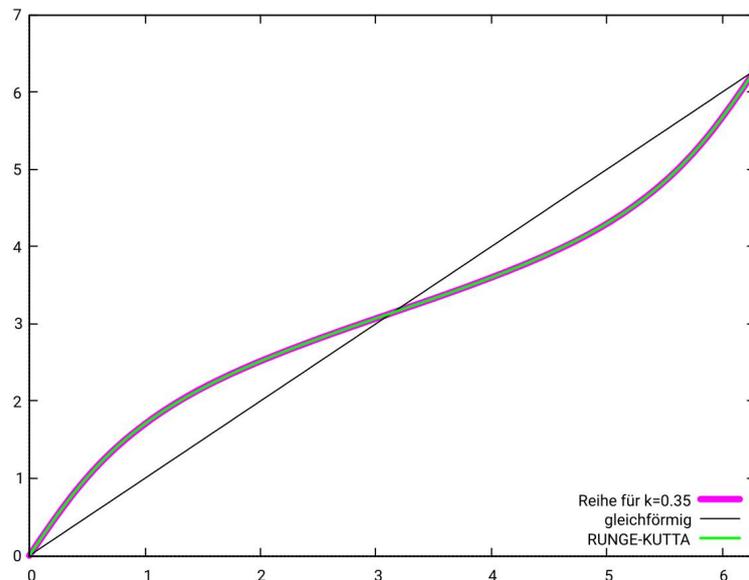


Abb.150 : Reihen-Näherung vs. Runge-Kutta für $\kappa = 0.35$ - noch ist alles OK

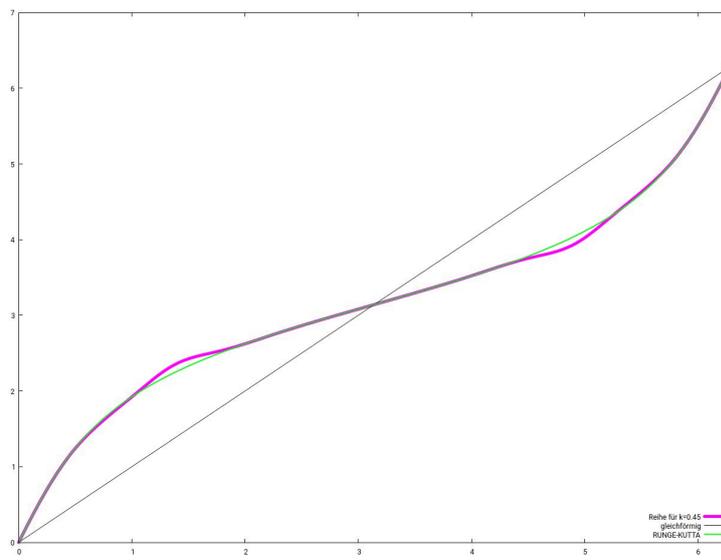


Abb.151 : Reihen-Näherung vs. Runge-Kutta für $\kappa = 0.45$ - die Abweichungen beginnen!

14.5 Alternative Reihenformel nach Stenlund

Das Problem mit Formel 14.3 sind nicht die Ableitungen der Potenzen, sondern dass sie anschl. durch Brüche ersetzt werden.

Man kann das im *wxMaxima*-Programm bei (% i4) bis (% i7) deutlich sehen, wie die Sache immer komplexer wird. Brüche mehrere Male hintereinander ableiten ergibt halt komplizierte Terme - überhaupt wenn die Funktion f noch eine Reihe beachtlicher Länge ist! Im obigen Programm war bei der 4. Ableitung das Leistungslimit von *wxMaxima* bzw. meines Computers erreicht.

Auf der Suche die Komplexität der Ableitungen in der Lagrange-Bürmann Formel zu umgehen, bin ich auf ein Paper von Henrik Stenlund (2007) gestoßen, der eine Rekursionsformel für das Inversionsproblem in diesem Artikel erarbeitet, die dieses Komplexitätsproblem in weit geringerem Aumaß hat.

Eine verkürzte Version möchte ich hier vorstellen:

Unser Ausgangspunkt: $f(\psi) = t \quad f(\psi_0) = t_0$ wobei wir wieder verwenden $f_N \approx f$

angenommen es gäbe eine analytische Funktion g mit

$$g(t) = \psi \quad g(t_0) = \psi_0 \quad \text{also } g = f^{-1}$$

wobei wir die Konvergenzbetrachtung in die experimentelle Mathematik von *wxMaxima* "auslagern".

Wir können dann g bei t_0 nach Taylor entwickeln

$$\begin{aligned} \psi = g(t) &= \sum_{n \geq 0} \frac{(t - t_0)^n}{n!} \frac{d^n}{dt^n} g(t) \Big|_{t_0} = g(t_0) + \sum_{n \geq 1} \frac{(t - t_0)^n}{n!} \frac{d^n}{dt^n} g(t) \Big|_{t_0} = \\ &= \psi_0 + \sum_{n \geq 1} \frac{(t - f(\psi_0))^n}{n!} \frac{d^n}{dt^n} g(t) \Big|_{f(\psi_0)} \Rightarrow \\ \text{unsere spezielle Situation: } \psi(t) &\approx t + \sum_{n=1}^M \frac{(t - f_N(t))^n}{n!} \frac{d^n}{dx^n} g(x) \Big|_t \end{aligned} \quad (14.4)$$

Da g bzw. ψ für kleine κ die identische Funktion annähert wird als Entwicklungspunkt ψ_0 das Argument t der Funktion genommen. Ebenfalls ersetzt das Argument t auch $f(\psi_0)$. Um Verwechslungen zu vermeiden wurde die "dummy"-Variable beim Ableiten von g durch x ersetzt. Je größer M bzw. N umso besser die Näherung - bei soviel Näherungen schauen wir dann, ob das "Zeugs" überhaupt noch konvergiert.

Der "Trick" besteht jetzt darin, die Ableitungen von g durch die Ableitungen von f auszu-drücken mit der Kettenregel:

$$1 = \frac{d}{d\psi} \psi = \frac{d}{d\psi} g(t) = \frac{d}{dt} g(t) \cdot \frac{d}{d\psi} t = \frac{d}{dt} g(t) \cdot \frac{d}{d\psi} f(\psi) \Rightarrow (*) \frac{d}{dt} g(t) = \frac{1}{\frac{d}{d\psi} f(\psi)} \Big| \frac{d}{dt}$$

die weitere Ableitung ergibt mit der Kettenregel(chain rule CR)

$$\frac{d^2}{dt^2} g(t) = \frac{d}{dt} \frac{1}{\frac{d}{d\psi} f(\psi)} \stackrel{CR}{=} \frac{d}{d\psi} \frac{1}{\frac{d}{d\psi} f(\psi)} \cdot \frac{d}{dt} g(t) \stackrel{(*)}{=} \frac{d}{d\psi} \left[\frac{1}{\frac{d}{d\psi} f(\psi)} \right] \frac{1}{\frac{d}{d\psi} f(\psi)}$$

weiteres Ableiten nach t ergibt

$$\frac{d^3}{dt^3} g(t) = \frac{1}{\frac{d}{d\psi} f(\psi)} \frac{d}{d\psi} \left[\frac{1}{\frac{d}{d\psi} f(\psi)} \frac{d}{d\psi} \left[\frac{1}{\frac{d}{d\psi} f(\psi)} \right] \right]$$

Mit dem Differentialoperator $D_\psi(f) := f \cdot \frac{d}{d\psi} f(\psi)$ ergibt sich

$$\frac{d^3}{dt^3} g(t) = D_\psi^2 \left(\frac{1}{\frac{d}{d\psi} f(\psi)} \right)$$

Damit wird 14.4 zur Formel, die nur mehr von $f \approx f_N$ abhängt:

$$\psi(t) \approx t + \sum_{n=1}^M \frac{(t - f_N(t))^n}{n!} D_\psi^{n-1} \left(\frac{1}{\frac{d}{d\psi} f_N(\psi)} \right) \Big|_t \quad (14.5)$$

14. Erdbahngleichung

(% i1) L:[ratsimp(1/diff(f(z),z))]; $\left[\frac{1}{\frac{d}{dz}f(z)}\right]$

(% i2) fold(L1,n):= if n>0 then
 fold(endcons(ratsimp(first(L1) *diff(last(L1),z)), L1), n-1)
 else L1\$

(% i3) Lz:fold(L,7)\$

Blick auf's dritte Listenelement

(% i4) Lz[3];
$$-\frac{\left(\frac{d}{dz}f(z)\right)\left(\frac{d^3}{dz^3}f(z)\right)-3\left(\frac{d^2}{dz^2}f(z)\right)^2}{\left(\frac{d}{dz}f(z)\right)^5}$$

Alle Elemente werden bei t ausgewertet!

(% i5) Lt:map(lambda([x],at(x,z=t)),Lz) \$

Wie im letzten Programm wird f_N als Cauchy-Summe ermittelt

(% i6) a[n]:=3*binomial(2*n-4,n-2)/((n-1)*n*2^(2*n-2))*k^(2*n); $a_n := \frac{3 \binom{2n-4}{n-2}}{(n-1)n2^{2n-2}} k^{2n}$

(% i7) (a[0]:1,a[1]:-3/2*k^2,a[2]:3/8*k^4)\$

(% i8) b[n](x):=xthru((n+1)*(-k)^n * integrate((cos(m))^(n,x))\$

(% i9) c[m](x):=sum(a[j]*b[m-j](x),j,0,m) ; $c_m(x) := \sum_{j=0}^m a_j b_{m-j}(x)$

So f_N mit $N = 10$ steht jetzt fest

(% i10) f(z):=sum(c[i](z),i,0,10); $f(z) := \sum_{i=0}^{10} c_i(z)$

Die höchste Potenz von k

(% i11) hipow(rat(f(z),k),k); 20

(% i12) showtime:true\$

Wir evaluieren jetzt Lt mit unserer Funktion f_{10} - dauert eine Weile

(% i13) Lt1:ev(Lt,nouns)\$ Evaluation took 29.8249 seconds using 1014.312 MB.

Legen jetzt die Formel 14.5 fest

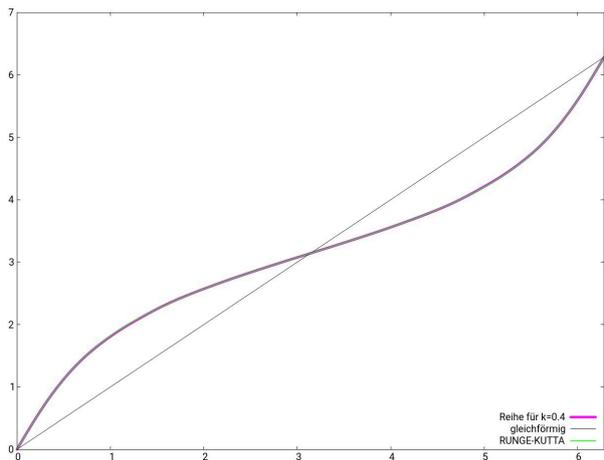
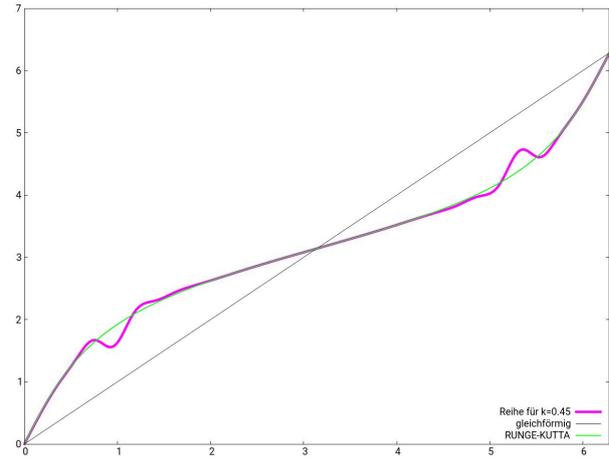
(% i14) define(psi(t,k),t+sum((t-f(t))^n/n!*Lt1[n],n,1,8))\$

Kontrolle mit RUNGE-KUTTA

(% i15) h(k,u):=(1-k^2)^(-3/2)*(1+k*cos(u))^2;

(% i16) pointsRK45:rk(h(0.4,u),u,0,[t,0,2*%pi,0.01])\$

```
(% i17) plot2d([psi(x,0.4),x,[discrete, pointsRK45]], [x,0,2*pi],
[style,[lines,5,4,1], [lines,1,5,2], [lines,2,3,2] ],
[legend, " Reihe für k=0.4", " gleichförmig", " RUNGE-KUTTA"],
[gnuplot_preamble, "set key bottom right; set xtics font \", 15\";
set ytics font \", 15\"; set key font \", 15\" "],
[gnuplot_term, "qt 0"]
)$
```

Abb.152 : $\kappa = 0.4$ - die Abweichungen kaum sichtbar!Abb.153 : $\kappa = 0.45$ - Konvergenzverlust!

14.6 Näherungsterm für $\psi(t)$

Die obige Reihenentwicklung der Umkehrfunktion bringt uns auf die Idee, ob wir $\psi(t)$ - das ja nur eine leichte Störung $\delta(t)$ der identischen Funktion ist - direkt als Reihe entwickeln können:

$$\psi(t) \approx t + \underbrace{\kappa f_1(t) + \kappa^2 f_2(t)}_{\delta(t)}$$

da wir später $\cos \psi$ benötigen und dies mit κ multipliziert wird, berücksichtigen wir bei der Näherung nur die linearen Terme in κ :

$$\cos \psi = \cos(t + \delta) = \cos t \underbrace{\cos \delta}_{\approx 1} - \sin t \underbrace{\sin \delta}_{\approx \delta} \approx \cos t - \sin t (\kappa f_1(t)) \Rightarrow$$

$$1 + \kappa \cos \psi \approx 1 + \kappa \cos t - \kappa^2 f_1(t) \sin t \Rightarrow$$

$$(1 + \kappa \cos \psi)^2 \approx 1 + 2\kappa \cos t - 2\kappa^2 f_1(t) \sin t + \kappa^2 \cos^2 t$$

Jetzt vereinfachen wir noch den Koeffizienten $(1 - \kappa^2)^{-\frac{3}{2}}$: nach Taylor gilt

$$(1 - x)^{-\frac{3}{2}} \approx 1 + \frac{3}{2}x \Rightarrow (1 - \kappa^2)^{-\frac{3}{2}} \approx 1 + \frac{3}{2}\kappa^2$$

14. Erdbahngleichung

damit ergibt sich

$$(1 - \kappa^2)^{-\frac{3}{2}} (1 + \kappa \cos \psi)^2 \approx 1 + 2\kappa \cos t - 2\kappa^2 f_1(t) \sin t + \kappa^2 \cos^2 t + \frac{3}{2}\kappa^2$$

Jetzt nehmen wir die Differentialgleichung als Bedingung

$$\begin{aligned} 0 &= \dot{\psi}(t) - (1 - \kappa^2)^{-\frac{3}{2}} (1 + \kappa \cos \psi)^2 = \\ &= 1 + \kappa \dot{f}_1(t) + \kappa^2 \dot{f}_2(t) - \left(1 + 2\kappa \cos t - 2\kappa^2 f_1(t) \sin t + \kappa^2 \cos^2 t + \frac{3}{2}\kappa^2 \right) = \\ &= \boxed{\kappa(\dot{f}_1(t) - 2 \cos t) + \kappa^2 \left(\dot{f}_2(t) + 2f_1(t) \sin t - \cos^2 t - \frac{3}{2} \right) = 0} \end{aligned}$$

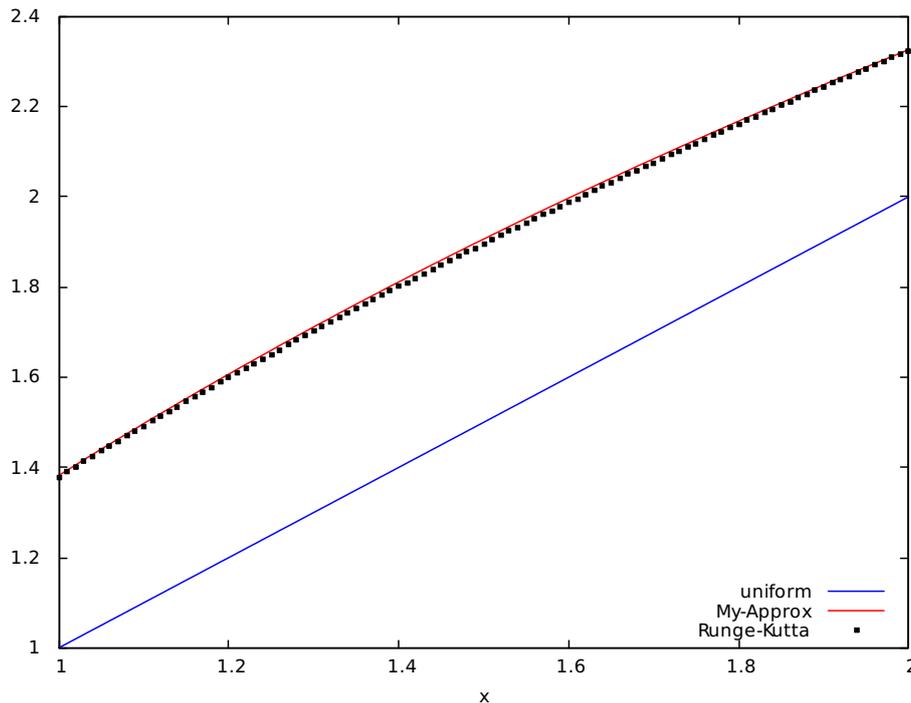
Da linke und rechte Seite für alle Zeiten verschwinden müssen, haben wir Bedingungen für f_1 und f_2 gefunden - das sind zwar wieder Differentialgleichungen - die sind aber trivial:

$$\begin{aligned} (1) \quad \dot{f}_1(t) &= 2 \cos t \Rightarrow f_1(t) = 2 \sin t \\ (2) \quad \dot{f}_2(t) &= \underbrace{\cos^2 t}_{1 - \sin^2 t} - 4 \sin^2 t + \frac{3}{2} = \frac{5}{2} \left(-2 \sin^2 t + \underbrace{1}_{\sin^2 t + \cos^2 t} \right) = \frac{5}{2} (\cos 2t) \Rightarrow \\ &\Rightarrow f_2(t) = \frac{5}{4} \sin 2t \end{aligned}$$

Also halten wir fest

$$\boxed{\psi(t) \approx t + \underbrace{\kappa 2 \sin t + \kappa^2 \frac{5}{4} \sin 2t}_{\delta(t)}}$$

Wie gut ist nun unsere Näherung?

Abb.154 : Term Näherung für $\Psi(t)$

Wir sehen selbst für $\kappa = 0.2$ ist unser Näherungsterm vom Runge-Kutta-Verfahren kaum mehr zu unterscheiden, obwohl der Graph bei der stärksten Abweichung (ca. $t = 1.5$) gezeichnet wurde. Bei "unserem" $\kappa = 0.016$ wäre ein Unterschied zum Runge-Kutta-Graphen nicht mehr erkennbar!

Hier noch der *wxMaxima*-Code:

14.7 Alternative Herleitung von $\psi(t)$

1. Wir vereinfachen den Koeffizienten $(1 - \kappa^2)^{-\frac{3}{2}}$ nach Taylor:

$$(1 - x)^{-\frac{3}{2}} \approx 1 + \frac{3}{2}x \Rightarrow (1 - \kappa^2)^{-\frac{3}{2}} \approx 1 + \frac{3}{2}\kappa^2$$

2. Es gilt die Identität (wenn Sie wollen die Fourierreihenentwicklung)

$$2 \cos^2 x = \cos 2x + 1$$

3. Damit ergibt sich mit 1) und 2)

$$\begin{aligned} (1 - \kappa^2)^{-\frac{3}{2}} (1 + \kappa \cos \psi)^2 &\approx \left(1 + \frac{3}{2}\kappa^2\right) (1 + 2\kappa \cos \psi + \kappa^2 \cos^2 \psi) = \\ &= \left(1 + \frac{3}{2}\kappa^2\right) \left(1 + 2\kappa \cos \psi + \kappa^2 \frac{1}{2} (\cos 2\psi + 1)\right) \end{aligned}$$

14. Erdbahngleichung

4. Ausmultiplizieren und höhere Terme als κ^2 vernachlässigen und $\psi \approx t$:

$$\dot{\psi}(t) \approx 1 + 2\kappa \cos t + 2\kappa^2 + \kappa^2 \frac{1}{2} \cos 2t$$

5. Jetzt gilt aber $\psi(\pi) = \pi$ und $\psi(2\pi) = 2\pi$, d. h. aber, dass die Zusatzterme neben 1 dort verschwinden (bzw. sich aufheben) müssen, wenn wir $2\kappa^2$ einfach weglassen ist das zwar der Fall, aber wir lassen den "größeren Teil" der κ^2 -Glieder weg, da ist es besser wir schlagen ihn zum Cosinusterm dazu

$$\dot{\psi}(t) \approx 1 + 2\kappa \cos t + \frac{5}{2}\kappa^2 \cos 2t$$

das ist zwar etwas ungenauer, aber damit sind unserer Bedingungen erfüllt - die Integration liefert dann unsere Näherung!

14.8 Historische Variante von Kepler

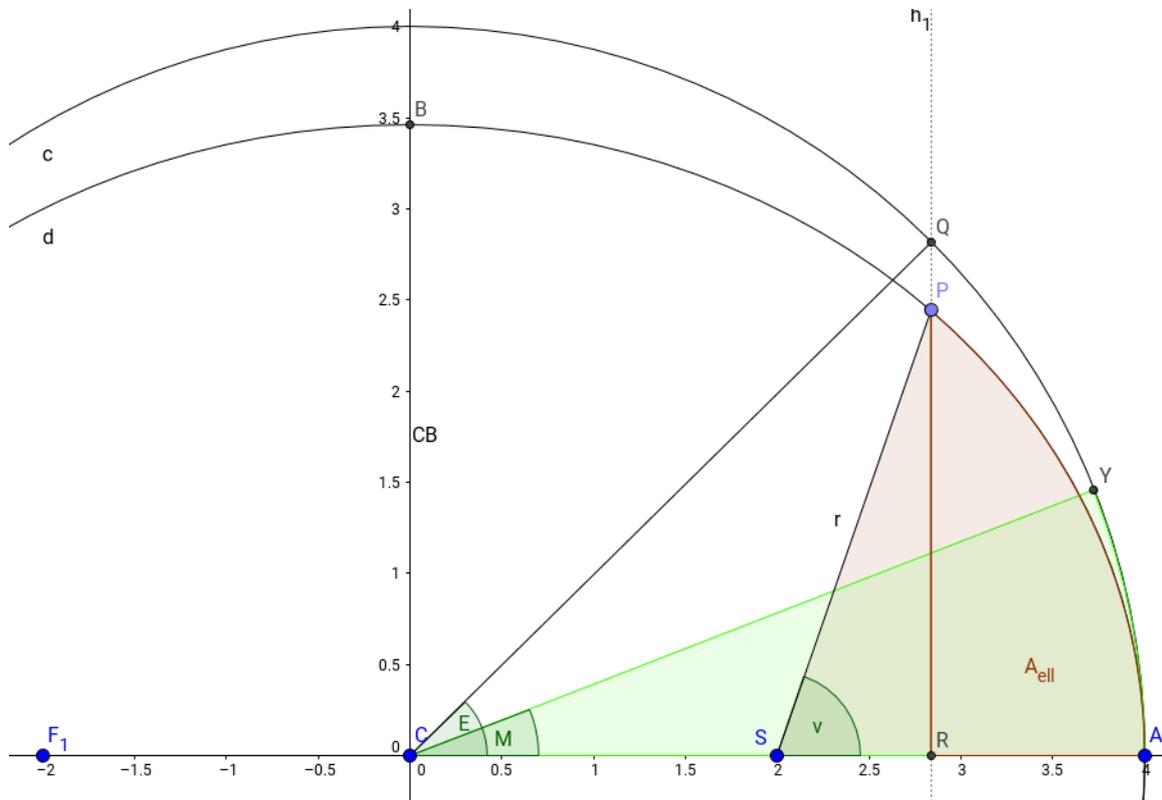


Abb.155 : Keplers Nomenklatur

- d Umlaufellipse (des Planeten)
- S Brennpunkt (Sonne)
- c Umkreis
- C Mittelpunkt (Ellipse/Umkreis)
- CA große Halbachse a
- CB kleine Halbachse b
- v reale Anomalie (bei uns ψ)
- P umlaufender Planet
- E exzentrische Anomalie
- M mittlere Anomalie

M ist ein mit der Zeit linear zunehmender Winkel (Y bewegt sich gleichförmig) - wie wir zeigen werden. Kepler gelang folgende Berechnungskette:

$$M(t) = \frac{2\pi}{T}t \rightarrow E(t) = f(M(t)) \rightarrow v(t) = g(E(t))$$

Obwohl es keine elementare Darstellung $v(t) = \psi(t)$ gibt, ist das Problem über diese Zwischenschritte lösbar. Meine Hochachtung vor Kepler. Erst später gelang Lagrange bzw. Bessler eine Darstellung von $v(t)$, die allerdings alles andere als elementar ist.

14. Erdbahngleichung

Argumentationskette

- Kreissektor CYA (Fläche A_{CYA}) hat pro Zeiteinheit immer den gleichen Flächenzuwachs (2. Kepler) - daher bewegt sich Y gleichförmig $M = \frac{2\pi}{T}t$ (T ist Umlaufdauer)
- $\overline{PR} = \frac{b}{a}\overline{QR}$ - die Ellipse ist eine affine Abbildung des Kreises!
- Laut Kepler ist der Flächenzuwachs von A_{PSA} pro Zeit konstant, formal

$$A_{PSA}(t) = k \cdot t \quad \text{und} \quad A_{PSA}(T) = ab\pi \Rightarrow A_{PSA}(t) = \frac{2\pi}{T} \frac{ab}{2} = M \frac{ab}{2}$$

$$\Rightarrow \boxed{M = \frac{2}{ab} A_{PSA}} \quad (14.6)$$

- Wir zerlegen jetzt A_{PSA}

$$A_{PSA} = A_{PRA} + A_{PSR} = \frac{b}{a} \underbrace{A_{QRA}}_{\text{Sektor-Dreieck}} + A_{PSR} =$$

$$= \frac{b}{a} \left[\frac{a^2}{2} E - \frac{1}{2} (e + r \cos v) \overline{QR} \right] + \frac{1}{2} (r \cos v) \frac{b}{a} \overline{QR} \quad \underbrace{=}_{\overline{QR} = a \sin E}$$

$$= \frac{ab}{2} \left(E - \frac{e}{a} \sin E \right) = \boxed{\frac{ab}{2} (E - \varepsilon \sin E) = A_{PSA}} \quad (14.7)$$

- Wir setzen 14.7 in 14.6 ein und erhalten die "Kepler-Gleichung":

$$\boxed{M = E - \varepsilon \sin E} \quad (14.8)$$

Für Kepler war hier Schluss - nur numerisch gelang eine Lösung von 14.8. Um die Argumentationskette nicht zu durchbrechen, setzen wir voraus $E(M(t))$ sei bekannt.

- Da bei der Erde $T = 2\pi$ gilt, wird durch ableiten nach t aus 14.8

$$1 = \dot{E}(t)(1 - \varepsilon \cos E(t)) \Rightarrow (1 - \varepsilon \cos E(t)) = \dot{E}(t)^{-1} \quad (14.9)$$

- Aus der Ellipsengleichung $r(\psi) = \frac{a(1 - \varepsilon^2)}{1 + \varepsilon \cos \psi}$, $a = 1$ und $\kappa = \varepsilon$ folgt

$$1 + \varepsilon \cos \psi = \frac{1 - \varepsilon^2}{r} \quad \text{damit wird 14.1 zu} \quad \dot{\psi}(t) = (1 - \varepsilon^2)^{\frac{1}{2}} r^{-2} \quad (14.10)$$

- Aus Abb. 155 sehen wir, dass man die Länge \overline{CR} auf 2 verschiedene Arten darstellen kann

$$\overline{CR} = a \cos E = e + r \cos v \Rightarrow r \cos v = a \cos E - e \tag{14.11}$$

- Gleichung 14.11 setzen wir in die Ellipsengleichung ein:

$$r(v) = \frac{a(1 - \varepsilon^2)}{1 + \varepsilon \cos v} \Rightarrow r + \varepsilon r \cos v = a(1 - \varepsilon^2) \xrightarrow{14.11} r + \frac{e}{a}(a \cos E - e) = a - \frac{e^2}{a} \Rightarrow$$

$$\Rightarrow r = a - e \cos E = a(1 - \varepsilon \cos E) \xrightarrow{14.9} r = a \dot{E}(t)^{-1} \tag{14.12}$$

- Mit 14.12 und $a = 1$ wird 14.10 zu

$$\dot{\psi}(t) = (1 - \varepsilon^2)^{\frac{1}{2}} \dot{E}^2 \xrightarrow{\text{Taylor}} \left(1 - \frac{1}{2}\varepsilon^2\right) \dot{E}^2 \tag{14.13}$$

Kennen wir $E(t)$ können wir vielleicht durch Integrieren $\psi(t)$ zu erhalten. Jetzt wird es Zeit aus 14.8 einen Ausdruck für $E(t)$ herzuleiten. Wir benutzen dazu ein Theorem von Lagrange.

Theorem 14.1 Lagrange reversion theorem

Hat die Gleichung

$$E = M + \varepsilon \phi(E)$$

eine Lösung, so gilt für eine Funktion f

$$f(E) = f(M) + \sum_{n=1}^{\infty} \frac{\varepsilon^n}{n!} \frac{d^{n-1}}{dM^{n-1}} \{f'(M) [\phi(M)]^n\} \xrightarrow{\text{hier}} E = M + \sum_{n=1}^{\infty} \frac{\varepsilon^n}{n!} \frac{d^{n-1}}{dM^{n-1}} (\sin M)^n$$

Das ist genau das, was wir brauchen um die Kepler-Gleichung zu knacken, mit $f(x) = id(x) = x$ und $\phi = \sin$. Um die Konvergenz brauchen wir uns kein Kopfzerbrechen machen, da $\varepsilon \ll 1$ und \sin beschränkt ist! (Bei der Erde gilt $M = t$)

14. Erdbahngleichung

So jetzt liegt der Weg klar vor uns:

- Wir berechnen mit Lagrange näherungsweise $E(t)$
- Mit 14.13 näherungsweise $\dot{\psi}(t) = g(t)$, wobei wir nur Terme einer maximalen Ordnung von ε berücksichtigen
- Folgender Punkt kann mit *wxMaxima* übersprungen werden, da man $\dot{\psi}(t) = g(t)$ mit `trigreduce` "vereinfachen" kann. Zu Demonstrationszwecken hab ich es im folgenden Programm belassen, obwohl dieser Teil redundant ist!
Anschließend unterwerfen wir diese gerade Funktion (nur Cosinusterme kommen in Frage) einer Fourierreihenentwicklung:

$$g(t) = a_0 + a_1 \cos(t) + a_2 \cos(2t) + a_3 \cos(3t) + \dots \left| \int_0^{2\pi} \cos(i \cdot t) dt \quad i \in \{0, 1, 2, \dots\} \right.$$

$$\Rightarrow \int_0^{2\pi} g(t) dt = a_0 2\pi \quad \text{und}$$

$$\int_0^{2\pi} \cos(i \cdot t) \cos(j \cdot t) dt = \begin{cases} 0, & \text{für } i \neq j \\ \pi, & \text{für } i = j \end{cases}$$

- Nach Berechnung der Fourierkoeffizienten $a_0 \dots a_n$ integrieren wir um $\psi(t)$ zu erhalten.

So das setzen wir jetzt in *wxMaxima* um:

Bis zur welchen Potenz von ε wollen wir Lagrange anwenden?

(%i1) `eps_max_order:3$`

Lagrange's Theorem für E

(%i2) `E_1:M+sum(e^n/n!*diff(sin(M)^n,M,n-1),n,1,eps_max_order);`

$$\frac{(6\cos(M)^2 \sin(M) - 3\sin(M)^3) e^3}{6} + \cos(M) \sin(M) e^2 + \sin(M)e + M \quad (\text{E}_1)$$

Wir vereinfachen und bekommen eine Näherungslösung für Keplers Equation

(%i3) `KE:trigreduce(E_1);` $\frac{(3 \sin(3M) - \sin(M)) e^3}{8} + \frac{\sin(2M) e^2}{2} + \sin(M)e + M$ (KE)

Für die Erde gilt $M = t$

(%i4) `define(E(t),subst(t,M,KE));`

$$E(t) := \frac{e^3 (3 \sin(3t) - \sin(t))}{8} + \frac{e^2 \sin(2t)}{2} + e \sin(t) + t \quad (\%o4)$$

Wir entwickeln den Wurzelfaktor nach Taylor bis zur gewünschten Ordnung

(%i5) `eps_factor(e):=taylor(sqrt((1-e^2)),e,0,eps_max_order);`

$$\text{eps_factor}(e) := \text{taylor}(\sqrt{1 - e^2}, e, 0, \text{eps_max_order}) \quad (\%o5)$$

(%i6) eps_taylor:expand(sum(coeff(eps_factor(e),e,i)*e^i,i,0,eps_max_order));

$$1 - \frac{e^2}{2} \quad (\text{eps_taylor})$$

Wir ermitteln $\dot{E}(t) = E_{pt}(t)$

(%i7) define(E_pt(t),diff(E(t),t));

$$E_{pt}(t) := \frac{e^3 (9 \cos(3t) - \cos(t))}{8} + e^2 \cos(2t) + e \cos(t) + 1 \quad (\%o7)$$

Unsere Näherung für $\dot{\psi}(t)$

(%i8) T1:eps_taylor*(E_pt(t))^2;

$$\left(1 - \frac{e^2}{2}\right) \left(\frac{e^3 (9 \cos(3t) - \cos(t))}{8} + e^2 \cos(2t) + e \cos(t) + 1\right)^2 \quad (\text{T1})$$

Für Kontrollzwecke können wir uns den Term anschauen - hier ausdrucken hat keinen Sinn

(%i9) h(t):=expand(T1)\$

Wir holen uns die Koeffizienten von ε^i expandieren und vereinfachen sie

(%i10) define(g(t),sum(trigreduce(ratsimp(trigexpand(coeff(h(t),e,i))))*e^i,i,0,eps_max_order));

$$g(t) := -\frac{e^3 (\cos(t) - 13 \cos(3t))}{4} + \frac{5e^2 \cos(2t)}{2} + 2e \cos(t) + 1 \quad (\%o10)$$

Wir integrieren "komponentenweise" und bekommen eine Näherung für $\psi(t)$

(%i11) define(psi(t),sum(ratsimp(integrate(coeff(g(t),e,i),t))*e^i,i,0,eps_max_order));

$$\psi(t) := \frac{e^3 (13 \sin(3t) - 3 \sin(t))}{12} + \frac{5e^2 \sin(2t)}{4} + 2e \sin(t) + t \quad (\%o11)$$

zur Probe mit Fourier - wie im Text angegeben!

(%i12) a[0]:(1/(2*%pi)*integrate(g(t),t,0,2*%pi));

$$1 \quad (\%o12)$$

(%i13) declare(n,integer,i,integer,j,integer);

done (%o13)

(%i14) integrate(cos(i*x)*cos(j*x),x,0,2*%pi);

$$0 \quad (\%o14)$$

14. Erdbahngleichung

(%i15) c:=integrate(cos(i*x)*cos(i*x),x,0,2*%pi);

$$\pi \quad (c)$$

(%i16) a[n]:=(1/%pi*integrate(g(t)*cos(n*t),t,0,2*%pi));

$$a_n := \frac{1}{\pi} \int_0^{2\pi} g(t) \cos(nt) dt \quad (%o16)$$

Die Fourierreihe für $g_f(t) := \dot{\psi}(t)$

(%i17) define(g_f(t), expand(ratsimp(a[0]+sum(a[i]*cos(i*t),i,1,eps_max_order))));

$$g_f(t) := \frac{13e^3 \cos(3t)}{4} + \frac{5e^2 \cos(2t)}{2} - \frac{e^3 \cos(t)}{4} + 2e \cos(t) + 1 \quad (%o17)$$

(%i18) define(g_f_orderedBy_e_(t),sum(coeff(g_f(t),e,i)*e^i,i,0,eps_max_order));

$$g_f_orderedBy_e_ (t) := e^3 \left(\frac{13 \cos(3t)}{4} - \frac{\cos(t)}{4} \right) + \frac{5e^2 \cos(2t)}{2} + 2e \cos(t) + 1 \quad (%o18)$$

(%i19) define(psi(t),integrate(g_f_orderedBy_e_(t),t));

$$\psi(t) := e^3 \left(\frac{13 \sin(3t)}{12} - \frac{\sin(t)}{4} \right) + \frac{5e^2 \sin(2t)}{4} + 2e \sin(t) + t \quad (%o19)$$

Wie wir sehen stimmen %o11 und %o19 überein und sind von der gewünschten Genauigkeit!
Ist eine höhere Genauigkeit gewünscht, ist `eps_max_order` entsprechend zu erhöhen!

14.9 Implemtation in *Geogebra*

Wir benutzen das Koordinatsystem(KS) von Kepler. Bei der Gleichung

$$r(\psi) = \frac{a^2 - e^2}{a + e \cos(\psi)}$$

liegt der Ursprung im Brennpunkt S (Sonne) - daher erhalten wir die Ellipse in “unserem” KS durch eine Verschiebung

$$(x(t), y(t)) = r(\psi(t)) \left[\frac{a^2 - e^2}{a + e \cos(\psi(t))} \right] + (e, 0)$$

Die große Halbachse a benutzen wir wieder als Längeneinheit und als Zeitskala benutzen wir $\frac{T}{2\pi}t$ (im Perihel wird gestartet!), sodass wenn unsere Uhr $2\pi \text{ rad} \approx 1256 * 0,005 \text{ rad}$ anzeigt, 1 Umlauf vollendet ist. Der kleinste “Tick” unserer Uhr ist 0.005 rad - damit teilen wir die Umlaufellipse in 1257 Ortspunkte \vec{r}_i (die Position von P (Planet)) ein!

14.9.1 Die Konstruktion

- $a=1$, Schieberegler für $e \in [0, 0.99]$ und $b=\text{sqrt}(a^2-e^2)$
- Schieberegler für unsere Zeit $t \in [0, 2\pi]$, Schrittweite 0.005; $\varepsilon = \frac{e}{a}$
- Ursprung und Scheitelpunkt: $O=(0,0)$ $A=(a,0)$
- Ellipse, Umkreis und Sonnenposition:
 $(x/a)^2+(y/b)^2=1$, $x^2+y^2=a^2$ und $S=(e,0)$
- Jetzt können wir mit der eigentlichen Konstruktion von Kepler beginnen:
 $Y=a*(\cos(t), \sin(t))$
- $M=\text{Angle}(A, O, Y)$
- Jetzt geht es an die Berechnung des Winkels E aus der Kepler-Gleichung:

$$\underbrace{M + \varepsilon \sin E}_{f(E)} = E$$

E ist also ein Fixpunkt von f . Man kann sich vergewissern, dass f die Voraussetzungen des Banach'schen Fixpunktsatzes erfüllt und daher eine Iteration gegen E konvergiert - als Startwert wählen wir M

$$f(x)=M+\varepsilon\sin(x) \quad \text{und} \quad E=\text{Iteration}(f, M, 20)$$

- Damit ist Q jetzt festgelegt: $Q=a*(\cos(E), \sin(E))$
- P liegt auf der Ellipse mit $x_P = x_Q$ und seine y -Koordinate y_P hat dasselbe Vorzeichen wie Q :
 $y_P=\text{sqrt}(1-((x(Q)/a)^2)*b*\text{sgn}(y(Q)))$ und $P=(x(Q), y_P)$

Überprüfung des Flächensatzes: Dabei denkt man als Mathematiker sofort an ein Integral, leichter zum Zeichnen ist allerdings ein Polygonzug mit einer Punktliste - nicht so genau, aber für unsere Zwecke reicht es hier.

Wir brauchen also $P(t_i)$ - die Position von P beim i -ten Uhrentick. Dazu basteln wir uns ein Benutzerwerkzeug:

```
output: P
name: getP
```

Geogebra meint für dessen Konstruktion benötigt es den "Zeitwinkel" t , a , e und den Ursprung O - also glauben wir es.

- Die gesamte Punktliste der Umlaufellipse:
 $L_1=\text{Sequence}(\text{getP}(i* 0.005 \text{ rad}, a, e, 0), i, 0, 1257)$
- Wo befinden wir uns gerade?
 $\text{lastIndex}=t / 0.005 \text{ rad} + 1$

14. Erdbahngleichung

- Ellipsensegment als Polygonzug zeichnen:

`L_2=Sequence(Element(L_1, i), i, 1, lastIndex)` und `L_3={S,A}`
`L_{Poly}={L_2,L_3}` und `q1=Polygon(Join(L_{Poly}))`

Geogebra berechnet uns mit $q1$ gleich die Fläche des Polygonzugs - eine Untersumme des Integrals

Für $e = 0.7$ ergeben sich z.B.

$t[\text{rad}]$	$q1$
0.25	0.0875
0.5	0.1767

 - eine gute Übereinstimmung!



Bei der Erdbahn gilt $\varepsilon = 0.0167$ - dann liegt die Umlaufellipse innerhalb der Linie des Umkreises! Beachtlich was Tycho Brahe und Kepler da geleistet haben!



Eine Animation der Konstruktion ist nicht sinnvoll, da sie einen falschen Eindruck vermittelt - die Berechnungen dauern in späteren Umlaufteilen länger - dadurch benötigt der Planet bei der "Simulation" für die zweite Umlaufhälfte mehr Zeit als für die erste (was natürlich nicht stimmt).

Daher habe ich mich entschlossen, die Geschwindigkeit des Planeten durch einen Geschwindigkeitsvektor zu veranschaulichen (die Zeit t muss man manuell "verstellen"):

$$\vec{v}(t_i) \approx \frac{\vec{r}_{i+1} - \vec{r}_i}{t_{i+1} - t_i}$$

- Implementiert mit dem Hilfsvektor

`v_1=scaleFact*Vector(Element(L_1, lastIndex), Element(L_1, lastIndex + 1))`

wobei `scaleFact` - wie der Name schon nahelegt - ein Skalierungsfaktor für die Vektorenlänge ist. Und dann der "eigentliche" Vektor

`velocity = Vector(P, P + v1)`

So jetzt bleibt noch die "Approximation" für $\varepsilon < 0.2$ zu überprüfen:

- $\psi = \varepsilon^3 / 12 * (13 * \sin(3*t) - 3 * \sin(t)) + 5/4 * \varepsilon^2 * \sin(2*t) + 2 * \varepsilon * \sin(t) + t$

- $r = (a^2 - e^2) / (a + e * \cos(\psi))$

- $P_{\text{Approx}} = r * (\cos(\psi), \sin(\psi)) + (e, 0)$

Man sieht, dass unsere Formel die Planetsposition ganz gut wiedergibt für kleine ε ! Das vollständige Arbeitsblatt bei <https://www.geogebra.org/m/gQX7Z838>!



Wie kommt man von Umlaufdauer T , großer Halbachse a und Exzentrizität e auf die tatsächliche Planetenposition?

14.10 ANHANG1: Keplergleichung mit Integralrechnung

14.10.1 Ellipsenabschnitt

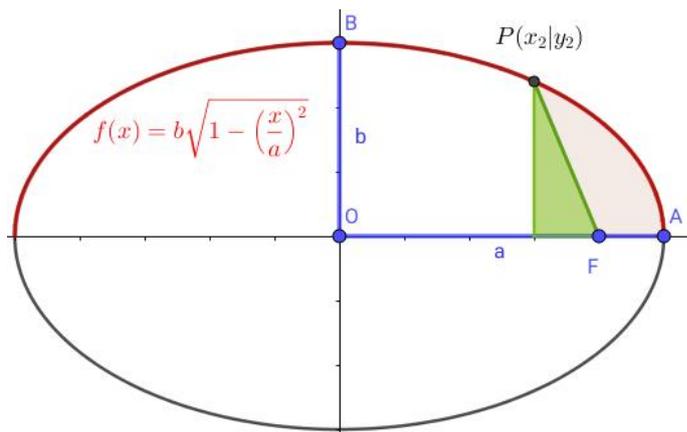


Abb.156 : Ellipsenabschnitt

$$\begin{aligned}
 A &= b \int_{x_2}^a \sqrt{1 - \left(\frac{x}{a}\right)^2} dx && u = \frac{x}{a} \\
 &&& \downarrow \\
 &= ab \int_{\frac{x_2}{a}}^1 \sqrt{1 - u^2} du && u = \cos t \\
 &&& \downarrow \\
 &= ab \int_0^{\arccos \frac{x_2}{a}} \sin^2 t dt && \text{partiell} \\
 &&& \downarrow \\
 &= \frac{ab}{2} [t - \cos t \sin t] \Big|_0^{\arccos \frac{x_2}{a}} = \\
 &= \frac{ab}{2} [t - \cos t \sqrt{1 - \cos^2 t}] \Big|_0^{\arccos \frac{x_2}{a}}
 \end{aligned}$$

$$A = \frac{ab}{2} \left[\arccos \frac{x_2}{a} - \frac{x_2}{a} \sqrt{1 - \left(\frac{x_2}{a}\right)^2} \right] = \frac{1}{2} \left[ab \arccos \frac{x_2}{a} - x_2 y_2 \right] \quad (14.14)$$

14.10.2 Keplerproblem

Ich beziehe mich auf die Bezeichnung von Abb. 14.8

- Fläche des Kreissektors mit Winkel M nimmt linear zu: $A_K = \frac{a^2}{2} M$
- $M \propto t$ und die Ellipsenfläche in Abhängigkeit von M (in obiger Zeichnung begrenzt mit den Punkten F, P und A und hellbraun dargestellt und in Abb. 14.8 bezeichnet mit $A_{ell} \pm A_{\Delta SRP}$ - wobei die Differenz selbst das Vorzeichen der Dreiecksfläche regelt)

$$A_e = A_K \cdot \frac{b}{a} = \frac{a^2}{2} M \cdot \frac{b}{a}$$

14. Erdbahngleichung

■ Ellipsenordinate = Kreisordinate $\times \frac{b}{a} \Rightarrow y_E = y_K \cdot \frac{b}{a}$

■ Keplers Flächensatz (Brennpunkt $F = (e, 0)$):

$$\underbrace{\frac{1}{2} \left[ab \arccos \frac{x_2}{a} - x_2 y_2 \right]}_{\text{halber Ellipsenabschnitt}} + \underbrace{\frac{1}{2} (x_2 - e) y_2}_{\text{grünes Dreieck}} = \underbrace{\frac{a^2}{2} M \cdot \frac{b}{a}}_{\text{Ellipsenfläche}} \quad (14.15)$$

■ Aus Abb. 14.8 ist ersichtlich

$$\cos E = \frac{x_2}{a} \quad \text{und} \quad \sin E = \frac{y_K}{a} = \frac{1}{a} \frac{y_2 a}{b} = \frac{y_2}{b}$$

■ Damit ergibt sich für Gleichung 14.15 und $\varepsilon = \frac{e}{a}$

$$abE - eb \sin E = abM \quad | : ab \Rightarrow$$

$$E - \varepsilon \sin E = M$$

14.11 ANHANG2: Das 2. Gesetz Kepler's - "der Flächensatz"

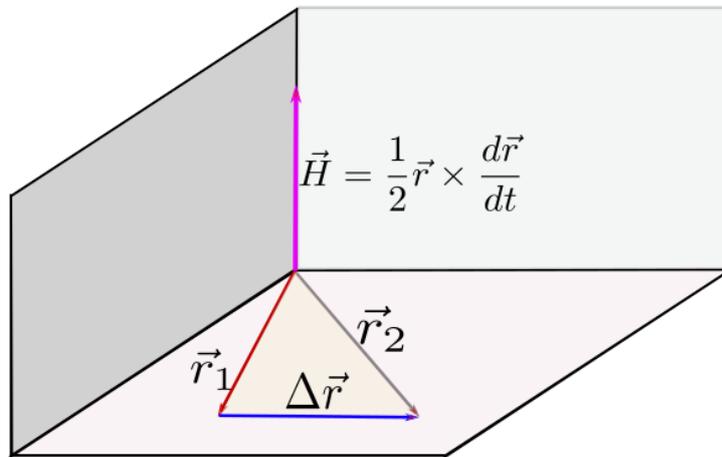


Abb.157 : 2. Gesetz Kepler's

In der Zeit Δt bewegt sich ein Körper von Position \vec{r}_1 nach \vec{r}_2 . Die Fläche, die der Positionsvektor dabei überstreicht ist annähernd $\frac{1}{2} \vec{r} \times \frac{\Delta \vec{r}}{\Delta t}$. Es gilt daher:

$$\lim_{\Delta t \rightarrow 0} \frac{1}{2} \vec{r} \times \frac{\Delta \vec{r}}{\Delta t} = \frac{1}{2} \vec{r} \times \frac{d\vec{r}}{dt} = \vec{H}$$

Das Bewegungsgesetz eines Körpers der Masse m sei

$$m \frac{d^2 \vec{r}}{dt^2} = f(r) \vec{r}_0 \quad \text{wobei } f(r) > 0 \text{ und } |\vec{r}_0| = 1 \text{ und } \vec{r}_0 \parallel \vec{r}$$

Wir multiplizieren obige Gleichung mit $\vec{r} \times$ - dies ergibt

$$\vec{r} \times m \frac{d^2 \vec{r}}{dt^2} = \vec{r} \times f(r) \vec{r}_0 = \vec{0}$$

also

$$\vec{r} \times \frac{d^2 \vec{r}}{dt^2} = \vec{0} \Rightarrow \frac{d}{dt} \left(\vec{r} \times \frac{d\vec{r}}{dt} \right) = \vec{0} \Rightarrow \boxed{\vec{r} \times \frac{d\vec{r}}{dt} = \vec{c} = 2\vec{H}} \quad \text{wobei } \vec{c} \text{ ein konstanter Vektor}$$

Oft wird auch die Drehimpulserhaltung bemüht:

$$\vec{L} = m\vec{r} \times \vec{v} \Rightarrow \vec{H} = \frac{\vec{L}}{2m} \quad \text{und} \quad \frac{d\vec{L}}{dt} = \vec{M} \quad \text{und} \quad \vec{M} = \vec{r} \times \vec{F} \stackrel{\vec{r} \parallel \vec{F}}{\underset{\downarrow}{=}} \vec{0}$$

Da das Drehmoment verschwindet, ist der Drehimpuls \vec{L} und damit \vec{H} konstant!

14.12 ANHANG3: $(1+x)^{-2}$ als Reihe für $0 \leq x \ll 1$

Wir entwickeln in eine Taylorreihe bei $x_0 = 0$ mit *wxMaxima*:

- (% i2) $f(x):=(1+x)^{-2}$; $f(x) := (1+x)^{-2}$
 (% i5) $taylor(f(x),x,0,9)$; $1 - 2x + 3x^2 - 4x^3 + 5x^4 - 6x^5 + 7x^6 - 8x^7 + 9x^8 - 10x^9$
 (% i6) $sum((n+1)*(-x)^n,n,0,9)$; $-10x^9 + 9x^8 - 8x^7 + 7x^6 - 6x^5 + 5x^4 - 4x^3 + 3x^2 - 2x + 1$

Dies führt uns zu folgender Behauptung:

Theorem 14.2

$$(1+a)^{-2} = \sum_{n \geq 0} (n+1)(-a)^n \quad |a| < 1 \quad (14.16)$$

Beweis: Wir benutzen dazu die geometrische Wahrscheinlichkeitsverteilung: Ein Versuch wird solange wiederholt bis das gewünschte Ergebnis (welches mit einer Wahrscheinlichkeit p eintritt) kommt.

Die Zufallsvariable X ist die Anzahl der benötigten Wiederholungen:

$$P(X = n) = \underbrace{(1-p)^{n-1}}_q p \quad \text{wobei gilt} \quad (*) \quad \sum_{n=1}^{\infty} P(X = n) = \sum_{n=1}^{\infty} p q^{n-1} = p \frac{1}{1 - (1-p)} = 1$$

Die erzeugende Funktion

$$f(x) = \sum_{n \geq 1} p (q x)^{n-1} = \frac{p}{1 - xq} \Big| \frac{d}{dx} \quad \text{konvergiert sicher für } x \in [0, 1]$$

$$f'(x) = \sum_{n \geq 2} p(n-1)(q x)^{n-2} q = p(-1)(1-xq)^{-2}(-q) \quad \text{mit } m := n-2 \text{ folgt}$$

$$f'(1) = \sum_{m \geq 0} p q (m+1)(q)^m = p q (1-q)^{-2} \quad \text{mit } a := -q \text{ folgt die Behauptung}$$

□



Die Voraussetzung $|a| < 1$ ist bei uns mit $a = \kappa \cos \psi$ erfüllt

14.13 ANHANG4: $(1-x)^{\frac{3}{2}}$ als Reihe für $0 \leq x \ll 1$

Sehen wir uns die Koeffizienten der Taylorreihenentwicklung bei $x_0 = 0$ an:

Hier ein kleine Programm in *wxMaxima*, welches die Produkte nicht ausrechnet, damit man die Gesetzmäßigkeit besser sieht!

Also lässt sich die Taylorreihe folgendermaßen schreiben:

$$(1-x)^{\frac{3}{2}} = 1 - \frac{3}{2}x + \frac{3}{8}x^2 + 3 \sum_{n \geq 3} \left[\prod_{i=1}^{n-2} (2i-1) \right] \frac{x^n}{n! 2^n} \quad (14.17)$$

Mit Theorem 14.3 können wir 14.17 umschreiben

$$(1-x)^{\frac{3}{2}} = \dots + 3 \sum_{n \geq 3} \left[\frac{(2n-4)!}{2^{n-2} (n-2)!} \right] \frac{x^n}{n! 2^n} = \dots + 3 \sum_{n \geq 3} \left[\frac{(2n-4)!}{(n-2)! (n-2)!} \right] \frac{x^n}{(n-1)n 2^{n-2} 2^n}$$

$$(1-x)^{\frac{3}{2}} = 1 - \frac{3}{2}x + \frac{3}{8}x^2 + 3 \sum_{n \geq 3} \binom{2n-4}{n-2} \frac{x^n}{2^{2n-2} (n-1)n} \quad (14.18)$$

Wir überprüfen 14.18 mit *wxMaxima*

(% i1) `f(x):=(1-x)^(3/2)$`

(% i6) `makelist(coeff(taylor(f(x),x,0,10),x,i),i,0,10) ;`

$$\left[1, -\frac{3}{2}, \frac{3}{8}, \frac{1}{16}, \frac{3}{128}, \frac{3}{256}, \frac{7}{1024}, \frac{9}{2048}, \frac{99}{32768}, \frac{143}{65536}, \frac{429}{262144} \right]$$

(% i5)

`append([1,-3/2,3/8],3*makelist(binomial(2*n-4,n-2)/((n-1)*n*2^{2*n-2}),n,3,10));`

$$\left[1, -\frac{3}{2}, \frac{3}{8}, \frac{1}{16}, \frac{3}{128}, \frac{3}{256}, \frac{7}{1024}, \frac{9}{2048}, \frac{99}{32768}, \frac{143}{65536}, \frac{429}{262144} \right]$$

14. Erdbahngleichung

$\binom{2n-4}{n-2}$ bzw. $\binom{2m}{m}$ ist der mittlere Binomialkoeffizient (central binomial coefficient)

- er ist der größte für alle k mit $\binom{2m}{k}$ für ein bestimmtes m !



Stellt sich die Frage, ob 14.18 überhaupt konvergiert? - Im nachfolgenden *wxMaxima*-Programm zeigen wir, dass die Koeffizienten von x^n doch langsam kleiner werden, sodass für $|x| < 1$ die Reihenglieder gegen Null gehen (notwendige Bedingung für Konvergenz).

Der mittlere Binomialkoeffizient ergibt sich auch als Koeffizient der Taylorreihe der erzeugenden Funktion $\frac{1}{\sqrt{1-4x}}$

(% i1) `fpprintprec:3$`

the first 10 central binomial coefficients

(% i2) `makelist(binomial(2*m,m),m,1,10);`

[2, 6, 20, 70, 252, 924, 3432, 12870, 48620, 184756]

the first 10 taylor coefficients for the generating function

(% i3) `makelist(at(diff(1/sqrt(1-4*x),x,m),x=0)/m!, m,1,10) ;`

[2, 6, 20, 70, 252, 924, 3432, 12870, 48620, 184756]

Is there a chance that series 14.18 converges? Yes, there is!

(% i4) `makelist(binomial(2*m,m)/((m+1)*(m+2)*2^(2*m+2)),m,1,100),numer;`

[0.0208, 0.00781, 0.00391, 0.00228, ..., 1.4410⁻⁶, 1.410⁻⁶, 1.3710⁻⁶]

Theoretische Begründung:



$$\lim_{m \rightarrow \infty} \left[\binom{2m}{m} - \frac{4^m}{\sqrt{\pi m}} \right] = 0 \quad \text{d.h. für große } n \text{ gilt in 14.18}$$

$$\binom{2n-4}{n-2} \frac{1}{2^{2n-2}(n-1)n} \approx \frac{1}{4n(n-1)\sqrt{\pi(n-2)}} \rightarrow 0$$

Theorem 14.3 Produkt der ungeraden Zahlen

$$\prod_{i=1}^n 2i - 1 = \frac{(2n)!}{2^n n!}$$

Beweis: Wir ergänzen das Produkt der ungeraden Zahlen mit den geraden Zahlen und $2n$:

$$1 \cdot 3 \cdot 5 \dots (2n-1) = \frac{1 \cdot 2 \cdot 3 \cdot 4 \dots (2n-1) \cdot (2n)}{2 \cdot 4 \cdot 6 \cdot 8 \dots (2n)} =$$

aus den n -Faktoren im Nenner können wir jeweils die Zahl 2 “herausheben”:

$$1 \cdot 3 \cdot 5 \dots (2n-1) = \frac{(2n)!}{2^n \cdot 1 \cdot 2 \cdot 3 \cdot 4 \dots n} = \frac{(2n)!}{2^n n!}$$

□

15 | Rotationskörper im Wasser



Warnung: Die meisten Berechnungen oder Simulationen in den folgenden *Geogebra*-Arbeitsblättern sind so rechenintensiv, dass sie weder auf einer Handy-App noch als webpage “flüssig” laufen. Das beste “Simulationserlebnis” hat man mit einer Desktop-Version von *Geogebra* und Arbeitsblatt runterladen!

Abstract:

Ein Rotationskörper (oben offen, erzeugende Funktion f mit $x \in [x_1, x_2]$) wird in einen Zylinder (Radius R), der mit Wasser bis zu einer Höhe H gefüllt ist, eingetaucht.

Dabei tauchen einige Fragestellungen auf:

- Wie hoch steht das Wasser im Zylinder bei einer bestimmten Eintauchtiefe?
- Man misst die Dicke des ebenen Bodens. Mißt das Gewicht. Mißt das Volumen des Materials (Außenvolumen minus Innenvolumen). Wie dick ist der Körper, falls sie konstant ist?
- Wie weit taucht der leere Hohlkörper ein?
- Bei welcher eingefüllten Wassermenge geht er unter?
- Wie hoch steht eine gewisse Wassermenge im Behälter?
- Wann kippt er um?

15.1 Wasserspiegel

Eine rotationssymmetrische “Vase” wird in einen halb mit Wasser gefüllten Zylinder getaucht. Wie hoch steht der Wasserspiegel? Die Umrissfunktion der Vase sei

$$f(x) = e^{-0.2x} \sin(x) + 1 \quad x \in [0, width] \quad width = 2\pi$$

Wir simulieren die Situation mit *Geogebra*. Den Ursprung legen wir auf den Wasserspiegel ohne Eintauchen, die x -Achse ist die Zylinderachse. Die Umrisspolygonpunkte des Zylinders sind $Z_i = (\pm H, \pm R)$. Für die Eintauchtiefe nehmen wir natürlich einen Schieberegler e . Er ist der Betrag, welchen die Vase unter Normalpegel liegt. Er geht von 0 bis $e_{max} = ?$ Verschieben wir dieses Problem und nehmen für e_{max} eine harmlose “Hausnummer” z.B. 1.

Mit etwas Rechnen (dividieren durch $R^2\pi$ und 15.1) ergibt sich

$$w = \underbrace{\int_0^{e+w} \left(\frac{f}{R}\right)^2}_{G_e(w)} \Rightarrow w \text{ ist Fixpunkt von } G_e \quad (15.2)$$

Im ANHANG 15.1.4 wird gezeigt, dass eine Iteration von G_e gegen ihren Fixpunkt w konvergiert. Was nehmen wir als Startwert?

Wir nehmen hier den Mittelwert des Radius des bisher eingetauchten Körpers als Prognose - aber es gibt natürlich sehr verschiedene Möglichkeiten (z.B. Stichwort "Untersummen", Gesamtmittelwert der quadratischen Funktion, Approximation von f durch einen Polynomzug oder einer Polynomfunktion, usw.):

$$m_e = \text{If}(e==0, 0, 1/e * \text{NIntegral}(f, 0, e))$$

Damit ergibt sich der Startwert w_s für den Wasserstand

$$\underbrace{m_e^2 e \pi}_{\text{verdrängt}} = \underbrace{(R^2 - m_e^2) w_s \pi}_{\text{aufgestaut}} \Rightarrow w_s = e \frac{m_e^2}{(R^2 - m_e^2)}$$

Jetzt noch - wie versprochen - die maximale Eintauchtiefe e_{max} :

maximale Eintauchtiefe plus Wasserstandshöhe ist Gefäßhöhe - in Mathematik

$$e_{max} = width - w = width - \int_0^{e+w} \left(\frac{f}{R}\right)^2 = width - \int_0^{width} \left(\frac{f}{R}\right)^2$$

Außerdem können wir jetzt auch die Niveaupunkte W_i des Wasserspiegels setzen!

In *Geogebra* schaut das dann so aus

```
startValue = e*m_e^2/(R^2-m_e^2)
waterLevel = Iteration(NIntegral((f(x)/R)^2, 0, e+w), w, {startValue}, 15)
e_{max} = width - NIntegral((f(x)/R)^2, 0, width)
W_1 = (waterLevel, R)
W_2 = (waterLevel, -R)
```

- Als "Probe" können wir uns das Gesamte Wasservolumen während der Simulation ausrechnen - es muss sich immer derselbe Wert ergeben, d. h. obwohl er ständig neu berechnet wird, bleibt er unverändert!

$$V_w = R^2 * (H - e) + \text{NIntegral}(R^2 - f(x)^2, 0, e + \text{waterLevel})$$

15. Rotationskörper im Wasser

15.1.2 3D Simulation als Zugabe



Für das Zeichnen von Rotationskörpern verweise ich auf mein Arbeitsblatt <https://www.geogebra.org/m/Z32jMPjM> und dem zugehörigen Begleittext!

- Zuerst der zylindrische Wasserbehälter

$$a(t, \theta) = \text{Surface}(t, \cos(\theta) R, \sin(\theta) R, t, -H, H, \theta, 0, 2\pi)$$

Er wird weitgehend durchsichtig gestaltet - damit er den Blick auf das eigentliche Geschehen kaum stört!

- Dann der "Wasserzylinder" - er reicht nur bis *waterLevel*

$$c(t, \theta) = \text{Surface}(t, \cos(\theta) R, \sin(\theta) R, t, -H, \text{waterLevel}, \theta, 0, 2\pi)$$

- Jetzt die Wasseroberfläche - sie besteht aus einer Liste von konzentrischen Kreisen, deren gemeinsamer Mittelpunkt (*waterLevel*,0,0) ist. Ein Kreis mit Radius *r* hat dann die Darstellung

$$\text{Curve}(\text{waterLevel}, r \cos(t), r \sin(t), t, 0, 2\pi)$$

r lassen von $f(e + \text{waterLevel}) + \varepsilon$ bis *R* laufen mit einer Schrittweite von 0.1:

$$\text{Sequence}(\text{Curve}(\text{waterLevel}, r * \cos(t), r * \sin(t), t, 0, 2 * \pi), r, f(e + \text{waterLevel}) + 0.1, R, 0.1)$$

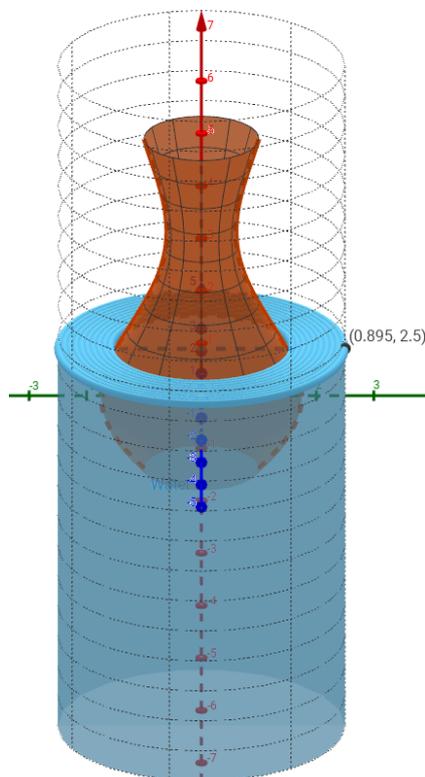


Abb.159 : Vase beim Abtauchen

15.1.3 Der Graph $w(e)$

Mit *Geogebra*

Um nur den Graphen von $w(e)$ zu erhalten - also die Wasserstandshöhe in Abhängigkeit von der Eintauchtiefe - können wir uns von dem ganzen Simulations-“Schnickschack” verabschieden und konzentrieren uns auf das Wesentliche:

- Die Zahlen H, R und $width$
- Natürlich unsere Funktion: $f(x) = e^{-0.2 \cdot x} \cdot \sin(x) + 1$
- Den maximalen Wert für die Eintauchtiefe und Slider $e \in [0, e_{max}]$
 $e_{max} = width - NIntegral((f(x) / R)^2, 0, width)$
- Mittelwert m_e von f bis zur Eintauchtiefe e
 $m_e = If(e == 0, 0, 1/e \cdot NIntegral(f, 0, e))$
- Hätte der Körper überall einen Radius von m_e ergäbe sich eine Wasserstandshöhe w_s

$$e \cdot m_e^2 \cdot \pi = (R^2 - m_e^2) \cdot w_s$$

$$startValue = e \cdot m_e^2 / (R^2 - m_e^2)$$

- Jetzt die Berechnung des Wasserspiegels durch Iteration
 $waterLevel = Iteration(NIntegral((f(x)/R)^2, 0, e+w), w, \{startValue\}, 5)$
- Schließlich der Graph als Ortslinie
 $W = (e, waterLevel)$ $graph = Locus(W, e)$

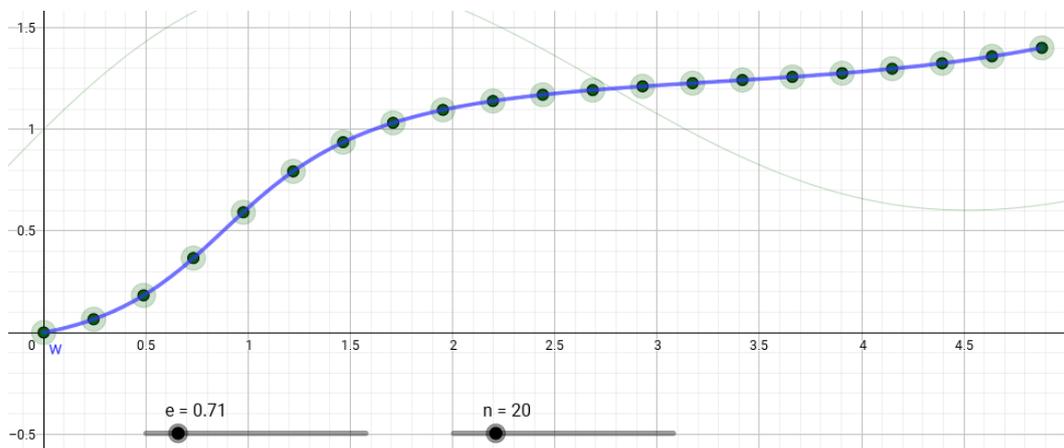


Abb.160 : Der Graph $w(e)$ als Ortslinie

Aus welchen Punkten besteht nun die Ortslinie? Dazu erzeugen wir ein “Custom-Tool”:

Output: W Input: f, R, e Name: `getLevelPoint`
 $pointL = Sequence(getLevelPoint(f, R, depth), depth, 0, e_{max}, e_{max}/n)$
 Jetzt kann man die Datenpunkte kopieren!

15. Rotationskörper im Wasser

Mit *wxMaxima*

Wir schreiben 15.2 um

$$\underbrace{w - G_e(w)}_{F(w)} = 0 \Rightarrow F'(w) = 1 - \left(\frac{f(e+w)}{R} \right)^2 \xrightarrow{\text{Newton}} w_{i+1} = w_i - \frac{F(w_i)}{F'(w_i)}$$

und lösen diese Gleichung mit dem Newton-Verfahren

Die Ableitung von F ist

 $F'(w) = \frac{d}{dw}(w - G_e(w)) = 1 - \frac{d}{dw}G_e(w)$
 $\frac{d}{dw}G_e(w)$ wird im Unterkapitel 15.1.4 behandelt!

Hier nun das entsprechende *wxMaxima* Programm für obigen Berechnungsweg:

Umrissfunktion und Integrand

```
(%i1) f_contour(x) := float(exp(-0.2*x)* sin(x) + 1)$
```

```
(%i3) R:2.5$width:2*$pi$
```

```
(%i4) integrand(t):=(f_contour(t)/R)^2$
```

Definieren die numerische Integration, fixe Variable t , fixe Untergrenze 0, 3 \rightarrow mäßige Oszillation, erster Teil der Liste (Ergebnis) wird zurückgegeben

```
(%i5) nInt(b):=block([h:0],h:quad_qag(integrand,t,0,b,3), first(h))$
```

```
(%i6) G(e,x):=nInt(e+x)$
```

Zähler (numerator) vom Newton-Verfahren

```
(%i7) Num(e,x):=x-G(e,x)$
```

$$e_{max} = width - \int_0^{width} \left(\frac{f}{R} \right)^2 = width - G(0, width) = Num(0, width)$$

```
(%i8) e_max:float(Num(0,width)); 4.881025246780005 (e_max)
```

Nenner (denominator) vom Newton-Verfahren

```
(%i9) Denom(e,x):=1-integrand(e+x)$
```

Newton's Iterationsformel

```
(%i10) newton(e,x):=x-Num(e,x)/Denom(e,x)$
```

Solange die Differenz größer als err ist, wird iteriert

```
(%i11) recurse(e,x,err):=block(
      x_new:newton(e,x),
      if (abs(x - x_new) > err) then recurse(e,x_new,err),
      x_new)\$
```

Wir erstellen 100 Plotpunkte mit dem Newtonverfahren

```
(%i12) plot_newton:makelist([i,recurse(i,i,0.0001)],i,0,e_max,e_max/100)$
```

Aus *Geogebra* wurde die Punktliste in einen Editor kopiert, runde und geschwungene Klammern in eckige umgewandelt, ausgeschnitten und hierher kopiert

```
(%i13) points_geogebra : [[0, 0], [0.0488, 0.01], [0.0976, 0.0213], ...
```

Jetzt kommt die Stunde der Wahrheit: wir vergleichen

```
(%i14) plot2d([ [discrete,plot_newton], [discrete,points_geogebra] ],
      [x,0,e_max],[same_xy],
      [title, "Comparison Iteration({\Geogebra}) vs. Newton({\wxMaxima})"],
      [legend, "solution by Newton","solution by Fixpoint-Theorem"],
      [style,[lines,6,1,2], [lines,3,2,2]],
      [gnuplot_preamble, "set key bottom right; set xtics font \", 15\";
      set ytics font \", 15\"; set key font \", 15\";
      set title font \", 20\" "]
      );
```

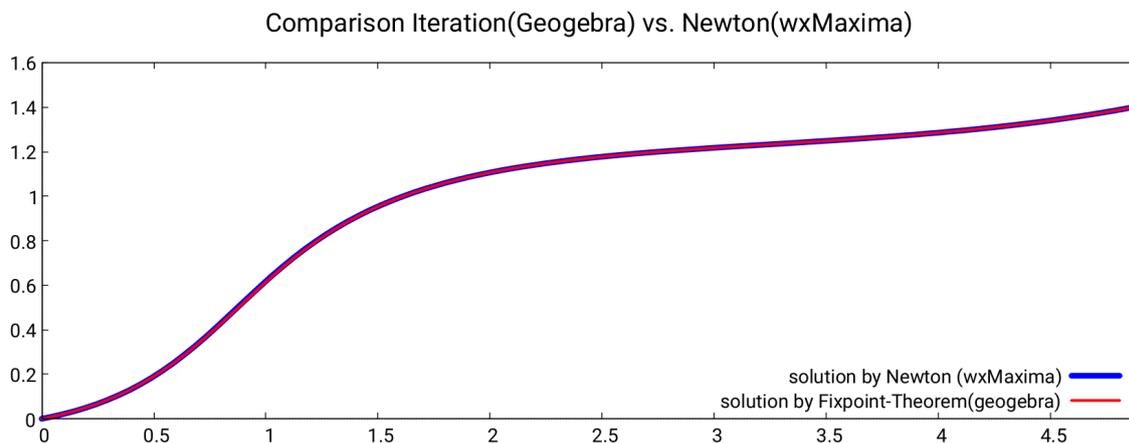


Abb.161 : Beide Lösungen im Vergleich

Besser hätte es nicht ausgehen können.

15. Rotationskörper im Wasser

15.1.4 Konvergiert die Iteration von G_e ?

$$G(x) := \int_0^x \left(\frac{f(t)}{R} \right)^2 dt \Rightarrow G_e(x) = G(x + e)$$

Der Graph von G_e ist der um e -Einheiten nach links verschobene Graph von G ! Ist dieser einigermaßen "flach", sollte sich ein eindeutiger Schnittpunkt mit der 1. Mediane (Fixpunkt) ergeben.

Wir verschaffen uns einen Überblick mit *Geogebra*.

Wir zeichnen den Graph als Ortlinie eines Punktes!

Also zuerst einen Schieberegler für e , dann einen - nennen wir in t - für die Ortslinie (Locus).

Unsere Funktion f im Intervall $[0, 2\pi]$ und R .

```
P=(t, NIntegral((f(x) / R)^2, 0, e + t))
locus = Locus(P, t)
G_{e1}= Sequence((i/10, NIntegral((f(x)/R)^2, 0, 1+i/10)), i, 0, 100)
```

Der letzte Befehl zeichnet eine Punktliste für den Graphen von $G_1(x)$. Damit können wir mit verschieben von e leicht erkennen, dass eine Linksverschiebung stattfindet. Die Sache mit dem eindeutigen Schnittpunkt mit der 1. Mediane sieht recht gut aus - es gibt nur 2 Extremfälle:

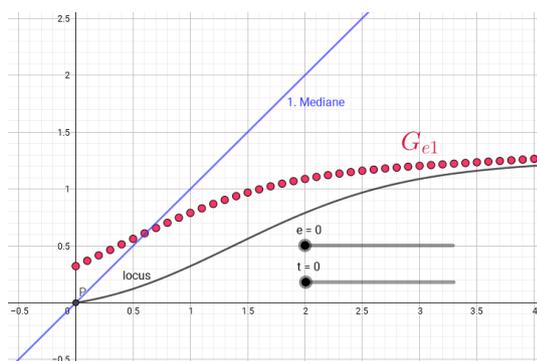


Abb.162 : Extremfall $e = 0$

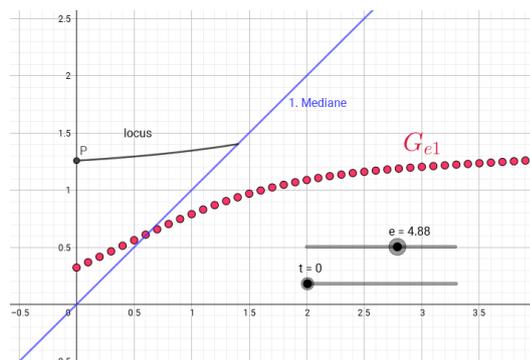


Abb.163 : Extremfall $e \approx 4.88$

Bei $e = 0$ ist der Fixpunkt 0 - das haben wir - da wir ihn als Startwert verwenden - sichergestellt. Werte für e größer e_{max} haben wir nicht zugelassen - sie verändern ja auch den Wasserspiegel nicht mehr, da ja ohnehin schon der gesamte Behälter unter Wasser ist!

Die Lipschitz-Bedingung des Banach'schen Fixpunktsatzes ist erfüllt, wenn die Ableitung kleiner als 1 ist. Mit $g(x) = x + e$ folgt:

$$\forall (x + e) \in [0, width] : \frac{d}{dx} G_e(x) = \frac{d}{dx} G(g(x)) = G'(g(x)) \cdot g'(x) = \left(\frac{f(x + e)}{R} \right)^2 < 1$$

da $\|f\| < R$ in $[0, width]$

Schließlich muss der Körper in den Zylinder hineinpassen!

15.1.5 Wir haben keinen Funktionsterm

Es ist unwahrscheinlich, dass wir von einem Körper die Umrissfunktion als Funktionsterm kennen - also bleibt messen.

Hier ist Einfallsreichtum gefragt, wie man zu seinen Messwerten kommt:

- Laserlängenmessgerät aus dem Baumarkt (auf Messgenauigkeit achten)
- Schattenwurf aus größerer Entfernung mit "punktförmiger" Lichtquelle
-

Wir machen es uns hier leicht - wir greifen von der "Originalfunktion" einzelne Datenpunkte ab!

Ausgangspunkt ist unsere Messwertliste der Länge m :

$$\{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$$

Zwei Verfahren sollen exemplarisch vorgeführt werden

Auffinden eines Ausgleichspolynoms

Wir suchen ein Polynom $p_n(x)$ vom Grad $n < m$, sodass die Summe der Fehlerquadrate ein Minimum wird:

$$\sum_{i=1}^m (p_n(x_i) - y_i)^2 \rightarrow \min$$

Da $m > n$ ist das Gleichungssystem für die Koeffizienten von p_n überbestimmt. Die Gleichungsmatrix A ist aber vom Rang n , da die Vandermonde-Matrix regulär ist.

Wie wir im Kapitel Householder-Transformation in 18.3 festgestellt haben, erfüllt eine Q - R -Zerlegung von A unsere Ausgleichsbedingung. Mit etwas experimentieren findet man heraus, dass für $m = 8$ und $n = 7$ bereits eine hervorragende Übereinstimmung mit der "Originalfunktion" erzielt wird.

Unser Gleichungssystem lautet

$$\forall i \in \{1 \dots m\} p_n(x_i) = y_i \Rightarrow \sum_{j=1}^{n+1} a_j \cdot (x_i)^{j-1} = y_i \Rightarrow \underbrace{\sum_{j=1}^{n+1} c_{ij} a_j}_{A \cdot \vec{x} = \vec{b}} = y_i \quad (15.3)$$

Lassen Sie sich nicht täuschen - die a_j sind die Variablen, die x_i sind bekannt und die Matrix $A = c_{ij} = (x_i)^{j-1}$ - würde sie quadratisch sein, heißt Vandermonde-Matrix. Bei uns ist A eine $(m \times n)$ -Matrix - hat also mehr Zeilen als Spalten(überbestimmt).



Beachte, dass in Formel 15.3 nicht wie üblich der erste Koeffizient mit a_0 beginnt - dies würde zu einer "null-ten"-Spalte der Matrix A führen!

Außerdem gilt: $c_{i1} = (x_i)^0 = 1$, d.h. die erste Spalte von A besteht aus lauter Einsen!

15. Rotationskörper im Wasser

Nun zum Programm:

Für die Q-R-Zerlegung brauchen wir das Lineare Algebra Package

```
(%i2) load(lapack)$ fpprintprec:3$
```

Anzahl der Messwerte $\rightarrow m$, Polynomgrad $\rightarrow n$

```
(%i5) width:2*%pi$ number_of_data:8$ approx_degree:7$
```

Unsere Umrissfunktion

```
(%i6) f(x) := float(exp(-0.2*x)* sin(x) + 1)$
```

Wir führen die "Messung" durch

```
(%i7) pointList:makelist([i,f(i)],i,0,width,width/number_of_data)$
```

```
(%i8) getXY(L,func):=map(lambda([x],func(x)),L)$
```

Wir holen die $y_i \rightarrow \vec{b}$

```
(%i9) b:getXY(pointList,second)$
```

Eine Zeile der Vandermonde-Matrix wird mit dem Wert x_i generiert

```
(%i10) vm_row(x,deg):=block([l:[1],p:1], for i thru deg do ( p:p*x, l:cons(p,l)), reverse(l))$
```

Jetzt wird die Matrix A erzeugt, initialisiert mit einer "dummy" Zeile - sie wird zum Schluss gelöscht

```
(%i11) vm_matrix(pL,deg):=block([M:matrix(makelist(i,i,1,deg+1)), rows:length(pL)],
    for j thru rows do M:addrow(M,vm_row(getXY(pL,first)[j],deg)),
    M:submatrix(1,M)
)$
```

```
(%i12) A:vm_matrix(pointList,approx_degree)$
```

Q-R-Zerlegung wird durchgeführt - fragen Sie mich nicht, was *dgeqrf* heißt

```
(%i13) [q,r]:dgeqrf(A)$
```

$QR\vec{x} = \vec{b} \rightarrow Q^T QR\vec{x} = Q^T \vec{b} \rightarrow R\vec{x} = Q^T \vec{b}$

```
(%i14) b_T:transpose(q) . b$
```

Die Berechnung des Lösungsvektors \vec{x} durch "Rückeinsetzen" kann man bei Initialisierung als Nullvektor ($x:zeromatrix(cols,1)$) die Summe als Skalarprodukt der c -ten Matrixzeile mit dem bisherigen Lösungsvektor berechnen.

$$x_c = \left(b_c - \underbrace{\sum_{k=c-1}^n r_{ck} \cdot x_k}_{\vec{r}_c \cdot \vec{x}} \right) / r_{cc}$$

```
(%i15) backwardSubstitution(r,b):=block([cols:second(matrix_size(r)), x],
    x:zeromatrix(cols,1),
    for c:cols thru 1 step -1 do x[c]:((b[c]-row(r,c) . x)/r[c,c]),
    x)$
```

```
(%i16) a:backwardSubstitution(r,b_T)$
```

Berechnung der Näherung; a ist eine Matrix und braucht daher Zeilen- und Spaltenindex!

```
(%i17) define(p(x),sum(a[i][1]*x^(i-1),i,1,length(a)))$
```

Ergebnis des Plots siehe Abbildung unten

```
(%i18) plot2d([[discrete,pointList],f(x), p(x)], [x,0,width],
  [legend, " data values"," original function", " approximation poly"],
  [style,[points,4,1,1], [lines,5,2,2], [lines,2,3,2] ],
  [gnuplot_preamble, "set key top right"])$
```

Wir schauen uns das Näherungspolynom als Term an

```
(%i19) p(x);
```

$$2.7410^{-5} x^7 - 7.310^{-5} x^6 - 0.00704x^5 + 0.0757x^4 - 0.227x^3 - 0.128x^2 + 0.976x + 1.0 \quad (\%o19)$$

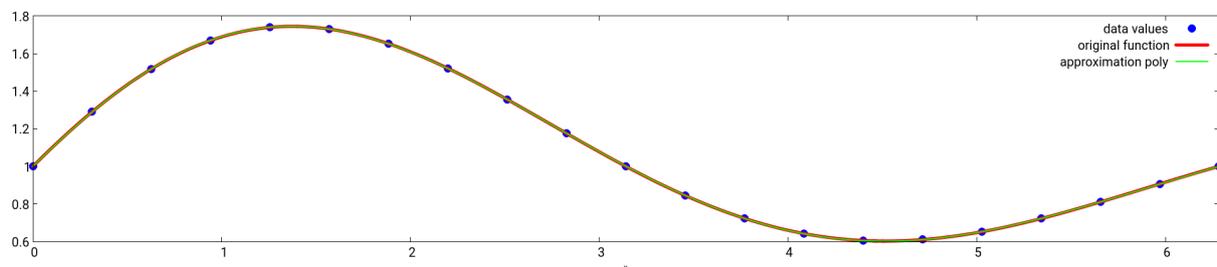


Abb.164 : Approximationspolynom vs. Originalfunktion



Man sollte m (Anzahl der Messungen) groß und n (Grad des Polynoms) so klein wie möglich machen, um dem RUNGE-Phänomen (Oszillationen am Rand) auszuweichen! Also gleich mit großem n "darüberzufahren" ist keine gute Idee!

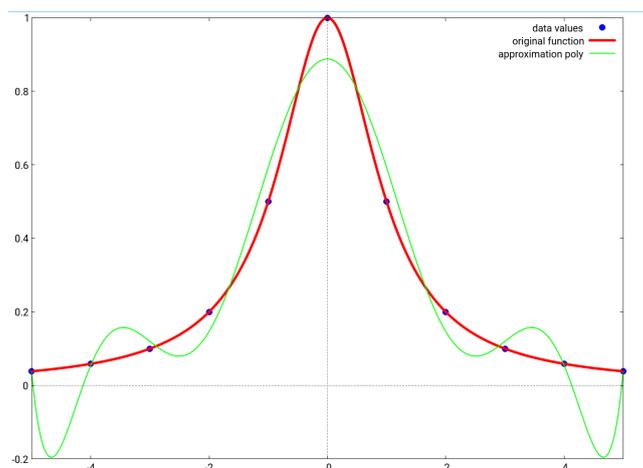


Abb.165 : RUNGE-Phänomen $f(x) := \frac{1}{1+x^2}$, $m = 11$, $n = 8$

Kubische Splines

Dieses Verfahren umgeht die Oszillationen (RUNGE-Phänomen), da es nur mit kubischen Polynomen arbeitet. Wir benützen hier nur das Javascript-Modul für Splines - nähere Ausführung zu der Theorie befindet sich im Kapitel 16!

Unser Ausgangspunkt ist wieder die Gleichung 15.2, also

$$w = \int_0^{e+w} \left(\frac{f}{R}\right)^2 \quad z := e + w \quad \Leftrightarrow \quad \underbrace{\int_0^z \left(\frac{f}{R}\right)^2}_{F(z)} = z - e =: h_e(z) \Leftrightarrow F(z) = h_e(z)$$

Wir benötigen eine zweidimensionale Punktliste *pointL* als Input. Also sieht unsere Vorgangsweise so aus:

- Punktliste der Umrissfunktion *pointL*; $f_h(x) = e^{-0.2x} \sin(x) + 1$
`pointL = Sequence((6.28/10*i, f_h(6.28*i/10)), i, 0, 10)`
- Spline-Funktion mit Button erzeugen und umbenennen in *sp1*; *R* festlegen
- Datenpunkte von $\int_0^x \left(\frac{sp1}{R}\right)^2$ erzeugen
`pointL = Sequence((6.28/10*i, NIntegral((sp_1(x)/R)^2, 0, 6.28/10*i)), i, 0, 10)`
- Spline davon anfertigen

- e_{max} berechnen und Schieberegler e für die Eintauchtiefe erzeugen

$$e_{\max} = 6.28 - NIntegral((sp_1(x)/R)^2, 0, 6.28)$$

- Hilfsfunktion $h_e(x) = x - e$

- Schnittpunkt h_e mit Spline f_s berechnen; hier ist zu beachten, dass momentan (April 2018) *Geogebra* nur innere Schnittpunkte zwischen Funktionen berechnet, d.h. für die Randpunkte muss man es "händisch" erledigen:

$$S = \text{If}(e==0, (0,0), e==e_{\max}, \text{Element}(\text{pointL}, \text{length}(\text{pointL})), \text{Intersect}(h_e, f_s, a, b))$$

$x \in [a, b]$ ist das Intervall, in dem eindeutigen Schnittpunkt gesucht wird - in unserem Fall $a = 0$ und $b = 6.28$

- Schließlich noch die Zuordnung $e \rightarrow \text{waterLevel}$:

$$W = (e, y(S))$$

Schaltet man die Spur von W ein bekommt man den Graph!

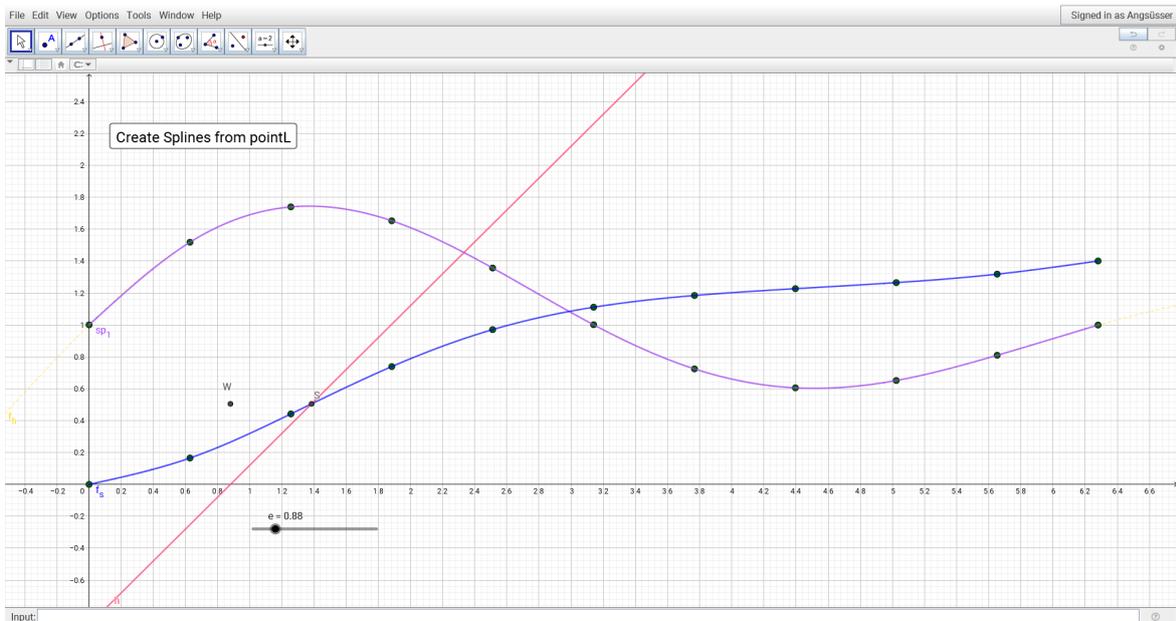


Abb.166 : Zweimal mit Spline-Funktionen approximiert



Man beachte, dass hier die Spline-Funktion sp_1 integriert wurde - dies ist meines Wissens mit dem in *Geogebra* eingebauten *Spline*-Befehl nicht möglich, da die Spline-Funktion dort in Parameterform vorliegt!

16 | Spline-Interpolation

Hier beziehe ich mich auf die Lecture Notes von Ruye Wang - im Speziellen auf seine Ausführungen über Splines in <http://fourier.eng.hmc.edu/e176/lectures/ch7/node6.html>.

Dieser Abschnitt ist in weiten Teilen nur eine Übersetzung seiner Ausführungen! Hinzugefügt von mir wurde der Tridiagonalmatrix-Algorithmus und die Implementation des kubischen Spline-Verfahrens in Javascript für *Geogebra!*

Definition 16.1 Splinefunktion vom Grad m

Eine reelle Funktion f sei an $(n + 1)$ Stellen bekannt: $f(x_i) = y_i$, $i \in \{0, 1, \dots, n\}$
Wir bestimmen n Polynome P_i vom Grad $m \ll n$, die mit f an den Stützstellen übereinstimmen und dort glatt ineinander übergehen (Ableitungen stimmen überein):

$$S(x) = \begin{cases} P_1(x) & x_0 \leq x \leq x_1 \\ \vdots & \vdots \\ P_i(x) & x_{i-1} \leq x \leq x_i \\ \vdots & \vdots \\ P_n(x) & x_{n-1} \leq x \leq x_n \end{cases} \quad (16.1)$$

Es muss also gelten

$$(1) \quad P_i(x_i) = P_{i+1}(x_i) = f(x_i) = y_i, \quad i \in \{1, \dots, (n-1)\} \quad (16.2)$$

$$(2) \quad P_1(x_0) = y_0 \quad \wedge \quad P_n(x_n) = y_n \quad (16.3)$$

$$(3) \quad P_i^{(k)}(x_i) = P_{i+1}^{(k)}(x_i), \quad i \in \{1, \dots, (n-1)\}, \quad k \in \{0, \dots, k_m\} \quad (16.4)$$

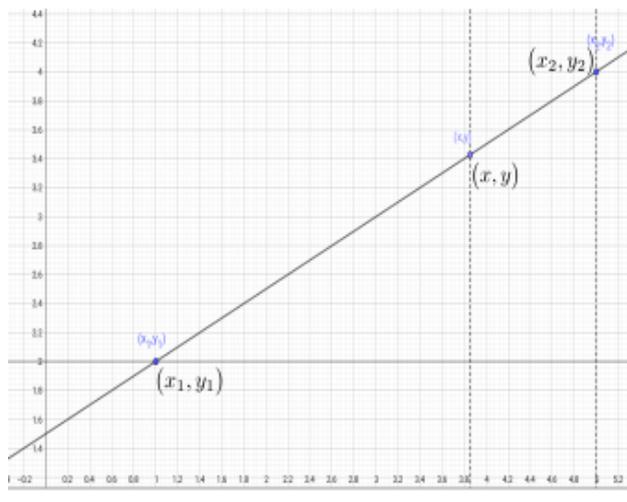
Die maximale Übereinstimmung der Ableitungen k_m sollte dabei so groß wie möglich sein und hängt natürlich von m - dem Grad der Teilpolynome ab.

S heißt dann die Splinefunktion von f vom Grad m .

16.1 Kubische Splines

Wir beschäftigen uns hier nur mit kubischen Splines - also $m = 3$

Zuerst leiten wir die hier verwendete lineare Interpolationsformel zwischen 2 Punkten her:



$$\begin{aligned} \frac{y - y_1}{x - x_1} &= \frac{y_2 - y_1}{x_2 - x_1} \Rightarrow \\ y &= \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1 = \\ &= \frac{y_2 - y_1}{x_2 - x_1}x + \frac{y_2 - y_1}{x_2 - x_1}(-x_1) + y_1 \frac{x_2 - x_1}{x_2 - x_1} = \\ &= \frac{1}{\underbrace{x_2 - x_1}_{h_1}} = (y_2x - y_1x - x_1y_2 + y_1x_2) \Rightarrow \\ & \boxed{y = \frac{1}{h_1} \left((x - x_1)y_2 + (x_2 - x)y_1 \right)} \quad (16.5) \end{aligned}$$

Abb.167 : Lineare Interpolation

Wir suchen also für $C_i(x) := P_i(x) = a_i x^3 + b_i x^2 + c_i x + d_i$ die 4 Parameter a_i , b_i , c_i und d_i . Die Bedingungen 16.2 bis 16.4 werden jetzt zu

$$C_i(x_i) = y_i, \quad C_i(x_{i-1}) = y_{i-1}, \quad C_i'(x_i) = C_{i+1}'(x_i) \quad \text{und} \quad C_i''(x_i) = C_{i+1}''(x_i)$$

$C_i''(x) = 6a_i x + 2b_i$ ist dann eine lineare Funktion. Die unbekannt Krümmungen dieser Polynome seien

$$C_i''(x_{i-1}) =: M_{i-1} \quad \text{und} \quad C_i''(x_i) =: M_i$$

Mit 16.5 können die $C_i''(x)$ geschrieben werden als

$$C_i''(x) = \frac{x_i - x}{h_i} M_{i-1} + \frac{x - x_{i-1}}{h_i} M_i \quad h_i := x_i - x_{i-1} \quad (16.6)$$

Wir integrieren 2 mal und erhalten

$$C_i(x) = \int \left(\int C_i''(x) dx \right) dx = \frac{(x_i - x)^3}{6h_i} M_{i-1} + \frac{(x - x_{i-1})^3}{6h_i} M_i + c_i x + d_i \quad (16.7)$$

Mit $C_i(x_{i-1}) = y_{i-1}$, $C_i(x_i) = y_i$ und $h_i := x_i - x_{i-1}$ wird daraus

$$C_i(x_{i-1}) = \frac{h_i^2}{6} M_{i-1} + c_i x_{i-1} + d_i = y_{i-1}, \quad C_i(x_i) = \frac{h_i^2}{6} M_i + c_i x_i + d_i = y_i \quad (16.8)$$

Wir lösen diese beiden Gleichungen für c_i und d_i und erhalten

$$c_i = \frac{y_i - y_{i-1}}{h_i} - \frac{h_i}{6}(M_i - M_{i-1}) \quad (16.9)$$

$$d_i = \frac{x_i y_{i-1} - x_{i-1} y_i}{h_i} - \frac{h_i}{6}(x_i M_{i-1} - x_{i-1} M_i) \quad (16.10)$$

Wir setzen 16.9 und 16.10 wieder in 16.7 ein und fassen zusammen

$$\begin{aligned} C_i(x) &= \frac{(x_i - x)^3}{6h_i} M_{i-1} + \frac{(x - x_{i-1})^3}{6h_i} M_i + \left(\frac{y_i - y_{i-1}}{h_i} - \frac{h_i}{6}(M_i - M_{i-1}) \right) x \\ &\quad + \frac{x_i y_{i-1} - x_{i-1} y_i}{h_i} - \frac{h_i}{6}(x_i M_{i-1} - x_{i-1} M_i) \\ &= \frac{(x_i - x)^3}{6h_i} M_{i-1} + \frac{(x - x_{i-1})^3}{6h_i} M_i + \left(\frac{y_{i-1}}{h_i} - \frac{M_{i-1} h_i}{6} \right) (x_i - x) + \left(\frac{y_i}{h_i} - \frac{M_i h_i}{6} \right) (x - x_{i-1}) \end{aligned}$$

also haben wir jetzt einen Ausdruck für die $C_i(x)$ in Abhängigkeit von den M_i :

$$\begin{aligned} C_i(x) &= \frac{(x_i - x)^3}{6h_i} M_{i-1} + \frac{(x - x_{i-1})^3}{6h_i} M_i + \left(\frac{y_{i-1}}{h_i} - \frac{M_{i-1} h_i}{6} \right) (x_i - x) + \\ &\quad + \left(\frac{y_i}{h_i} - \frac{M_i h_i}{6} \right) (x - x_{i-1}) \quad (16.11) \end{aligned}$$

Um in 16.11 M_i , $i \in \{1, 2, \dots, (n-1)\}$ zu bestimmen, verwenden wir die Bedingung, dass an den "Schnittstellen" der C_i die Ableitungen übereinstimmen müssen - $C'_i(x_i) = C'_{i+1}(x_i)$.

Wir leiten 16.11 ab und erhalten

$$C'_i(x) = -\frac{(x_i - x)^2}{2h_i} M_{i-1} + \frac{(x - x_{i-1})^2}{2h_i} M_i - \frac{1}{h_i} \left(y_{i-1} - \frac{M_{i-1} h_i^2}{6} \right) + \frac{1}{h_i} \left(y_i - \frac{M_i h_i^2}{6} \right) \quad (16.12)$$

$$= -\frac{(x_i - x)^2}{2h_i} M_{i-1} + \frac{(x - x_{i-1})^2}{2h_i} M_i + \frac{y_i - y_{i-1}}{h_i} - \frac{h_i}{6}(M_i - M_{i-1}) \quad (16.13)$$

Mit der mittleren Steigung \bar{k}_f von f zwischen 2 Stützstellen x_{i-1} und x_i

$$\bar{k}_f[x_{i-1}, x_i] := \frac{y_i - y_{i-1}}{h_i} \quad \text{ergibt sich dann}$$

$$\begin{aligned} C'_i(x_i) &= \frac{h_i}{3} M_i + \frac{y_i - y_{i-1}}{h_i} + \frac{h_i}{6} M_{i-1} = \frac{h_i}{6}(2M_i + M_{i-1}) + \bar{k}_f[x_{i-1}, x_i] \\ C'_i(x_{i-1}) &= -\frac{h_i}{3} M_{i-1} + \frac{y_i - y_{i-1}}{h_i} - \frac{h_i}{6} M_i = -\frac{h_i}{6}(2M_{i-1} - M_i) + \bar{k}_f[x_{i-1}, x_i] \quad (16.14) \end{aligned}$$

16. Spline-Interpolation

Wenn wir in 16.14 i um 1 erhöhen, erhalten wir $C'_{i+1}(x_i)$:

$$C'_{i+1}(x_i) = -\frac{h_{i+1}}{3}M_i - \frac{h_{i+1}}{6}M_{i+1} + \bar{k}_f[x_i, x_{i+1}] \quad (16.15)$$

Durch Gleichsetzen mit 16.12 erhalten wir

$$\begin{aligned} \frac{h_i}{3}M_i + \bar{k}_f[x_{i-1}, x_i] + \frac{h_i}{6}M_{i-1} &= -\frac{h_{i+1}}{3}M_i + \bar{k}_f[x_i, x_{i+1}] - \frac{h_{i+1}}{6}M_{i+1} \Big| \cdot 6 \\ h_i M_{i-1} + 2(h_{i+1} + h_i)M_i + h_{i+1} M_{i+1} &= 6(\bar{k}_f[x_i, x_{i+1}] - \bar{k}_f[x_{i-1}, x_i]) \end{aligned} \quad (16.16)$$

Mit $\frac{1}{x_{i+1} - x_{i-1}} (\bar{k}_f[x_i, x_{i+1}] - \bar{k}_f[x_i, x_{i-1}]) = \frac{\bar{k}_f[x_i, x_{i+1}] - \bar{k}_f[x_i, x_{i-1}]}{h_{i+1} + h_i} := \bar{c}_f[x_{i-1}, x_{i+1}]$

als mittlere Krümmung (Mittelwert der mittleren Steigungen) von f in $[x_{i-1}, x_{i+1}]$ lässt sich 16.16 schreiben

$$\begin{aligned} h_i M_{i-1} + 2(h_{i+1} + h_i)M_i + h_{i+1} M_{i+1} &= 6 \bar{c}_f[x_{i-1}, x_{i+1}] (h_{i+1} + h_i) \Big| : (h_{i+1} + h_i) \\ \frac{h_i}{h_{i+1} + h_i} M_{i-1} + 2M_i + \frac{h_{i+1}}{h_{i+1} + h_i} M_{i+1} &= \underbrace{6 \bar{c}_f[x_{i-1}, x_{i+1}]}_{d_i} \end{aligned}$$

Wir landen schließlich bei

$$\begin{aligned} \mu_i M_{i-1} + 2M_i + \lambda_i M_{i+1} &= d_i, \quad (i = 1, \dots, n-1) \\ \mu_i &= \frac{h_i}{h_{i+1} + h_i}, \quad \lambda_i = \frac{h_{i+1}}{h_{i+1} + h_i} = 1 - \mu_i, \quad d_i = 6 \bar{c}_f[x_{i-1}, x_{i+1}] \end{aligned} \quad (16.17)$$

Wir haben also ein Gleichungssystem mit $(n-1)$ Gleichungen und $(n+1)$ Variable M_0, M_1, \dots, M_n . Wir benötigen also 2 zusätzliche Bedingungen. Wir konzentrieren uns hier auf die "natürliche Randwertbedingung": $M_0 = 0 \quad \wedge \quad M_n = 0$

Damit wird 16.17 zu folgendem tridiagonalen Gleichungssystem:

$$\begin{pmatrix} 2 & \lambda_1 & 0 & 0 & \dots & 0 \\ \mu_2 & 2 & \lambda_2 & 0 & \dots & 0 \\ 0 & \mu_3 & 2 & \lambda_3 & 0 & 0 \\ \vdots & 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & 0 & \mu_{n-2} & 2 & \lambda_{n-2} \\ 0 & \dots & 0 & 0 & \mu_{n-1} & 2 \end{pmatrix} \begin{pmatrix} M_1 \\ M_2 \\ \vdots \\ \vdots \\ M_{n-2} \\ M_{n-1} \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ \vdots \\ d_{n-2} \\ d_{n-1} \end{pmatrix} \quad (16.18)$$

Der Lösung dieses Problems widmen wir ein eigenes Unterkapitel!

16.2 Tridiagonal-Matrix Algorithmus(TDMA)

Definition 16.2 Tridiagonales Gleichungssystem

Ein Gleichungssystem heißt *tridiagonal*, wenn es wie folgt strukturiert ist:

$$\begin{pmatrix} b_1 & c_1 & 0 & 0 & \dots & 0 \\ a_2 & b_2 & c_2 & 0 & \dots & 0 \\ 0 & a_3 & b_3 & c_3 & 0 & 0 \\ \vdots & 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \vdots & 0 & a_{N-1} & b_{N-1} & c_{N-1} \\ 0 & \dots & 0 & 0 & a_N & b_N \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_{N-1} \\ x_N \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ \vdots \\ d_{N-1} \\ d_N \end{pmatrix} \quad (16.19)$$

bzw.

$$\begin{aligned} a_i x_{i-1} + b_i x_i + c_i x_{i+1} &= d_i \\ a_i, b_i, c_i, d_i &\in \mathbb{R}, i \in \{1, \dots, n\}, a_1 = 0 \wedge c_n = 0 \end{aligned} \quad (16.20)$$

Schauen wir uns so ein Gleichungssystem näher an. Die erste Gleichung lautet

$$b_1 x_1 + c_1 x_2 = d_1 \Rightarrow x_1 = P_1 x_2 + Q_1 \quad \text{mit } P_1 \left(= -\frac{c_1}{b_1} \right), Q_1 \left(= \frac{d_1}{b_1} \right) \in \mathbb{R}$$

Halten wir fest: x_1 hängt linear von x_2 ab: $x_1 = x_1(x_2)_\ell$

Die zweite Gleichung lautet

$$a_2 x_1 + b_2 x_2 + c_2 x_3 = d_2 \Rightarrow a_2 x_1(x_2)_\ell + b_2 x_2 + c_2 x_3 = d_2 \Rightarrow x_2 = x_2(x_3)_\ell$$

Der letzte Schluss folgt aus der Abgeschlossenheit der linearen Funktionen. Wir können dies auch wieder schreiben als

$$x_2 = P_2 x_3 + Q_2$$

Allgemein können wir formulieren

$$\boxed{x_i = P_i x_{i+1} + Q_i \Leftrightarrow x_{i-1} = P_{i-1} x_i + Q_{i-1}} \quad (16.21)$$

Die letzte Gleichung lautet

$$a_N x_{N-1} + b_N x_N = d_N \quad \text{jetzt gilt aber } x_{N-1} = x_{N-1}(x_N)_\ell$$

damit ist diese Gleichung für x_N lösbar, mit Kenntnis von $x_{N-1} = x_{N-1}(x_N)_\ell$ bekommen wir x_{N-1} , und so weiter bis x_1 . Es gilt also die P_i bzw. Q_i zu bestimmen - dazu setzen wir den

16. Spline-Interpolation

rechten Teil von 16.21 in 16.20 ein:

$$a_i (P_{i-1} x_i + Q_{i-1}) + b_i x_i + c_i x_{i+1} = d_i \quad \Rightarrow$$

$$x_i = -\frac{c_i}{b_i + a_i P_{i-1}} x_{i+1} + \frac{d_i - a_i Q_{i-1}}{b_i + a_i P_{i-1}} \quad (16.22)$$

Vergleichen wir 16.22 mit 16.21 (linker Teil) ergeben sich die Rekursionsformeln für P und Q

$$P_i = -\frac{c_i}{b_i + a_i P_{i-1}} \quad Q_i = \frac{d_i - a_i Q_{i-1}}{b_i + a_i P_{i-1}} \quad (16.23)$$

Jetzt liegt der TDMA (**Thomas Algorithmus**) vor uns

- INPUT: Felder a , b , c , und d
- $P_1 = -\frac{c_1}{b_1} \quad Q_1 = \frac{d_1}{b_1}$
- Berechnung der nächsten P_i und Q_i mit 16.23
- $x_N = Q_N$, da $P_N = 0$ wegen $c_N = 0$ und eingesetzt in 16.21
- Berechnung der weiteren Lösungen mit 16.21
- OutPut: Lösungsvektor

```

1 function TDMA(a, b, c, d){
2 // fields a,b,c,d start at Index zero!!
3 // so the formula in the text above must be adapted
4 var P=[], Q=[], u=[], denom, n=a.length;
5
6 P[0]=0; Q[0]=0; // special case j=0 yields correct result
7 for (j=0;j<n; j++) {
8     denom = b[j]+a[j]*P[j];
9     P[j+1]= - c[j]/denom;
10    Q[j+1]= (d[j]-a[j]*Q[j])/denom;
11 }
12
13 u[n]=Q[n];
14 for (i=n-1; i >0; i--) u[i]=P[i]*u[i+1]+Q[i];
15 // u[0] is "undefined"; u[1] ... u[n]
16 return u
17 }

```

Änderungen im Code: $a_i := \mu_i$, $b_i := 2$, $c_i := \lambda_i$, $d_i := 6 \bar{c}_f[x_{i-1}, x_{i+1}]$ starten bei Index 0! Dadurch dekrementieren sich deren Indices in 16.23 um 1 (Zeilen 9 und 10):



$$P_i = -\frac{c_{i-1}}{b_{i-1} + a_{i-1} P_{i-1}} \quad Q_i = \frac{d_{i-1} - a_{i-1} Q_{i-1}}{b_{i-1} + a_{i-1} P_{i-1}}$$

16.3 Implementierung in *Geogebra (Javascript)*

`importPointList` in Zeile 4 erstellt aus der *Geogebra*-Punktliste einen Javascript-Array
`buildSplineSum` in Zeile 10 erstellt aus den Funktionszweigen den *Geogebra*-Befehlsstring und
 ist im Wesentlichen die Abbildung der Formeln 16.11 - anders als dort wurde die Indizierung
 der C_i generell (für Javascript typisch) bei Null begonnen (siehe Zeile 28), auch das h -Feld
 wird jetzt bei Index 0 gestartet (siehe Schleife Zeile 28)!

```

1  /***** MAIN *****/
const getX = 0, getY = 1;
3  var p_x=[], p_y=[], h=[], mu=[], lambda=[], M=[], cmdStr = "f_s(x)=";
  var pL=importPointList ();
5
  p_x=getCoord(getX,pL); p_y=getCoord(getY,pL);
7  h=getDelta( p_x);
  setMuLambda(h); // mu, lambda are set
9  M = TDMA(mu, lambda, getConstVec(h, p_y));
  cmdStr += buildSplineSum(M, h, p_x, p_y);
11 ggbApplet.evalCommand(cmdStr);
  /***** END-MAIN *****/
13
function importPointList () {
15   var pL = "ggbApplet.getValueString("pointL");
    pL=pL.split("=")[1];
17   pL=pL.replace(/{/g, "[");
    pL=pL.replace(/}/g, "]");
19   pL=pL.replace(/\\(/g, "[");
    pL=pL.replace(/\\)/g, "]");
21   return eval(pL);
  }
23
function buildSplineSum(M,h,p_x,p_y){
25   var splineStr="If(";
    var branches=[], N=[]; //M/6
27   for (var j=0; j<M.length; j++) N[j]=M[j]/6;
    for ( var k=0; k<M.length-1; k++) branches.push(""+p_x[k]+ "<=x<=" +p_x[k+1]+",
      "+ buildFuncTerm(N, k,h,p_x,p_y));
29   return splineStr+branches.join(",")+";";
  }
31 // a_s ... d_s are the coefficients in each spline-branch
function buildFuncTerm(N, i, h,p_x,p_y){
33   var a_s, b_s, c_s, d_s, f="(";
    a_s = N[i]/h[i];
35   b_s = N[i+1]/h[i];
    c_s = p_y[i]/h[i]-N[i]*h[i];
37   d_s = p_y[i+1]/h[i]-N[i+1]*h[i];
    f += a_s + ")*(x + p_x[i+1] + "-x)^3 + (";
39   f += b_s + ")*(x- + p_x[i] + ")^3 + (";
    f += c_s + ")*(x + p_x[i+1] + "-x) + (";
41   f += d_s + ")*(x- + p_x[i] + ")";
    return f;
43 }

```

16. Spline-Interpolation

Hier die Funktionen für den eigentlichen Algorithmus um 16.17 für M_1, \dots, M_{N-1} zu lösen:

```
46 function getConstVec(hList, yList){
47     var m = [], k = [], n = yList.length - 1 ;
48     for (j=0; j<n; j++) m[j]=(yList[j+1]-yList[j])/hList[j];
49     for (j=0; j<n-1; j++) k[j]=6*(m[j+1] - m[j])/(hList[j+1] + hList[j] );
50     return k;
51 }
52
53 function getCoord(xOrY, pL){
54     var val=[];
55     for (i=0; i<pL.length; i++) val[i]=pL[i][xOrY];
56     return val;
57 }
58
59 function getDelta(list){
60     var r=[];
61     for (i=1; i<list.length; i++) r[i-1]=list[i]-list[i-1];
62     return r;
63 }
64
65 function setMuLambda(list){
66     var m=[], e=list.length-1;
67     for (var i=0; i < e; i++) {
68         mu[i]=list[i]/(list[i+1]+list[i]);
69         lambda[i]=1-mu[i];
70     }
71     mu[0]=0; lambda[e-1]=0; // first mu amd last lambda are reset
72 }
73
74 // TriDiagonalMatrixAlgorithm
75 function TDMA(a, c, d){
76     var P=[], Q=[], u=[], denom, n=a.length;
77     // for (i = 0; i < n; i++) b[i]=2; //diagonal is set
78
79     P[0]=0; Q[0]=0; // special case i=0 yields correct result
80     for (j=0; j<n; j++) {
81         denom = 2+a[j]*P[j]; // b[j]=2
82         P[j+1]= - c[j]/denom;
83         Q[j+1]= (d[j]-a[j]*Q[j])/denom;
84     }
85     u[n]=Q[n];
86     for (i=n-1; i >0; i--) u[i]=P[i]*u[i+1]+Q[i];
87     u[0]=0; u[n+1]=0; //M[0]=0 and M[n]=0
88     return u
89 }
90 }
```

TDMA benötigt als Parameter keine Diagonalelemente und vor der Rückgabe wird der Lösungsvektor um die natürliche Lösung ergänzt.

`getConstVec` berechnet das 6-fache der mittleren Krümmungen.

Der Zweck aller anderer Funktionen ist hoffentlich selbsterklärend.

Signed in as Angluster

File Edit View Options Tools Window Help

Algebra

Function

$f_1(x) = \begin{cases} 0 & 0.2 \leq x \leq 0.2 \\ -0.0557(2.2-x)^3 + 0.1101(x-0.2)^3 + 0.2729(2.2-x) + 0.0596(x-0.2) & 0.2 \leq x \leq 2.2 \\ 0.2202(3.2-x)^3 - 0.5482(x-2.2)^3 + 0.7798(3.2-x) + 2.5482(x-2.2) & 2.2 \leq x \leq 3.2 \\ -0.7831(3.9-x)^3 - 0.1442(x-3.2)^3 + 3.2409(3.9-x) + 2.2135(x-3.2) & 3.2 \leq x \leq 3.9 \\ -0.1121(4.8-x)^3 + 1.6171(x-3.9)^3 + 1.7575(4.8-x) + 0.2457(x-3.9) & 3.9 \leq x \leq 4.8 \\ 7.2772(5-x)^3 + 0(x-4.8)^3 + 6.7089(5-x) + 10(x-4.8) & 4.8 \leq x \leq 5 \end{cases}$

List

- pointL = {(0, 0), (0.2, 0.1), (2.2, 2), (3.9, 1.5), (4.8, 1.4), (5, 2)}
- Number
- a = 4.4085

Graphics

Create Splines from pointL

Input:

16. Spline-Interpolation

Auf der vorigen Seite sieht man die Ausgabe angewandt auf die Test-Punktliste

`pointL = {(0, 0), (0.2, 0.1), (2.2, 1), (3.2, 2), (3.9, 1.5), (4.8, 1.4), (5, 2)}`

Das Javascript-Programm wird dabei in *Geogebra* hinter der Click-Methode des Buttons *Create Splines For pointL* “versteckt”.

Außerdem wurde das Integral $\int_{0.5}^{4.5} f_s(x) dx = 4.4085$ mit `Integral(f_s, 0.5, 4.5)` und die

Ableitungsfunktion $f'_s(x)$ mit `Derivative(f_s)` berechnet.

Beides wäre mit der “eingebauten” *Spline*-Funktion nicht möglich gewesen.

16.4 Gegencheck mit *wxMaxima* (native invert)

Wir halten uns an 16.18, statt *TDMA* benutzen wir allerdings einfach die Matrixinversion!

Keine Kovertierungswarnungen und höchstens 5 Ziffern Genauigkeit ausgeben

```
(%i2) ratprint:false$ fpprintprec: 5$
```

Wir kopieren die Punktliste aus *Geogebra* und passen die Syntax an

```
(%i3) pL:[[0, 0], [0.2, 0.1], [2.2, 1], [3.2,2],[3.9,1.5], [4.8, 1.4],[5,2]]$
```

Liste der x- und y-Werte

```
(%i4) xList:makelist(first(pL[i]),i,1,length(pL));          [0, 0.2, 2.2, 3.2, 3.9, 4.8, 5]          (xList)
```

```
(%i5) yList:makelist(second(pL[i]),i,1,length(pL));          [0, 0.1, 1, 2, 1.5, 1.4, 2]          (yList)
```

Start bei Index 0, sodass wir die Formeln aus dem Text verwenden können - $p_x[0] \dots p_x[6]$

```
(%i6) p_x[i]:=xList[i+1]$          (%i7) p_y[i]:=yList[i+1]$
```

Die Stützstellen - hier wäre auch die auskommentierte Version denkbar

```
(%i8) /*xLimits:append([-inf],makelist(xList[i],i,2,length(xList)-1),[inf])*/  
      xLimits:append([first(xList)],makelist(xList[i],i,2,length(xList)-1),[last(xList)]);  
                                          [0, 0.2, 2.2, 3.2, 3.9, 4.8, 5]          (xLimits)
```

Jetzt kommen die Zwischenergebnisse um den Konstantenvektor zu bestimmen

```
(%i9) h:makelist(xList[i+1]-xList[i],i,1,length(xList)-1);          [0.2, 2.0, 1.0, 0.7, 0.9, 0.2]          (h)
```

```
(%i10) hy:makelist(yList[i+1]-yList[i],i,1,length(yList)-1);          [0.1, 0.9, 1, -0.5, -0.1, 0.6]          (hy)
```

```
(%i11) hyOverH:makelist(hy[i]/h[i],i,1,length(h));          [0.5, 0.45, 1.0, -0.71429, -0.11111, 3.0]          (hyOverH)
```

```
(%i12) deltaHyOverH:makelist(hyOverH[i+1] - hyOverH[i],i,1,length(hyOverH)-1);  
                                          [-0.05, 0.55, -1.7143, 0.60317, 3.1111]          (deltaHyOverH)
```

Der Konstantenvektor \vec{d} ist jetzt bestimmt - er hat 5 Komponenten für $M_1 \dots M_5$

```
(%i13) dV:makelist(6*1/(h[i+1]+h[i])*deltaHyOverH[i],i,1,length(deltaHyOverH));  
                                          [-0.13636, 1.1, -6.0504, 2.2619, 16.97]          (dV)
```

Wie benötigen ihn als Spaltenvektor

```
(%i14) dVec:transpose(matrix(dV))$
```

$\vec{\mu}$ und $\vec{\lambda}$ werden berechnet, von den 5 Komponenten werden jeweils 4 gebraucht

```
(%i15) mu:makelist(h[i]/(h[i+1]+h[i]),i,1,length(h)-1); [0.090909,0.66667,0.58824,0.4375,0.81818] (mu)
```

```
(%i16) makelist(1-mu[i],i,1,length(pL)-2); [0.90909,0.33333,0.41176,0.5625,0.18182] (%o16)
```

Die Gleichungsmatrix wird erstellt - siehe 16.18 im Text

```
(%i17) setMatrix():=block([dim:length(pL)-2,A],
  A:diagmatrix(dim,2),
  for i thru dim do
    for j thru dim do block(
      if i-j=1 then A[i,j]:mu[i],
      if j-i=1 then A[i,j]:1-mu[i]
    ),
  A
)$
```

```
(%i18) A:setMatrix();
```

$$\begin{pmatrix} 2 & 0.90909 & 0 & 0 & 0 \\ 0.66667 & 2 & 0.33333 & 0 & 0 \\ 0 & 0.58824 & 2 & 0.41176 & 0 \\ 0 & 0 & 0.4375 & 2 & 0.5625 \\ 0 & 0 & 0 & 0.81818 & 2 \end{pmatrix} \quad (A)$$

```
(%i19) Die Lösungen für  $M_1 \dots M_5$  - statt TDMA verwenden wir die
Matrixinversion
M:invert(A).dVec;
```

$$\begin{pmatrix} -0.66867 \\ 1.3211 \\ -3.2891 \\ -0.6056 \\ 8.7326 \end{pmatrix} \quad (M)$$

Wir ergänzen M_0 und M_6 und nennen dieses Feld m ; $length(pL) = 7$

```
(%i21) m[0]:0$ m[length(pL)-1]:0$
```

Zugriff auf M wird kodiert (M ist eine Matrix - für ein Feld wird nur die erste Komponente benötigt)

```
(%i22) m[i]:=first(M[i])$
```

```
(%i23) makelist(m[i],i,1,length(M)); [-0.66867,1.3211,-3.2891,-0.6056,8.7326] (%o23)
```

Die Koeffizienten für die C_i werden erstellt: a_s, b_s, c_s, d_s

```
(%i24) a_s:makelist(m[i-1]/(6*h[i]),i,1,length(h)); [0.0,-0.055722,0.22018,-0.78311,-0.11215,7.2772] (a_s)
```

```
(%i25) b_s:makelist(m[i]/(6*h[i]),i,1,length(h)); [-0.55722,0.11009,-0.54818,-0.14419,1.6171,0.0] (b_s)
```

```
c_s:makelist(p_y[i-1]/h[i]-m[i-1]*h[i]/6,i,1,length(h)); [0,0.2729,0.78,3.241,1.76,6.71] (c_s)
```

```
d_s:makelist(p_y[i]/h[i]-m[i]*h[i]/6,i,1,length(h)); [0.5223,0.05964,2.5482,2.2135,0.2457,10] (d_s)
```

Der Term für die einzelnen C_i - $charfun(A)$ ist die charakteristische Funktion (Indikatorfunktion)

$x \in A \rightarrow 1 \quad x \notin A \rightarrow 0$

```
(%i28) C[i](x):=( a_s[i]*(p_x[i]-x)^3 + b_s[i]*(x-p_x[i-1])^3 + c_s[i]*(p_x[i]-x)+
d_s[i]*(x-p_x[i-1]))*charfun("and" (x>xLimits[i],x<=xLimits[i+1]))$
```

Für Vergleichszwecke lassen wir uns die Terme ausgeben

16. Spline-Interpolation

(%i29) showSplineBranches(last):= for i thru last do display(C[i](x))\$

(%i30) showSplineBranches(length(xList)-1)\$

$$C_1(x) = (0.52229x - 0.55722x^3) \text{ charfun}(x > 0 \text{ and } x \leq 0.2)$$

$$C_2(x) = (0.059644(x - 0.2) + 0.27289(2.2 - x) + 0.11009(x - 0.2)^3 - 0.055722(2.2 - x)^3) \text{ charfun}(x > 0.2 \text{ and } x \leq 2.2)$$

$$C_3(x) = (2.5482(x - 2.2) + 0.77982(3.2 - x) - 0.54818(x - 2.2)^3 + 0.22018(3.2 - x)^3) \text{ charfun}(x > 2.2 \text{ and } x \leq 3.2)$$

$$C_4(x) = (2.2135(x - 3.2) + 3.2409(3.9 - x) - 0.14419(x - 3.2)^3 - 0.78311(3.9 - x)^3) \text{ charfun}(x > 3.2 \text{ and } x \leq 3.9)$$

$$C_5(x) = (0.24567(x - 3.9) + 1.7575(4.8 - x) + 1.6171(x - 3.9)^3 - 0.11215(4.8 - x)^3) \text{ charfun}(x > 3.9 \text{ and } x \leq 4.8)$$

$$C_6(x) = (10.0(x - 4.8) + 6.7089(5 - x) + 7.2772(5 - x)^3) \text{ charfun}(x > 4.8 \text{ and } x \leq 5)$$

The whole Spline ist the sum of alle spline-branches

(%i31) define(S(x),sum(C[i](x),i,1,length(xList)-1))\$

wxMaxima does not know the derivative of the characteristic function, so we must tell him - we declare a pattern match

(%i32) matchdeclare ([aa, bb], numberp, xx, symbolp)\$

The following pattern should be substituted by zero - no Dirac Delta-distribution is needed here

(%i33) tellsimp ('diff (charfun (xx > aa and xx <= bb), xx), 0)\$

Now the derivative can be calculated without any $\frac{d}{dx} \text{charfunc}$ terms in the result

(%i34) define (S_1(x), diff(S(x),x))\$

(%i35) plot2d([[discrete,pL],S(x),S_1(x)],[x,first(xList),last(xList)],[y,-1.3,2.8],[style,[points,3,1,1],[lines,3,3,2],[lines,2,4,2]])\$

$\int_{0.5}^{4.5} f_s(x) dx$ wird numerisch berechnet - zum Vergleich mit *Geogebra*

(%i36) quad_qag (S(x),x,0.5,4.5, 3, 'epsrel=5d-8); [4.4085, 1.212910⁻⁷, 465, 0] (%o36)

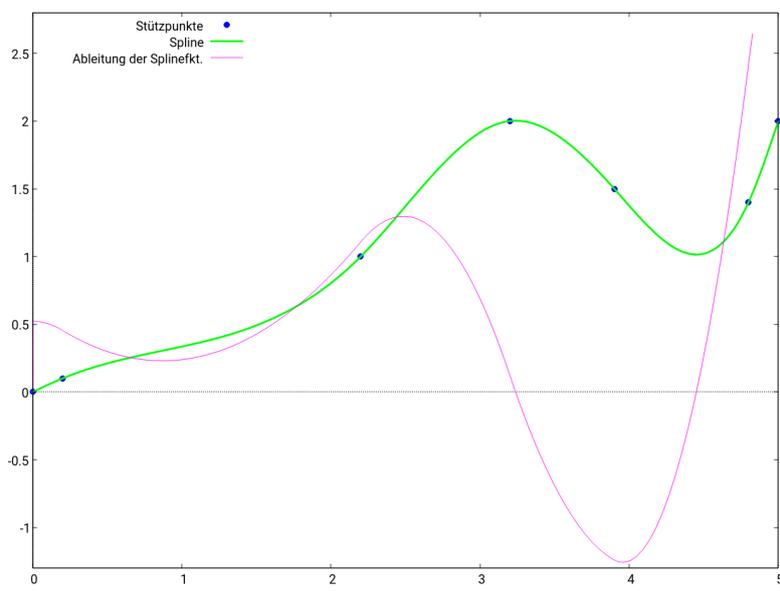


Abb.168 : Spline und seine Ableitung in *wxMaxima*-plot

Hier noch einmal die einzelnen Zweige der Spline-Funktion der beiden Programme zum Vergleich “herausgeschnitten”.

```
(%i33) showSplineBranches(length(xList)-1)$
C1(x)=(0.52229 x-0.55722 x3) charfun(x>0 ∧ x<=0.2)
C2(x)=(0.059644 (x-0.2)+0.27289 (2.2-x)+0.11009 (x-0.2)3-0.055722 (2.2-x)3) charfun(x>0.2 ∧ x<=2.2)
C3(x)=(2.5482 (x-2.2)+0.77982 (3.2-x)-0.54818 (x-2.2)3+0.22018 (3.2-x)3) charfun(x>2.2 ∧ x<=3.2)
C4(x)=(2.2135 (x-3.2)+3.2409 (3.9-x)-0.14419 (x-3.2)3-0.78311 (3.9-x)3) charfun(x>3.2 ∧ x<=3.9)
C5(x)=(0.24567 (x-3.9)+1.7575 (4.8-x)+1.6171 (x-3.9)3-0.11215 (4.8-x)3) charfun(x>3.9 ∧ x<=4.8)
C6(x)=(10.0 (x-4.8)+6.7089 (5-x)+7.2772 (5-x)3) charfun(x>4.8 ∧ x<=5)
```

Abb.169 : Spline Terme in wxMaxima

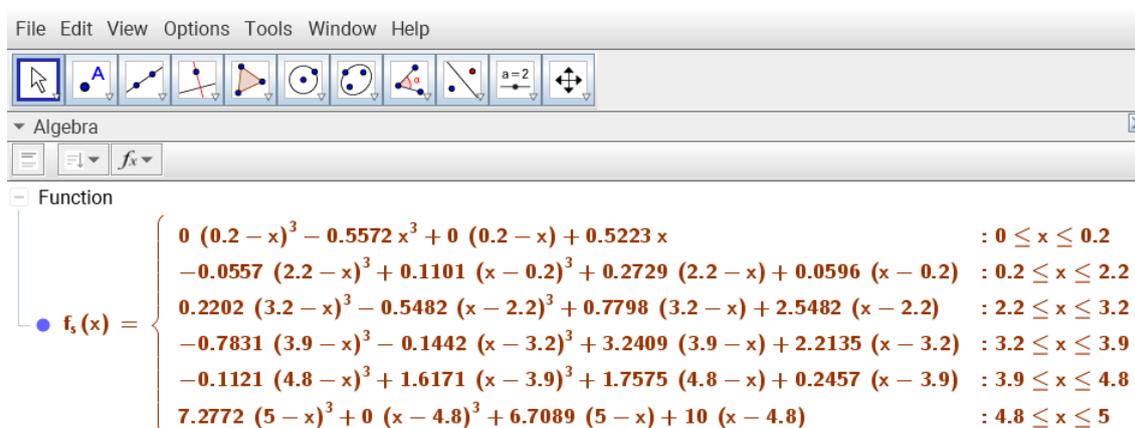


Abb.170 : Spline Terme in Geogebra

Die Zweige stimmen haarscharf überein. Bei jedem Programm ist allerdings Handarbeit nötig: Bei *Geogebra* ist es meiner Ansicht etwas mehr - man könnte sich zwar den TDMA-Algorithmus sparen (*Geogebra* kann Matrizen invertieren), aber den gesamten Algorithmus mit bordeigenen Mitteln nachzubilden ist doch recht mühsam - wie wir im nächsten Abschnitt sehen werden. Da bietet eine ausgereifte Programmiersprache wie Javascript doch mehr Möglichkeiten. Dafür lässt sich ein in *Geogebra*-Script implementierte Lösung mit einem “Custom-Tool” verknüpfen und damit als eigene Datei/Icon leicht benutzen.

In *wxMaxima* könnte man noch mehr verkürzen, indem man das Package “interpol” verwendet:

```
load(interpol);
pL:[[0, 0], [0.2, 0.1], [2.2, 1], [3.2,2],[3.9,1.5], [4.8, 1.4],[5,2]]$
define(S(x),cspline(pL));
```

Hier wird dann die Indikator-Funktion *charfun2()* verwendet (sie gibt statt Null oder Eins *true* oder *false* zurück), dies erlaubt numerische Integration, aber für das Differenzieren braucht man ebenfalls händische Vereinfachung durch “pattern-matching” wie oben.

16.5 Implementierung in *Geogebra (native invert)*

Weitestgehend wurde hier derselbe Weg wie bei *wxMaxima* beschritten - wobei das "human interface" mühsamer ist (wie weiter oben geschildert). Der Vollständigkeit halber sei aber auch dieser Weg hier gezeigt - vor allem für den reinen Anwender bleiben ja diese Mühen unsichtbar!

1. Am Beginn steht selbstverständlich die Punktliste:

```
pointL = {(0,0), (0.2,0.1), (2.2,1), (3.2,2), (3.9,1.5), (4.8,1.4), (5,2)}
```

2. Die Koordinatenlisten:

```
xList = Sequence(x(Element(pointL, i)), i, 1, Length(pointL))
yList = Sequence(y(Element(pointL, i)), i, 1, Length(pointL))
```

3. Jetzt die verschiedenen Differenzenlisten:

```
h = Sequence(Element(xList, i+1) - Element(xList, i), i, 1, Length(xList)-1)
hy = Sequence(Element(yList, i+1) - Element(yList, i), i, 1, Length(yList)-1)
hyOverH = Sequence(Element(hy, i) / Element(h, i), i, 1, Length(h))
deltaHyOverH = Sequence(Element(hyOverH, i+1) - Element(hyOverH, i), i, 1, Length(hyOverH)-1)
```

4. Die Konstantenliste d wird ermittelt:

```
dV = Sequence(6/(Element(h, i+1) + Element(h, i))*
             *Element(deltaHyOverH, i), i, 1, Length(deltaHyOverH))
```

5. Aus dV wird ein Spaltenvektor erzeugt (achten Sie auf die geschwungenen Klammern!):

```
dVec = Sequence({Element(dV, i)}, i, 1, Length(dV))
```

6. Nun wird μ bestimmt ($\lambda = 1 - \mu$; wir verzichten auf griechisch):

```
mu = Sequence(Element(h, i) / (Element(h, i+1) + Element(h, i)), i, 1, Length(h)-1)
```

7. Jetzt die Matrix (ein "furchtbarer" Befehl in *Geogebra*script):

```
A = Sequence(Sequence(If(i == j, 2, i-j == 1, 1-Element(mu, j),
                        j-i == 1, Element(mu, j), 0), i, 1, Length(mu)), j, 1, Length(mu))
```

Die innere Folge steht für die j -te Zeile der Matrix; i ist also der Spaltenindex; in der 1. Zeile ($j = 1$) kann es kein mu geben ($j - i \leq 0$)

8. Wir invertieren A ($B = A^{-1}$):

```
B = Invert(A)
```

9. Wir berechnen den Lösungs(spalten)vektor $M_1 \dots M_5$:

```
M = B * dVec
```

10. $M_0 = 0$ und $M_6 = 0$ wird hinzugefügt und Spaltenvektor auf *Liste m* konvertiert:

```
m = Sequence(If(i==0 || i==Length(M)+1, 0, Element(M, i, 1)), i, 0, Length(M)+1)
```

11. Die Koeffizienten von 16.11 werden als Liste von Listen (Matrix) berechnet (ein Monster):
- ```
coeffs = Sequence({Element(m,i)/(6*Element(h,i)),Element(m,i+1)/(6*Element(h,i)),
 Element(yList,i)/Element(h,i)-Element(m,i)*Element(h,i)/6,
 Element(yList,i+1)/Element(h,i)-Element(m,i+1)*Element(h,i)/6},
 i,1,Length(m)-1)
```

Man beachte das geschwungene Klammersymbol zum Erzeugen der inneren Liste (Zeile der Matrix). Die einzelnen Zeilen werden von *Sequence* erzeugt.

12. Noch ein Kraftakt und wir sind fast am Ziel: wir erstellen eine Liste der Spline-Zweige:

```
F = Sequence(If(Element(xList,i)<x<=Element(xList,i+1),
 Element(coeffs,i,1)*(Element(xList,i+1)-x)^3 +
 Element(coeffs,i,2)*(x-Element(xList,i))^3 +
 Element(coeffs,i,3)*(Element(xList,i+1)-x) +
 Element(coeffs,i,4)*(x-Element(xList,i)),0),i,1,Length(xList)-1)
```

Beachte das Kleinerzeichen beim *If*-Befehl - dadurch gehört die 1. Stützstelle nicht zur Spline-Funktion sondern zur konstanten Nullfunktion. Für ein Integral oder die Ableitungsfunktion hat dies - soweit ich weiß - keine Auswirkungen. Käme man auf die Idee das  $<$  durch  $\leq$  zu ersetzen (weil die Funktionswerte ohnehin an diesen Stellen übereinstimmen) passiert Folgendes:



Bei der anschließenden Summenbildung sind bei der zweiten, dritten bis zur letzten Stützstelle genau 2 Zweige zuständig, sodass sich der Funktionswert an diesen Stellen verdoppelt. Offenbar wird dies, wenn man einen Punkt auf die Spline-Funktion  $f_{sp}$  setzt und anschl. auf eine dieser Stützstellen verschiebt, springt dieser auf "wundersame Weise" auf den doppelten Funktionswert! Möchte man die erste Stützstelle unbedingt bei der Spline-Funktion dabei haben, müsste man die 1. Stützstelle dazugeben:

```
F_1 = Append(F,If(x==Element(xList,1),Element(yList,1),0)
```

13. So jetzt noch die eigentliche Spline-Funktion erstellen:  $f_{sp} = \text{Sum}(F)$

Sie verhält sich etwas anders als die Javascript-Version, weil sie außerhalb der Stützstellen den Wert 0 hat. Sollte das stören, kann man noch eine Indikator-Funktion (charakteristische Funktion) hinzufügen:

```
ind_f(x) = If(Element(xList, 1) < x <= Element(xList, Length(xList)), 1, 0)
g(x) = If(ind_f(x) > 0, f_{sp}(x))
```

Für das " $<$ " -Zeichen in der Indikator-Funktion siehe obige Zusatzbemerkung!

Der Vorteil dieses umständlichen Verfahrens gegenüber dem mit Javascript ist der, dass es auf eine Veränderung der Punktliste sofort wieder automatisch angewandt wird - während man bei der Javascript-Version erneut den Button anklicken muss!

Jetzt heißt es das Benutzer-Werkzeug (Custom Tool) erstellen. Da dies aber exemplarisch für die verschiedensten "Funktionen" gilt, ist dem ein eigener Abschnitt gewidmet.

## 16.6 Zerlegung in LU-Matrizen

Wir haben ein Gleichungssystem der Form  $A_T u = L \overbrace{U}^y u = f$  - wobei  $A_T$  unsere Tridiagonalmatrix ist.  $L$  ist eine untere und  $U$  eine obere (lower, upper) Dreiecksmatrix der folgenden Form:

$$\begin{pmatrix} b_1 & c_1 & & & 0 \\ a_2 & b_2 & c_2 & & \\ & \ddots & \ddots & \ddots & \\ 0 & & a_{n-1} & b_{n-1} & c_{n-1} \\ & & & a_n & b_n \end{pmatrix} = \begin{pmatrix} 1 & & & & \\ l_2 & 1 & & & \\ & l_3 & 1 & & \\ 0 & & \ddots & \ddots & \\ & & & l_n & 1 \end{pmatrix} \begin{pmatrix} v_1 & c_1 & & & \\ v_2 & c_2 & & & 0 \\ & \ddots & \ddots & \ddots & \\ 0 & & v_{n-1} & c_{n-1} & \\ & & & v_n & \end{pmatrix}$$

■ Wir bestimmen die  $L$  und  $U$  durch Ausmultiplizieren und Vergleichen

$$\begin{aligned} b_1 &= v_1 & \Rightarrow & v_1 = b_1 \\ a_k &= l_k v_{k-1} & \Rightarrow & l_k = a_k / v_{k-1} \quad k = 2, \dots, n \\ b_k &= l_k c_{k-1} + v_k & \Rightarrow & v_k = b_k - l_k c_{k-1} = b_k - a_k (c_{k-1} / v_{k-1}) \end{aligned} \quad (16.24)$$

■ Wir bestimmen  $y$  mit  $Ly = f$

$$\begin{aligned} y_1 &= f_1 \\ y_k &= f_k - l_k y_{k-1} \quad k = 2, \dots, n \end{aligned} \quad (16.25)$$

■ Wir bestimmen  $u$  mit  $Uu = y$

$$\begin{aligned} u_n &= \frac{y_n}{v_n} \\ u_k &= \frac{y_k - c_k u_{k+1}}{v_k} = \underbrace{\frac{y_k}{v_k}}_{p_k} - \underbrace{\frac{c_k}{v_k}}_{q_k} u_{k+1} \quad k = (n-1), \dots, 1 \end{aligned} \quad (16.26)$$

 Die normale Inversion einer Matrix ist von den Multiplikationen ca.  $\mathcal{O}(n^3)$ , die obige  $LU$ -Zerlegung hat ca.  $\mathcal{O}(3n)$ . Bei größeren  $n$  wirkt sich das beachtlich aus!

Während man in *wxMaxima* obige Rekursionformeln mit indizierten Funktionen direkt implementieren kann (Beachte die verschiedenen Zuweisungszeichen!)

```
v[1]:b[1]$ 1[k]:= a[k]/v[k-1]
```

ist dies in *Geogebra* etwas schwieriger

### 16.6.1 Rekursion in *Geogebra*

Eine Möglichkeit (sie geht auf eine Idee von *Michel Iroir* im *Geogebra*-Forum zurück) benutzt den Befehl *IterationList* und macht aus  $v_k$  einen "Punkt"  $(v, k)$  - also die  $x$ -Koordinate ist der Wert der Folge, die  $y$ -Koordinate ist der Index. Damit lässt sich mit

```
IterationList(<Expression>, <Variables>, <Start Values>, <Count>)
```

für Gleichungen 16.24 eine Liste erzeugen (beachte die "Punkteklammer" vor dem ersten *El...*

```
IterationList((Element(b, y(A)+1)-Element(a, y(A)+1)*Element(c, y(A)))/x(A),
 y(A)+1, A, {(Element(b, 1), 1)}, Length(c)-1)
```

Die erste "Iteration" (sie ist ja noch keine) liefert die Initialisierungswerte als Liste

```
{(b[1], 1)}
```

bei der ersten richtigen Iteration wird der Ausdruck ausgeführt und zur Liste hinzugefügt

```
{(b[2]-a[2]*c[1]/v[1], 2), (b[1], 1)}
```

die zweiten Iteration ergibt

```
{(b[3]-a[3]*c[2]/v[2], 3), (b[2]-a[2]*c[1]/v[1], 2), (b[1], 1)}
```

usw.

Wir müssen also  $(n - 1)$  Iterationen ausführen, um  $(v_n, n)$  zu erhalten, wobei  $n$  die Länge der Listen  $a$ ,  $b$  und  $c$  ist. Um die Folgenliste zu erhalten, brauchen wir von der Punktliste die  $x$ -Koordinaten, also lautet jetzt der vollständige Befehl für 16.24

```
x(IterationList((Element(b, y(A)+1)-Element(a, y(A)+1)*Element(c, y(A)))/x(A),
 y(A)+1, A, {(Element(b, 1), 1)}, Length(c)-1))
```

Wenn es sich nicht um eine Rekursion handelt - wie z.B. bei  $l$ -Feld von 16.24 (rechts vom Gleichheitszeichen sind ja alle Elemente bekannt), dann bietet sich der *Zip*-Befehl von *Geogebra* an: `l=Join({{0}, Zip(a(k) / v(k - 1), k, 2...Length(a))})`



Beachte, dass das Feld  $l$  erst beim Index 2 beginnt, also fügen wir vorne eine "Dummy-Null" ein. Außerdem erlaubt *Geogebra* (Desktop-Version 5.0.470) beim *Zip*-Befehl eine abgekürzte Schreibweise für den *Element*-Befehl

```
a(k) statt Element(a, k)
```

## 16.7 Spline in *Geogebra* (LU-Zerlegung)

Um die Rechnungen etwas zu vereinfachen benutzen wir hier 2 *Custom Tools*:

1. `Delta(<Liste>)` --> `{1[2]-1[1], 1[3]-1[2], ...1[n]-1[n-1]}`
2. `SuccSum(<Liste>)` --> `{1[2]-1[1], 1[3]-1[2], ...1[n]-1[n-1]}`

`Delta(<Liste>)` bildet aus einer Liste von Zahlen eine Liste der Differenz der aufeinanderfolgenden Zahlen.

`SuccSum(<Liste>)`(successor sum) bildet aus einer Liste von Zahlen eine Liste der Summe der aufeinanderfolgenden Zahlen.

## 16. Spline-Interpolation

Implementiert wurden diese beiden Benutzer-Funktionen mit folgenden Befehlen:

```
a = {1, 2, 3, 4, 5}
b = Zip(a(k), k, 1...(Length(a) - 1))
c = Zip(a(k), k, 2...Length(a))
r = c - b --> Delta
s = b + c --> SuccSum
```

Wie das im einzelnen funktioniert wird in 16.8 näher erklärt.

Hier das Befehlslisting

```
1 pointL={{(0,0), (0.2,0.1), (2.2,1), (3.2,2), (3.9,1.5), (4.8,1.4), (5,2)}}
2 xList = x(pointL)
3 yList = y(pointL)
4 // Einsatz der neuen Tools
5 h = Delta(xList)
6 hy = Delta(yList)
7 SuccSumH = SuccSum(h)
8 mu = Zip(h(k)/SuccSumH(k), k, 1...(Length(h)-1))
9 lambda = 1 - mu
10 hyOverH = hy/h
11 deltaHyOverH = Delta(hyOverH)
12 f = Zip(6*deltaHyOverH(k)/SuccSumH(k), k, 1...Length(deltaHyOverH))
13 // Rekursion wie im Text oben beschrieben
14 v = x(IterationList((2-Element(mu, y(A) + 1)*Element(lambda, y(A))/x(A),
15 y(A) + 1), A, {(2, 1)}, Length(mu) - 1))
16 l = Join({{0}}, Zip(mu(k)/v(k-1), k, 2...Length(mu)))
17 // Rekursion wie im Text oben beschrieben
18 yL = x(IterationList((Element(f, y(A) + 1) - Element(l, y(A) + 1) x(A),
19 y(A) + 1), A, {(Element(f, 1), 1)}, Length(f) - 1))
20 p = Zip(yL(k)/v(k), k, 1...Length(v))
21 q = Zip(lambda(k)/v(k), k, 1...Length(v))
22 M = Reverse(x(IterationList((Element(p,y(A)-1)-Element(q,y(A)-1)*x(A),
23 y(A)-1), A, {(Element(p, Length(p)), Length(p))}, Length(p)-1)))
24 m = Join({{0}}, M, {0})
25 h6 = 6*h
26 hOver6 = h/6
27 coeffs = Zip({m(k)/h6(k), m(k+1)/h6(k), yList(k)/h(k)-m(k)*hOver6(k),
28 yList(k+1)/h(k)-m(k+1)*hOver6(k)}, k, 1...(Length(m)-1))
29 //..... REST bleibt gleich
30
31
```

In Zeile 14 beginnt Implementierung von 16.24

In Zeile 18 beginnt Implementierung von 16.25

In Zeile 22 beginnt Implementierung von 16.26

Man erkennt, dass etwas andere Befehle verwendet wurden, wie bei der vorigen Version(native invert) . Es führen eben viele Wege nach Rom!

Es bleibt nach dem Erstellen der Splinefunktion wieder ein Benutzerwerkzeug zu bauen:

*csplineLU*

## 16.8 Erstellen eines Custom Tool in *Geogebra*

Ein *Custom Tool* (Benutzerwerkzeug) ist eine Funktion, deren Argumente (Parameter) *Geogebra*-Objekte sind und als Output 1 *Geogebra*-Objekt liefert.

$$ct(Obj1, Obj2, \dots, Objn) \mapsto Obj_{target}$$

Der Name des Tools (= Funktion) (hier *ct*) kann während des Erstellungsprozesses frei gewählt werden. Außerdem ist es hilfreich (aber nicht notwendig) ein eigenes 32x32 Icon zur Verfügung zu haben. Für das Spline-Tool hab ich mir schnell eines mit Gimp erstellt:

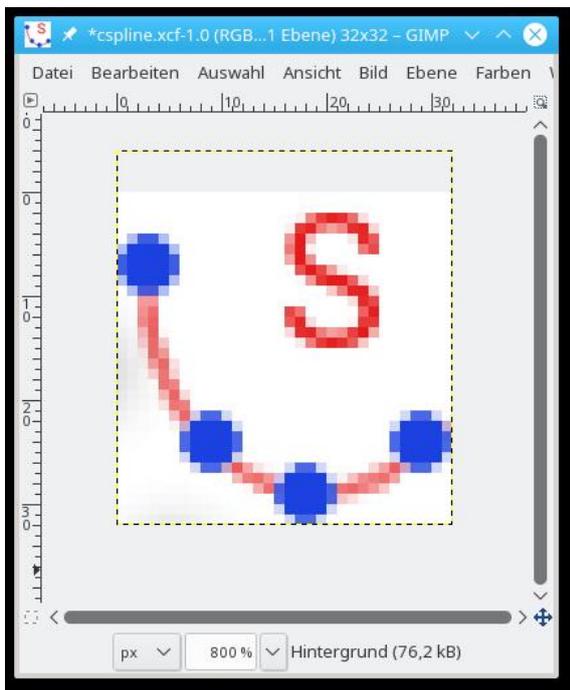


Abb.171 : Spline Icon für *Geogebra*-Menü

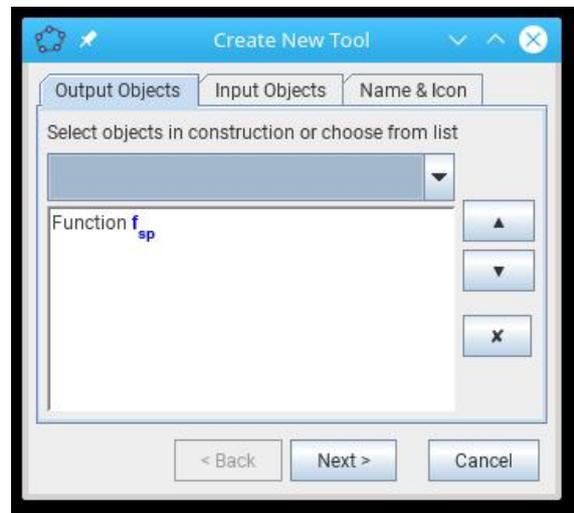


Abb.172 : Output des Custom Tool wird festgelegt

Damit *Geogebra* ein *Custom Tool* erzeugen kann, müssen alle Objekte (Parameter und Output) bereits existieren. Es wird als der Weg von den Parametern zum Output abgebildet!

Im *Tools*-Menü von *Geogebra* wird *Create New Tool* aufgerufen:

- Zuerst wird das Ergebnis(Output)-Objekt festgelegt (siehe Abb. 172 )  $\rightarrow f_{sp}$
- Klicken von *Next* zeigt, dass *pointL* als Parameter (Input) benötigt wird
- Wieder klicken von *Next* bringt uns in den Reiter *name & icon*:  
Wir wählen den Namen des Werkzeugs und Befehl: **cspline** (für cubic spline) und wählen unser vorhin konstruiertes Icon aus!

## 16. Spline-Interpolation

Wenn die Auswahlbox *Show in Toolbar* angeklickt ist, sollte das Werkzeug mit unserem Icon im Werkzeugkasten zur Verfügung stehen - sollte man dies vergessen haben, kann man im *Tool*-Menü → *Customize Toolbar* das Icon in einer Ansicht(hier *General*) an einer bestimmten Stelle einfügen!



Abb.173 : Name,Befehl,Icon und Eingabetipp

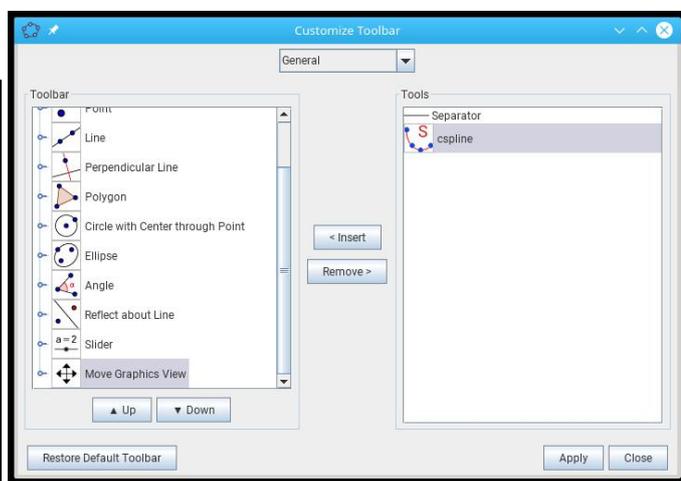


Abb.174 : Einfügen des Icons ins Menü

Nach dem Klicken von *Finish* kommt hoffentlich eine Erfolgsmeldung.

- Als nächstes sollte man das Werkzeug als eigenen File (\*.ggt, *geogebra tool*) speichern. Dies wird im *Tool*-Menü → *Manage Tools* mit *Save as* erledigt (ein aussagekräftiger Dateiname wäre nicht schlecht).
- Will man das Werkzeug verwenden, lädt man es wie mit dem *Datei*-Menü von *Geogebra*.  
 Für das verwenden eines Werkzeugs **nicht(!)** *Manage Tools* → *Open* benutzen. Dies benutzt man, um den "Quellcode" (das Zustandekommen) eines Benutzerwerkzeugs nachzuvollziehen!
- Will man dieses Werkzeug in Zukunft beim Öffnen von *Geogebra* bereits zur Verfügung haben, gehen wir auf *Options* → *Save Settings*.

### 16.8.1 2 Verwendungsmöglichkeiten

1. Man klickt das Icon und anschließend die Punktliste
2. Man gibt in die Kommandozeile: `<FuncName> =cspline(<List of Points>)`

## 16.9 Theoretische Überlegungen

### 16.9.1 Thomas Algorithmus (TDMA)

Der TDMA 16.23 kann nur scheitern, wenn der Nenner  $b_i + a_i P_{i-1}$  verschwindet. Wenn wir uns die Matrix in 16.18 anschauen stellen wir fest:

1. Die Hauptdiagonale besteht aus der Zahl 2 ( $b_i$ )
2. Die Nebendiagonalen sind positiv und kleiner 1:  $0 < a_i < 1 \wedge 0 < c_i < 1$
3. Damit können wir eine vollständige Induktion starten:

■  $|P_1| = \left| -\frac{c_1}{2} \right| < 1$

- Falls  $|P_{i-1}| < 1 \Rightarrow |P_i| < 1$ , weil der Nenner in 16.23 ist dann größer 1, bei einem Zähler der kleiner 1 ist.

Es gilt also

$$\forall i \in \{1, \dots, n\} : |P_i| < 1 \Rightarrow b_i + a_i P_{i-1} > 0$$

### 16.9.2 LU-Faktorisierung

Der Algorithmus von 16.24 scheitert wenn  $v_k$  verschwindet.

■  $v_1 = 2 > 1$

- Wir zeigen auch hier: Wenn  $v_{k-1} > 1$  ist, dann ist es auch  $v_k$

$$v_k = 2 - \underbrace{a_k}_{< 1} \underbrace{\frac{c_{k-1}}{v_{k-1}}}_{< 1} > 1 \neq 0$$

Auch hier zeigt sich, dass bei unser Ausgangsmatrix der Algorithmus immer zum Erfolg führt!



# 17 | Computertomografie-Basics

## 17.1 Einleitung

Die Computertomografie beschäftigt sich mit der Rekonstruktion des inneren Aufbaus eines Gegenstandes. Mit "inneren Aufbau" ist hier die räumliche Verteilung einer Materialeigenschaft gemeint. Die bekannteste Möglichkeit ist vermutlich die Vermessung des Absorptionskoeffizienten  $\alpha$  beim Durchschicken eines "dünnen" Röntgenstrahls durch menschliches Gewebe. Ziel also ist, das skalare Feld  $\alpha(\vec{x})$  zu rekonstruieren. Aber statt Absorption könnten auch andere Eigenschaften wie "time-of-flight" (Durchdringungszeit), Phasenverschiebung, Streuung, elektrischer Widerstand, usw. herangezogen werden.

Die Wikipedia zählt heute (2014) ca. 30 verschiedenen Tomografietypen auf. Alle benützen natürlich irgendeine physikalische Eigenschaft des "zu scannenden Gegenstandes" wie z.B. den elektrischen Widerstand. Die durch das "Scannen" gewonnenen Daten lassen sich bearbeiten (wo sind starke Veränderungen - Umrisse) und an Hand von Vergleichsdaten lassen sich bekannte Komponenten(Organe) identifizieren.

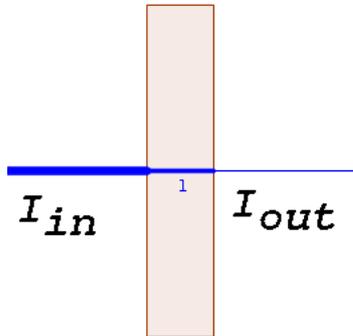
Die CT ist im Wesentlichen(Ausnahme: Einbringen von Kontrastmittel, radioaktives Material, usw.) ein "nicht-invasives" Verfahren, d.h. wir brauchen den Gegenstand nicht "Aufschneiden", um sein Inneres zu kennen. Wie weit das Einbringen ionisierender Strahlung als "nicht-invasiv" zu bezeichnen ist, ist natürlich Ansichtssache - aber klarerweise müssen wir irgendeine Wechselwirkung zwischen Gegenstand und Messgerät herbeiführen. Ultraschall schneidet in punkto "biologische Gefährlichkeit" besser ab, durch die weit größere Wellenlänge ist die Auflösung dafür weit geringer.

Im Folgenden werden wir die Röntgen-CT im Auge haben, da wir Reflexion und Streuung vernachlässigen können. Wir können die Welleneigenschaften vernachlässigen. Zur weiteren Vereinfachung unseres Modelles werden wir 1 Dimension vernachlässigen und nur Raumpunkte betrachten, die in einer Ebene liegen. Außerdem werden wir das skalare Feld  $\alpha(\vec{x})$  diskretisieren, d.h. wie bei einem Fernseher das Bild aus Pixel (Bildpunkten) aufgebaut ist, so werden wir unser kontinuierliches skalare Feld  $\alpha(\vec{x})$  durch  $\alpha(\vec{x}_i)$  annähern. Wir können die "Pixel" von links oben nach rechts unten durchnummerieren und so eine Folge von Variablen schaffen, die es zu berechnen gilt. Für die Vorstellung ist es allerdings leichter sich einem Matrix-Schema zu bedienen, also  $\alpha(\vec{x}_{ij})$  kurz  $\alpha_{ij}$  für  $i$ -te Zeile und  $j$ -te Spalte.

Heutige ct-Geräte bewegen sich auf Spiralbahnen, benutzen mehrere Röntgenstrahlquellen und -detektoren verschiedener Intensität, benutzen Vergleichsdaten, und ... Verwechseln Sie also nicht die Grundlagen mit "state of the art devices"!

## 17.2 Physikalische Grundlagen

Wir betrachten zuerst die Absorption von Röntgenstrahlung in einem homogenem Material der Dicke 1 Längeneinheit - wobei 90% der Strahlung durchgelassen werden:

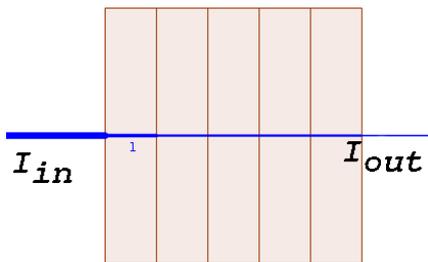


Intensität der durchgehenden Strahlung:

$$I_{out} = 0,9 \cdot I_{in}$$

Abb.175 : Absorption bei Einheitsdicke

Die Dicke der Schicht sei nun d Längeneinheiten. Die Ausgangsintensität der vorigen Schicht ist die Eingangsintensität der nachfolgenden:



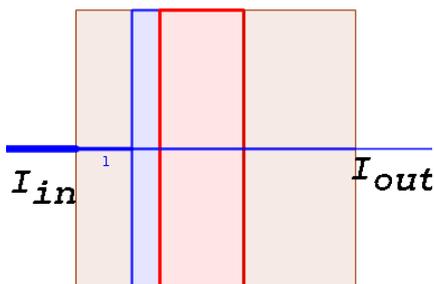
90% der Strahlung werden durchgelassen (pro Schicht-einheit), daraus ergibt sich:

$$I_{out} = 0,9^d \cdot I_{in} = \underbrace{e^{(\ln 0,9)d}}_{0,9^d} \cdot I_{in} = e^{-\alpha d} \cdot I_{in}$$

mit  $\alpha := -\ln 0,9 > 0$  (Absorptionskoeffizient)

Abb.176 : Absorption bei d Längeneinheiten

Sind die Schichten verschieden dick und aus verschiedenen Materialien



Die Dicke der Schichten sei  $d_1, d_2 \dots d_4$  Längeneinheiten mit den Absorptionskoeffizienten  $\alpha_1, \alpha_2, \dots \alpha_4$ . Daraus ergibt sich:

$$I_{out} = I_{in} e^{-\alpha_1 d_1} e^{-\alpha_2 d_2} \dots e^{-\alpha_4 d_4} = I_{in} e^{-\sum_i \alpha_i d_i}$$

dividieren durch  $I_{in}$  und logarithmieren liefert

$$\frac{I_{out}}{I_{in}} = e^{-\sum_i \alpha_i d_i} \Rightarrow -\ln \frac{I_{out}}{I_{in}} = \sum_i \alpha_i d_i$$

Abb.177 : Verschiedene Materialien

Bei bekannter linker Seite (Eingangs- und Ausgangsintensität) und bekannten Schichtdicken ( $d_i$ ) ist dies eine lineare Gleichung für die Absorptionskoeffizienten des Materials. Man kann natürlich nicht hoffen dieses Gleichungssystem durch variieren des Strahlwinkels zu lösen, weil

die Anordnung der Schichten beliebig permutiert werden kann! Allerdings ändert sich das durch hinzufügen einer Dimension!

## 17.3 Geometrische Grundlagen

Diese Situation wurde vom Mathematiker Radon bereits geklärt vor dem 1. Weltkrieg - ein typisches Beispiel, dass Grundlagenforschung gar nicht hoch genug einzuschätzen ist. Niemand kann ahnen, ob die Forschungsergebnisse von "Nutzen" sind ( ein anderes wichtiges Beispiel wären die " gekrümmten Mannigfaltigkeiten" von Riemann - ohne die Einstein niemals seine Allgemeine Relativitätstheorie formulieren hätte können). Wer sich näher damit aueinander setzen möchte, müsste sich mit der "Radon-Transformation" beschäftigen.

### 17.3.1 Arbeitsmodell

Jetzt zu unserer "Versuchsanordnung":

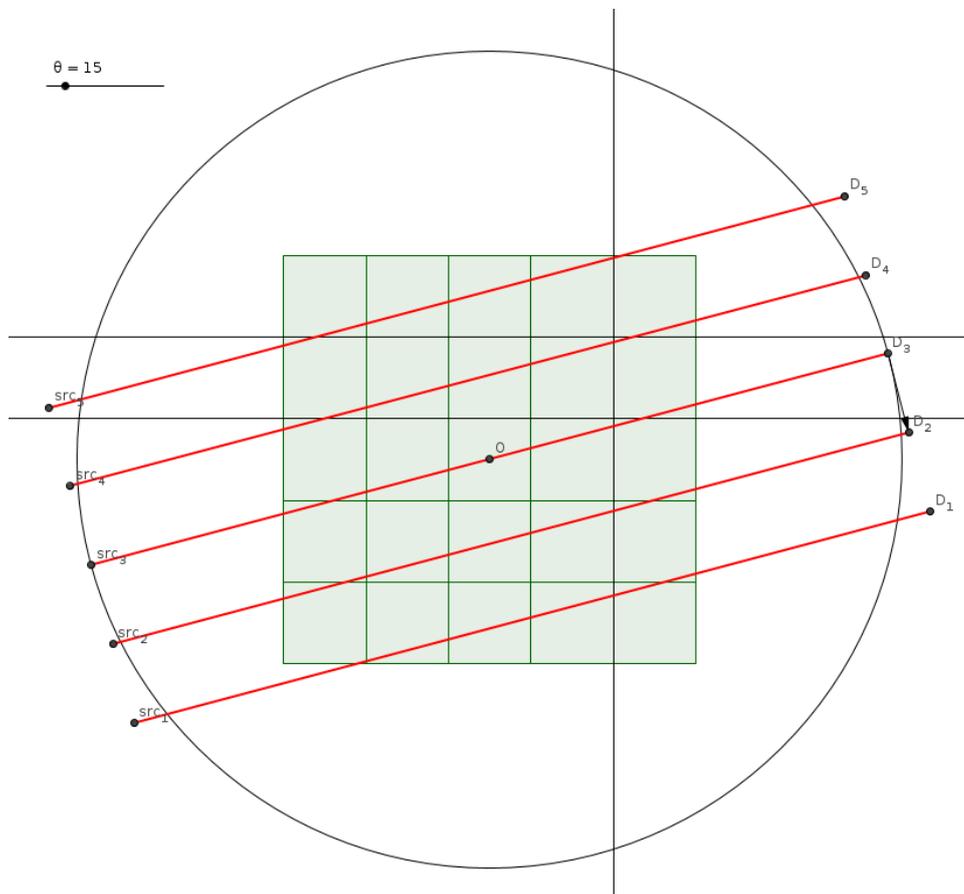


Abb.178 : 2 dimensionales Modell

Der Einfachheit halber besteht unser Objekt nur aus 25 quadratischen "Pixeln", die zu einem Quadrat angeordnet sind. Da die Anordnung eine Matrixschreibweise nahelegt, bezeichnen wir

die unbekanntenen Absorptionskoeffizienten mit

$$\begin{pmatrix} \alpha_{11} & \alpha_{12} & \dots & \alpha_{15} \\ \alpha_{21} & \alpha_{22} & \dots & \alpha_{25} \\ \vdots & & & \vdots \\ \alpha_{51} & \alpha_{52} & \dots & \alpha_{55} \end{pmatrix}$$

Die Seitenlänge der Pixel sei unsere Einheitslänge. Gescannt wird unser Target durch ein paralleles Strahlenbündel (5 Stück im Abstand 1), die den Winkel  $\theta$  mit der x-Achse bilden. Der Winkel  $\theta$  kann durch Drehung der Sender bzw. des Targets variiert werden. Der Richtungsvektor der Geraden  $\vec{a} = (\cos \theta, \sin \theta)$ , der nach rechts geklappte Einheitsnormalvektor beträgt  $\vec{n}_0 = (\sin \theta, -\cos \theta)$

Der Koordinatenursprung ist eingezeichnet.

Die Koordinaten der Sender  $S_i$  bei  $\theta = 0$  sind  $[-5, 2], [-5, 1], [-5, 0], [-5, -1], [-5, -2]$  und ihre aktuelle Position wird durch eine Rotationsmatrix bestimmt. Jetzt gilt es zu überlegen, welche Strecke die einzelnen Strahlen in den Pixeln zurücklegen:

### 17.3.2 Kein Treffer bei $0 \leq \theta \leq 90^\circ$

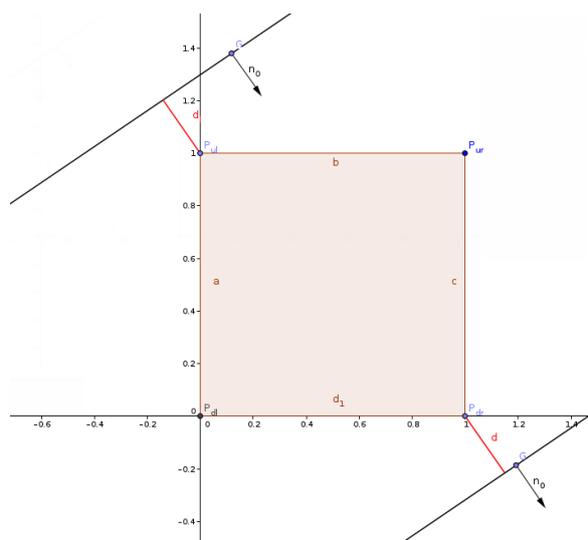


Abb.179 : Kein Treffer

Der Richtungsvektor der Geraden  $\vec{a} = (\cos \theta, \sin \theta)$ , der nach rechts geklappte Einheitsnormalvektor beträgt  $\vec{n}_0 = (\sin \theta, -\cos \theta)$ . Wenn der Strahl von "unten" hereinwandert, stößt er zuerst auf  $P_{dr}$  (down-right) und verlässt das Quadrat wieder bei  $P_{ul}$  (up-left). Das Ereignis "kein Treffer" lässt sich also so formulieren:

$$d(P_{dr}) := \overrightarrow{SP_{dr}} \cdot \vec{n}_0 \leq 0 \quad \text{oder} \quad d(P_{ul}) \geq 0$$

$S$  ist dabei der Ausgangspunkt des Strahls (Sender, Source). Also verbal ausgedrückt:  $P_{dr}$  liegt links vom Strahl oder  $P_{ul}$  liegt rechts von ihm!

**17.3.3 Einfallswinkel  $\theta \leq 45^\circ$**

Wenn der Strahl von “unten” hereinwandert, gibt es 2 Grenzfälle (blau) bei denen er eine horizontale und eine vertikale Grenze durchkreuzt, dazwischen liegt der Fall mit 2 vertikalen Schnitten. Kümmern wir uns um den Grenzfall a):

$$d(P_{dr}) = \sin \theta$$

Also bei  $d(P_{dr}) \leq \sin \theta$  sind wir unterhalb des unteren blauen Grenzstrahls, bei  $|d(P_{ul})| \leq \sin \theta$  sind wir oberhalb des oberen blauen Grenzstrahls, sonst durchschneiden wir das ganze Quadrat! Für die Länge  $\ell$ , die der Strahl im “Pixel” zubringt bedeutet das:

$$\left. \begin{array}{l} d(P_{dr}) \leq \sin \theta \\ \text{or} \\ |d(P_{ul})| \leq \sin \theta \end{array} \right\} \Rightarrow \ell = d \frac{2}{\sin 2\theta} \quad (17.1)$$

$d$  ist dabei der entsprechende positive Abstand. In allen anderen Fällen (also zwischen den blauen Linien) ergibt sich:

$$\ell = \frac{1}{\cos \theta} \quad (17.2)$$

**17.3.4 Einfallswinkel  $45^\circ \leq \theta \leq 90^\circ$**

Die genaue Analyse überlasse ich jetzt dem Leser - aber sie funktioniert nach obigem Vorbild. Nach wie vor sind unten-rechts(dr) und oben-links (ul) die 2 Bezugspunkte, damit ergeben sich die Fälle:

$$\begin{aligned} d(P_{dr}) \leq \cos \theta &\Rightarrow \ell = d(P_{dr}) \frac{2}{\sin 2\theta} \\ |d(P_{ul})| \leq \cos \theta &\Rightarrow \ell = |d(P_{ul})| \frac{2}{\sin 2\theta} \\ \text{sonst} &\Rightarrow \ell = \frac{1}{\sin \theta} \end{aligned}$$

Um dies in die obigen Formeln überzuführen können wir  $\theta$  einfach durch  $\theta' = \frac{\pi}{2} - \theta$  ersetzen. Es bleibt dem Wir landen wieder bei den Formeln 17.1 bzw. 17.2

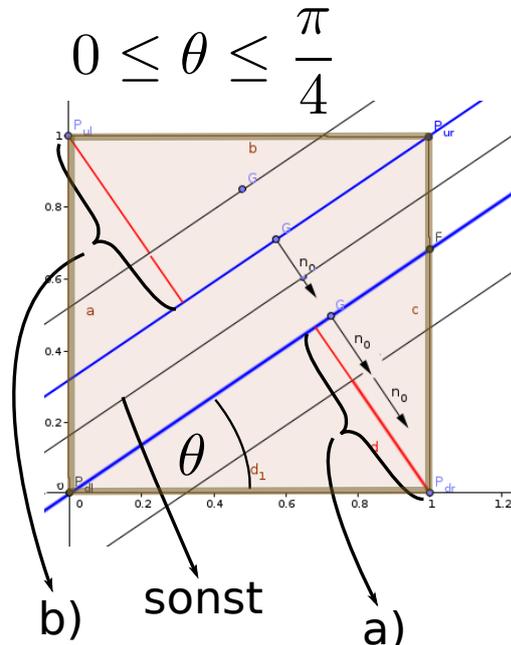


Abb.180 : Einfallswinkel  $\theta \leq 45^\circ$

### 17.3.5 Kein Treffer bei $90^\circ \leq \theta \leq 180^\circ$

Ersetze in obiger Formel  $P_{dr}$  durch  $P_{ur}$  (up-right) und  $P_{ul}$  durch  $P_{dl}$  (down-left) oder kurz: up  $\leftrightarrow$  down

**Diese Ersetzung gilt auch für die folgenden Punkte!**

### 17.3.6 Einfallswinkel $90^\circ \leq \theta \leq 135^\circ$

obige Ersetzung und  $\theta$  durch  $\theta' = \theta - \frac{\pi}{2}$

### 17.3.7 Einfallswinkel $135^\circ \leq \theta \leq 180^\circ$

obige Ersetzung und  $\theta$  durch  $\theta' = \pi - \theta$

## 17.4 Mathematische Grundlagen

### 17.4.1 Rotationen in der Ebene

Hier gibt es nicht viel zu sagen:

Rotation ist eine Abbildung des  $\mathbb{R}^2 \rightarrow \mathbb{R}^2$  und kann mit folgender Matrix durchgeführt werden:

$$R_\theta = \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix}$$

also  $P_\theta = R_\theta \cdot P$

Versuchen Sie das in *Geogebra*

$$A = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix}$$

indem Sie vorher einen Schieberegler  $\theta$  definieren. Einen Punkt  $P$  in der Nähe des Ursprungs setzen und  $P_1 = A * P$  festlegen und anschl. den Schieberegler betätigen. Damit haben Sie den Befehl **Drehe** von *Geogebra* mathematisch nachgebildet!

### 17.4.2 Mathematik der linearen Gleichungssysteme - LA

LA steht hier natürlich nicht für *Los Angeles* sondern für *Lineare Algebra*. Ein eindeutig lösbares lineares Gleichungssystem wird i. a. beschrieben als

$$A \cdot \vec{x} = \vec{b} \mid A^{-1}. \quad \Rightarrow I \cdot \vec{x} = A^{-1} \cdot \vec{b}$$

Das Auffinden des Lösungsvektors läuft sich auf das Invertieren einer quadratischen Matrix hinaus. Jedes CAS kann das - so auch das mit dem wir arbeiten wollen: *wxMaxima* - es ist für alle Plattformen zu haben und im Gegensatz zu *Mathematica* oder *Maple* ist es **kostenlos**

verfügbar. Sprache für die Dinger muss man sowieso für jedes lernen. (Stichwort: “Lebenslanges Lernen”)

Jetzt ist die Sache bei uns etwas komplizierter, denn wir haben es mit einem überbestimmten (hoffentlich - Rang der Matrix ist gleich Spaltenzahl) Gleichungssystem zu tun, d. h. unsere Koeffizientenmatrix  $A$  hat mehr Zeilen als Spalten (Anzahl der Messungen übersteigt die Anzahl der gesuchten Absorptionskoeffizienten). Natürlich gibt es hier keine Lösung im eigentlichen Sinne, sondern wir brauchen einen Vektor  $\vec{x}$ , der alle Gleichungen “einigermaßen gut” erfüllt. Ein Verfahren, dass dies erfüllt heißt *Q-R-Faktorisierung* bzw. Q-R-Zerlegung von  $A$  - und *wxMaxima* kann das mit `load(lapack)` und der Befehl dafür lautet `dgeqrf(A)` und liefert eine Liste mit  $Q$  und  $R$ .

Welche Eigenschaften haben nun die Matrizen  $Q$  und  $R$ ?

1.  $Q \cdot R = A$  - klar, wegen Faktorisierung
2.  $Q$  ist quadratisch mit derselben Zeilenanzahl von  $A$
3.  $Q$  ist orthonormal, also  $Q^T = Q^{-1}$
4.  $R$  hat dieselbe Zeilenanzahl und Spaltenzahl wie  $A$
5.  $R$  ist eine obere Dreiecksmatrix (engl. “upper triangular matrix”) - restliche Zeilen sind 0!

Die Vorgangsweise ist im Kern dann einfach:

$$A \cdot \vec{x} = \vec{b} \quad \Rightarrow \quad Q \cdot R \cdot \vec{x} = \vec{b} \quad \Rightarrow \quad R \cdot \vec{x} = Q^T \cdot \vec{b}$$

Die letzte Gleichung ist durch Rücksubstitution aber leicht zu lösen.

Stellen wir das Verfahren in *wxMaxima* auf die Probe: wir nehmen  $x = 1$  und  $y = 2$  und basteln ein nicht-widersprüchliches überbestimmtes Gleichungssystem aus 4 Gleichungen

$$\begin{array}{l} 1) \quad x + y = 3 \\ 2) \quad 2x + y = 4 \\ 3) \quad 2x + 2y = 6 \\ 4) \quad 5x + 2y = 9 \end{array} \quad \Rightarrow \quad \begin{pmatrix} 1 & 1 \\ 2 & 1 \\ 2 & 2 \\ 5 & 2 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 3 \\ 4 \\ 6 \\ 9 \end{pmatrix}$$

Bevor wir uns die Lösung in *wxMaxima* anschauen, noch ein kurzer Hinweis wie in diesem Programm Funktionen aufgebaut sind:

```
<funcName>(<par1>,<par2>, ...):=block([<localVar1>: <value1>,<localVar2>: <value2>],
 for c:10 thru 1 step -1 do <command>,
<var>)$
```

Also Funktionsname mit Übergabeparametern, `block(` ist dasselbe wie in vielen Programmiersprachen eine geschwungene Klammer und wird dort meist als *scope* bezeichnet (Gültigkeitsbereich von lokalen Variablen). Wie in anderen Programmiersprachen auch, kann `block(`

## 17. Computertomografie-Basics

---

weggelassen werden, wenn nur ein Befehl kommt. Innerhalb von `block(` können in eckigen Klammern *lokale Variablen* definiert werden (und ev. initialisiert), um globale Variablen nicht zu beeinflussen. Als Beispiel habe ich hier einmal eine Schleife mit der Schleifenvariablen `c` gewählt, wobei von 10 heruntergezählt wird. Die einzelnen Befehle werden durch Beistriche getrennt (in Javascript ist dies ein “;”). Der Wert der letzten Variable wird zurückgegeben! Das folgende Programm ist selbsterklärend bis auf die Funktion `backwardSubstitution`. Übergabeparameter sind die Dreiecksmatrix `r` und der Konstantenvektor `b`. In den lokalen Variablen holt man sich mit der Funktion `matrix_size` und `second` die Spaltenanzahl von `r` und definiert einen Nullvektor `x` mit 1 Spalte und `cols` Zeilen. Die Berechnung des Lösungsvektors durch “Rückeinsetzen” ergibt (bitte nachrechnen):

$$x_c = \left( b_c - \underbrace{\sum_{k=c-1}^n r_{ck} \cdot x_k}_{\vec{r}_c \cdot \vec{x}} \right) / r_{cc}$$

Die Summe kann also mit dem skalaren Produkt des `c`-ten Zeilenvektors von `r` (`=row(r,c)`) und dem bisher bekannten Lösungsvektor `x` berechnet werden.

```
(%i11) load(lapack)$
...
(%i12) A:matrix([1,1],[2,1],[2,2],[5,2]);

(%o12) $\begin{pmatrix} 1 & 1 \\ 2 & 1 \\ 2 & 2 \\ 5 & 2 \end{pmatrix}$
(%i13) b:matrix([3],[4],[6],[9])$
(%i14) [q,r]:dgeqrf(A)$
(%i15) b_T:transpose(q) . b;

(%o15) $\begin{pmatrix} -11.6619037896906 \\ 2.44948974278318 \\ -1.554312234475219 \cdot 10^{-15} \\ -8.881784197001252 \cdot 10^{-16} \end{pmatrix}$
(%i16) backwardSubstitution(r,b):=block([cols:second(matrix_size(r)),
 x:zeromatrix(second(matrix_size(r)),1)],
 for c:cols thru 1 step -1 do x[c]:((b[c]-row(r,c) . x)/r[c,c]),
 x)$
(%i17) backwardSubstitution(r,b_T);
```

```
(%o17) $\begin{pmatrix} 1.0 \\ 2.0 \end{pmatrix}$
```

Hat doch ganz gut geklappt! Jetzt könnte man noch den Vektor  $b$  verändern und so das System widersprüchlich machen - diese Experiment überlasse ich dem Leser.

Fazit: Weichen die Werte nicht allzustark ab, bleibt der Lösungsvektor in der Nähe von  $(1, 2)$   
Nun zum eigentlichen Programm - *wxMaxima* braucht dazu bei mir ca. 1 Minute - es handelt sich schließlich um 36 verschiedene Winkel bei 5 Strahlen - macht 180 Gleichungen für 25 Variable.

```
(%i79) ratprint:false$ loadprint:false$ load(lapack)$
```

0errors,

Bezugspunkt eines quadratischen Pixels ist unten-links, für die anderen Eckpunkte werden die Koordinaten entsprechend erhöht!

```
(%i82) down_left:[0,0]$ down_right:[1,0]$ up_right:[1,1]$ up_left:[0,1]$
```

```
(%i86) vecLength(P,Q):=sqrt((P-Q) . (P-Q))$
```

Eingabe Matrix-Indices und position z.B. `down_right` Rückgabe: $[x, y]$

```
(%i87) getPoint(i,j, position):=[-3.5+j,2.5-i]+position$
```

```
(%i88) getNormalUnitVec(theta):=block([t:float(theta*%pi/180)],
 [sin(t), -cos(t)])$
```

Hier bekommt man die Rotationsmatrix für einen bestimmten Winkel  $\theta$

```
(%i89) getRotMat(theta):=block([t:float(theta*%pi/180),m],
 m:matrix([cos(t), -sin(t)], [sin(t), cos(t)]), m)$
```

Der rotierte Punkt wird zurückgegeben

```
(%i90) rotatePoint(P,theta):=block([m, P_r],
 m:getRotMat(theta),
 P_r: m . P,
 P_r)$
```

Hier wird einfach eine Liste der 5 Scannerpositionen bei  $\theta = 0$  generiert

```
(%i91) createSourceList():=block([src:[], ystart:-2],
 for j: 0 thru 4 do src:cons([-5, ystart+j], src),
 src)$
```

Ausprobiert!

```
(%i92) srcL:createSourceList();
```

```
(%o92) [[-5, 2], [-5, 1], [-5, 0], [-5, -1], [-5, -2]]
```

## 17. Computertomografie-Basics

---

Der Befehl `map` wendet eine Funktion auf alle Listenelemente an - in diesem Fall `rotatePoint(P, theta)` - und gibt diese Liste zurück.

Übergabeparameter sind der Winkel und die Punktliste

```
(%i93) rotListBy(theta,1):=map(lambda([P],rotatePoint(P,theta)),1)$
```

Verwandelt die übergebene  $5 \times 5$  Matrix in einen Zeilenvektor. Das brauchen wir um die Absorptionskoeff.-Matrix und Penetrationslängen-Matrix in Zeilen der Koeffizientenmatrix umzuformen. `cons` fügt ein Listenelement vorne an eine Liste. Wichtig ist die Reihenfolge:  $[a_{11}, a_{12}, \dots, a_{15}, a_{21}, \dots, a_{55}]$

```
(%i94) convert2List(M):=block([H:[]],
 for i: 1 thru 5 do
 for j: 1 thru 5 do
 H: cons(M[6-i,6-j],H),
 H)$
```

Probe - man kann mit *Geogebra* überprüfen, ob das passt!

```
(%i95) srcL_Rot:reverse(rotListBy(30,srcL));
```

```
(%o95) [(-3.330127018922194), (-3.830127018922194),
 (-4.232050807568877), (-3.366025403784438), ...]
```

$\overrightarrow{SP} \cdot \vec{n}_0$  wird berechnet, wobei  $P$  mit den Marixkoordinaten angegeben wird.  $S$  ist eine "Senderposition"

```
(%i96) getScalarProd(S,n0,i,j, position):=block([P, vecSP],
 P:getPoint(i, j, position),
 vecSP:P - S,
 n0 . vecSP)$
```

Hier wird jetzt bestimmt, welche Weglänge in einem Pixel zurückgelegt wird(die Fallunterscheidungen) - die Theorie dazu haben wir oben besprochen

```
(%i97) evalScalarProd(dr,ul,theta):=block([t:float(theta*%pi/180), pi2:float(%pi/2),
 pi4:float(%pi/4), pi34:float(3*%pi/4)],
 if (t > pi4) then t:pi2 - t else
 if (t > pi2) then t: t - pi2 else if (t > pi34) then t: 2*pi2 - t,
 if ((ul >= 0) or (dr <= 0)) then 0 else
 if (dr <= sin(t)) then dr*2/sin(2*t) else
 if (abs(ul) <= sin(t)) then abs(ul)*2/sin(2*t) else 1/cos(t))$
```

Hier wird jetzt für einen bestimmten Sender die Weglängenmatrix berechnet - die meisten Elemente werden natürlich Null sein

```
(%i98) getPenetrationMatrix(S,theta):=block([m:zeromatrix(5,5), n0],
 n0:getNormalUnitVec(theta),
 for i thru 5 do
 for j thru 5 do block(
 d_dr:getScalarProd(S,n0,i,j,down_right),d_ur:getScalarProd(S,n0,i,j,down_right),
 d_ul:getScalarProd(S,n0,i,j,up_left),d_dl:getScalarProd(S,n0,i,j,down_left),
 if (theta <= 90) then m[i,j]: evalScalarProd(d_dr, d_ul, theta)
 else m[i,j]: evalScalarProd(d_ur, d_dl, theta)), m)$
```

Hier wieder eine Probeausgabe, die man mit *Geogebra* verifizieren kann!

```
(%i99) getPenetrationMatrix(srcL_Rot[3],30);
```

```
(%o99)
$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & .7320 & 1.1547 \\ 0 & 0.4226 & 1.1547 & .4226 & 0 \\ 1.1547 & .73205 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

```

Hier haben wir das "Arbeitspferd" vor uns:

Auf die rotierte Senderliste wird die Funktion `getPenetrationMatrix(S,angle)` angewendet und diese Matrix wird mit `append` an die Liste `l` angehängt und diese dann zurückgegeben,

```
--> getMatrixList():=block([srcL:createSourceList(), l:[]],
 for angle:0 thru 175 step 5 do
 l:append(map(lambda([S],
 getPenetrationMatrix(S,angle)),rotListBy(angle,srcL)),l)
 ,l)$
```

Jetzt wird gearbeitet - die Matrizenlist wird erstellt!

```
(%i101)mList:getMatrixList()$
```

Die Liste der Penetrationslängen-Matrizen wird zur Koeffizientenmatrix umgestaltet. Schlüsselbefehl ist hier `addrow`: nimmt eine Matrix und fügt eine Zeile an

```
(%i102)createCoeffMatrix(ml):=block([m:matrix(convert2List(ml[1])), mat],
 for i:2 thru length(ml) do m:addrow(m, convert2List(ml[i])),
 m)$
```

Wird jetzt ausgeführt!

```
(%i103)MM:createCoeffMatrix(mList)$
```

Zur Sicherheit schauen wir uns den Rang der Matrix an - sind wir nicht unterbestimmt?

```
(%i104)rank(MM);
```

```
(%o104)25
```

Wir denken uns Versuchsdaten aus!

## 17. Computertomografie-Basics

---

```
(%i105)absorbMat:matrix(
 [0.1,0.2,0.3,0.4,0.5],
 [0.3,0.4,0.5,0.6,0.7],
 [0.5,0.6,0.7,0.8,0.9],
 [1.1,1.2,1.3,1.4,1.5],
 [1.9,2.0,2.1,2.2,5.3])$
```

Verwandeln sie in einen Vektor

```
(%i106)absorbVec:matrix(convert2List(absorbMat));
```

```
(%o106)(0.1 0.2 0.3 0.4 0.5 0.3 ... 2.0 2.1 2.2 5.3)
```

Bestimmen die “Ablesewerte”

```
(%i107)readingVec:MM . absorbVec$
```

Um den Median zu berechnen brauchen wir ein Paket

```
(%i108)load(descriptive)$
```

Wir fügen zu den exakten Daten zufällige Ablesefehler im Ausmaß von 5% bei 10 “Messungen” und nehmen dann aus dieser Liste den Medianwert

```
(%i117)addNoise(rv):=block(
 for i thru length(rv) do block([sign:0, readingList:[]],
 for j thru 10 do block(
 if (floor(random(2.0)) = 1) then sign:1 else sign: -1,
 readingList:cons(rv[i,1]+sign*random(abs(rv[i,1])/20.0+0.001), readingList)),
 rv[i,1]:median(readingList)),
 rv)$
```

Es ist soweit:  $Q - R$ -Faktorisierung. Jetzt heißt es warten!

```
(%i110)[q, r]: dgeqrf(MM)$
```

```
(%i111)b:addNoise(readingVec)$
```

Den Rest kennen wir von unserem Einführungsgleichungssystem

```
(%i112)q_t:transpose(q)$
```

```
(%i113)b_1:q_t . b$
```

```
(%i114)transposeAndRound(x):=block([x1:transpose(x), 1:[]],
 1:map(lambda([p], floor((100*p+0.5))/100.0), x1),
 1)$
```

```
--> backwardSubstitution(r,b):=block([cols:second(matrix_size(r)),
 x:zeromatrix(second(matrix_size(r)),1)],
 for c:cols thru 1 step -1 do x[c]:((b[c]-row(r,c) . x)/r[c,c]),
 transposeAndRound(x))$
```

Schaut nicht schlecht aus!

```
(%i116)solutionVec:backwardSubstitution(r,b_1);
```

```
(%o116)(0.09 0.2 0.31 0.41 0.5 ... 1.41 1.48 1.91 2.03 2.14 2.22 5.34)
```

Hier das vollständige Programm “zum runterladen”

Bis jetzt haben wir einige Aufgaben (Median, Q-R-Faktorisierung) *wxMaxima* ausführen lassen (packages: descriptive, lapack), aber wir können natürlich auch wissen wollen, was dahintersteckt:

Im nächsten Kapitel werden wir die Householder-Q-R-Faktorisierung und das Backtracking etwas genauer unter die Lupe nehmen.



# 18 | QR Faktorisierung

## 18.1 Die Householder-Transformation

Die Q-R-Faktorisierung mit Hilfe der Householder-Transformation ist bei *wxMaxima* in dem package “lapack” als Funktion “dgeqrf” implementiert, welches die Q- bzw. die R-Matrix in einer Liste zurückgibt. Sie ist numerisch etwas stabiler als das Gram-Schmidt Verfahren oder Givens-Rotationen (2 andere Möglichkeiten).

Zur Erinnerung: die Q-Matrix ist orthogonal also  $Q^t = Q^{-1}$  und  $R$  besteht aus einer oberen Dreiecksmatrix und einer Nullmatrix, d.h. alle Elemente unterhalb der Hauptdiagonale verschwinden:

$$R = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22} & a_{23} & \dots & a_{2n} \\ 0 & 0 & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & a_{nn} \\ 0 & \dots & \dots & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & \dots & \dots & 0 \end{pmatrix} = \begin{pmatrix} \hat{R} \\ 0 \end{pmatrix} \quad \text{wobei } \hat{R} \text{ eine obere Dreiecksmatrix ist}$$

Nocheinmal die Ausgangssituation kurz zusammengefasst:

Gegeben ist ein (womöglich) überbestimmtes Gleichungssystem

$$A \cdot x = b \quad \text{mit} \quad A \in \mathbb{R}^{m \times n} \wedge m \geq n, x \in \mathbb{R}^n, b \in \mathbb{R}^m, \text{Rang}(A) = n$$

Im allgemeinen wird es zu keiner eindeutigen Lösung  $x$  führen (überbestimmt), aber wir wären schon mit der Lösung des linearen Ausgleichsproblems zufrieden, nämlich wir finden ein  $x$  mit der Eigenschaft

$$|Ax - b|^2 \rightarrow \min$$

Weiter unten (nach Theorem 2) werden wir zeigen, dass die Q-R-Zerlegung genau so ein  $x$  liefert.

Gelingt nun eine Q-R-Zerlegung von  $A = Q \cdot R$  mit  $Q \in \mathbb{R}^{m \times m}$  und  $R \in \mathbb{R}^{m \times n}$  so ergibt sich:

$$Q \cdot R x = b \mid Q^t \cdot \Rightarrow I \cdot R x = Q^t \cdot b \Rightarrow R x = Q^t \cdot b$$

## 18. QR Faktorisierung

Die letzte Gleichung ist aber leicht durch Rücksubstituierung zu lösen, also  $x_n$  aus letzter Zeile, dann  $x_{n-1}$  aus vorletzter Zeile usw. bis schließlich  $x_1$  aus 1. Zeile.

Übrigens ist die Q-R-Zerlegung insofern eindeutig, dass sie immer die gleiche Dreiecksmatrix  $\hat{R}$  liefert, also

$$A = Q \cdot R = (Q_1, Q_2) \begin{pmatrix} \hat{R} \\ 0 \end{pmatrix} = Q_1 \hat{R}$$

d.h.  $Q_1$  ist eindeutig  $Q_2$  i.a. nicht. Für diese Eindeutigkeit brauchen wir nur die positive Definitheit der Diagonalelemente von  $\hat{R}$ .

$$A = Q_1 R_1 = Q_2 R_2 \Rightarrow \boxed{R_1^t R_1} = R_1^t (Q_1^t Q_1) R_1 = A_1^t A_1 = R_2^t (Q_2^t Q_2) R_2 = \boxed{R_2^t R_2}$$

aus

$$R_1^t R_1 = R_2^t R_2 \Rightarrow (R_2^{-1})^t R_1^t = R_2 R_1^{-1}$$

auf der linken Gleichungsseite stehen untere Diagonalmatrizen, auf der rechten Seite obere Diagonalmatrizen - also müssen sie Diagonalmatrizen sein. Seien  $\alpha_i$  die Diagonalelemente von  $R_1$  und  $\beta_i$  die Diagonalelemente von  $R_2$ , dann ergibt obige Matrixmultiplikation

$$\forall i \in \{1, 2, \dots, n\} \quad \frac{1}{\beta_i} \alpha_i = \beta_i \frac{1}{\alpha_i} \stackrel{\alpha_i, \beta_i \geq 0}{\Rightarrow} \alpha_i = \beta_i$$

Für die letzte Behauptung haben wir herangezogen, dass die Inverse einer Dreiecksmatrix wieder eine Dreiecksmatrix vom gleichen Typ ist (Untergruppe) und beim Invertieren sich der Reziprokwert in der Hauptdiagonale ergibt, dies kann man leicht einsehen, wenn man sich die Inverse als Unbekannte darstellt:

$$A X = I \Rightarrow x_{ii} = \frac{1}{a_{ii}}$$

Wir werden sehen, dass wir für eine erfolgreiche Zerlegung die Ausgangsmatrix  $A$  noch etwas verändern müssen (indem wir einige Zeilen vertauschen) - dies werden wir mit einer Permutationsmatrix  $P$  besorgen, sodass das Problem sich dann so stellt:

$$\underbrace{P \cdot A}_{A'} \underbrace{P \cdot x}_{x'} = \underbrace{P \cdot b}_{b'} \Rightarrow R x' = Q^t b'$$

aus dem Lösungsvektor  $x'$  lässt sich aber wieder leicht  $(P \cdot P^t = I) x$  berechnen! Jetzt bleibt nur mehr zu zeigen, dass diese Lösung  $x$  ganz gut durch das Ausgangsgleichungssystem "laviert" - das ja im strengen Sinn widersprüchlich ist, weil die Messwerte  $b$  ja mit Fehlern behaftet sind! Dazu benutzen wir folgendes

**Theorem 18.1**

Ist  $Q$  orthogonal,  $Q \in \mathbb{R}^{n \times n}$  und  $u, v \in \mathbb{R}^n \Rightarrow (Qu) \cdot (Qv) = u \cdot v$

*Beweis:*

$$(Qu) \cdot (Qv) = (Qu)^t(Qv) = u^t \underbrace{Q^t Q}_I v = u \cdot v$$

□

**Theorem 18.2**

Ist  $Q$  orthogonal,  $Q \in \mathbb{R}^{n \times n}$  und  $u \in \mathbb{R}^n \Rightarrow |(Qu)| = |u|$   
wobei  $|\cdot|$  die euklidische Norm ist.

*Beweis:*  $v := u$  in Theorem 18.1

□

Mit dieser Eigenschaft lässt sich jetzt das lineare Ausgleichsproblem umformen:

$$|Ax - b|^2 = |QAx - (QQ^t)b|^2 \stackrel{Q^t \text{ ist orthogonal}}{=} |Rx - Q^t b|^2 = \left| \begin{pmatrix} \hat{R}x \\ 0 \end{pmatrix} - \begin{pmatrix} c \\ d \end{pmatrix} \right|^2 = |\hat{R}x - c|^2 + |d|^2$$

Diese Summe wird offensichtlich ein Minimum, wenn wir das Dreieckssystem  $\hat{R}x - c = 0$  lösen und der Wert des Minimums sind die Quadrate von  $(Q^t b)_j$   $j \in \{n+1, n+2, \dots, m\}$ . Mit variieren von  $x$  lässt sich das Minimum nicht weiter verringern, weil  $d$  nicht von  $x$  abhängt. In diesem Anhang versuchen wir die “blackbox” *dgeqrf* in eine “whitebox” zu verwandeln. Dazu werden wir uns etwas theoretisches Rüstzeug zulegen und das ganze Verfahren dann in *wxMaxima* implementieren.

**Definition 18.3 Householder Matrix**

Sei  $\vec{w} \in \mathbb{R}^n$  ein Einheitsspaltenvektor, d. h. es gilt  $\vec{w}^t \cdot \vec{w} = 1$ .

Eine  $n \times n$  Matrix  $P$  mit der Eigenschaft

$$P = I - 2 \vec{w} \vec{w}^t \quad p_{ij} = \delta_{ij} - 2 w_i w_j$$

heißt Householder Matrix.

Beachte: Während  $\vec{w}^t \cdot \vec{w}$  das innere Produkt darstellt (Zahl), ist  $\vec{w} \vec{w}^t$  das äußere Produkt ( $n \times n$  Matrix).

**Theorem 18.4**

Ist  $P$  eine Householder-Matrix  $\Rightarrow P$  ist symmetrisch und orthogonal ( $P = P^t = P^{-1}$ )

*Beweis:*

$$P^t = (I - 2 \vec{w} \vec{w}^t)^t = I^t - (2 \vec{w} \vec{w}^t)^t \stackrel{(A \cdot B)^t = B^t \cdot A^t}{=} I - 2 (\vec{w}^t)^t \vec{w} = P \Rightarrow P \text{ ist symmetrisch}$$

$$\begin{aligned} P \cdot P^t &\stackrel{P^t=P}{=} (I - 2 \vec{w} \vec{w}^t) (I - 2 \vec{w} \vec{w}^t) = I - 4 \vec{w} \vec{w}^t + 4 \vec{w} \vec{w}^t \vec{w} \vec{w}^t \\ &= I - 4 \vec{w} \vec{w}^t + 4 \vec{w} \underbrace{(\vec{w}^t \vec{w})}_{=1} \vec{w}^t = I \Rightarrow P \text{ ist orthogonal} \end{aligned}$$

in Tensorschreibweise ist die Symmetrie eine Folge der Kommutativität in  $\mathbb{R}$  und der Symmetrie von  $\delta_{ij}$ :

$$p_{ij} = \delta_{ij} - 2 w_i w_j = p_{ji}$$

auch die Orthogonalität ist relativ einfach zu beweisen:

$$p_{ik} p_{jk} = (\delta_{ik} - 2 w_i w_k)(\delta_{jk} - 2 w_j w_k) = \delta_{ik} \delta_{jk} - 2 \delta_{jk} w_i w_k - 2 \delta_{ik} w_j w_k + 4 w_i w_j \underbrace{w_k w_k}_1 = \delta_{ij}$$

□

**Theorem 18.5**

Sind  $A$  und  $B$  orthogonal  $\Rightarrow A \cdot B$  ist orthogonal

*Beweis:* Wird dem Leser überlassen □

**Theorem 18.6**

$\vec{w}$  ist ein Eigenvektor der Householder-Matrix  $P = I - 2 \vec{w} \vec{w}^t$

*Beweis:*

$$P\vec{w} = (I - 2 \vec{w} \vec{w}^t) \vec{w} = \vec{w} - 2 \vec{w} \underbrace{(\vec{w}^t \vec{w})}_{=1} = -\vec{w}$$

in Tensorschreibweise:

$$p_{ij} w_j = (\delta_{ij} - 2 w_i w_j) w_j = \delta_{ij} w_j - 2 w_i \underbrace{w_j w_j}_1 = -w_i$$

□

**Theorem 18.7**

Jeder zu  $\vec{w}$  orthogonale Vektor  $\vec{x}$  ist ein Fixpunkt (Eigenvektor mit Eigenwert 1) der Householder-Matrix  $P = I - 2 \vec{w} \vec{w}^t$

*Beweis:*

$$P\vec{x} = (I - 2 \vec{w} \vec{w}^t) \vec{x} = \vec{x} - 2 \vec{w} \underbrace{(\vec{w}^t \vec{x})}_{=0} = \vec{x}$$

in Tensorschreibweise:

$$p_{ij} x_j = (\delta_{ij} - 2 w_i w_j) x_j = \delta_{ij} x_j - 2 w_i \underbrace{w_j x_j}_0 = x_i$$

□

## 18. QR Faktorisierung

Im Folgenden sei  $w := \vec{w}$ ; wir lassen die Vektorpfeile weg, weil es nicht zu Verwechslungen führen kann.

### Theorem 18.8

Sei  $x, y \in \mathbb{R}^n$  und  $|x| = |y|$  und  $w = \frac{x-y}{|x-y|}$  der Householder-Matrix  $P = I -$

$$2ww^t \Rightarrow \boxed{Px = y}$$

*Beweis:*

$$\begin{aligned} Px = y &\Leftrightarrow x - 2 \frac{x-y}{|x-y|} \frac{(x-y)^t}{|x-y|} x = y \Leftrightarrow \\ &\Leftrightarrow (x-y) - 2 \frac{(x-y)(x-y)^t x}{|x-y|^2} = 0 \Leftrightarrow \\ &\Leftrightarrow \frac{(x-y)}{|x-y|^2} (|x-y|^2 - 2(x-y)^t x) = 0 \Leftrightarrow \\ &\Leftrightarrow \frac{(x-y)}{|x-y|^2} ((x-y)^t(x-y) - 2(x-y)^t x) = 0 \Leftrightarrow \\ &\Leftrightarrow \frac{(x-y)}{|x-y|^2} (\cancel{x^t x} - x^t y - y^t x + \cancel{y^t y} - 2\cancel{x^t x} + 2y^t x) = 0 \\ &\Leftrightarrow \frac{(x-y)}{|x-y|^2} (-\cancel{x^t y} - \cancel{y^t x} + 2y^t x) = 0 \end{aligned}$$

Die letzte Zeile gilt wegen der Kommutativität des skalaren Produkts  $\vec{x} \cdot \vec{y} = \vec{y} \cdot \vec{x}$ , die vorletzte wegen  $|x| = |y|$ . Wenn die letzte Zeile also wahr ist, können wir zurückschließen auf die 1. Zeile - was die Behauptung ist!

In Tensorschreibweise:

$$\left( \delta_{ij} - 2 \frac{(x_i - y_i)(x_j - y_j)}{x_k x_k - 2x_k y_k + y_k y_k} \right) x_j = y_i \Rightarrow \text{mit } x_k x_k = y_k y_k$$

$$2(\cancel{x_i - y_i})(x_k x_k - x_k y_k) = 2(\cancel{x_i - y_i})(x_j - y_j) x_j$$

Die letzte Gleichheit folgt wieder aus der Kommutativität des skalaren Produkts.  $\square$

**Beispiel 18.9**

Wir haben nun ein Verfahren, um mit einer Householder-Matrix (symmetrisch und orthogonal) alle Komponenten bis auf eine zum Verschwinden zu bringen:

$$P = P^t = P^{-1}$$

$$x = (1, 1, 3, 3, 4)^t \xrightarrow{\uparrow} y = (y_1, 0, 0, 0, 0)^t$$

Nachdem die Normen der beiden Vektoren übereinstimmen müssen, muss gelten:

$$y_1 = 6.$$

Führen wir das in *wxMaxima* durch:

```
(%i1) x:[1,1,3,3,4]$
(%i2) y:[6,0,0,0,0]$
(%i3) norm2(x):=sqrt(x . x)$
(%i4) w1:(x-y)/norm2(x-y);
(%o4) [-5/2*sqrt(15), 1/2*sqrt(15), 3/2*sqrt(15), 3/2*sqrt(15), 2/sqrt(15)]
(%i5) vec2Matrix(x):=block([1:length(x), m],
 m:zeromatrix(1,1),
 for i thru 1 do m[i,1]:x[i],
 m
)$
(%i6) w:vec2Matrix(w1);
(%o6) (
 -5/2*sqrt(15)
 1/2*sqrt(15)
 3/2*sqrt(15)
 3/2*sqrt(15)
 2/sqrt(15)
)
(%i7) w_t:transpose(w);
(%o7) (-5/2*sqrt(15) 1/2*sqrt(15) 3/2*sqrt(15) 3/2*sqrt(15) 2/sqrt(15))
(%i8) outerPr:w . w_t;
(%o8) (
 5/12 -1/12 -1/4 -1/4 -1/3
 -1/12 60 1/20 1/20 1/15
 -1/4 1/20 3/20 3/20 1/5
 -1/4 1/20 3/20 3/20 1/5
 -1/3 1/15 1/5 1/5 4/15
)
(%i9) I:diagmatrix(5,1)$
(%i10) P:I-2*outerPr;
```

## 18. QR Faktorisierung

```
(%o10)
$$\begin{pmatrix} \frac{1}{6} & \frac{1}{6} & \frac{1}{2} & \frac{1}{2} & \frac{2}{3} \\ \frac{1}{6} & \frac{29}{30} & -\frac{1}{10} & -\frac{1}{10} & -\frac{1}{15} \\ \frac{1}{2} & -\frac{1}{10} & \frac{7}{10} & -\frac{3}{10} & -\frac{2}{5} \\ \frac{1}{2} & -\frac{1}{10} & -\frac{3}{10} & \frac{7}{10} & -\frac{2}{5} \\ \frac{2}{3} & -\frac{2}{15} & -\frac{2}{5} & -\frac{2}{5} & \frac{7}{15} \end{pmatrix}$$

(%i11) transpose(P . vec2Matrix(x));
```

```
(%o11) (6 0 0 0 0)
```

Den letzten Vektor haben wir transponiert, um Platz zu sparen. Hat doch super geklappt - nun weiter.

Da es bei Matrixumformungen meist darum geht, Spaltenvektoren zu erzeugen, die bis auf die 1. Komponente verschwinden, wird in der mathematischen Literatur obiges Theorem meist gleich mit

$$y = -\operatorname{sgn}(x_1)|x|e_1 \quad \text{wobei } e_1 := (1, 0, 0, \dots, 0)^t$$

angegeben. Allerdings muss gelten  $x_1 \neq 0$  (d.h. Hauptdiagonale besetzt). Es ist klar, dass  $|x| = |y|$  erfüllt ist, die Householder-Matrix ergibt sich dann

$$P = I - \underbrace{\frac{2}{v^t v}}_{\beta} v v^t \quad \text{mit } v := x - y = x + \underbrace{\operatorname{sgn}(x_1)|x|}_{\alpha} e_1$$

### 18.2 Implementation in *wxMaxima*

```
(%i1) signValue(r) := block([s:sign(r)], /* sgn-Fkt wird erzeugt */
if s='pos then 1 else if s='zero then 0 else -1)$
```

ematrix(n,m,x,i,j) – erzeugt eine  $(n \times m)$ -Matrix, die überall verschwindet, nur an der Position  $(i, j)$  steht  $x$

```
(%i2) unitVector(n) := ematrix(n,1,1,1,1)/* Einheitsvektor in x-Richtung */$
```

```
(%i3) householder(A) := block([m : length(A),
alpha,v,beta, a:col(A,1)],
alpha : signValue(A[1,1])*sqrt(a . a),
v : a + alpha*unitVector(m),
beta : 2/(v . v),
diagmatrix(m,1) - beta*(v . transpose(v))$
```

$$v = x + \operatorname{sgn}(x_1)|x|e_1$$

Der 2-te Spaltenvektor von  $A$  ist nur ein “dummy”

```
(%i4) A:matrix([1,0], [1,0], [3,0], [3,0], [4,0])$
```

```
(%i5) P:householder(A)$
```

$P \cdot A$  liefert in der 1.-ten Spalte  $y = -\text{sgn}(x_1)|x|e_1 = (-6, 0, 0, 0, 0)^t$

```
(%i6) P . A;
```

```
(%o6)
$$\begin{pmatrix} -6 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{pmatrix}$$

```

Die numerische Stabilität des Algorithmus ist natürlich verbesserbar - aber vernachlässigen wir das fürs erste.

Betrachten wir noch einmal die Matrixmultiplikation (mit der Einsteinschen Summationskonvention: über alle mehrfach vorkommenden Indices wird summiert!)

$$c_{ij} := a_{ik} b_{kj}$$

umgedeutet auf Vektoren bedeutet dies:

$$\underbrace{c_{ij}}_y := \underbrace{q_{ik}}_P \underbrace{a_{kj}}_x$$

Für den  $j$ -ten Spaltenvektor der Matrix  $A$  gibt es eine Householder-Matrix  $P$ , sodass der  $j$ -te Spaltenvektor der Produktmatrix obige Eigenschaft (nur 1 Komponente ungleich 0) besitzt. Damit können wir folgendes Verfahren durchführen:

1. Wir formen mit Hilfe des Backtracking Algorithmus (Anhang C) die Ausgangsmatrix  $A$  so um, dass die Hauptdiagonale besetzt ist. Dies muss möglich sein, sonst wären in einer Spalte nur Nullen und das würde bedeuten das der Rang nicht  $n$  sein kann! Um es am Anfang nicht gleich zu verkomplizieren, gehen wir davon aus  $A$  würde diese Bedingung bereits erfüllen.
2. Wir wenden auf den 1. Spaltenvektor von  $A$  die Householder-Matrix  $Q_1$  an, sodass nur die 1. Komponente nicht verschwindet (wie bei unserem oberen Beispiel).
3. Wir streichen von  $A$  1. Zeile und 1. Spalte und erhalten  $A_2$
4. Wir wenden auf den 1. Spaltenvektor von  $A_2$  die Householder-Matrix  $Q_2$  an, sodass nur die 1. Komponente nicht verschwindet.
5. Wir "blähen"  $Q_2$  zur ursprünglichen Größe auf (Ergebnis  $Q'_2$ ), sodass andere Spalten unbeeinflusst bleiben.
6. Wir gehen weiter bei Punkt 3) mit einem höherem Index-Zähler

7. Wir brechen ab, wenn wir bei der letzten Spalte angelangt sind

Wir haben also folgendes erreicht:

$$\underbrace{Q'_n Q'_{n-1} Q'_{n-1} Q'_{n-2} \dots Q'_2 Q_1}_Q A = R = \begin{pmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22} & a_{23} & \dots & a_{2n} \\ 0 & 0 & a_{33} & \dots & a_{3n} \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & 0 & a_{nn} \\ 0 & \dots & \dots & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & \dots & \dots & \dots & 0 \end{pmatrix}$$

wobei  $Q$  als Produkt von orthogonalen Matrizen selbst orthogonal ist (Theorem 4), daher gilt

$$A = Q^t R$$

Jetzt müssen wir nur mehr Punkt 5) unseres Verfahrens näher erläutern: das ‘‘Aufblähen’’ der Householder-Matrix. Dazu müssen wir wissen, dass die Multiplikationsformel für Matrizen auch für die Unterteilung in Blockmatrizen gilt, also:

$$\left( \begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array} \right) \cdot \left( \begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array} \right) = \left( \begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array} \right) \quad \text{wobei } C_{ij} := A_{ik} \cdot B_{kj}$$

wobei in der letzten Formel Einsteinsche Summationskovention gilt und das Multiplikationszeichen eine Matrixmultiplikation darstellt. Zeilen- und Spaltenanzahl der einzelnen Matrizen müssen natürlich derart sein, dass die Produkte existieren!

Zurück zu unserem Verfahren: durch Streichen der linken Spalte und oberen Zeile ergibt sich z.B.: im vierten Schritt folgende Situation:

$$\left( \begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & Q_4 \cdot A_{22} \end{array} \right)$$

wobei  $A_{11}$  eine  $3 \times 3$  (obere Dreiecks-)Matrix,  $A_{21}$  eine  $(m - 3) \times 3$  Null-Matrix,  $A_{12}$  eine  $3 \times (n - 3)$  Matrix und  $A_{22}$  eine  $(m - 3) \times (n - 3)$  Matrix (wobei  $a_{11}$  nicht verschwinden darf!). Unser Verfahren liefert  $Q_4$  wir benötigen aber  $Q'_4$ , welche bei Multiplikation  $A_{11}$ ,  $A_{12}$  und  $A_{21}$  unverändert lässt - dass lässt sich aber leicht mit folgender Matrix erreichen:

$$\underbrace{\left( \begin{array}{c|c} I & 0 \\ \hline 0 & Q_4 \end{array} \right)}_{Q'_4} \cdot \left( \begin{array}{c|c} A_{11} & A_{12} \\ \hline 0 & A_{22} \end{array} \right)$$

Ausrechnen ergibt obiges Ergebnis!

Also zusammengefasst: Aus  $Q'_i$  wird  $Q_i$  indem man in der Hauptdiagonale soviele ‘‘1’’-er hinzufügt, bis man die ursprüngliche Größe ( $m \times m$ ) erreicht hat - alles andere wird mit ‘‘0’’ aufgefüllt.

Wir berechnen mit diesem Verfahren ein überbestimmtes Gleichungssystem - wie immer mit *wxMaxima*:

Altbekanntes ....

```
(%i28) signValue(r) := block([s:sign(r)],
 if s='pos then 1 else if s='zero then 0 else -1)$

(%i29) unitVector(n) := ematrix(n,1,1,1,1)$

(%i30) householder(A) := block([m : length(A),alpha,v,beta, a:col(A,1)],
 alpha : signValue(A[1,1])*sqrt(a . a),
 v : a + alpha*unitVector(m),
 beta : 2/(v . v),
 diagsmatrix(m,1) - beta*(v . transpose(v)))$
```

Hier jetzt das Zurücksetzen auf die ursprüngliche Größe *size*, *M* ist die *Q*-Matrix und *s* der “Schrumpfungsgangrad” der Matrix, es gilt:  $0 \leq s < size$

```
(%i31) setOrigSize(M,s, size):=block(
 genmatrix(lambda([i,j], if (i>s and j>s) then M[i-s,j-s]
 else if (i=j) then 1
 else 0),size,size))$
```

Von der Matrix *A* wird *j*-mal 1-te Zeile und 1-te Spalte gelöscht

```
(%i32) getSubMatrix(A,j):=block([M:A],
 for i thru j do M:=submatrix(1,M,1),
 M)$
```

Matrixelemente die kleiner als *threshold* werden, werden durch Null ersetzt!

```
(%i33) setZero(M):=block([mat:M, m:first(matrix_size(M)), n:second(matrix_size(M)),
 threshold:10^(-15)],
 for i thru m do
 for j thru n do
 if (abs(mat[i,j]) < threshold) then mat[i,j]:0,
 mat)$
```

Hier jetzt die Debugging-Version des rekursiven Householder-Algorithmus:

Input ist die bisherige “obere Dreiecksmatrix” *R*, in der *recNr* Spalten unterhalb der Hauptdiagonalen Null gesetzt sind, *Q* ist das Produkt der einzelnen Householder-Matrizen, *recNrHalt* ist ein “Breakpoint” (for debugging only), und *origSize* ist die Dimension von *Q*, oben  $m \times m$  genannt!

```
(%i34) getQR_debug(R,Q,recNr,recNrHalt,origSize):=block([subMat, q],
 if recNr < recNrHalt then block(
 subMat:getSubMatrix(R, recNr),
 q: householder(subMat),
 q: setOrigSize(q, recNr, origSize),
 getQR_debug(q . R, Q . q, recNr+1, recNrHalt, origSize)
)
 else [Q,setZero(R)]$
```

## 18. QR Faktorisierung

---

Hier jetzt die eigentliche Q-R-Faktorisierung:  $R := A$ ,  $Q := I$  und  
*recNrHalt* := alle n-Spalten von  $A$ , *origSize* =  $m$

```
(%i35) Q_R_Fact(A):=block([m:first(matrix_size(A)), n:second(matrix_size(A))],
 getQR_debug(A,diagmatrix(m,1),0, n, m))$
```

Hier eine Koeffizientenmatrix mit Rang 4,  $m = 5$ ,  $n = 4$

```
(%i36) A:matrix(
 [2,1,0,0],
 [1,1,0,0],
 [0,0,1,1],
 [0,0,3,2],
 [0,0,0,1]
)$
```

Wir berechnen  $Q$  und  $R$  und schauen uns  $R$  an(obere Dreiecksmatrix):

```
(%i37) [q,r]:float(Q_R_Fact(A))$
```

```
(%i38) r;
```

```
(%o38)
$$\begin{pmatrix} -2.23606797749979 & -1.341640786499874 & 0.0 & 0.0 \\ 0.0 & -.4472135954999579 & 0.0 & 0.0 \\ 0.0 & 0.0 & -3.162277660168379 & -2.213594362117866 \\ 0.0 & 0.0 & 0.0 & 1.048808848170151 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

```

Wir setzen den Ergebnisvektor  $\vec{b}$  so, dass sich ein Lösungsvektor  $\vec{x} = (1, 2, 3, 4)^t$  ergibt

```
(%i39) b:matrix([4],[3],[7],[17],[4])$
```

Durch "Rückwärtseinsetzen" wird die Lösung bestimmt:

```
(%i40) backwardSubstitution(r,b):=
 block([cols:second(matrix_size(r)), x:zeromatrix(second(matrix_size(r)),1)],
 for c:cols thru 1 step -1 do x[c]:((b[c]-row(r,c) . x)/r[c,c]),
 x)$
```

```
(%i41) solutionVec:backwardSubstitution(r, transpose(q) . b);
```

```
(%o41)
$$\begin{pmatrix} 1.0 \\ 2.0 \\ 3.0000000000000001 \\ 4.0 \end{pmatrix}$$

```

Jetzt wird der Ergebnisvektor abgeändert, dass er widersprüchlich wird:

```
(%i42) b:matrix([4.5],[3],[7.5],[16],[3.4])$
```

Neuer Lösungsvektor wird bestimmt

```
(%i43) solutionVec:backwardSubstitution(r, transpose(q) . b);
```

$$(%o43) \begin{pmatrix} 1.5 \\ 1.5 \\ 2.972727272727274 \\ 3.681818181818181 \end{pmatrix}$$

Erfüllt dieser Lösungsvektor die *Normalengleichung*? (Theorie siehe unten im Text!)

```
(%i44) transpose(A) . b;
```

$$(%o44) \begin{pmatrix} 12.0 \\ 7.5 \\ 55.5 \\ 42.9 \end{pmatrix}$$

Offensichtlich

```
(%i46) transpose(A) . (A . solutionVec);
```

$$(%o46) \begin{pmatrix} 12.0 \\ 7.5 \\ 55.500000000000001 \\ 42.9 \end{pmatrix}$$

## 18.3 Das lineare Ausgleichsproblem

Wir haben ein überbestimmtes lineares Gleichungssystem

$$a_{ij} x_j = b_i \quad 1 \leq i \leq m, 1 \leq j \leq n, n \leq m$$

Wir suchen  $\hat{x}_j$ , sodass die Summe der Residuenquadrate minimal wird:

$$r_i = b_i - a_{ij} x_j \quad S = r_i r_i \rightarrow \min$$

Dazu muss der Gradient von  $S$  verschwinden:

$$\frac{\partial S}{\partial x_j} = \frac{\partial r_i}{\partial x_j} \cdot r_i + r_i \cdot \frac{\partial r_i}{\partial x_j} = 2 r_i \cdot \frac{\partial r_i}{\partial x_j} \stackrel{\frac{\partial r_i}{\partial x_j} = -a_{ij}}{\uparrow} = 2 (b_i - a_{ik} \hat{x}_k) (-a_{ij}) = 0$$

Ausmultiplizieren führt zu den Normalgleichungen

$$a_{ij} a_{ik} \hat{x}_k = b_i a_{ij}$$

in Matrixschreibweise

$$A^t A \vec{x} = A^t \vec{b}$$

Dies haben wir oben in *wxMaxima* durchgeführt!

### 18.4 ANHANG Backtracking

Der Algorithmus für Backtracking erinnert an die Aufgabe in einem Labyrinth vom Eingang zum Ausgang zu kommen, indem man den Eingang mit Wasser flutet und am Ausgang wartet bis das Wasser herauskommt. Anschließend verfolgt man die Spur des Wassers zurück (“Backtracking”). Das ist zwar in der Praxis kein gangbarer Weg aber falls man ein Modell des Labyrinths besitzt schon.

Falls ein Computer ein Problem lösen soll (welches als Labyrinth modelliert werden kann), muss es ein mathematisches Modell geben und der Computer kann nach obigem Algorithmus vorgehen - mit dem Unterschied allerdings, dass er bei einer Verzweigung nicht gleichzeitig alle Wege weiterbeschreitet, sondern eine Möglichkeit nach der anderen durchschreitet (Es ist Aufgabe der Compiler, diese Aufgabe auf mehrere Prozessorkerne zu parallelisieren - der Programmierer sollte sich nicht darum kümmern müssen).

Hier einmal der Algorithmus:

```
boolean FindeLoesung(int index, Lsg loesung, ...) {
 // index ist die aktuelle Schrittzahl
 // Teillösungen loesung werden als Referenz übergeben.
 1. Solange es noch neue Teil-Lösungsschritte gibt:
 a) Wähle einen neuen Teil-Lösungsschritt schritt; // Heuristik
 b) Falls schritt gültig ist:
 I) Erweitere loesung um schritt;
 II) Falls loesung vollständig ist, return true, sonst:
 if (FindeLoesung(index+1, loesung)) { // rekursiv
 return true; // Lösung gefunden
 } else { // Wir sind in einer Sackgasse
 Mache schritt rückgängig; // Backtracking
 }
 }
 }
 2. Gibt es keine neuen Teil-Lösungsschritte mehr, dann: return false
}
```

Wie bei jedem nicht trivialen rekursiven Algorithmus ist es schwer den Überblick zu bewahren. Am besten man denkt die Fälle mit Hilfe eines Diagramms durch. Einen Einblick gibt es bei [http://www.swisseduc.ch/informatik/diskrete\\_mathematik/backtracking/demos.html](http://www.swisseduc.ch/informatik/diskrete_mathematik/backtracking/demos.html).

Wozu brauchen wir Backtracking? Damit unser Q-R-Algorithmus funktioniert darf kein Hauptdiagonalelement unserer Matrix verschwinden. Dies muss selbstverständlich möglich sein, sonst wären bei einer Spalte alle Elemente Null, damit bricht der Rang der Matrix auf kleiner

$n$  ein. Am besten wir machen ein Beispiel:

$$A = \begin{pmatrix} 2 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Dass  $\text{Rang}(A) = 4$  ist leicht ersichtlich. Sie eignet sich aber nicht für unser Verfahren, weil in Zeile 2 und 4 die Hauptdiagonalen nicht besetzt sind - also müssen wir umordnen. Umordnen macht man mit einer Permutationsmatrix. Wie diese beschaffen sind schaut man sich z.B. in der Wikipedia an.

Im Wesentlichen kann man sich den Aufbau so erklären: Sei  $e_i = (0, 0, \dots, 1, 0, \dots, 0)$  die Verallgemeinerung des kartesischen Basisvektors, dann gilt für jeden Spaltenvektor  $b$ :

$$e_i b = b_i \quad \text{hat man k Spaltenvektoren zur Verfügung } e_i b_k = b_{ik}$$

also das Ergebnis des Vektorprodukts ist der Skalar in der  $i$ -ten Zeile.

Denken wir uns jetzt bei einem Matrixprodukt  $A \cdot B$  die Matrix  $A$  aus Zeilenvektoren  $a_i$  und  $B$  aus Spaltenvektoren  $b_k$  aufgebaut. Die  $i$ -te Zeile der Produktmatrix entsteht dann durch

$$a_i b_k = c_{ik} \quad \text{wobei } i \text{ fest ist und } k \text{ durchläuft}$$

Wenn wir jetzt bei dieser Operation  $a_i$  durch  $e_j$  (beachte die verschiedene Bedeutung der Indices!) ersetzen, entsteht nach obiger Regel

$$e_j b_k = c_{jk} = b_{jk}$$

dadurch ist die  $j$ -te Zeile von  $B$  in die  $i$ -te gewandert. Analoges machen wir für die  $j$ -te Zeile von  $A$  - wir ersetzen sie durch  $e_i$ . Den Rest der Matrix  $A$  nehmen wir von der Einheitsmatrix - damit bleibt alles unverändert. Oder man macht es wie im nachfolgenden Programm - man vertauscht die  $i$ -te und  $j$ -te Zeile der Einheitsmatrix.

Um die 1.-te und 2.-te Zeile von  $A$  zu vertauschen, brauchen wir bei der Einheitsmatrix  $I$  auch nur die 1.-te und 2.-te Zeile zu vertauschen, um die Permutationsmatrix zu erhalten - das sieht man leicht an folgendem Code-Snippet aus *wxMaxima*:

```
(%i7) A:matrix(
 [2,1,0,0],
 [1,0,0,0],
 [0,0,1,1],
 [0,0,3,0],
 [0,0,0,1]
);
```

## 18. QR Faktorisierung

---

$$(\%o7) \begin{pmatrix} 2 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

```
(%i8) swap(k,1,A):=block([m:first(matrix_size(A)), M,B],
 M:diagmatrix(m,1),
 B:copymatrix(M),
 for j thru m do B[k,j]:M[1,j], /* swap row 1 with row k */
 for j thru m do B[1,j]:M[k,j],
 B)$
```

```
(%i9) P:swap(1,2,A);
```

$$(\%o9) \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

```
(%i10) P . A;
```

$$(\%o10) \begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Wird nur eine Vertauschung durchgeführt (elementare Permutation) ergeben sich sofort einige Eigenschaften:

1.  $P$  ist symmetrisch  $\Rightarrow P^t = P$
2. Zwei Vertauschungen führen auf die Identität:  $P \cdot P = I \Rightarrow P^{-1} = P$
3. Aus obigen Eigenschaften folgt unmittelbar:  $P \cdot P^t = I \Rightarrow P$  ist orthogonal.

Durch Ausmultiplizieren können nun sämtliche Vertauschungen durchgeführt werden und die Orthogonalität bleibt erhalten.

Wir implementieren jetzt Backtracking in *wxMaxima* und lösen obiges lineares Gleichungssystem mit Rücksubstitution.

Einige Besonderheiten:

- Die Funktion  $setZero(M)$  setzt alle Matrixelemente von  $M$ , die kleiner als  $10^{-15}$  sind Null. Damit kann
- die Funktion  $nonZeroEntries(M,col)$  überprüfen kann, in welcher Spalte keine Nullelemente vorhanden sind und

- die Funktion *upperRowsUntouched(list,k)* sucht aus dieser Liste jene aus, die noch vertauscht werden dürfen - die oberen Zeilen sind ja schon abgearbeitet.
- Eine Eigenheit von *wxMaxima*: der Befehl *return(var)* verlässt nur den Block (also bei uns die for-Schleife) und nicht die *procedure*  
Der Befehl *throw(var)* verlässt zwar die *procedure*, muss aber im aufrufenden Programmteil mit *catch(procedure)* “abgefangen” werden.  
Beide Möglichkeiten wurden mit *btrDiag(k,n)* und *btrDev(k,n)* implementiert, wobei *k* die Rekursionstiefe und *n* die Spaltenanzahl der Matrix ist. Das Ziel ist also erreicht, wenn die Bedingung ( $k = n$ ) wahr ist.

```
(%i21) signValue(r) := block([s:sign(r)],
 if s='pos then 1 else if s='zero then 0 else -1)$

(%i22) unitVector(n) := ematrix(n,1,1,1,1)$

(%i23) householder(A) := block([m : length(A),alpha,v,beta, a:col(A,1)],
 alpha : signValue(A[1,1])*sqrt(a . a),
 v : a + alpha*unitVector(m),
 beta : 2/(v . v),
 diagmaatrix(m,1) - beta*(v . transpose(v)))$

(%i24) setOrigSize(M,s, size):=block(
 genmatrix(lambda([i,j], if (i>s and j>s) then M[i-s,j-s]
 else if (i=j) then 1
 else 0),size,size))$

(%i25) getSubMatrix(A,j):=block([M:A],
 for i thru j do M:submatrix(1,M,1),
 M)$

(%i26) setZero(M):=block([mat:M, m:first(matrix_size(M)), n:second(matrix_size(M)), threshold:10^(-15)],
 for i thru m do
 for j thru n do
 if (abs(mat[i,j]) < threshold) then mat[i,j]:0,
 mat)$

(%i27) getQR_debug(R,Q,recNr,recNrHalt,origSize):=block([subMat, q],
 if recNr < recNrHalt then block(
 subMat:getSubMatrix(R, recNr),
 q: householder(subMat),
 q: setOrigSize(q, recNr, origSize),
 getQR_debug(q . R, Q . q, recNr+1, recNrHalt, origSize)
)
 else [Q,setZero(R)]$
```

## 18. QR Faktorisierung

---

```
(%i28) Q_R_Fact(A):=block([m:first(matrix_size(A)), n:second(matrix_size(A))],
 getQR_debug(A,diagmatrix(m,1),0, n, m))$
```

```
(%i29) A:matrix(
 [2,1,0,0],
 [1,0,0,0],
 [0,0,1,1],
 [0,0,3,0],
 [0,0,0,1]
)$
```

```
(%i30) nonZeroEntries(M,col):=block([m:first(matrix_size(M)), rowList:[]],
 M:setZero(M),
 for i thru m do
 if M[i,col] # 0 then rowList:cons(i, rowList),
 reverse(rowList))$
```

```
(%i31) upperRowsUntouched(list,k):=sublist(list, lambda ([x], x >= k))$
```

Erstellen der elementaren Permutationsmatrix - wie weiter oben beschrieben

```
(%i32) swap(k,l,A):=block([m:first(matrix_size(A)), M,B],
 M:diagmatrix(m,1),
 B:copymatrix(M),
 for j thru m do B[k,j]:M[l,j], /* swap row l with row k */
 for j thru m do B[l,j]:M[k,j],
 B)$
```

Hier Backtracking mit "Ausbruch" aus der for-Schleife

```
(%i33) btrDiag(k,n):=block([choices, P_saved,M_saved, sol_found:false, goAhead:false],

 choices:upperRowsUntouched(nonZeroEntries(M,k),k),
 for i:1 while i <= length(choices) do (
 P_saved:P, M_saved:M, /* save step back */
 Pm: swap(k, choices[i], M), M:Pm . M, P: Pm .P, /* next step */
 if k=n then (sol_found:true, return()) /* goal reached --> break and return true */
 else (
 if btrDiag(k+1,n) then (goAhead:true, return()) /* if next step possible go ahead */
 else (P:P_saved, M:M_saved) /* go back */
)
),
 if goAhead then return(goAhead),
 if sol_found then return(sol_found),
 false
)$
```

Hier Backtracking mit “Ausbruch” aus der Funktion

```
(%i34) btrDev(k,n):=block([choices, P_saved,M_saved],
 choices:upperRowsUntouched(nonZeroEntries(M,k),k),
 for i:1 while i <= length(choices) do (
 P_saved:P, M_saved:M, /* save step back */
 Pm: swap(k, choices[i], M), M:Pm . M, P: Pm .P, /* next step */
 if k=n then throw(true) /* goal reached --> break and return true */
 else (
 if btrDiag(k+1,n) then throw(true) /* if next step possible go ahead */
 else (P:P_saved, M:M_saved) /* go back */
)
),
 false
)$
```

Hier der “Wrapper” für das Backtracking

```
(%i35) backtrackingDiag(A):=block([k:1,m:first(matrix_size(A)), n:second(matrix_size(A)),M,P],
 P:diagmatrix(m,1),
 M:copymatrix(A),
 if catch(btrDev(k,n))then [M,P]
 else false
);
```

```
(%i36) [m,p]:backtrackingDiag(A) ;
```

```
(%o36) [$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 3 & 0 \end{pmatrix}$, $\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$]
```

Die r-Matrix ist obere Dreiecksmatrix

```
(%i37) [q,r]:float(Q_R_Fact(m));
```

```
(%o37)
```

$$\left[ \begin{pmatrix} -0.4472135954999579 & .8944271909999159 & 0.0 & 0.0 & 0.0 \\ -0.8944271909999159 & -0.447213595499958 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & -0.316227766016838 & -.6529286250990104 & -.6882472016116853 \\ 0.0 & 0.0 & 0.0 & -.7254762501100116 & .6882472016116853 \\ 0.0 & 0.0 & -.9486832980505138 & .2176428750330036 & .2294157338705619 \end{pmatrix}, \right.$$

$$\left. \begin{pmatrix} -2.23606797749979 & -.8944271909999159 & 0.0 & 0.0 \\ 0.0 & -0.447213595499958 & 0.0 & 0.0 \\ 0.0 & 0.0 & -3.162277660168379 & -0.316227766016838 \\ 0.0 & 0.0 & 0.0 & -1.378404875209022 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix} \right]$$

Der “Messwert”-Vektor wird gesetzt

```
(%i38) b:matrix([4],[1],[7],[9],[4])$
```

## 18. QR Faktorisierung

Auf das ‘Rückwärts-Einsetzen’ wurde im Haupttext eingegangen

```
(%i39) backwardSubstitution(r,b):=
 block([cols:second(matrix_size(r)), x:zeromatrix(second(matrix_size(r)),1)],
 for c:cols thru 1 step -1 do x[c]:((b[c]-row(r,c) . x)/r[c,c]),
 x)$
```

Beachte, dass jetzt der Messwert-Vektor  $\vec{b}$  mit  $p$  permutiert werden muss - es ergibt sich die exakte Lösung!

```
(%i40) solutionVec:backwardSubstitution(r, transpose(q) . (p . b));
```

```
(%o40) $\begin{pmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \end{pmatrix}$
```

## 18.5 Givens-Rotationen

### 18.5.1 Rotationen in 3D in einer Ebene

Wir bezeichnen (um die spätere Erweiterung zu erleichtern) die  $x$ -Achse mit 1, die  $y$ -Achse mit 2 und die  $z$ -Achse mit 3.

#### Beispiel 18.10



Wir berechnen die Rotation in der  $(2,3)$ -Ebene um den Winkel  $\theta$ . Seien  $\vec{e}_i$  die Einheitsvektoren in Richtung der Koordinatenachsen.

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \vec{e}_i = \vec{x}_i \quad \text{mit} \quad \vec{x}_i = \begin{cases} \vec{e}_1 & \text{für } i = 1 \\ (0, \cos \theta, \sin \theta)^t & \text{für } i = 2 \\ (0, -, \sin \theta, \cos \theta)^t & \text{für } i = 3 \end{cases}$$

Das Ergebnis ist eine  $(3 \times 3)$ -Einheitsmatrix mit mit folgenden Änderungen:

$$a_{22} = a_{33} = \cos \theta \quad a_{32} = \sin \theta \quad a_{23} = -a_{32}$$

### 18.5.2 Rotationen im $\mathbb{R}^n$ in der Ebene aufgespannt durch $i,k$ ( $i < k$ )

Obiges Beispiel lässt sich leicht erweitern auf den  $\mathbb{R}^n$  :

#### Definition 18.11 Givens Rotation

Eine  $n \times n$  Matrix heißt Givens Rotation (Givens Matrix), wenn gilt

$$G[i, k] = I_n \text{ außer } a_{ii} = a_{kk} = c \wedge a_{ik} = -s \wedge a_{ki} = s \wedge c^2 + s^2 = 1 \quad (i < k)$$

**Theorem 18.12**

Givens Matrizen sind orthogonal, also  $G[i, j] \cdot (G[i, j])^t = I$  oder  $G^t = G^{-1}$

*Beweis:* Alle Spaltenvektoren von  $G[i, j]$ ,  $i < j$  sind die Standard-Basisvektoren  $\vec{e}_k$  (mit den Koordinaten  $\delta_{nk}$ ,  $k \in \{1, 2, \dots, n\}$ ) nur der  $i$ -te Spaltenvektor  $\vec{v}_i$  und der  $j$ -te Spaltenvektor  $\vec{v}_j$  sehen anders aus, nämlich

$$\vec{v}_i = \delta_{ik}c + \delta_{jk}s \quad \vec{v}_j = \delta_{ik}(-s) + \delta_{jk}c \quad k \in \{1, 2, \dots, n\}$$

Hier ein Beispiel für  $G[i = 2, j = 5]$ :

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & c & 0 & 0 & -s & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & s & 0 & 0 & c & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Die Orthogonalität:

$$\vec{v}_i \cdot \vec{e}_\ell \stackrel{\ell \neq i, j}{=} \sum_{k=1}^n (\delta_{ik}c + \delta_{jk}s) \delta_{k\ell} = \sum_{k=1}^n \delta_{ik}c\delta_{k\ell} + \delta_{jk}s\delta_{k\ell}$$

Bei beiden Summanden verschwindet jeweils eines der beiden Deltas! Analoges gilt für  $\vec{v}_j \cdot \vec{e}_\ell$   
Bleibt noch

$$\vec{v}_i \cdot \vec{v}_j = \sum_{k=1}^n (\delta_{ik}c + \delta_{jk}s) (\delta_{jk}c + \delta_{ik}(-s)) = \sum_{k=1}^n (\delta_{ik}c\delta_{jk}c + \delta_{ik}c\delta_{ik}(-s) + \delta_{jk}s\delta_{jk}c + \delta_{jk}s\delta_{ik}(-s))$$

Erster und letzter Summand verschwinden wegen  $i \neq j$

Zweiter und dritter Summand liefern für  $k = i$  und  $k = j$  die Ergebnisse:  $c(-s) + cs = 0$

Dass  $\vec{v}_i$  und  $\vec{v}_j$  Einheitsvektoren sind verdanken wir der Voraussetzung  $c^2 + s^2 = 1$ .

Für  $c = \cos \theta \wedge s = \sin \theta$  ist dies erfüllt.

□

Wir können die Givens-Matrizen dazu benutzen um aus einer Matrix mit vollem Rang eine Dreiecksmatrix zu formen:

## Beispiel 18.13



$$G[1,3] \cdot \begin{pmatrix} a_{11} & 9 & -1 \\ 8 & -3 & 6 \\ a_{31} & -2 & 5 \end{pmatrix} = \begin{pmatrix} a_{11}c - a_{31}s & 2s + 9c & -(5s) - c \\ 8 & -3 & 6 \\ a_{11}s + a_{31}c & 9s - 2c & 5c - s \end{pmatrix}$$

mit  $s = \frac{-a_{31}}{\sqrt{a_{11}^2 + a_{31}^2}}$  und  $c = \frac{a_{11}}{\sqrt{a_{11}^2 + a_{31}^2}}$  verschwindet  $a_{31}$

Es gilt  $s^2 + c^2 = 1$ . Bei vollem Rang von  $A$  verschwindet dabei kein anderes Element.

Erinnern wir uns, dass die Givens-Matrix eine Drehung darstellt. Wir drehen also den ersten Spaltenvektor von  $A$  so, dass seine letzte Koordinate verschwindet. Gute geometrische Darstellungen finden sich im Internet - die Spaltenvektoren der Matrix müssen natürlich linear unabhängig sein. Hier eine „naive“ Implementierung in *wxMaxima*:

```
(% i1) rnd_matrix(m,n, arr_gen):= block([td, p_td, st, s],
 td:timedate(), p_td:parse_timedate(td),
 st:mod(p_td, 1000) * mod(p_td, 37), s:make_random_state (st),
 set_random_state(s),
 genmatrix(arr_gen, m,n)
)$

(% i2) my_round3(A):=block([N:matrix_size(A)[1]],
 for i thru N do
 for j thru N do
 if abs(A[i,j]) < 10^-12 then A[i,j]: 0.0,
 A
)$

(% i3) h[i,j]:= float((-1)*random(20)+10)$
(% i4) (N:4, I: float(diagmatrix(N,1)), A:rnd_matrix(N,N,h));

 \begin{pmatrix} -5.0 & 7.0 & 4.0 & 10.0 \\ -8.0 & -5.0 & -7.0 & -9.0 \\ -5.0 & 0.0 & -7.0 & 6.0 \\ 8.0 & 0.0 & 2.0 & -5.0 \end{pmatrix} (% o4)

(% i5) given(i,j,a):=block([r,c,s,g],
 g:diagmatrix(matrix_size(a)[1],1),
 r:sqrt(a[j,j]^2+a[i,j]^2),
 c:-a[j,j]/r, s:a[i,j]/r,
 g[i,i]:c, g[j,j]:c,
 g[i,j]:s, g[j,i]:-s,
 g
)$
```

```
(% i6) QR_decomp(A,Q):=block(
 [N:matrix_size(A)[1], R:A, G],
 for col:1 thru N-1 do
 for row:N thru col+1 step -1 do block(
 G:given(row,col,R) ,
 Q: Q . transpose(G) ,
 R:G . R
),
 [my_round3(Q),my_round3(R)]
)$
(% i7) fpprintprec:3$
(% i8) [Q,R]:QR_decomp(A,I);
```

$$\left[ \begin{pmatrix} 0.375 & -0.831 & -0.116 & 0.395 \\ 0.6 & 0.556 & -0.162 & 0.553 \\ 0.375 & -0.0163 & 0.919 & -0.12 \\ -0.6 & 0.0261 & 0.34 & 0.724 \end{pmatrix}, \begin{pmatrix} -13.3 & -0.375 & -6.52 & 3.6 \\ 0.0 & -8.59 & -7.05 & -13.5 \\ 0.0 & 0.0 & -5.08 & 4.12 \\ 0.0 & 0.0 & 0.0 & -5.37 \end{pmatrix} \right] \quad (\% \text{ o8})$$

```
(% i9) my_round3(Q . transpose(Q));
```

$$\begin{pmatrix} 1.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{pmatrix} \quad (\% \text{ o9})$$

```
(% i10) my_round3(Q . R);
```

$$\begin{pmatrix} -5.0 & 7.0 & 4.0 & 10.0 \\ -8.0 & -5.0 & -7.0 & -9.0 \\ -5.0 & 0.0 & -7.0 & 6.0 \\ 8.0 & 0.0 & 2.0 & -5.0 \end{pmatrix} \quad (\% \text{ o10})$$

Hier gibt es viel Raum für Optimierung (Beispiele):

1. Es wird  $\sqrt{a^2 + b^2}$  berechnet - dies ist numerisch instabil, da es zu *overflow* bzw. *underflow* für große bzw. kleine Zahlen neigt - dies sollte man durch geeignete Algorithmen verhindern.
2. Für große Matrizen: Die Givens-Matrix verändert nur 2 Zeilen der Ausgangsmatrix - es reicht also die Multiplikation ausführen:

$$\begin{pmatrix} c & -s \\ s & c \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix}$$

und das Ergebnis in den entsprechenden Zeile der Ausgangsmatrix zu ersetzen.

## 18. QR Faktorisierung

---

```
using LinearAlgebra # needed for identity matrix

we build an N×N matrix with rank N and 1 decimal digit
function getSquareMat(N::Int64)
 mat=zeros(N,N); r=0
 while r!=N
 mat=floor.(rand(N,N)*100)/10
 r=rank(mat)
 end
 mat
end

e.g (row_col)=(5,1) - this element has to vanish row 1 and row 5 are affected
row 1 of M in B, row 5 of M in B
function extract2x2(M,row_col::Tuple{Int64, Int64})
 columns = size(M)[2]
 B = zeros(2,columns)::Matrix{Float64}
 B[1,:]=M[row_col[2],:]
 B[2,:]=M[row_col[1],:]
 return B
end

there are only 2 rows now
function givesRot(B::Matrix{Float64}, col::Int64)
 r = sqrt(B[1,col]^2 + B[2,col]^2)
 c = B[1,col]/r
 s = B[2,col]/r
 return [c s; -s c]
end

see comment of "extract2x2" - this is the inverse
function swapBack(M,B, row_col::Tuple{Int64, Int64})
 M[row_col[2],:] = B[1,:]
 M[row_col[1],:] = B[2,:]
 return M
end

small numbers are substituted by zero
function substZero(nr::Float64)::Float64
 if abs(nr) < 10^(-12)
 return 0.0
 end
 return nr
end
```

```

we build the "large" transposed Givens Rotation Matrix
function buildOrig_G(G,row,col)
 G_t =1.0* Matrix(I, N, N)
 c = G[1,1]; s = G[1,2]
 G_t[row,row] = c
 G_t[col,col] = c
 G_t[row,col] = s
 G_t[col,row] = -s
 return G_t
end

function decompose(M::Matrix{Float64},N::Int64)
 R = M; Q =1.0* Matrix(I, N, N)
 # N=5: → (5, 1),(4, 1),(3, 1),(2, 1),
 # (5, 2),(4, 2),(3, 2)
 # (5, 3) (4, 3)
 # (5, 4) = (row, col) has to vanish
 for c in range(1,N-1)
 for r in range(start=N, stop=c+1, step=-1)
 B = extract2x2(R, (r,c))
 G = givensRot(B,c)
 B = G * B # (2×2)×(2×N)=2×N → 2 rows, N columns
 R = swapBack(R,B, (r,c))
 Q = Q * buildOrig_G(G,r,c)
 end
 end
 return (substZero.(R), Q)
end

N=5
M=getSquareMat(N)
display(M)
R,Q=decompose(M,N)
display(R)
display(Q * R)

```



## 19 | Der schiefe Wurf

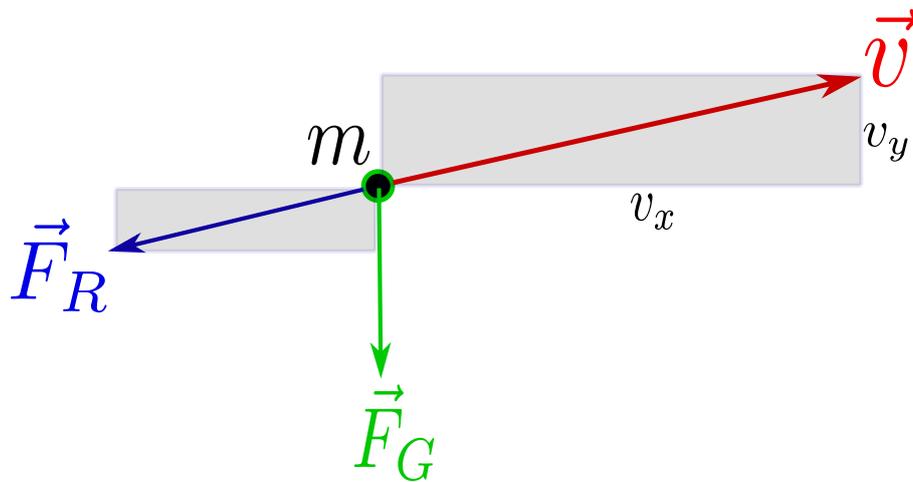


Abb.181 : angreifende Kräfte

Ausgangspunkt ist natürlich die Newton'sche Bewegungsgleichung:

$$m_t \ddot{\vec{x}} = \sum_i \vec{F}_i \quad (19.1)$$

$\vec{F}_i$  sind natürlich die angreifenden Kräfte.  $m_t$  ist die träge Masse des Körpers.

Wir betrachten hier nur den Luftwiderstand  $\vec{F}_R$ , der dem Geschwindigkeitsvektor  $\vec{v}$  des Körpers entgegengerichtet ist und die Gravitationskraft  $\vec{F}_G$ . Mit dem Newton'schen Gravitationsgesetz folgt für die Gravitationsbeschleunigung in Richtung Erdzentrum

$$m_t \vec{a}_r(r) = \vec{F}_G = -G \frac{M_E m_t}{(R_E + r)^2} \vec{r}_0 \quad (19.2)$$

## 19. Der schiefe Wurf

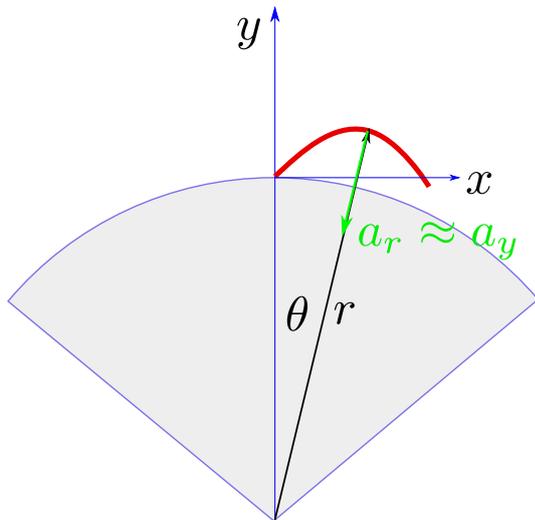


Abb.182 : Die Erde als Scheibe

Wir machen einige Vereinfachungen:

- Die Wurfweiten  $w$  seien extrem klein gegenüber dem Erdradius  $R_E$  ( $w \ll R_E$ ) - also wir machen aus der Erde eine Scheibe (eine über Jahrtausende beliebte Näherung). Damit können wir die Wurfbahn statt mit Polarkoordinaten  $(r(t), \theta(t))$  mit kartesischen Koordianten  $(x, y)$  beschreiben.
- Die Wurfhöhen  $r$  (in obiger Formel) seien extrem klein gegenüber dem Erdradius  $R_E$  ( $r \ll R_E$ ), damit können  $r$  in Formel 19.2 vernachlässigen und es ergibt sich:

$$\begin{cases} a_x = 0 \\ a_y = -G \frac{M_E}{R_E^2} = -g \approx -9.81 \end{cases} \quad (19.3)$$

Unsere Formeln sind daher für ballistische Geschöße und Satelliten vollkommen ungeeignet - das wäre wieder eine andere Geschichte!

- Wir betrachten keine Auftriebs- bzw. Abtriebskräfte - ganz gleich ob sie sich durch schnelle Rotation (Spin) (Frisbie-Scheibe) oder Tragflächen (Bumerang), rotierender Tennisball usw. ergeben.
- Als Näherungsstufen betrachten wir
  - $\vec{F}_R = 0$  (Bewegung im Vakuum, z.B. am Mond)
  - $|\vec{F}_R| \propto |\vec{v}|$  (laminare Strömung, langsame Geschwindigkeit z.B. Kugel im Honigglas, Auto in Schrittgeschwindigkeit - Karosserie möglichst ohne Kanten)
  - $|\vec{F}_R| \propto |\vec{v}|^2$  (turbulente Strömung - geworfene Steine, Geschöße, Autos bei "Reise-geschwindigkeit" )

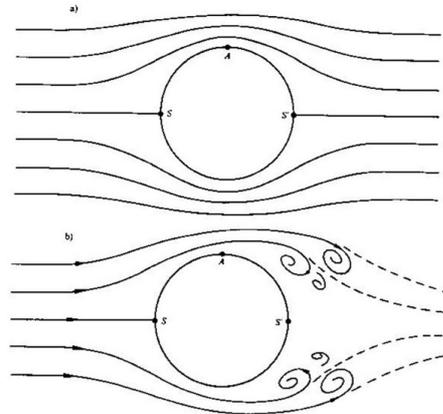


Abb.183 : laminare und turbulente Strömung

## 19.1 Ohne Luftwiderstand auf der Erdoberfläche

Uns bleibt einfach Gleichung 19.3 zu lösen:

$$\ddot{\vec{x}} = \begin{pmatrix} 0 \\ -g \end{pmatrix} \quad (19.4)$$

### 19.1.1 Mit Differentialgleichung und *wxMaxima*

Obige gewöhnliche Differentialgleichung 19.4 führt zu einem einfachen Anfangswertproblem:

$$\ddot{\vec{x}} = \begin{pmatrix} 0 \\ -g \end{pmatrix} \quad \vec{x}(0) = \vec{0} \quad \vec{v}(0) = \begin{pmatrix} v_{x0} \\ v_{y0} \end{pmatrix} \quad (19.5)$$

Dies lässt sich natürlich leicht “per Hand” integrieren, doch zur Übung machen wir es mit *wxMaxima*:

Keine Meldungen bei Umwandlungen von Brüchen in Dezimalzahlen

(%i1) `ratprint:false$`

Beschleunigungsvektor  $\vec{a}(t)$  wird festgelegt

(%i2) `a(t):=[0,-g]$`

(%i3) `define(v(t),integrate(a(t),t)+[v_x0,v_y0]);`

$$\mathbf{v}(t) := [v_{x0}, v_{y0} - gt] \quad (\%o3)$$

Zeit für Wurfhöhe  $t_H$

(%i4) `t_H:rhs(linsolve(v(t)[2],t)[1]);`

$$\frac{v_{y0}}{g} \quad (t\_H)$$

Bahn in Parameterform

(%i5) `define(x(t),integrate(v(t),t));`

$$\mathbf{x}(t) := [t v_{x0}, t v_{y0} - \frac{g t^2}{2}] \quad (\%o5)$$

Wurfhöhe

(%i6) `Height:subst(t_H,t,x(t)[2])$`

von Parameterform in Funktionsdarstellung

(%i7) `define(y_1(x),subst(x/v_x0,t,x(t)[2]));`

$$y_1(x) := \frac{v_{y0}x}{v_{x0}} - \frac{g x^2}{2v_{x0}^2} \quad (\%o7)$$

(%i8) `eq1:(ratsimp(y_1(x)/x=0,x));`

$$-\frac{gx - 2v_{x0}v_{y0}}{2v_{x0}^2} = 0 \quad (eq1)$$

Berechnung der Wurfweite

(%i9) `width:rhs(linsolve(eq1,x)[1])$`

(%i10) `display(Height,width)$`

$$Height = \frac{v_{y0}^2}{2g} \quad width = \frac{2v_{x0}v_{y0}}{g}$$

(%i11) `t_max:width/v_x0;`

$$\frac{2v_{y0}}{g} \quad (t\_max)$$

## 19. Der schiefe Wurf

statt der Wurfparameter  $v_{x0}$ ,  $v_{y0}$  wechseln wir auf  $|\vec{v}| = v$  und Steigungswinkel  $\alpha$

```
(%i12) subst(v*cos(alpha),v_x0,y_1(x))$
```

```
(%i13) define(y_2(x),trigsimp(subst(v*sin(alpha),v_y0,%)));
```

$$y_2(x) := -\frac{g x^2 - 2 \cos(\alpha) \sin(\alpha) v^2 x}{2 \cos(\alpha)^2 v^2} \quad (\%o13)$$

```
(%i16) v_x0:6$v_y0:8$g:9.81$
```

```
(%i17) v:sqrt(v_x0^2+v_y0^2)$
```

```
(%i18) alpha:atan(v_y0/v_x0)$
```

Jetzt plotten wir alle 3 "Kurven" mit den Linienstärken 8,4 und 1 und den Farben 2,3 und 1

```
(%i19) plot2d([[parametric,x(t)[1],x(t)[2],[t,0,t_max],[nticks, 100]],y_1(x),y_2(x)],
[x,0,width],[legend,"parametric","with v_x,v_y","with v and {/Symbol a}"],
[style, [lines, 8,2],[lines, 4,3], [lines, 1,1]])
```

```
(%i20) ev(Height); 3.261977573904179
```

```
(%i21) ev(width); 9.785932721712538
```

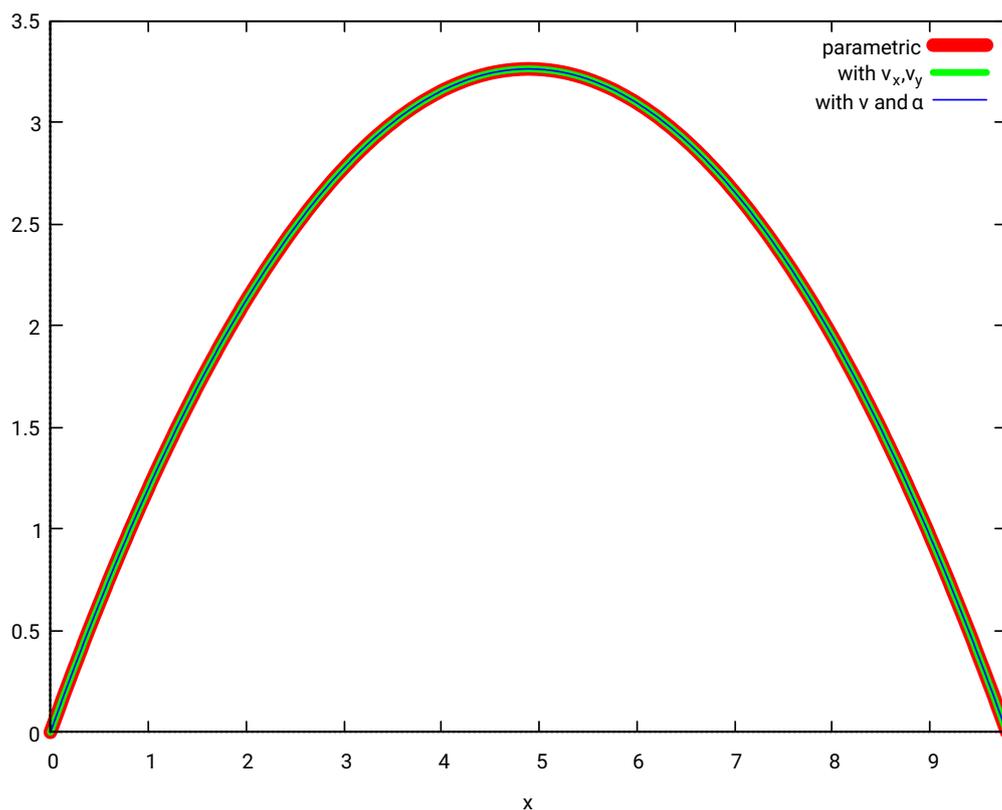


Abb.184 : alle 3 Darstellungen der Bahn



Alle 3 Graphen führen zum selben Ergebnis!

### 19.1.2 Mit Differenzengleichung und *wxMaxima*

$\Delta t \ll 1$  sei der diskrete Simulationszeitraum. Folgende Iterationsformeln finden Verwendung:

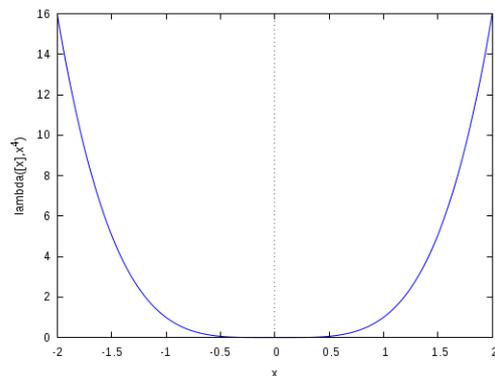
$$\begin{aligned} \frac{\vec{v}_{i+1} - \vec{v}_i}{\Delta t} &= \vec{a}_i & \Rightarrow & \vec{v}_{i+1} = \vec{v}_i + \vec{a}_i \Delta t & (19.6) \\ \frac{\vec{x}_{i+1} - \vec{x}_i}{\Delta t} &= \vec{v}_i & \Rightarrow & \vec{x}_{i+1} = \vec{x}_i + \vec{v}_i \Delta t \\ \vec{x}_0 &= \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \vec{v}_0 &= \begin{pmatrix} v_{x0} \\ v_{y0} \end{pmatrix} = \begin{pmatrix} 6 \\ 8 \end{pmatrix} & \vec{a}_0 &= \begin{pmatrix} 0 \\ -g \end{pmatrix} \end{aligned}$$

Natürlich lassen sich obige Vektorfolgen leicht in die entsprechenden Folgen für die Koordinaten zerlegen (so werden wir das in der Tabellenkalkulation machen) - in *wxMaxima* lassen sich allerdings obige rekursive Vektorfolgen durch Array-Funktionen implementieren. Hier eine kleine Demonstration:

#### Array Functions

|                                                                                                                                                                                              |                                                                                                                           |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|
| <p>(%i5) <code>cc[x,y]:= x/y;</code></p> <p>(%i2) <code>cc[u,z];</code></p> <p>(%i4) <code>u:1\$ z:2\$</code></p> <p>(%i5) <code>cc[u,z];</code></p> <p>(%i6) <code>f[n](x):=x^n;</code></p> | $cc_{x,y} := \frac{x}{y} \quad (\%o1)$ $\frac{u}{z} \quad (\%o2)$ $\frac{1}{2} \quad (\%o5)$ $f_n(x) := x^n \quad (\%o6)$ |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------|

(%i7) `wxplot2d(f[4],[x,-2,2]);`



(%t7)

#### Recursion

|                                                                                                                                                                                                |                                                                                                                                      |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|
| <p>(%i8) <code>v[i]:=v[i-1]+1;</code></p> <p>(%i9) <code>v[0]:0;</code></p> <p>(%i10) <code>listarray(v);</code></p> <p>(%i11) <code>v[8];</code></p> <p>(%i12) <code>listarray(v);</code></p> | $v_i := v_{i-1} + 1 \quad (\%o8)$ $0 \quad (\%o9)$ $[0] \quad (\%o10)$ $8 \quad (\%o11)$ $[0, 1, 2, 3, 4, 5, 6, 7, 8] \quad (\%o12)$ |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------|

## 19. Der schiefe Wurf

Aber jetzt zurück - der Lösung unseres Problems mit Differenzengleichung:

```
(%i3) Delta_t:0.001$ a:[0,-9.81]$ fpprintprec : 3 $
```

```
(%i4) v[i]:=v[i-1] + a * Delta_t;
```

$$v_i := v_{i-1} + a \Delta_t \quad (\%o4)$$

```
(%i5) x[i]:=x[i-1]+v[i-1] * Delta_t;
```

$$x_i := x_{i-1} + v_{i-1} \Delta_t \quad (\%o5)$$

```
(%i6) x[0]:[0,0]$
```

```
(%i7) v[0]:[6,8]$
```

```
(%i8) trajectory():=block([list:[x[0]]],
 for i:1 while x[i-1][2] >= 0 do
 list: cons(x[i],list),
 list
)$
```

```
(%i9) plotlist:trajectory()$
```

```
(%i10) plot2d([discrete, plotlist])$
```

```
(%i11) Height:lmax(map(lambda([x],x . [0,1]),plotlist));
```

3.27

(Height)

```
(%i12) width:(plotlist[1][1]+plotlist[2][1])/2;
```

9.79

(width)

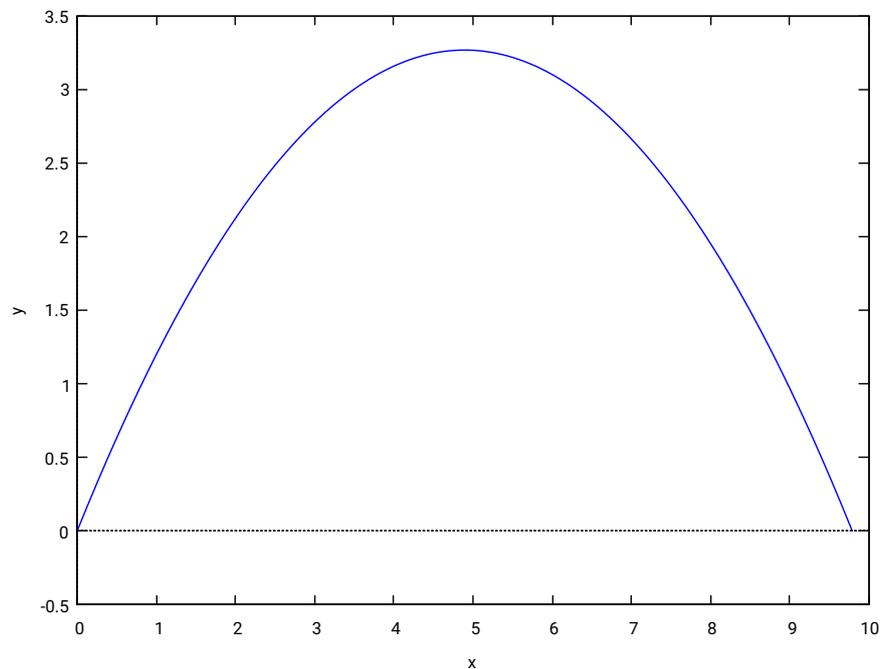


Abb.185 : Jetzt als Punktplot mit Differenzengleichung

19.1.3 Mit LibreOffice-Calc

|    | A          | B         | C                | D                | E                | F                | G        |
|----|------------|-----------|------------------|------------------|------------------|------------------|----------|
| 1  |            |           |                  |                  |                  |                  |          |
| 2  | Angaben    |           |                  |                  |                  |                  |          |
| 3  | $\Delta t$ | $x_0$     | $y_0$            | $v_{x0}$         | $v_{y0}$         | $a_{x0}$         | $a_{y0}$ |
| 4  | 0,01       | 0         | 0                | 6                | 8                | 0                | -9,81    |
| 5  |            |           |                  |                  |                  |                  |          |
| 6  | $a_{xi}$   | $a_{yi}$  | $v_{xi}$         | $v_{yi}$         | $x_i$            | $y_i$            | Schritt  |
| 7  | $=\$F\$4$  | $=\$G\$4$ | $=\$D\$4$        | $=\$E\$4$        | $=\$B\$4$        | $=\$C\$4$        | 0        |
| 8  | $=A7$      | $=B7$     | $=C7+A8*\$A\$4$  | $=D7+B8*\$A\$4$  | $=E7+C8*\$A\$4$  | $=F7+D8*\$A\$4$  | $=G7+1$  |
| 9  | $=A8$      | $=B8$     | $=C8+A9*\$A\$4$  | $=D8+B9*\$A\$4$  | $=E8+C9*\$A\$4$  | $=F8+D9*\$A\$4$  | $=G8+1$  |
| 10 | $=A9$      | $=B9$     | $=C9+A10*\$A\$4$ | $=D9+B10*\$A\$4$ | $=E9+C10*\$A\$4$ | $=F9+D10*\$A\$4$ | $=G9+1$  |

Abb.186 : Tabellenkalkulation

Bis zur 4.-ten Zeile sind die Angaben einzugeben. In Zeile 7 werden die Angaben kopiert. Zeile 8 ist die wichtigste - hier werden die Rekursionsformeln ausgedrückt. Diese Zeile wird nach unten ausgefüllt (beachte die absolute Adressierung für  $\Delta t$ ) bis die  $y_i$  negativ werden. Anschließend wird für die x-y-Spalte ein x-y-Diagramm erstellt:

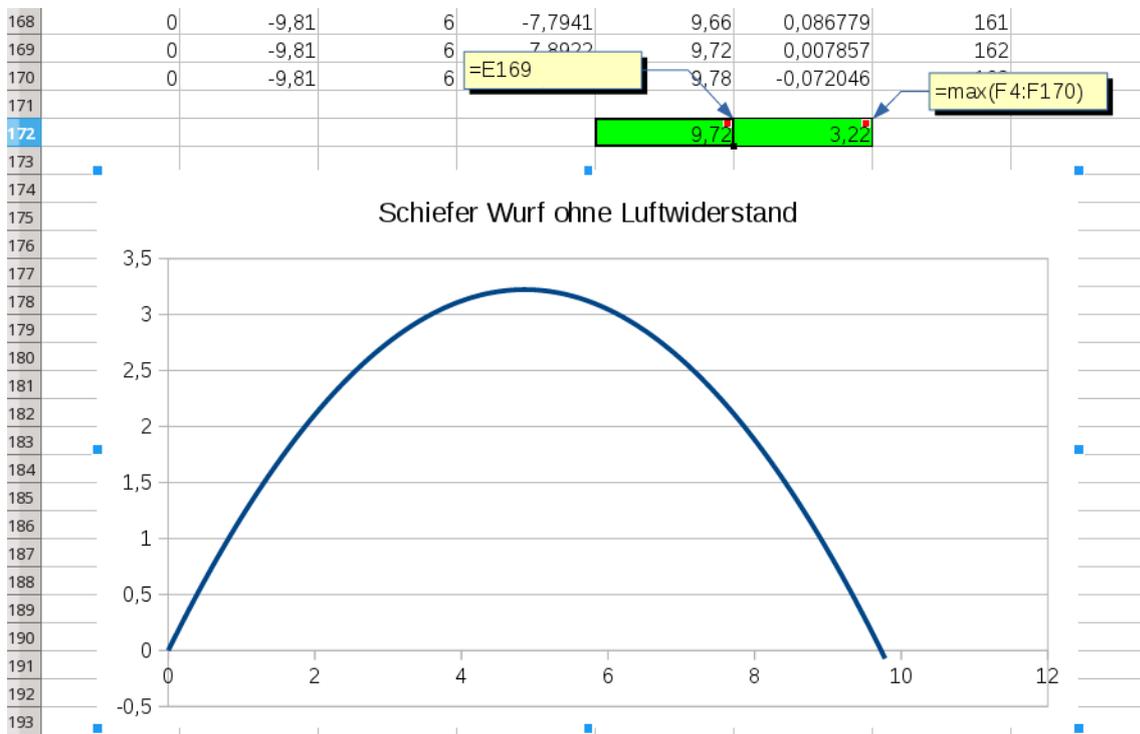


Abb.187 : Weite, Höhe und Bahn in LibreOffice-Calc

## 19. Der schiefe Wurf

---

### 19.1.4 Mit Hand und Kettenregel

Wir schreiben 19.4 mit  $\vec{v}$  an:

$$\begin{cases} \dot{v}_x = 0 \\ \dot{v}_y = -g \end{cases} \quad (19.7)$$

Wir versuchen gleich die Differentialgleichung für die Flugbahn  $y(t(x))$  mit der Kettenregel herzuleiten:

$$\frac{dy}{dx} = \frac{dy}{dt} \frac{dt}{dx} = \frac{dy}{dt} \frac{1}{\frac{dx}{dt}} \Rightarrow \boxed{\frac{d}{dx} = \frac{1}{v_x} \frac{d}{dt}} \quad (19.8)$$

Wir benutzen die letzte Operatorgleichung

$$\frac{d^2y}{dx^2} = \frac{d}{dx} \frac{dy}{dx} = \frac{1}{v_x} \frac{d}{dt} \frac{v_y}{v_x} = \frac{\dot{v}_y v_x - v_y \dot{v}_x}{v_x^3} \stackrel{19.7}{=} = \frac{-g}{v_x^2} = \frac{-g}{v_{x0}^2} \quad (19.9)$$

Das letzte Gleichheitszeichen gilt, da  $v_x$  eine Konstante ist. Mit den Anfangsbedingungen

$$y(0) = 0 \quad y'(0) = \tan \alpha = \frac{v_{y0}}{v_{x0}}$$

folgt durch einfaches Integrieren

$$\boxed{y(x) = x \tan \alpha - \frac{g}{2v_{x0}^2} x^2} \quad (19.10)$$

Dies stimmt natürlich mit dem mit *wxMaxima* errechneten  $y_1(x)$  überein!

## 19.2 Luftwiderstand $\propto |\vec{v}|$

Gleichung 19.4 wird jetzt zu

$$\vec{a} = \vec{g} - \frac{1}{\tau} \vec{v} \quad (19.11)$$

### 19.2.1 Mit Hand und Variablentransformation

Wie in 19.11 schon angedeutet, muss der Proportionalitätsfaktor  $\tau$  die Dimension einer Zeit haben. Wir führen daher dimensionslose Größen ein (Genauerer im ANHANG 19.3.11:

$\tau$  als neue Zeiteinheit,  $g\tau$  als neue Geschwindigkeitseinheit und  $g\tau^2$  als neue Längeneinheit. Das bedeutet in Gleichung 19.11 wird  $g$  und  $\tau$  durch 1 ersetzt!

$$T = \frac{t}{\tau} \Rightarrow t = T \cdot \tau \quad (X(T), Y(T)) = \frac{1}{g\tau^2} (x(t(T)), y(t(T))) \quad (19.12)$$

damit können wir eine neue Geschwindigkeit festlegen

$$\frac{dX}{dT} = \frac{1}{g\tau^2} \frac{dx}{dt} \frac{dt}{dT} = \frac{1}{g\tau} v(t) \Rightarrow \boxed{V = \frac{1}{g\tau} v} \quad (19.13)$$

Dasselbe machen wir mit den Beschleunigungen indem wir 19.13 differenzieren:

$$\dot{V} = \frac{dV}{dT} = \frac{1}{g\tau} \frac{dv}{dt} \frac{dt}{dT} = \frac{1}{g} \dot{v}(t) \Rightarrow \boxed{\dot{V} = \frac{1}{g} \dot{v}} \quad (19.14)$$

Die x-Koordinate von 19.11 wird dann mit 19.13 und 19.14 zu

$$\dot{v}_x = g\dot{V}_x = -\frac{1}{\tau} g\tau V_x \Rightarrow \dot{V}_x = -V_x$$

und die y-Koordinate zu

$$\dot{v}_y = g\dot{V}_y = -g - \frac{1}{\tau} g\tau V_y \Rightarrow \dot{V}_y = -1 - V_y$$

wir haben also folgendes entkoppeltes Anfangswertproblem zu lösen

$$\begin{cases} \dot{V}_x = -V_x & Y(0) = 0, X(0) = 0 \\ \dot{V}_y = -1 - V_y & \frac{dY}{dX}(0) = \tan \alpha = \frac{V_0 \sin \alpha}{V_0 \cos \alpha} = \frac{V_{y0}}{V_{x0}} \end{cases} \quad (19.15)$$

### Lösung über die Zeitdimension $T$ durch Variablentrennung

x-Koordinate von 19.15

$$\int \frac{dV_x}{V_x} = \int -dT \Rightarrow \ln V_x = -T + C_1 \Rightarrow V_x(T) = e^{-T} e^{C_1} \Rightarrow V_x(T) = V_{x0} e^{-T}$$

$$X(T) = -V_{x0} e^{-T} + C_2 \Rightarrow \boxed{X(T) = V_{x0} (1 - e^{-T})} \quad (19.16)$$

y-Koordinate von 19.15

$$\int \frac{dV_y}{1 + V_y} = \int -dT \Rightarrow \ln(1 + V_y) = -T + C_1 \Rightarrow 1 + V_y(T) = e^{-T} e^{C_1} \Rightarrow V_y(T) = (1 + V_{y0}) e^{-T} - 1$$

$$Y(T) = -(1 + V_{y0}) e^{-T} - T + C_2 \Rightarrow \boxed{Y(T) = (1 + V_{y0}) (1 - e^{-T}) - T} \quad (19.17)$$

## 19. Der schiefe Wurf

Um  $Y(X)$  zu erhalten, machen wir aus 19.16  $T$  explizit und setzen in 19.17 ein:

$$Y(X) = (1 + V_{y0}) \frac{X}{V_{x0}} + \ln \left( 1 - \frac{X}{V_{x0}} \right) \quad (19.18)$$

Gleichungen 19.16 bis 19.18 können wir mit den Transformationsgleichungen 19.12 und 19.13 in die üblichen physikalischen Einheiten umformen - als Beispiel sei das für  $X(T)$  vorgeführt:

$$\frac{1}{g\tau^2}x(t) = \frac{1}{g\tau}v_{x0} \left( 1 - e^{-\frac{t}{\tau}} \right)$$

Mit etwas Rechnen kommt man auf folgende Ergebnisse:

$$\begin{cases} x(t) = \tau v_{x0} \left( 1 - e^{-\frac{t}{\tau}} \right) \\ y(t) = (g\tau^2 + \tau v_{y0}) \left( 1 - e^{-\frac{t}{\tau}} \right) - t g \tau \\ y(x) = (g\tau + v_{y0}) \frac{x}{v_{x0}} + \ln \left( 1 - \frac{x}{\tau v_{x0}} \right) g\tau^2 \end{cases} \quad (19.19)$$



Überprüfen Sie in 19.19 die einzelnen Terme auf ihre physikalische Dimension (Plausibilitätscheck!)

Für  $\tau \rightarrow \infty$  müsste sich in 19.19 der reibungsfreie Fall ergeben also die Formeln von (%o5) im *wxMaxima*-Programm! Wir zeigen hier eine Möglichkeit:

Wir ersetzen die Exponentialfkt. durch eine Taylorreihe:

$$1 - e^{-\frac{t}{\tau}} = - \sum_{i=1}^{\infty} \left( -\frac{t}{\tau} \right)^i \frac{1}{i!} \quad \text{damit wird } x(t) \text{ von 19.19}$$

$$x(t) = v_{x0}t - \sum_{i=2}^{\infty} \left( -\frac{t}{\tau} \right)^i \frac{1}{i!} \quad \text{die Summe verschwindet für } \tau \rightarrow \infty$$

Für  $y(t)$  machen spalten wir von der Taylorsumme die ersten 2 Summande an:

$$y(t) = (g\tau^2) \left[ \frac{t}{\tau} - \left( \frac{t}{\tau} \right)^2 \frac{1}{2} - \sum_{i=3}^{\infty} \left( -\frac{t}{\tau} \right)^i \frac{1}{i!} \right] - \cancel{t g \tau} + v_{y0}t - \sum_{i=2}^{\infty} \left( -\frac{t}{\tau} \right)^i \frac{1}{i!}$$

Nach dem Grenzübergang verschwinden jetzt ebenfalls die Summen und es bleibt

$$y(t) = v_{y0}t - \frac{1}{2}g t^2$$

Damit haben die Formeln von 19.19 einen weiteren Plausibilitätscheck bestanden!

**Lösung von 19.15 über die Wurfbahn**

Wir erhalten folgende Operatorengleichung mit der Kettenregel:

$$\frac{dY}{dX} = \frac{dY}{dT} \frac{dT}{dX} = \frac{dY}{dT} \frac{1}{\frac{dX}{dT}} \Rightarrow \boxed{\frac{d}{dX} = \frac{1}{V_x} \frac{d}{dT}} \quad (19.20)$$

andererseits aus dem ersten Teil obiger Gleichung

$$\frac{dY}{dX} = \frac{dY}{dT} \frac{dT}{dX} = \frac{dY}{dT} \frac{1}{\frac{dX}{dT}} = \frac{V_y}{V_x} \quad (19.21)$$

Nun kombinieren wir 19.20 und 19.21 zu

$$\frac{d^2Y}{dX^2} = \frac{d}{dX} \frac{dY}{dX} = \frac{1}{V_x} \frac{d}{dT} \frac{V_y}{V_x} = \frac{\dot{V}_y V_x - V_y \dot{V}_x}{V_x^3} \stackrel{19.15}{=} -\frac{1}{V_x^2} \quad (19.22)$$

Wir benötigen also  $V_x(X)$  um weiter zu kommen. Diese Funktion bekommen wir mit der Kettenregel und dem Anfangswertproblem 19.15:

$$\begin{aligned} \frac{dV_x}{dX} &= \frac{dV_x}{dT} \frac{dT}{dX} = \dot{V}_x \frac{1}{V_x} \stackrel{19.15}{=} -1 \Rightarrow \\ \Rightarrow V_x(X) &= C - X = V_{x0} - X \end{aligned} \quad (19.23)$$

Damit wird 19.22 zu

$$\boxed{\frac{d^2Y}{dX^2} = -\frac{1}{(V_{x0} - X)^2}} \quad (19.24)$$

Gleichung 19.24 ist aber leicht durch Integrieren zu lösen:

$$\frac{dY}{dX} = (V_{x0} - X)^{-1} + C_1 \xrightarrow{Y'(0)=\tan \alpha} \frac{dY}{dX} = (V_{x0} - X)^{-1} + \underbrace{\frac{V_{y0}}{V_{x0}}}_{\tan \alpha} - \frac{1}{V_{x0}}$$

und neuerliches integrieren führt zu

$$Y(X) = \ln(V_{x0} - X) + X \tan \alpha - \frac{1}{V_{x0}} X + C_2 \xrightarrow{Y(0)=0} C_2 = -\ln V_{x0}$$

dies führt schließlich zum Endergebnis, welches Gott sei Dank mit 19.18 identisch ist:

$$Y(X) = \ln \left( 1 - \frac{X}{V_{x0}} \right) + X \tan \alpha - \frac{X}{V_{x0}} \quad (19.25)$$

### 19.3 Luftwiderstand $\propto |\vec{v}|^2$

$$\vec{F}_R = - \underbrace{\frac{1}{2} \rho_L A c_W}_{p} v^2 \vec{v}_0 \quad \Rightarrow \quad \vec{F}_R = -p v \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$

$A$  ist die "Schattenfläche", bei einer Kugel also die Fläche eines Großkreises  
 $\rho_L$  ist die Dichte des Mediums (bei uns Luft) - diese schwankt etwas mit der Temperatur und der Seehöhe ( $1,3 \text{ kg/m}^3$  ist aber hier ein brauchbarer Wert)

$$\frac{1}{2} \rho_L v^2 \quad \text{wird oft als Staudruck bezeichnet}$$

$c_W$  ist eine dimensionslose Zahl, die von der Form des Körpers abhängt (auch von der Art der Strömung - laminar oder turbulent - sie hängt mit der Reynolds-Zahl  $Re$  zusammen. Am besten man bestimmt sie experimentell: für eine Kugel ist 0,4 ein brauchbarer Wert!)  
 mit  $\mu = p/m_t$  und  $\vec{F} = m_t \vec{a}$  folgt für die Beschleunigung

$$\vec{a} = \dot{\vec{v}} = \begin{pmatrix} 0 \\ -g \end{pmatrix} - \mu v \begin{pmatrix} v_x \\ v_y \end{pmatrix} \quad (19.26)$$

Das Produkt aus  $\mu \cdot v$  ist eine Größe, die den Einfluss der Reibung beschreibt. Für  $\mu = 0$  ergibt sich der reibungsfreie Wurf. Bei einer Billardkugel  $m = 170 \text{ g}$  und  $\varnothing \approx 58 \text{ mm}$  und unserer Anfangsgeschwindigkeit ist dieser Einfluss sehr gering! Aber schon der Austausch der Billardkugel durch eine aus Papier ( $m \approx 10 \text{ g}$ ) würde diesen Einfluss erheblich vergrößern!  
 Während die exakte Lösung des Problems sich ziemlich verkompliziert hat, bleibt die Lösung mit Differenzgleichung beinahe gleich:  
 Man muss zu 19.6 nur Folgendes hinzufügen:

$$\vec{a}_i = -\mu |\vec{v}_i| \vec{v}_i + \begin{pmatrix} 0 \\ -g \end{pmatrix}$$

$\mu$  muss natürlich wie oben definiert sein!  
 Damit sich bei den Gleichungen keine zirkulären Definitionen einschleichen muss man etwas aufpassen:

$$\begin{aligned} \vec{a}_0 &= f_1(\vec{v}_0) \\ \vec{v}_1 &= f_2(\vec{v}_0, \vec{a}_0) \\ \vec{x}_1 &= f_3(\vec{x}_0, \vec{v}_0) \quad \text{wobei } f_2 = f_3 \\ \vec{a}_1 &= f_1(\vec{v}_1) \\ \vec{v}_2 &= f_2(\vec{v}_1, \vec{a}_1) \quad \text{usw.} \end{aligned}$$

Beachte die übereinstimmenden Indices! In *wxMaxima* muss es daher heißen (beachte den Index bei der Beschleunigung!):



```
v[i] := v[i-1] + a[i-1]*Delta_t
```

### 19.3.1 Mit *wxMaxima* 1 Bahnkurve

Bei einer Billardkugel ist der Einfluss der Luftreibung gering  $\mu \approx 0$

```
(%i1) mu:0.5*1.3*30^2*10^(-6)*%pi*0.4/0.17, numer;
 0.004324309887882421 (mu)
```

```
(%i2) a[i]:= -mu*sqrt(v[i] . v[i])*v[i] + g;
 a_i := (-mu) sqrt(v_i.v_i) v_i + g (%o2)
```

```
(%i5) Delta_t:0.001$ g:[0,-9.81]$ fpprintprec : 3 $
```

```
(%i6) v[i]:=v[i-1] + a[i-1] * Delta_t;
 v_i := v_{i-1} + a_{i-1} \Delta_t (%o6)
```

```
(%i7) x[i]:=x[i-1]+v[i-1] * Delta_t;
 x_i := x_{i-1} + v_{i-1} \Delta_t (%o7)
```

Interessanterweise wertet *wxMaxima* die Rekursionsformeln passend aus den Angaben selbst aus!

```
(%i8) x[0]:[0,0]$
```

```
(%i9) v[0]:[6,8]$
```

Solange die  $y$ -Werte der Lagevektoren  $\vec{x}$  größer/gleich Null sind, geben wir sie in eine Liste

```
(%i10) trajectory():=block([list:[x[0]]],
 for i:1 while x[i-1][2] >= 0 do
 list: cons(x[i],list),
 list
)$
```

```
(%i11) plotlist:trajectory()$
```

Die Liste der  $(x, y)$ -Werte wird mit *discrete* ausgedruckt

```
(%i12) plot2d([discrete, plotlist])$
```

Wir ermitteln das Maximum der  $y$ -Werte - es weicht kaum von dem ohne Luftwiderstand ab!

```
(%i13) height:lmax(map(lambda([x],x . [0,1]),plotlist));
 3.2 (height)
```

Bei der Weite nehmen wir einfach den Mittelwert der "zwei letzten"  $x$ -Werte

```
(%i14) width:(plotlist[1][1]+plotlist[2][1])/2;
 9.45 (width)
```

## 19. Der schiefe Wurf

---

### 19.3.2 Mit *wxMaxima* mehrere Bahnkurven

Jetzt wollen wir für verschiedene Luftwiderstände die Graphen ausgeben, dafür schreiben wir unser Programm etwas um:

```
(%i4) Delta_t:0.0001$ g:[0,-9.81]$ fpprintprec:3$ stringdisp:true$
```

```
(%i7) radius:30*10^(-3)$c_W:0.4$m:0.17$
```

```
(%i8) mu(r,c,m):=0.5*1.3*r^2*3.1416*c/m$
```

Wir weisen den Einfluss der Luftreibung einfach der Variablen  $k$  zu!

```
(%i9) k:-float(mu(radius,c_W,m));
```

−0.00432

(k)

Nun reservieren wir etwas Platz für die Fließkomma-Arrays  $a$ ,  $v$  und  $x$

```
(%i12) array (a, flonum,100000)$array (v, flonum,100000)$array (x, flonum,100000);$
```

```
(%i13) v[0]:[6,8]$
```

Jetzt die Anfangsbedingungen

```
(%i14) x[0]:[0,0]$
```

Berechnungsmethode für die Beschleunigung

```
(%i15) get_acc(i):=k*sqrt(v[i] . v[i])*v[i] + g$
```

Berechnungsmethode für Ort und Geschwindigkeit

```
(%i16) next_xv(i,base, incr):=base[i-1]+incr[i-1]*Delta_t$
```

$a[0]$  gehört zwar nicht zu den Anfangsbedingungen wird aber für die Berechnung der  $..[1]$ -Werte gebraucht!

```
(%i17) a[0]:get_acc(0);
```

[−0.259, −10.2]

(%o17)

Liste für die “Legende” im Diagramm - wird in *trajectories* gleich mitausgefüllt!

```
(%i18) myLegend:[]$
```

Berechnung der Plotliste und der Legende; Parameter: Liste der zu berechnenden  $k$ -Werte

```
(%i19) trajectories(k_list):=block([plist:[],list:[]],
 for j in k_list do block(
 k:-j,
 list:[x[0]],
 for i:1 while second(x[i-1]) >= 0 do block(
 v[i]:next_xv(i,v,a),
 x[i]:next_xv(i,x,v),
 a[i]:get_acc(i),
 list: cons(x[i],list)
),
 plist:cons(cons('discrete,[list]),plist),
 myLegend:cons(concat("k = ",j),myLegend)
),
 myLegend:cons('legend,myLegend),
 plist
)$
```

Wir berechnen die Bahnen für  $k = 0$  (reibungsfrei),  $k = 0.01$ ,  $k = 0.1$  und  $k = 1$  (starke Luftreibung)

```
(%i20) plotlist:trajectories([0,0.01,0.1,1])$
```

```
(%i21) plot2d(plotlist,myLegend)$
```

Liste von Vektoren; Betrachte nur die Liste mit einer bestimmten -nr- Komponente

```
(%i22) get_component(nr, l):= map(nr,l)$
```

```
(%i23) max_y_ofList(l):=lmax(get_component(second,l))$
```

```
(%i24) removed_discrete:get_component(second,plotlist)$
```

```
(%i25) heights:map(max_y_ofList,removed_discrete);
```

```
[0.851, 2.32, 3.12, 3.26]
```

```
(heights)
```

```
(%i26) width:get_component(first,(get_component(first,removed_discrete)));
```

```
[1.46, 5.59, 9.04, 9.79]
```

```
(width)
```

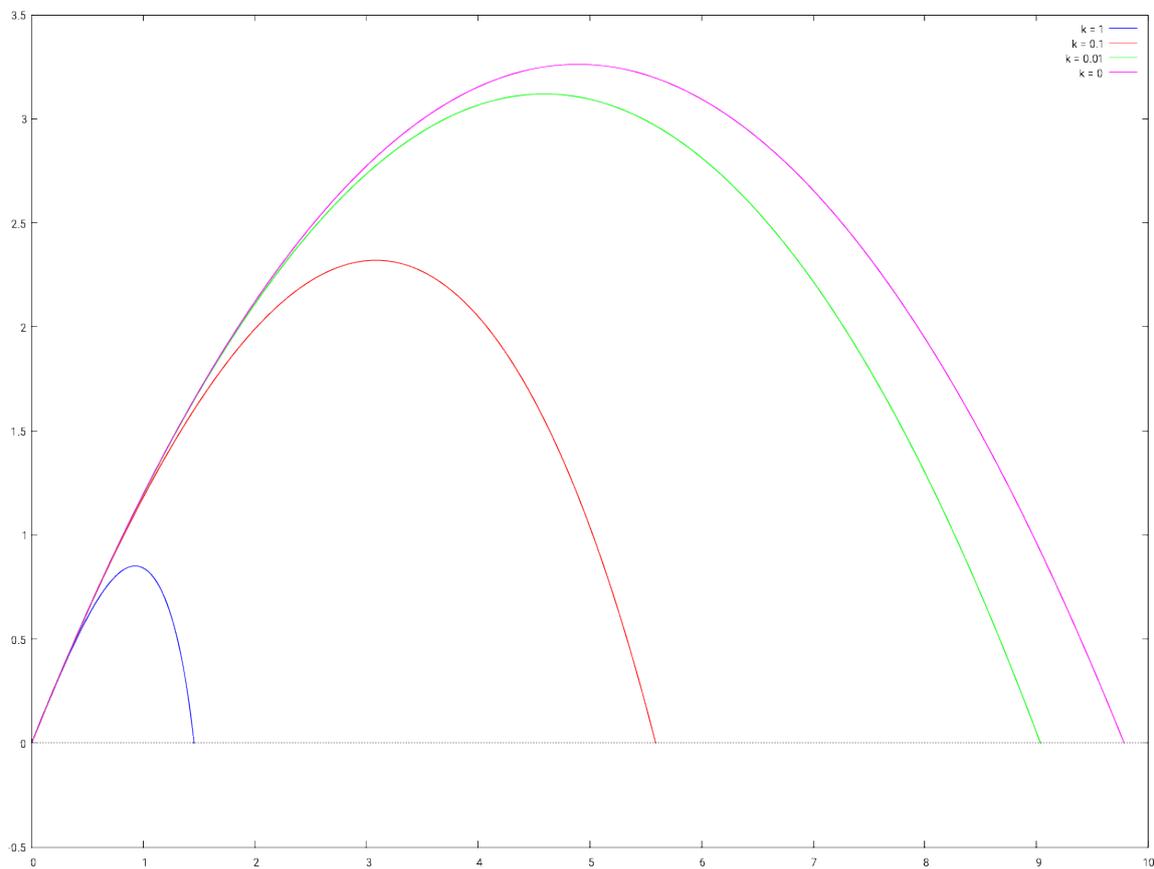


Abb.188 : Vergleich der Flugkurven - genaue Höhen und Weiten siehe in der Berechnung oben



Mit steigendem  $k$  geht die Symmetrie der Wurfbahn verloren und Höhen und Weiten werden geringer!

$k$  ist proportional zu  $A$ ,  $\rho_L$  und  $c_W$  und umgekehrt proportional zu  $m$  !

### 19.3.3 Mit *Geogebra* mehrere Bahnkurven

Vorweg - es ist mir nicht gelungen ohne etwas Javascript dieses Ziel mit *Geogebra* zu erreichen. Für mich war es daher auch sehr lehrreich das Zusammenspiel von *Geogebra* mit Javascript zu studieren. Dabei gab es für mich auch einige Überraschungen! Auf dem Arbeitsblatt werden auch untenstehende Differenzgleichungen und Flugbahn ohne Luftreibung angezeigt.

Zur Erinnerung noch einmal die Differenzgleichungen:

$$\overbrace{m\vec{a} = \vec{F}_R + \vec{F}_G}^{\text{Newtons Law}} = - \underbrace{\frac{1}{2}\rho_L|\vec{v}|^2 A c_W}_{p\text{Stau}} \frac{\vec{v}}{|\vec{v}|} + m\vec{g} \Rightarrow \vec{a} = - \underbrace{\frac{1}{2m}\rho_L A c_W}_{k} |\vec{v}| \vec{v} + \vec{g}$$

$$(1) \quad \vec{a}_i = -k|\vec{v}_i|\vec{v}_i + \vec{g} \quad (2) \quad \vec{v}_{i+1} = \vec{v}_i + \Delta t\vec{a}_i \quad (3) \quad \vec{x}_{i+1} = \vec{x}_i + \Delta t\vec{v}_i$$

(1) in (2) eingesetzt: (19.27)

$$\vec{v}_{i+1} = \vec{f}(\vec{v}_i) = (f_x, f_y)(v_{ix}, v_{iy}) \quad \text{wobei}$$

$$f_x(x, y) := x \left(1 - k \Delta t \sqrt{x^2 + y^2}\right) \quad f_y(x, y) := f_x(y, x) - g \Delta t$$

Um die Bahnkoordinaten  $\vec{x}_i$  koordinatenweise zu berechnen definieren wir die Funktion:

$$\text{incrBy}(x, y) := x + \Delta t y$$

Diese 3 Funktionen sind sozusagen die Arbeitspferde. Die einzelnen Ergebnisse unserer Rekursionsschritte speichern wir in den Listen

*vxList*, *vyList*, *xList*, *yList*, *PList*

wobei *PList* die Punktliste aus *xList* und *yList* ist. Auf *PList* arbeitet ein *Polygon*-Befehl:

`trajectory = Polygon(PList)`

Die Vorgangsweise zum Zeichnen der Wurfbahn mit den aktuellen Parametern (*k* als Schieberegler implementiert) ist:

- Button erstellen → “Create Trajectory”
- Im “On Mouseclick” dieses Buttons folgende Tätigkeiten durchführen:
  - Formeln wegblenden und Flugbahn ohne Luftreibung einblenden
  - Die Listen *vxList*, *vyList*, *xList*, *yList*, *PList* leeren
  - Diese Listen mit den Startwerten füllen
  - Mit den Funktionen  $f_x$ ,  $f_y$  und *incrBy* die Listen *vxList*, *vyList*, *xList*, *yList* berechnen bis ein Wert der *yList* negativ wird
  - Bei obiger Berechnung auch gleich die Punktliste *PList* erstellen
  - Gezeichnet wird dann automatisch wegen `trajectory = Polygon(PList)`

In Javascript (nicht vergessen umschalten von *Geogebra*-Script - notwendig, da in *Geogebra*-Script als einzige Schleife nur der *Folge*-Befehl existiert und wir brauchen eine Schleife mit Abbruchbedingung) schaut das dann so aus:

```

2 // hide explanation text
 setVisibility(false);
4 // set vxList, vyList, xList, yList and PList to start-values
 initLists();

6 //now do the recursion as long as body's height is positive - save the points in
 PList
 // the content of PList is drawn by 'trajectory'
8 var i=1;
 do {
10 i = i+1;
 ggbApplet.evalCommand("SetValue(vxList, "+i+", f_x(vxList("+(i-1)+"),vyList("+(
 i-1)+")))");
12 ggbApplet.evalCommand("SetValue[vyList, "+i+", f_y(vxList("+(i-1)+"),vyList("+(i
 -1)+"))]");
 ggbApplet.evalCommand("SetValue(xList, "+i+", incrBy(xList("+(i-1)+"),vxList("+(
 i-1)+")))");
14 ggbApplet.evalCommand("SetValue(yList, "+i+", incrBy(yList("+(i-1)+"),vyList("+(
 i-1)+")))");
 currentWidth = ggbApplet.getListValue("xList", i);
16 currentHeight = ggbApplet.getListValue("yList", i);
 ggbApplet.evalCommand("SetValue(PList, "+i+", (" +currentWidth+", "+currentHeight+"
))");
18 } while (currentHeight >= 0);

```

Wie man sieht, sind die Funktionen *setVisibility(boolean v)* und *initLists()* in den Reiter “global Javascript” ausgelagert, um den workflow klarer zu machen. Die Hauptarbeit leistet die *do - while*-Schleife.



Nicht vergessen: wir starten beim zweiten Listenelement, da im ersten Listenelement die Initialisierungen stehen!

Schauen wir uns die “global Javascript” näher an:

Da ist die *function initLists()* welche die Listen leert und anschl. initialisiert!

```

2 function initLists() {
 ggbApplet.evalCommand("vxList = {}");
4 ggbApplet.evalCommand("vyList = {}");
 ggbApplet.evalCommand("xList = {}");
6 ggbApplet.evalCommand("yList = {}");
 ggbApplet.evalCommand("PList = {}");

8

10 ggbApplet.evalCommand("SetValue(vxList,1,v_{x0})");
 ggbApplet.evalCommand("SetValue[vyList,1,v_{y0}]");
 ggbApplet.evalCommand("SetValue(xList,1,x_{0})");
12 ggbApplet.evalCommand("SetValue[yList,1,y_{0}]");
 ggbApplet.evalCommand("SetValue(PList,1,(x_{0},y_{0}))");
14 }

```



Das “Leeren” der Liste kann nicht weggelassen werden, da sonst nur das erste Listenelement verändert wird!

## 19. Der schiefe Wurf

```
function toggleVisibility(){
2 setVisibility(! ggbApplet.getVisible("Text1"));
}
4
function setVisibility(v){
6 ggbApplet.setVisible("Text1", v);
 ggbApplet.setVisible("Text2", v);
8 ggbApplet.setVisible("Text3", v);
 ggbApplet.setVisible("Text4", v);
10 ggbApplet.setVisible("s", ! v);
}
```

Außerdem ist hier auch die *function setVisibility(boolean v)* implementiert - die den Text aus- bzw. Parameterwurfkurve (ohne Luftwiderstand) einblendet. Diese Funktion wird auch für die *function toggleVisibility()* verwendet (die Mausklick-Routine für einen Button)

So jetzt haben wir alles beieinander, um jeweils 1(!) Bahnkurve zu zeichnen (für das jeweilig mit dem Schieberegler eingestellte  $k$ ). Wir wollen aber eine Kurve - wenn sie uns gefällt - weiter anzeigen (speichern) und den  $k$ -Wert dazuschreiben. Dazu benutzen wir folgende Dinge:

- `SavedPLists={}` eine Liste, in der wir die aktuelle *PList* einfügen (eine Liste von Listen)
- `SavedPoly=Folge(Polygonzug(SavedPLists(i)), i, 1, Länge(SavedPLists))`  
zeichnet die *SavedPLists*
- `MyLabels= {"", (0, 0)}` Liste, die Text und Punkte enthält - was wird wo geschrieben; ist mit einem Leerstring initialisiert, da sonst der nächste Befehl nicht funktioniert!
- `display=Folge(Text(Element(MyLabels, i, 1), Element(MyLabels, i, 2)), i, 1, Länge(MyLabels))` schreibt die Einträge von *MyLabels* auf den Schirm (arbeitet NICHT auf einer leeren Liste!
- `beforeLastPos=Element(PList, Länge(PList) - 1)` der vorletzte Punkt der aktuellen Wurfbahn (zu Beginn undefiniert!) - das wird die Position, wo wir unseren  $k$ -Wert hinschreiben!
- `savedLength=Länge(SavedPLists)` die Anzahl der gespeicherten Kurven - in Javascript benötigt

Jetzt brauchen wir nur etwas Javascript-Kitt, um diese Objekte zu verbinden: *SavedPLists* und *MyLabels* füllen - den Rest erledigt *Geogebra* automatisch!

So jetzt erzeugen wir einen Button - ich habe ihn "Retain Trajectory" genannt - mit folgender *onMouseClicked* Methode:

```
// get the length of list 'SavedPLists' and increment list-pointer
2 var savedLength = ggbApplet.getValue("savedLength");
```

```

savedLength++;
4
// PList is inserted by value NOT as pointer
6 ggbApplet.evalCommand("SetValue[SavedPLists, "+savedLength+", PList]");

8 // we construct the current label - but we cannot use the point 'beforeLastPos'
 // directly,
 // because it's a pointer NOT a value! So we have to construct it from it's
 // coordinates!
10 var frictionCoeff=ggbApplet.getValue("k");
 var myLabel = "k = " + frictionCoeff;
12 var xPos = ggbApplet.getXcoord("beforeLastPos");
 var yPos = ggbApplet.getYcoord("beforeLastPos");
14 var pointPos = "("+xPos+", "+yPos+")";

16 // note the double quotes - they are important here!
 ggbApplet.evalCommand('SetValue[MyLabels, ' +savedLength+ ', { " ' +myLabel+ ' " , '
 + pointPos + '}]');

```

So - geschafft! Das Ergebnis können Sie bei <https://www.geogebra.org/m/MsVRs4pn> runterladen!

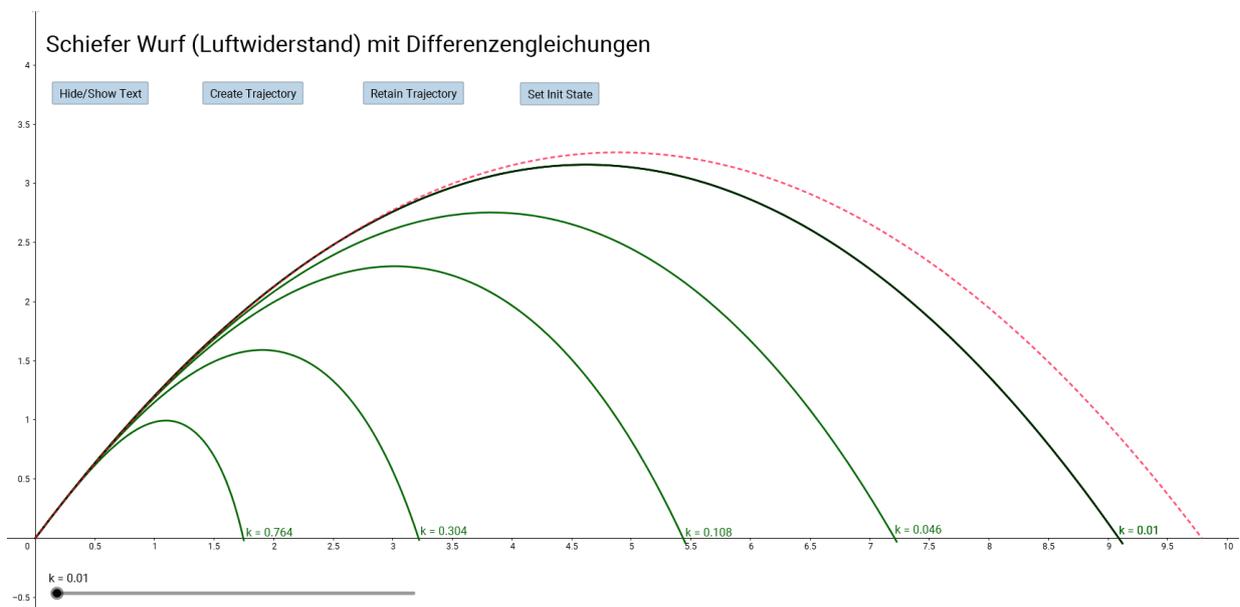


Abb.189 : Vergleich der Flugkurven in *Geogebra*

Den Aufwand mit *Geogebra* und *wxMaxima* vergleichend, scheint mir der in *Geogebra* höher zu sein - um eine Programmierung konnte ich mich in keinem von beiden drücken! Dies gelingt natürlich mit einer Tabellenkalkulation - aber der Preis dafür ist hoch. Hier die Übersicht über die vielen unterschiedlichen Zellen zu behalten, setzt eine gute Organisation voraus!

### 19.3.4 Mit Runge-Kutta Verfahren

Wir können 19.27 (1) als Differentialgleichungssystem anschreiben:

$$\dot{\vec{v}} = -k|\vec{v}| \vec{v} + \vec{g} \quad (19.28)$$

aufgeschlüsselt in Komponenten ergibt das

$$\begin{aligned} \dot{v}_x &= -k\sqrt{v_x^2 + v_y^2} v_x \\ \dot{v}_y &= -k\sqrt{v_x^2 + v_y^2} v_y - 9.81 \end{aligned}$$

Für obiges System können wir das in *wxMaxima* eingebaute Runge-Kutta-Verfahren benützen.

Die Syntax ist einfach: `rk([eq1,eq2], [v1,v2], [x0,y0], [t,t0,tf,Delta_t])`

Übergeben werden die Liste Differentialgleichungen rechte Seite, Liste der abhängigen Variablen, Liste der Anfangswerte und die Liste der unabhängigen Variablen mit ihrem Anfangswert, Endwert und der Schrittweite. Als Output bekommt man eine Liste der Form:

$$[ [t_0, v_1(t_0), v_2(t_0)], \dots [t_i, v_1(t_i), v_2(t_i)] \dots [t_f, v_1(t_f), v_2(t_f)] ]$$

Einige Probleme dieses Verfahrens liegen auf der Hand:

- Wie groß soll man  $t_f$  (t-final) wählen bei bestimmter Auftreffhöhe? (Bleibt nur Probieren)
- Wie groß soll die Schrittweite  $\Delta t$  für eine "genügende"(?) Genauigkeit gewählt werden? Strategie:  $\Delta t_0 = 10$  und  $\Delta t_{i+1} = \Delta t_i \cdot 2$  - das Verfahren für jedes  $\Delta t_i$  durchführen, die Wurfhöhen  $h_i$  und Wurfweiten  $w_i$  bilden jeweils eine Folge. Abbruch wenn für beide folgen gilt

$$|a_{i+1} - a_i| < \varepsilon \quad \text{Cauchy-Kriterium}$$

- Wie integriert man die Geschwindigkeiten  $\vec{v}_i$ ? Von Unter- bzw. Obersumme, Trapez, Simpson, Romberg, Gauß und Spline Verfahren ist hier alles möglich - auch das wirkt sich natürlich auf obige Folgen aus!

Wir verwenden hier Unter- bzw. Obersumme - was ja gleichbedeutend ist mit unserer Differenzengleichung (bzw. Rekursionsformel):

$$\begin{aligned} \vec{x}_{i+1} &= \vec{x}_i + \vec{v}_i \Delta t \quad \Rightarrow \\ \vec{x}_1 &= \vec{x}_0 + \vec{v}_0 \Delta t \quad \vec{x}_2 = \vec{x}_1 + \vec{v}_1 \Delta t = \vec{x}_0 + \vec{v}_0 \Delta t + \vec{v}_1 \Delta t \quad \text{also} \\ \vec{x}_{n+1} &= \vec{x}_0 + \Delta t \sum_{i=0}^n \vec{v}_i \quad \Leftrightarrow \quad \dot{\vec{x}}(t) = \vec{v}(t) \Leftrightarrow \vec{x}(t) = \int_0^t \vec{v}(\tau) d\tau + \vec{x}_0 \end{aligned} \quad (19.29)$$

Wenn man in der letzten Zeile linken und rechten Term vergleicht, erkennt man, dass unsere Summe eine Näherung des Integrals über  $\vec{v}$  darstellt!

Wir implementieren das jetzt in *wxMaxima*:

Zuerst die Angaben, wobei  $\Delta t$  erst beim letzten Durchlauf so klein gewählt wird!

```
(%i6) Delta_t:0.001$ k:0.1$v_x0:6$v_y0:8$x[0]:[0,0]$g:9.81$targetHeight:0$
```

```
(%i8) eq1:-k*sqrt(v_x^2+v_y^2)*v_x; -0.1v_x*sqrt(v_y^2+v_x^2) (eq1)
```

```
(%i9) eq2:-k*sqrt(v_x^2+v_y^2)*v_y-9.81; -0.1v_y*sqrt(v_y^2+v_x^2)-9.81 (eq2)
```

die Liste der Stützpunkte - 3 dimensional! siehe Text!

```
(%i10) sol:rk([eq1,eq2],[v_x,v_y],[v_x0,v_y0],[t,0,1.4,Delta_t])$
```

die rekursive Methode und alternativ die Näherungssumme nach 19.29;  $rest([1,2,3] \rightarrow [2,3])$

```
(%i11) x[i]:=x[i-1] + rest(sol[i])*Delta_t$
```

```
(%i12) y[i]:=x[0]+rest(lreduce("+",firstn(sol,i))*Delta_t$
```

beide liefern natürlich das gleiche Ergebnis - hier ein Beispiel

```
(%i13) x[15]; [0.08937836805567448, 0.1181460047265376] (%o13)
```

```
(%i14) y[15]; [0.08937836805567449, 0.1181460047265376] (%o14)
```

```
(%i15) points:makelist(x[i],i,0,length(sol))$
```

```
(%i16) onlyPosY:sublist(points,lambda([x], (second(x)-targetHeight) >= 0))$
```

```
(%i17) width:lmax(map(lambda([x],first(x)), onlyPosY)); 5.5922 (width)
```

```
(%i18) flightTime:first(last(firstn(sol,length(onlyPosY)))); 1.372 (flightTime)
```

```
(%i19) height:lmax(map(lambda([x],second(x)),points)); 2.3241 (height)
```

```
(%i20) plot2d([discrete,onlyPosY])$
```

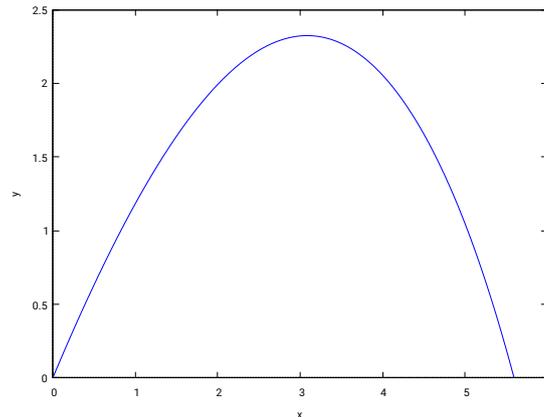


Abb.190 : Bahn mit Runge-Kutta und Differenzgleichung

### 19.3.5 Ermittlung der Bahnkurvengleichung (dimensionslos)

Wir schreiben 19.26 etwas anders an:

$$\vec{a} = \vec{g} - \frac{1}{\ell} v \vec{v} \quad v := |\vec{v}| \quad (19.30)$$

Nach Division durch  $g$  (dimensionslos) ergibt sich folgendes System

$$\begin{cases} \dot{V}_x = -V V_x & (V_x, V_y) := \frac{1}{\sqrt{g\ell}} (v_x, v_y) & V := |\vec{V}| \\ \dot{V}_y = -1 - V V_y & (X, Y) := \frac{1}{\ell} (x, y) & T := t \sqrt{\frac{g}{\ell}} \end{cases} \quad (19.31)$$

(Durch Substitution zeigt man leicht  $V_x \cdot T = X \Leftrightarrow v_x \cdot t = x$ )

Jetzt gehen wir genau so vor wie bei 19.20 und 19.21 und erhalten

$$\begin{aligned} \frac{d^2 Y}{dX^2} &= \frac{d}{dX} \frac{dY}{dX} = \frac{1}{V_x} \frac{d}{dT} \frac{V_y}{V_x} = \frac{\dot{V}_y V_x - V_y \dot{V}_x}{V_x^3} = \\ &= \frac{(-1 - V V_y) V_x - V_y (V V_x)}{V_x^3} = -\frac{1}{V_x^2} \end{aligned} \quad (19.32)$$

Leider ist jetzt  $V_x'$  keine Konstante mehr, sodass uns der Weg des linearen Falles versperrt bleibt:

$$V_x' = \frac{dV_x}{dX} = \frac{dV_x}{dT} \frac{1}{\frac{dX}{dT}} = \dot{V}_x \frac{1}{V_x} = -V(X) = -V_x \sqrt{1 + \left(\frac{V_y}{V_x}\right)^2} = -V_x \sqrt{1 + \left(\frac{dY}{dX}\right)^2}$$

Ein anderer Weg ergibt sich durch ableiten nach  $X$  von 19.32:

$$\frac{d^3 Y}{dX^3} = \frac{2}{V_x^3} \frac{dV_x}{dX} = -\frac{2}{V_x^2} \sqrt{1 + \left(\frac{dY}{dX}\right)^2}$$

Wir benützen nochmal 19.32 und erhalten (zusammen mit den Anfangsbedingungen:

$$\frac{d^3 Y}{dX^3} = 2 \frac{d^2 Y}{dX^2} \sqrt{1 + \left( \frac{dY}{dX} \right)^2} \quad Y(0) = 0, Y'(0) = \tan \alpha, Y''(0) = -\frac{1}{V_{x0}^2} \quad (19.33)$$

Leider hat auch 19.33 keine geschlossene Lösungsformel, aber es lässt sich leicht eine Näherung angeben, wenn  $\frac{dY}{dX} \ll 1$  gilt - die sog. Kurzzeit(“short-time” *st*) Lösung (Abschusswinkel annähernd Null und Flugzeit zu kurz, dass sich viel ändert!) - dann gilt

$$\frac{d^3 Y_{st}}{dX^3} \approx 2 \frac{d^2 Y_{st}}{dX^2} \quad \text{short time approximation} \quad (19.34)$$

### 19.3.6 Lösung der short-time Näherung im Ortsraum

Wir lösen obige short-time-approximation mit *wxMaxima* und überprüfen sie mit Runge-Kutta.

Vorher aber einige erklärende Bemerkungen:

In *wxMaxima* werden gewöhnliche Differentialgleichungen erster und zweiter Ordnung (ordinary differential equations of first or second order) mit

`ode2(<eq>, <dvar>, <ivar>)` gelöst, dabei steht `<dvar>` bzw. `<ivar>` für die abhängige (dependent) bzw. unabhängige (independent) Variable.

Die Differentiation bei der Gleichung ( `<eq>` ) darf nicht ausgewertet werden (Apostroph), d. h. *wxMaxima* setzt hier nur das Operatorensymbol (noun form of operator) ohne es auszuwerten - da ja keine Funktionen eingesetzt werden (!) sondern nur Variable, würde die Differentiation einfach Null ergeben!

Um von der allgemeinen Lösung die *ode2* (die Integrationskonstante wird mit `%c` bezeichnet) liefert, auf die Lösung des Anfangswertproblems zu kommen, bedient man sich der Funktion `ic1(<gsol>, <ivar>=<ival>, <dvar>=<dval>)` – sie erzeugt aus der allgemeinen Lösung (general solution) die spezielle Lösung! Der “1”-er bedeutet, dass es sich um eine Differentialgleichung 1.-er Ordnung handelt. Hier jetzt das Programm:

Bei Dezimalzahlen werden nur 3 Ziffern ausgegeben!

**(%i29)** `fpprintprec:3$`

$z_1 := Y''$  - wir holen uns Schritt für Schritt die nächste Funktion!

**(%i2)** `diffEq1:'diff(z_1,X)=2*z_1;`  $\frac{d}{dX} z_1 = 2z_1$  (diffEq1)

**(%i3)** `gSol1:ode2(diffEq1,z_1,X);`  $z_1 = \%c \%e^{2X}$  (gSol1)

**(%i4)** `spSol1:ic1(gSol1,X=0,z_1=-1/V_X0^2);`  $z_1 = -\frac{\%e^{2X}}{V_{X0}^2}$  (spSol1)

## 19. Der schiefe Wurf

---

$z_2 := Y'$

(%i5) `diffEq2:'diff(z_2,X)=rhs(spSol1);`  $\frac{d}{dX} z_2 = -\frac{v_0 e^{2X}}{V_{X0}^2}$  (diffEq2)

(%i6) `gSol2:ode2(diffEq2,z_2,X);`  $z_2 = v_0 c - \frac{v_0 e^{2X}}{2V_{X0}^2}$  (gSol2)

(%i7) `spSol2:ic1(gSol2,X=0,z_2=tan(%alpha));`  $z_2 = -\frac{v_0 e^{2X} - 2 \tan(\alpha) V_{X0}^2 - 1}{2V_{X0}^2}$  (spSol2)

$z_3 := Y$

(%i8) `diffEq3:'diff(z_3,X)=rhs(spSol2);`  $\frac{d}{dX} z_3 = -\frac{v_0 e^{2X} - 2 \tan(\alpha) V_{X0}^2 - 1}{2V_{X0}^2}$  (diffEq3)

(%i9) `gSol3:ode2(diffEq3,z_3,X);`  $z_3 = v_0 c - \frac{\frac{v_0 e^{2X}}{2} - 2 \tan(\alpha) V_{X0}^2 X - X}{2V_{X0}^2}$  (gSol3)

Hier haben wir jetzt die spezielle Lösung für unsere Funktion  $Y(X)$

(%i10) `spSol3:ic1(gSol3,X=0,z_3=0);`  $z_3 = -\frac{v_0 e^{2X} + (-4 \tan(\alpha) V_{X0}^2 - 2) X - 1}{4V_{X0}^2}$  (spSol3)

Wir schreiben den Term um

(%i11) `solEqXY:Y=expand(rhs(spSol3));`

$$Y = -\frac{v_0 e^{2X}}{4V_{X0}^2} + \frac{X}{2V_{X0}^2} + \tan(\alpha)X + \frac{1}{4V_{X0}^2}$$

(solEqXY)

Substituieren die "alten" Variablen

(%i12) `solEqxy:subst(v_x0/sqrt(g*1),V_X0,(subst(y/l,Y,subst(x/l,X,solEqXY))));`

$$\frac{y}{l} = -\frac{gl v_0 e^{\frac{2x}{l}}}{4v_{x0}^2} + \frac{gx}{2v_{x0}^2} + \frac{\tan(\alpha)x}{l} + \frac{gl}{4v_{x0}^2} \quad (\text{solEqxy})$$

Machen  $y$  explizit

(%i13) `expand(solEqxy*1);`  $y = -\frac{gl^2 v_0 e^{\frac{2x}{l}}}{4v_{x0}^2} + \frac{glx}{2v_{x0}^2} + \tan(\alpha)x + \frac{gl^2}{4v_{x0}^2}$  (%o13)

Definieren  $y$  als Funktion

(%i14) `define(y(x),rhs(%));`  $y(x) := -\frac{gl^2 v_0 e^{\frac{2x}{l}}}{4v_{x0}^2} + \frac{glx}{2v_{x0}^2} + \tan(\alpha)x + \frac{gl^2}{4v_{x0}^2}$  (%o14)

Löschen die Variablen  $z_i$  - wir brauchen sie später für RUNGE-KUTTA

(%i15) `kill(z_1,z_2,z_3)$`

Abschusswinkel in Grad

(%i16) alphaDeg:5\$

Winkel in Bogenmaß, Billardkugel:  $V_{X0} = 1 \leftrightarrow v_{x0} \approx 50 \text{ m/s}$ , Simulationsweite  $0.4 \leftrightarrow \approx 900 \text{ m}$

(%i19) %alpha:alphaDeg\*pi/180\$V\_X0:1\$simWidth:0.4\$

Jetzt Runge-Kutta für Systeme (nur rechte Seite):  $z'_1 = z_2$ ;  $z'_2 = z_3$ ;  $z'_3 = 2z_3 \dots$  mit Variablenliste und Anfangsbedingungen

(%i20) sol5:rk([z\_2,z\_3, 2\*z\_3\*sqrt(1+z\_2^2)], [z\_1,z\_2,z\_3], [0,tan(%alpha),-1/V\_X0^2], [x,0,simWidth,0.01])\$

Zum Zeichnen brauchen wir nur  $X$  und  $z_1 = Y$  - das sind die ersten 2 Listenelemente!

(%i21) plotPoints5:map(lambda([x],firstn(x,2)),sol5)\$

Zum Vergleich unsere Näherungsfunktion!

(%i22) define(Y\_5(X),rhs(ev(solEq-  
XY))); 
$$Y_5(X) := -\frac{e^{2X}}{4} + \tan\left(\frac{\pi}{36}\right)X + \frac{X}{2} + \frac{1}{4} \quad (\%o22)$$

Jetzt dasselbe nochmal mit 15 Grad Abschusswinkel:

(%i24) alphaDeg:15\$%alpha:alphaDeg\*pi/180\$

(%i25) sol15:rk([z\_2,z\_3, 2\*z\_3\*sqrt(1+z\_2^2)], [z\_1,z\_2,z\_3], [0,tan(%alpha),-  
1/V\_X0^2], [x,0,simWidth,0.01])\$

(%i26) plotPoints15:map(lambda([x],firstn(x,2)),sol15)\$

-> define(Y\_15(X),rhs(ev(solEqXY)));

(%i28) plot2d([[discrete,plotPoints5], [discrete,plotPoints15], Y\_5(X), Y\_15(X)], [X,0,simWidth],  
[legend, " rk-5Grad", " rk-15Grad", " Näherung-5", " Näherung-15"],  
[gnuplot\_preamble, "set key bottom left; set xtics font \", 20\";  
set ytics font \", 20\"; set key font \", 20\" "]);

Wie man bei Abb. 191 sieht: Beim 5 Grad totale Überdeckung, bei 15 Grad bei genauem Hinsehen (Billardkugel bei ca. 50 m/s)

## 19. Der schiefe Wurf

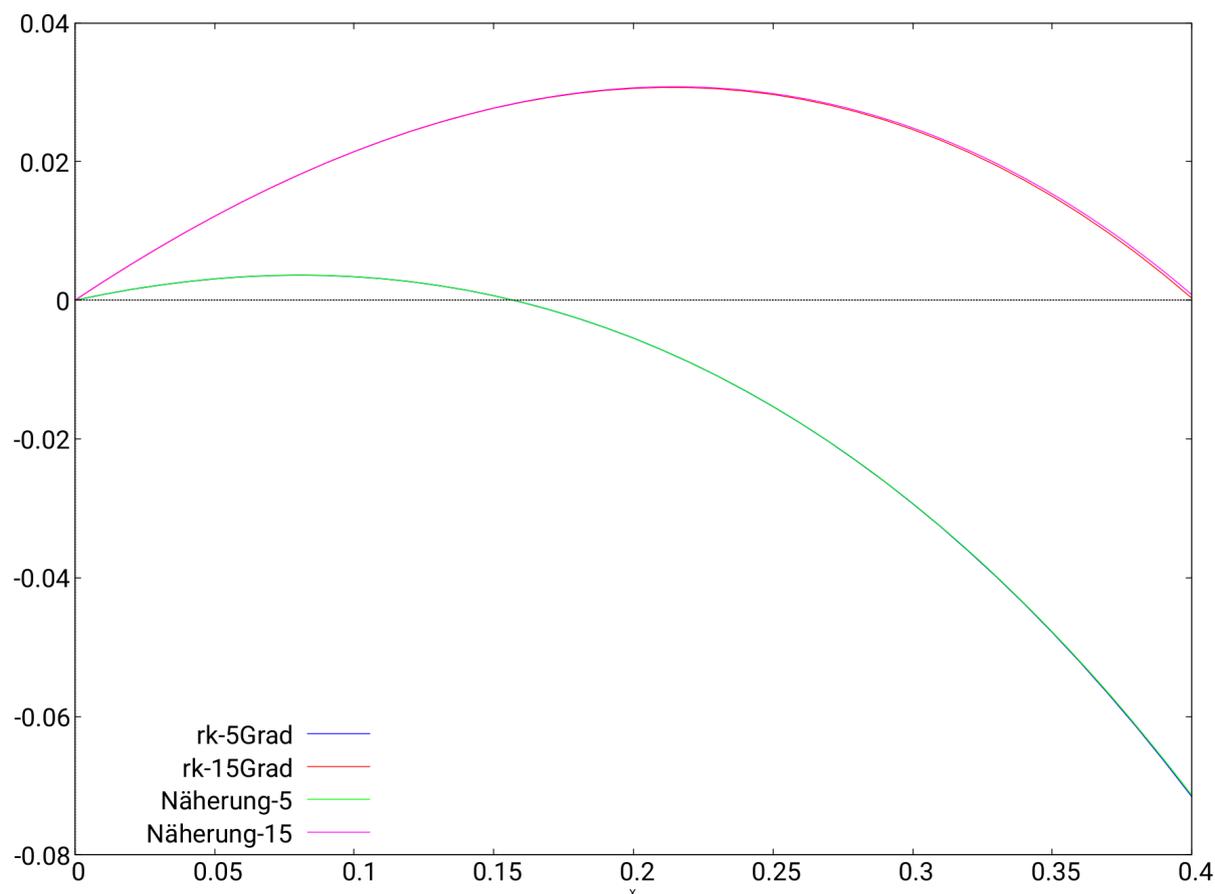


Abb.191 : Short-Time Näherungen vs. Runge-Kutta: kaum unterscheidbar

### 19.3.7 Herleitung und Lösung der short-time Näherung im Zeitraum

Ausgangspunkt sei wieder unser Differentialgleichungssystem 19.31:

$$\begin{cases} \dot{V}_x = -V V_x \\ \dot{V}_y = -1 - V V_y \end{cases}$$

Beachte, dass es sowohl für  $V_x$  als auch für  $V_y$  stabile Endgeschwindigkeiten gibt, bei denen die Zeitableitung auf der linken Seite verschwindet - nennen wir sie  $V_{xf}$  und  $V_{yf}$  (f für "final"):

$$\begin{cases} 0 = -V V_{xf} & \Rightarrow V_{xf} = 0 \\ 0 = -1 - V V_{yf} & \Rightarrow 1 = -|V_{yf}| V_{yf} \Rightarrow V_{yf} = -1 \end{cases}$$

Die Horizontalgeschwindigkeit verschwindet und die Vertikalgeschwindigkeit strebt betragsmäßig gegen 1.

Für z.B. Hagelkörner macht das einen beträchtlichen Unterschied - nehmen wir eine Wolkenhöhe von  $H = 10$  km an (was nicht ungewöhnlich ist für Gewittertürme) ergibt sich bei  $r = 1$  cm

ein

$v_{yf} \approx 22 \text{ m/s}$  gegenüber im Vakuum mit Überschallgeschwindigkeit  $v_{vac} \approx 443 \text{ m/s} \approx 1.3c$

In Luft schmerzt es zwar, im Vakuum würde man von einer ‘‘Gewehrkugel’’ getroffen. Hier die Rechnung:

(%i9) (%rho\_Eis:0.92\*10^3,r:10^(-2),c\_W:0.4,%rho\_Air:1.2,H:10^4,g:9.81)\$

Berechnung von  $\ell$

(%i7) (V:4/3\*r^3\*pi,m:V\*%rho\_Eis, A:r^2\*pi, l:2\*m/(%rho\_Air\*c\_W\*A))\$

(%i10) v\_yf:sqrt(g\*l); 22.39196284384198

(%i12) v\_vac:box(sqrt(2\*H\*g)); 442.944691807002

Zurück zur short-time Näherung: Ist  $\frac{V_y^2}{V_x^2} \ll 1$  über große Teile der Geschosßbahn, kann man 19.31 wegen  $V \approx V_x$  vereinfachen zu

$$\begin{cases} \dot{V}_x = -V_x^2 \\ \dot{V}_y = -1 - V_x V_y \end{cases} \quad (19.35)$$

Die erste Gleichung von 19.35 ist jetzt entkoppelt und kann leicht durch Variablentrennung integriert werden.

Zur leichteren Lösung der Gleichung für  $V_y$  verwenden wir folgenden Trick (ohne Trick siehe ANHANG 19.3.14):

$$h(T) := (\tan \alpha)(T) = \frac{V_y}{V_x} \Rightarrow V_y = h \cdot V_x \Rightarrow \dot{V}_y = \dot{h} V_x + h \dot{V}_x = \dot{h} V_x + \left(\frac{V_y}{V_x}\right) (-V_x^2)$$

wobei beim letzten Gleichheitszeichen die Definition von  $h$  und die erste Zeile von 19.35 verwendet wurden. Vergleich mit der zweiten Zeile von 19.35 führt zu:

$$\dot{h} V_x = -1 \Rightarrow \boxed{h(T) = \frac{V_{y0}}{V_{x0}} - \int_0^T \frac{1}{V_x(t)} dt} \Rightarrow V_y(T) = h(T) \cdot V_x(T) \quad (19.36)$$

Die Ortskoordinaten bekommen wir mit den beiden Formeln

$$X(T) = X_0 + \int_0^T V_x(t) dt \quad Y(T) = Y_0 + \int_0^T V_y(t) dt \quad (19.37)$$

Wenn wir jetzt noch den Zeitparameter eliminieren, müssten wir die short-time Näherung im Ortsraum vor uns haben. Ziehen wir obigen Algorithmus in mit *wxMaxima* durch:

## 19. Der schiefe Wurf

---

Formeln um die Ortskoordinaten zu berechnen

(%i1) `X(T):=X_0+integrate(V_x(t),t,0,T);`

$$X(T) := X_0 + \int_0^T V_x(t) dt \quad (\%o1)$$

(%i2) `Y(T):=Y_0+integrate(V_y(t),t,0,T);`

$$Y(T) := Y_0 + \int_0^T V_y(t) dt \quad (\%o2)$$

Die 1. Gleichung von 19.35 wird gelöst:

(%i3) `diffEq1:'diff(V_x,T)=-V_x^2;`

$$\frac{d}{dT} V_x = -V_x^2 \quad (\text{diffEq1})$$

(%i4) `gSol1:ode2(diffEq1,V_x,T);`

$$\frac{1}{V_x} = T + \%c \quad (\text{gSol1})$$

(%i5) `spSol1:ic1(gSol1,T=0,V_x=V_x0);`

$$\frac{1}{V_x} = \frac{T V_{x0} + 1}{V_{x0}} \quad (\text{spSol1})$$

$V_x$  ist da!

(%i6) `define(V_x(T),rhs(linsolve (spSol1,V_x)[1]));`

$$V_x(T) := \frac{V_{x0}}{T V_{x0} + 1} \quad (\%o6)$$

Jetzt unsere "Hilfsfkt"  $h(T)$  aus 19.36 berechnet

(%i7) `define(h(T),V_y0/V_x0-integrate(1/V_x(t),t,0,T));`

$$h(T) := \frac{V_{y0}}{V_{x0}} - \frac{T^2 V_{x0} + 2T}{2V_{x0}} \quad (\%o7)$$

Nun wird damit  $V_y(T)$  bestimmt

(%i8) `define(V_y(T),ratsimp(V_x(T)*h(T)));`

$$V_y(T) := \frac{2V_{y0} - T^2 V_{x0} - 2T}{2T V_{x0} + 2} \quad (19.38)$$

Einige Voraussetzungen, um  $X$  bzw.  $Y$  zu berechnen

(%i9) `(X_0:0,Y_0:0)$`

(%i10) `assume(T>0,V_x0>0);`

$$[T>0, V_{x0}>0] \quad (\%o10)$$

Die X-Koordinate ist berechnet!

(%i11) eq1: $X=X(T);$   $X = \log(T V_{x0} + 1)$  (eq1)

Die Y-Koordinate wird berechnet und “zerteilt”!

(%i12) distrib(Y(T));  $\frac{\log(T V_{x0} + 1)V_{y0}}{V_{x0}} + \frac{\log(T V_{x0} + 1)}{2V_{x0}^2} - \frac{T}{2V_{x0}} - \frac{T^2}{4}$  (%o12)

Von obigem Ausdruck wird aus dem Term 1 und 2 herausgehoben

h1:factor(part(distrib(Y(T)),1)+part(distrib(Y(T)),2));  $\frac{\log(T V_{x0} + 1)(2V_{x0}V_{y0} + 1)}{2V_{x0}^2}$  (h1)

Wir zerteilen den zweiten Teil des obigen Bruches

(%i14) h2:expand(part(h1,1,2)/part(h1,2));  $\frac{V_{y0}}{V_{x0}} + \frac{1}{2V_{x0}^2}$  (h2)

So - jetzt schreiben wir die Gleichung so an wie sie uns “gefällt”

h3:Y=part(h1,1,1)\*h2+part(distrib(Y(T)),3)+part(distrib(Y(T)),4);

$$Y = \log(T V_{x0} + 1) \left( \frac{V_{y0}}{V_{x0}} + \frac{1}{2V_{x0}^2} \right) - \frac{T}{2V_{x0}} - \frac{T^2}{4} \quad (\text{h3})$$

Wir eliminieren T: dafür holen wir es aus X(T)

(%i16) eq2:solve(eq1,T)[1];  $T = \frac{\%e^X - 1}{V_{x0}}$  (eq2)

Jetzt wir zweimal substituiert: das “Endergebnis” stimmt mit vorigem Abschnitt überein - Gott sei Dank!

(%i17) expand(subst(rhs(eq2),T,subst(X,log(T\*V\_x0+1),h3)));

$$Y = -\frac{\%e^{2X}}{4V_{x0}^2} + \frac{V_{y0}X}{V_{x0}} + \frac{X}{2V_{x0}^2} + \frac{1}{4V_{x0}^2} \quad (\%o17)$$

### 19.3.8 Herleitung einer “long-time”-Näherung im Zeitraum

Um diese Gleichung herzuleiten brauchen wir einige Bausteine:

- Den zweiten Teil von 19.31 umgeschrieben:

$$\dot{V}_y = -1 - V_y (V_x^2 + V_y^2)^{\frac{1}{2}} \Rightarrow -\frac{\dot{V}_y + 1}{V_y V_x} = \left[ 1 + \left( \frac{V_y}{V_x} \right)^2 \right]^{\frac{1}{2}} \quad (19.39)$$

- Mit h(T) von 19.36 wird daraus

$$-\frac{\dot{V}_y + 1}{V_y V_x} = \left[ 1 + (h(T))^2 \right]^{\frac{1}{2}} \Rightarrow -\frac{\dot{V}_y + 1}{V_y V_x} = \left[ 1 + \left( \frac{V_{y0}}{V_{x0}} - \int_0^T \frac{1}{V_x(t)} dt \right)^2 \right]^{\frac{1}{2}} \quad (19.40)$$

## 19. Der schiefe Wurf

- Aus 19.31 können wir die “lästige” Wurzel  $V$  eliminieren:

$$\frac{\dot{V}_y + 1}{\dot{V}_x} = \frac{V_y}{V_x} \Rightarrow \frac{\dot{V}_y + 1}{V_y} = \frac{\dot{V}_x}{V_x} \quad (19.41)$$

das setzen wir in die vorige Gleichung ein und erhalten



$$-\frac{\dot{V}_x}{V_x^2} = \left[ 1 + \left( \frac{V_{y0}}{V_{x0}} - \int_0^T \frac{1}{V_x(t)} dt \right)^2 \right]^{\frac{1}{2}} \quad (19.42)$$

- Mit der Substitution  $z(t) := (V_x(t))^{-1}$  wird aus 19.42

$$\dot{z} = \left[ 1 + \left( \frac{V_{y0}}{V_{x0}} - \int_0^T z(t) dt \right)^2 \right]^{\frac{1}{2}} \quad (19.43)$$

Unsere Strategie ist jetzt:

- Lösung von 19.43 für  $T \gg 1$
- $V_x(T) = z^{-1}$
- 

$$V_y(T) = V_x(T) h(T) = V_x \left( \frac{V_{y0}}{V_{x0}} - \int_0^T z(t) dt \right)$$



$$\lim_{T \rightarrow \infty} V_x = 0 \Rightarrow \lim_{T \rightarrow \infty} z(T) = \infty \Rightarrow \lim_{T \rightarrow \infty} \int_0^T z(t) dt = \infty$$

Die rechte Seite von Gleichung 19.43 hat folgende Struktur für  $T \gg 1$ :

$$\left[ 1 + (a - x)^2 \right]^{\frac{1}{2}} \quad \text{wobei } x \text{ (das Integral) sehr groß ist!}$$

Wir können also obigen Ausdruck mit einer asymptotischen Taylorreihe (“Entwicklungspunkt ist  $\infty$ ”) annähern (siehe Anhang 19.3.12):

$$\left[ 1 + (a - x)^2 \right]^{\frac{1}{2}} \approx (x - a) + \frac{1}{2x} + \frac{a}{2x^2} + \frac{4a^2 - 1}{8x^3} + \dots \quad (19.44)$$

Wir brechen 19.44 bereits nach dem 1. Reihenglied ab und damit geht 19.43 über in

$$\dot{z} \approx \int_0^T z(t) dt - \frac{V_{y0}}{V_{x0}}$$

durch Differenzieren geht dies in folgendes Anfangswertproblem über

$$\ddot{z} = z \quad \dot{z}(0) = -\frac{V_{y0}}{V_{x0}} \quad z(0) = \frac{1}{V_{x0}} \quad (19.45)$$

So jetzt die Implementation in *wxMaxima*:

(%i1) `ratprint:false$;`

Berechnungsformel für  $V_y$

(%i2) `V_y(T):=V_x(T)*(V_y0/V_x0 - integrate(z(t),t,0,T));`  $V_y(T) := V_x(T) \left( \frac{V_{y0}}{V_{x0}} - \int_0^T z(t) dt \right)$

Anfangswertproblem 19.45 wird gelöst

(%i3) `eq:diff(z,t,2) - 'diff(z,t,0) = 0;`  $\frac{d^2}{dt^2} z - z = 0$  (eq)

(%i4) `sol1:ode2(eq,z,t);`  $z = \%k1 \%e^t + \%k2 \%e^{-t}$  (sol1)

(%i5) `expr:(ic2(sol1,t=0,z=1/V_x0,'diff(z,t)=-V_y0/V_x0));`  $z = \frac{(V_{y0} + 1) \%e^{-t}}{2V_{x0}} - \frac{(V_{y0} - 1) \%e^t}{2V_{x0}}$

Wir substituieren die hyperbolischen Funktionen für die Exponentialfkt.

(%i6) `define(z(t),rhs(expand(trigsimp(subst(cosh(t)+sinh(t),%e^t,(subst(-sinh(t)+cosh(t),%e^(-t),expr))))));`

$$z(t) := \frac{\cosh(t)}{V_{x0}} - \frac{V_{y0} \sinh(t)}{V_{x0}}$$

$V_x$  ist die reziproke Fkt. von  $z$ ,  $V_y$  wurde schon festgelegt

(%i7) `define(V_x(T),trigsimp(1/z(T)));`  $V_x(T) := -\frac{V_{x0}}{\sinh(T) V_{y0} - \cosh(T)}$  (%o7)

(%i8) `display(V_y(T))$`  $V_y(T) = -\frac{\cosh(T) V_{y0} - \sinh(T)}{\sinh(T) V_{y0} - \cosh(T)}$

Jetzt RUNGE-KUTTA

(%i10) `$V_x0:1$V_y0:0.2$`

(%i11) `eq1:-sqrt(V_x^2+V_y^2)*V_x$`

(%i12) `eq2:-sqrt(V_x^2+V_y^2)*V_y-1$`

## 19. Der schiefe Wurf

(%i14) Delta\_t:0.01\$ simWidth:10\$

(%i15) sol\_u: rk([eq1,eq2],[V\_x,V\_y],[V\_x0,V\_y0],[t,0,simWidth,Delta\_t])\$

(%i16) plotListVx:map(lambda([x],firstn(x,2)),sol\_u)\$

(%i17) plotListVy:map(lambda([x],[first(x),last(x)]),sol\_u)\$

Die short-time Näherungen zum Vergleich

(%i18) Vst\_x(T):=V\_x0/(T\*V\_x0+1);

$$V_{st_x}(T) := \frac{V_{x0}}{T V_{x0} + 1}$$

(%i19) Vst\_y(T):=(2\*V\_y0-T^2\*V\_x0-2\*T)/(2\*T\*V\_x0+2);

$$V_{st_y}(T) := \frac{2V_{y0} - T^2 V_{x0} + (-2) T}{2T V_{x0} + 2}$$

(%i20) plot2d([Vst\_y(x),V\_y(x),[discrete,plotListVy]], [x,0,8], [y,-1.1,0.22],  
 [gnuplot\_term,"qt-0"], [title, "velocity in y-direction"],  
 [legend, " short-time-V\_y", " long-time-V\_y", " RUNGE-KUTTA-V\_y"],  
 [style,[lines,3,1,2], [lines,3,2,2],[lines,3,3,2]],  
 [gnuplot\_preamble, "set key top right;  
 set xtics font \", 15\"; set ytics font \", 15\";  
 set key font \", 15\"; set title font \", 20\" "]  
 )\$

(%i21) plot2d([Vst\_x(x),V\_x(x),[discrete,plotListVx]], [x,0,8],  
 [gnuplot\_term,"qt-1"], [title, "velocity in x-direction"],  
 [legend, " short-time-V\_x", " long-time-V\_x", " RUNGE-KUTTA-V\_x"],  
 [style,[lines,3,1,2], [lines,3,2,2],[lines,3,3,2]],  
 [gnuplot\_preamble, "set key top right;  
 set xtics font \", 15\"; set ytics font \", 15\";  
 set key font \", 15\"; set title font \", 20\" "]  
 )\$

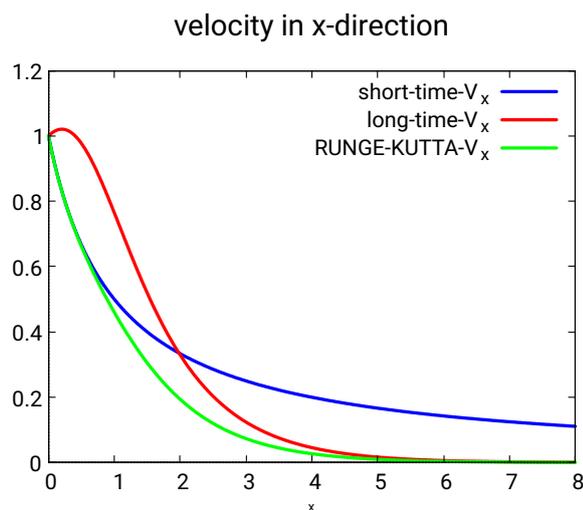


Abb.192 : Vergleich x

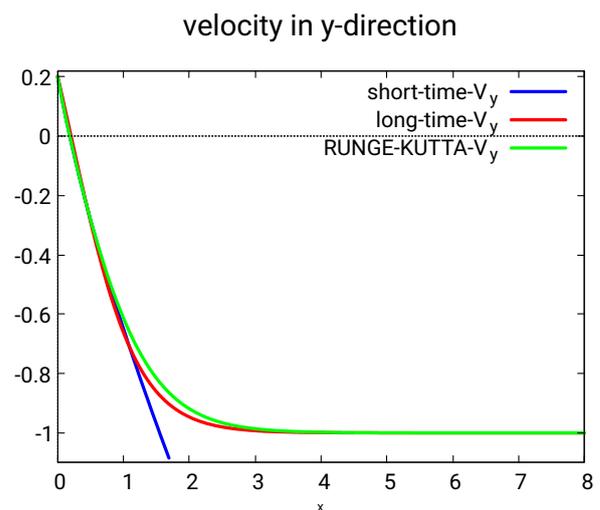
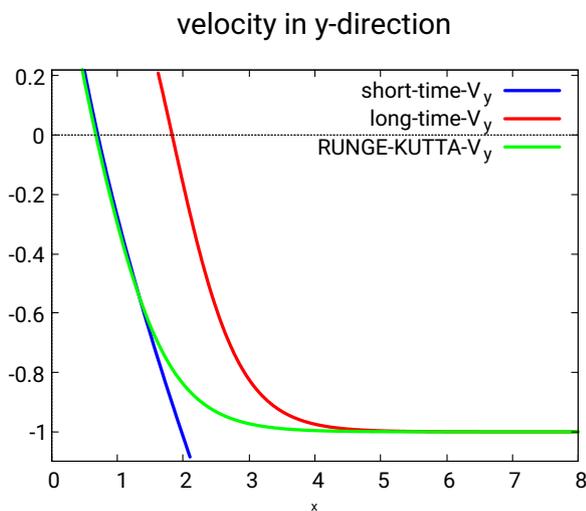
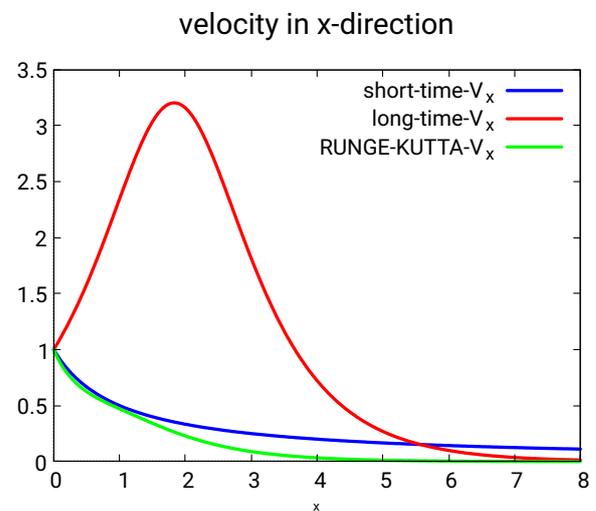


Abb.193 : Vergleich y

Wir haben vorausgesetzt, dass für  $T \gg 1$  das Integral  $\int_0^T z dt$  extrem groß wird, das schauen wir uns noch genauer an:

$$\int_0^T z dt \approx \frac{e^T \overbrace{(1 - V_{y0})}^{!!}}{2 V_{x0}} - \frac{e^{-T} (V_{y0} + 1)}{2 V_{x0}} + \frac{V_{y0}}{V_{x0}}$$

Also  $V_{y0}$  sollte kleiner 1 sein, wenn eine "ordentliche" Konvergenz gewährleistet sein soll! Der Graph für  $V_{y0} = 0.95$  zeigt gleich, wie schlecht die long-time-Näherung dann gleich ist:

Abb.194 :  $V_x$  für  $V_{y0} = 0.95$ Abb.195 :  $V_x$  für  $V_{y0} = 0.95$ 

Man sieht an beiden Graphen wie spät jetzt die long-time-Näherung sich an die tatsächliche Lösung schmiegt - die anfangs steigende x-Geschwindigkeit ist natürlich total daneben.



Eine Trajektorie mit Hilfe der long-time-Näherung ist nicht möglich, da sich der anfängliche Fehler so akkumuliert, sodass die Formeln 19.37 kein sinnvolles Ergebnis liefern. Nur wenn man zu einem späten Zeitpunkt  $t_1$  die Koordinaten  $X_1$  und  $Y_1$  kennen würde, könnte man diese Formeln sinnvoll anwenden!

19.3.9 Implizite Lösung der Bahnkurvengleichung im Zeitraum

Wir starten bei 19.43:

$$\dot{z} = \left[ 1 + \left( \frac{V_{y0}}{V_{x0}} - \int_0^T z(t) dt \right)^2 \right]^{\frac{1}{2}} \tag{19.46}$$

und wählen folgenden Substitution

$$w(T) = \int_0^T z(t) dt - \left( \frac{V_{y0}}{V_{x0}} \right) \quad \text{damit gilt } \dot{w} = z \tag{19.47}$$

damit wird 19.46 zu

$$\ddot{w} = [1 + w^2]^{\frac{1}{2}} \tag{19.48}$$

multiplizieren wir mit  $\dot{w}$  ergibt sich folgender Ausdruck für die noch unbekannte Funktion  $f(w)$ :

$$\ddot{w} \dot{w} = [1 + w^2]^{\frac{1}{2}} \dot{w} \Leftrightarrow \frac{1}{2} \frac{d}{dT} (\dot{w}^2) = \frac{df}{dw} \underbrace{\frac{dw}{dT}}_{\dot{w}} \Rightarrow \frac{df}{dw} = [1 + w^2]^{\frac{1}{2}} \tag{19.49}$$

$$\Rightarrow f(w) = \frac{1}{2} \left( \operatorname{asinh}(w) + w\sqrt{1 + w^2} \right) + C \tag{19.50}$$

Mit der Integration von 19.49 ergibt sich

$$\begin{aligned} \frac{1}{2} \frac{d}{dT} (\dot{w}^2) &= \frac{df}{dw} \frac{dw}{dT} \quad \Big| \int_0^t \cdot dT \\ \Rightarrow \frac{1}{2} [\dot{w}^2 - \dot{w}_0^2] &= f(w) - f(w_0) \Rightarrow \dot{w} = \underbrace{\sqrt{2f(w) + \dot{w}_0^2 - 2f(w_0)}}_{s(w)} \end{aligned} \tag{19.51}$$

dabei gilt

$$w_0 = w(0) \stackrel{19.47}{=} - \left( \frac{V_{y0}}{V_{x0}} \right) \quad \dot{w}_0 = \frac{1}{z_0} = \frac{1}{V_{x0}} \quad \dot{w} = s(w)$$

19.51 können wir mit Variablentrennung integrieren und erhalten

$$\int_{w_0}^w \frac{dv}{s(v)} = T \Leftrightarrow S(w) = T \rightarrow w = S^{-1}(T) \tag{19.52}$$

Selbst wenn es gelingen sollte  $S$  (das Integral) zu berechnen, ist die Umkehrung unmöglich! Numerisch ist es aber möglich (siehe ANHANG 19.3.13)

$$T \xrightarrow{19.52} w \xrightarrow{s(w)=\dot{w}} \dot{w} \xrightarrow{\dot{w}=z} z \xrightarrow{z=V_x^{-1}} V_x \xrightarrow{19.36} V_y \Rightarrow \begin{cases} T \longrightarrow V_x \\ T \longrightarrow V_y \end{cases}$$

Diese Möglichkeit schauen wir uns nun in *wxMaxima* an und überprüfen das Ergebnis mit dem Resultat des ursprünglichen Differentialgleichungssystems:

(%i2) `V_x0:1$V_y0:0.2$`

(%i5) `w_0:-V_y0/V_x0$w_dot_0:1/V_x0$w_f:150$`

(%i6) `invert(1):=map(lambda([x],[second(x),first(x)]),1)$ /* was never used */`

Differentialgleichung für  $f(w)$  wird gelöst

(%i7) `eq:diff(f,w)=sqrt(1+w^2);`  $\frac{d}{dw}f = \sqrt{w^2 + 1}$  (eq)

(%i8) `logarc:true$ /*asinh is substuted by log */;`

(%i9) `sol:ode2(eq,f,w);`  $f = \frac{\log(\sqrt{w^2 + 1} + w)}{2} + \frac{w\sqrt{w^2 + 1}}{2} + \%c$  (sol)

→ Integrationskonstante  $C$  wegen Differenz unwichtig!

(%i10) `soll:ex-pand(ic1(sol,w=0,f=0));`  $f = \frac{\log(\sqrt{w^2 + 1} + w)}{2} + \frac{w\sqrt{w^2 + 1}}{2}$  (sol1)

(%i11) `define(f(w), rhs(sol1));`  $f(w) := \frac{\log(\sqrt{w^2 + 1} + w)}{2} + \frac{w\sqrt{w^2 + 1}}{2}$  (%o11)

(%i12) `s(w):=sqrt(2*f(w)+w_dot_0^2-2*f(w_0));`  $s(w) := \sqrt{2f(w) + w_{dot_0}^2 + (-2)f(w_0)}$  (%o12)

Wir bekommen die  $w \leftrightarrow T$  Tabelle von Runge-Kutta

(%i13) `wT_list:rk(1/s(w),y,0,[w,w_0,w_f,0.1])$`

zur Erinnerung:  $\dot{w} = s(w) = 1/V_x$

(%i14) `get_T_To_Vx(rk_list):=map(lambda([x],[second(x),1/s(first(x))]),rk_list)$`

zur Erinnerung:  $V_y = -w/\dot{w} = -w/s(w)$

(%i15) `get_T_To_Vy(rk_list):=map(lambda([x],[second(x),-first(x)/s(first(x))]),rk_list)$`

(%i16) `T_max:last(last(wT_list));` 5.40561416361086 (T\_max)

Zum Vergleich lösen wir die originalen Differentialgleichungen

(%i17) `eq1:-sqrt(v_x^2+v_y^2)*v_x;`  $-v_x\sqrt{v_y^2 + v_x^2}$  (eq1)

## 19. Der schiefe Wurf

(%i18)  $1;$   $-v_y \sqrt{v_y^2 + v_x^2} - 1$  (eq2)

(%i19) `sol_rk: rk([eq1,eq2],[v_x,v_y],[V_x0,V_y0],[t,0,T_max,0.1])$`

Wir extrahieren  $V_x$  bzw.  $V_y$

(%i20) `only_t_Vx(l):=map(lamb-  
da([x],firstn(x,2)),l)$`

(%i21) `plot2d([[discrete,get_T_To_Vx(wT_list)],[discrete,only_t_Vx(sol_rk)] ],  
[y,0,1],[x,0,T_max], [gnuplot_term,"qt-1"], [title, "velocity in x-direction"],  
[legend, " V_x-via implicit equation"," V_x-via original diffEqu "],  
[style,[lines,6,1,2],[lines,3,2,2]], [gnuplot_preamble, "set key top right;  
set xtics font\", 15\"; set ytics font\", 15\"; set key font\", 15\";  
set title font\", 20\" "])$`

(%i22) `only_t_Vy(l):=map(lamb-  
da([x],[first(x),third(x)]),l)$`

(%i23) `plot2d([[discrete,get_T_To_Vy(wT_list)],[discrete,only_t_Vy(sol_rk)] ],  
[gnuplot_term,"qt-2"],[x,0,T_max], [title, "velocity in y-direction"],  
[legend, " V_y-via-implicit"," V_y-via-diffEq"],  
[style,[lines,6,1,2],[lines,3,2,2]], [gnuplot_preamble, "set key top right;  
set xtics font\", 15\"; set ytics font\", 15\"; set key font\", 15\";  
set title font\", 20\" "])$`

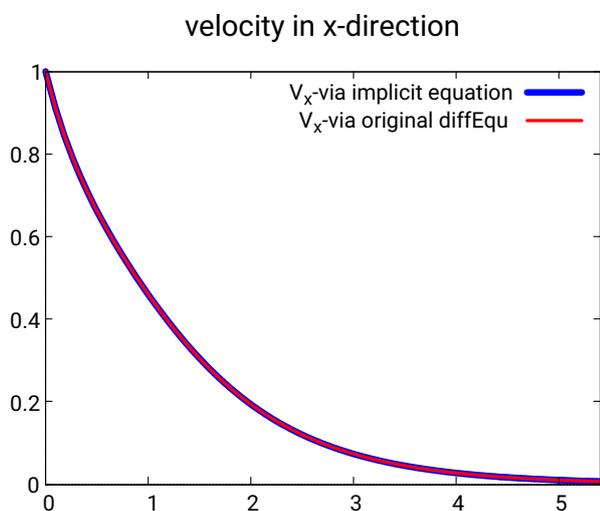


Abb.196 : Übereinstimmung der beiden Lösungen

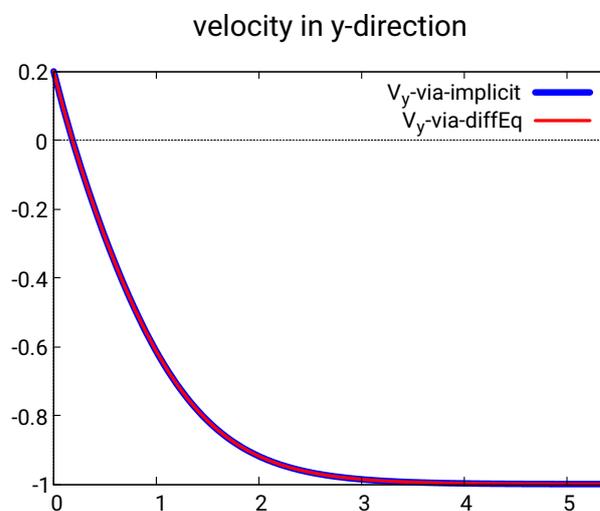


Abb.197 : Übereinstimmung der beiden Lösungen

Wir haben früher bei einer Billardkugel berechnet  $\frac{1}{\ell} \approx 4 \cdot 10^{-3} \Rightarrow \ell \approx 250$

Bei unseren Anfangsbedingungen

$V_x = 1, V_y = 0.2 \Rightarrow \alpha \approx 11^\circ$  wäre bei  $T \approx 5 \Rightarrow t \approx 5T = 25 s$

der Endzustand erreicht:

$$v_x \approx 0 \quad v_y = \sqrt{g\ell}(-1) \approx -50 \text{ m/s}$$

## 19. Der schiefe Wurf

Aber jetzt noch zur Bahnkurve: Natürlich geht es auch wieder mit 19.29, aber es geht auch anders:

$$V_x = \frac{dX}{dt} = \frac{dX}{dw} \frac{dw}{dt} \Rightarrow \frac{dX}{dw} = \frac{V_x}{\dot{w}} \quad \text{außerdem gilt} \quad V_x = \frac{1}{\dot{w}} = \frac{1}{s(w)} \quad (19.53)$$

damit ergibt sich

$$\begin{aligned} \int_0^T V_x(t) dt &= \int_{w_0}^w \frac{dX}{dw} dw = \int_{w_0}^w \frac{V_x}{\dot{w}} dw = \int_{w_0}^w \frac{dw'}{[s(w')]^2} \Rightarrow \\ X(T(w)) &= X_0 + \int_0^T V_x(t) dt = X_0 + \int_{w_0}^w \frac{dw'}{[s(w')]^2} \end{aligned} \quad (19.54)$$

Bei  $Y(T)$  läuft es ähnlich, nur gilt hier  $V_y = -\frac{w}{\dot{w}}$  damit ergibt sich

$$Y(T(w)) = Y_0 + \int_{w_0}^w \frac{V_y}{\dot{w}} dw = - \int_{w_0}^w \frac{w'}{[s(w')]^2} dw' \quad (19.55)$$

wobei die bijektive Abbildung  $w$  zu  $T$  bzw. umgekehrt nur als Tabelle vorliegt. Aber wir können für die Berechnung der Bahn nur auf die  $w$ -Werte bzw.  $s(w)$ -Werte zurückgreifen. Als "Probe" benutzen wir die *short-time* Approximation.

Diese Methode testen wir mit *wxMaxima*:

```
(%i2) V_x0:1$V_y0:0.2$
```

Hier noch einmal die short-time Näherung

```
(%i3) Y_st(X):=-%e^(2*X)/(4*V_x0^2)+X/(2*V_x0^2)+V_y0/V_x0*X+1/(4*V_x0^2);
```

$$Y_{st}(X) := \frac{-e^{2X}}{4V_{x0}^2} + \frac{X}{2V_{x0}^2} + \frac{V_{y0}}{V_{x0}}X + \frac{1}{4V_{x0}^2} \quad (%o3)$$

```
(%i6) w_0:-V_y0/V_x0$w_dot_0:1/V_x0$w_f:0.4$
```

```
(%i7) f(w):=log(sqrt(w^2+1)+w)/2+(w*sqrt(w^2+1))/2; f(w) := \frac{\log(\sqrt{w^2+1}+w)}{2} + \frac{w\sqrt{w^2+1}}{2}
```

```
(%i8) s(w):=sqrt(2*f(w)+w_dot_0^2-2*f(w_0)); s(w) := \sqrt{2f(w) + w_0^2 + (-2)f(w_0)}
```

Wir berechnen die Tabellen mit Runge-Kutta

```
(%i9) wX_list:rk(1/(s(w))^2,X,0,[w,w_0,w_f,0.01])$
```

```
(%i10) wY_list:rk(-w/(s(w))^2,Y,0,[w,w_0,w_f,0.01])$
```

Wir stellen die Plot-Liste zusammen

```
(%i11) getXY_list(xl,y1):=block([xy_list:[]],
 for i thru length(xl) do
 xy_list:cons([second(xl[i]),second(y1[i])],xy_list),
 reverse(xy_list))\$$
```

```
(%i12) XY_list:getXY_list(wX_list,wY_list)$
```

```
(%i13) plot2d([Y_st(w),[discrete,XY_list]], [w,0,0.4],
 [title, "implicit vs. short-time trajectory"],
 [legend, " short-time"," implicit"], [style,[lines,6,1,2], [lines,3,2,2]],
 [gnuplot_preamble, "set key top right; set xtics font \", 15\";
 set ytics font \", 15\"; set key font \", 15\"; set title font \", 20\" "])$
```

Wir berechnen das Integral  $\int_{w_0}^{w_f} 1/s(w) dw = T_f$  um die Endzeit zu bestimmen

```
print("w=0.4 entspricht die Zeit: T =",quad_qag(1/s(w),w,w_0,w_f,3)[1])$
```

w=0.4 entspricht die Zeit: T = 0.4825063259906707

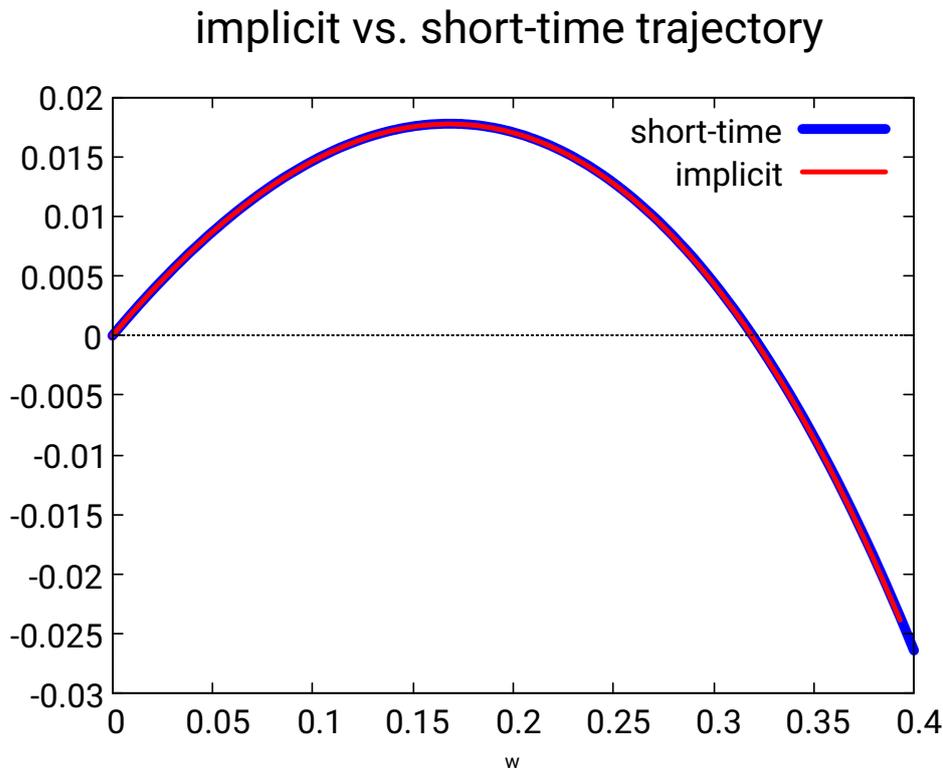


Abb.198 : Short-time vs. implicit trajectory

### 19.3.10 Allgemeine Lösung der Bahnkurvengleichung im Ortsraum

Wie schon gesagt, besitzt 19.33 keine geschlossene Lösung, aber es ist eine Reihenlösung a la Taylor angebbbar. Mit CAS-Tools wie *wxMaxima* sind ziemlich genaue Ergebnisse erzielbar. Aber jetzt der "Reihe" nach:

$$Y(X) = \sum_k^{\infty} \frac{a_k}{k!} X^k \quad \text{mit} \quad a_k := Y^{(k)}(0)$$

wobei  $a_0$ ,  $a_1$  und  $a_2$  bereits von 19.33 bekannt sind. Die folgenden kann man mit 19.33 rekursiv berechnen:

$$a_3 = 2a_2 \sqrt{1 + a_1^2}$$

um  $a_4$  zu erhalten, muss man 19.33 ableiten:

$$Y^{(4)} = 2 \frac{Y^{(3)}(1 + Y'^2) + Y'' Y'}{\sqrt{1 + Y'^2}} \Rightarrow a_4 = 2 \frac{a_3(1 + a_1^2) + a_2 a_1}{\sqrt{1 + a_1^2}}$$

Für  $a_5$  wieder ableiten und die bisher bekannten  $a_i$  einsetzen.

Bezeichnen wir 19.33 mit `eq[1]` und Ableitung davon mit `eq[2]` und Ableitung davon mit `eq[3]` ergibt sich folgendes Schema (`calcCoeff(till)`):

```

a[1], a[2] → eq[1] → a[3]
a[1], ... a[3] → eq[2] → a[4]
a[1], ... a[4] → eq[3] → a[5]
a[1], ... a[5] → eq[4] → a[6]
:

```

Dieses Verfahren ist zwar mühsam, eignet sich aber gut für ein Computer Algebra System (CAS) wie *wxMaxima*. Also muss ein Programm her - einige Besonderheiten vorweg:

`a:make_array(hashed,30)` reserviert ein Array mit maximal 30 Feldern, deren Daten mathematische Ausdrücke sind - keine Zahlen

`depends(Y,X)` gibt *wxMaxima* zu verstehen, dass  $Y$  von  $X$  abhängt, obwohl wir diese Abhängigkeit nicht kennen.



Hier nun der Vergleich zwischen dem numerischen RUNGE-KUTTA-Verfahren, unserer Potenzreihe bis zum Grad 16 und der short-time Näherung für 30 Grad - wir sehen in Abb. 199, das zwischen Potenzreihe und Runge-Kutta kein wahrnehmbarer Unterschied ist und auch die short-time-Näherung noch beachtlich gut ist.

Wäre der Abschusswinkel flacher könnte auch sie gut mithalten!

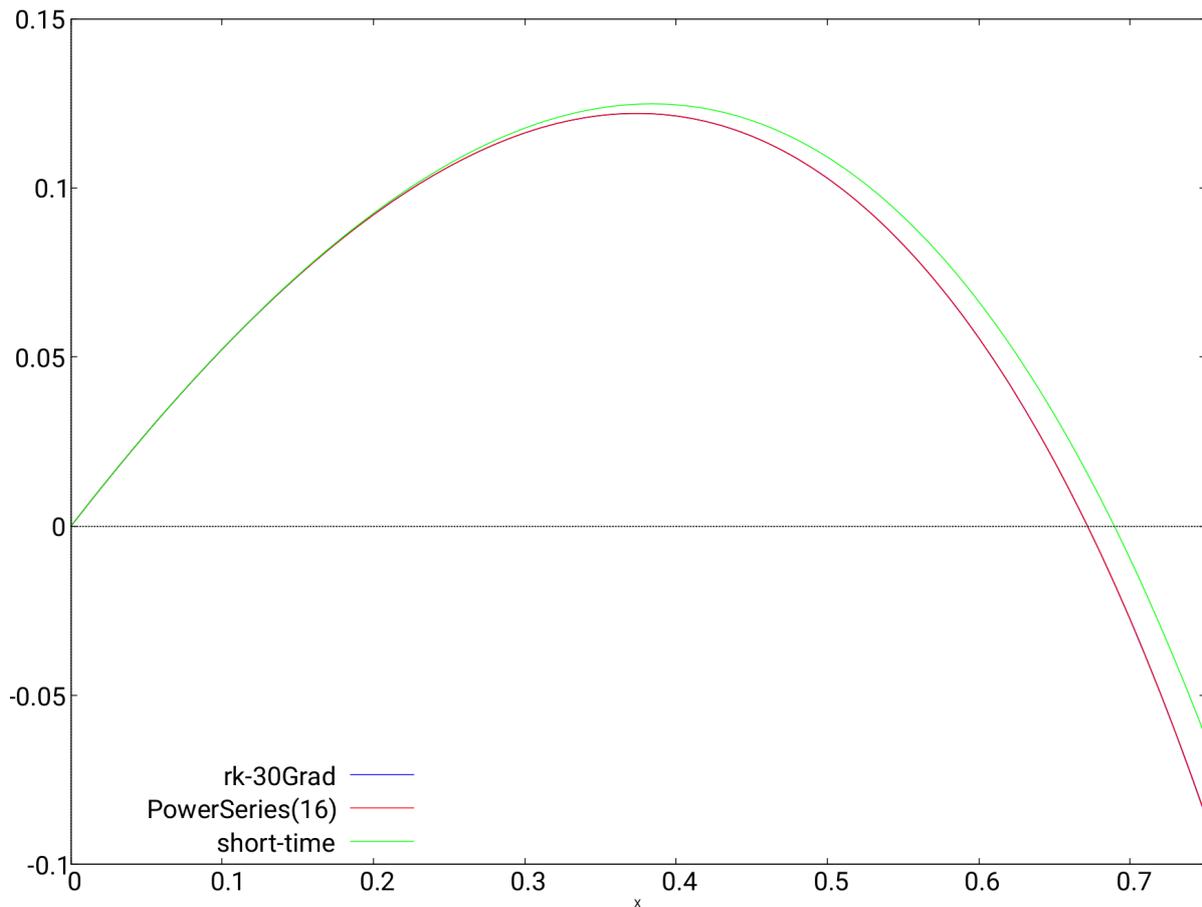


Abb.199 : Vergleich shorttime-Näherung, Potenzreihe und Runge-Kutta

Rechnet man die dimensionslosen Größen für eine Billardkugel um, ergeben die Wurfweite, Wurfhöhe und Anfangsgeschwindigkeit in x-Richtung für obige Bahn folgende Zahlen Zahlen:

$$\begin{aligned} \ell = \frac{1}{k} &\approx 230 & X = 0.67 &\Rightarrow x = X \cdot \ell \approx 154 \text{ m} \\ Y = 0.12 && Y = 0.12 &\Rightarrow y = Y \cdot \ell \approx 28 \text{ m} \\ V_{X0} = 1 && V_{X0} = 1 &\Rightarrow v_{x0} = 1\sqrt{g\ell} \approx 48 \text{ m/s} \approx 171 \text{ km/h} \end{aligned}$$

Hier nun der Code für obige Bahnkurve:

Wir reservieren “hashed-arrays” - siehe oben im Text!

```
(%i3) a:make_array(hashed,30)$eq:make_array(hashed,30)$fpprintprec:2$
```

Die Anfangswerte für die Potenzreihe werden festgelegt

```
(%i6) a[0]:0$a[1]:tan(%alpha)$a[2]:-1/V_X0^2$
```

## 19. Der schiefe Wurf

---

$Y$  ist abhängig von  $X$ ; mit *dependencies* werden alle Abhängigkeiten aufgelistet

```
(%i8) depends(Y,X)$dependencies;
```

```
[Y(X)] (o8)
```

Die Ausgangsdifferentialgleichung wird festgelegt

```
(%i9) eq[1]:'diff(Y,X,3)=2*'diff(Y,X,2)*sqrt(1+'diff(Y,X,1)^2);
```

$$\frac{d^3}{dX^3}Y = 2\sqrt{\left(\frac{d}{dX}Y\right)^2 + 1} \left(\frac{d^2}{dX^2}Y\right) \quad (\%o9)$$

Ersetzt in *eq* der Reihe nach  $Y^{(i)}$  durch  $a_i = Y^{(i)}(0)$  mit  $i \in \{1, 2, \dots, n\}$  und berechnet  $a_{n+1}$

```
(%i10) substn(eq,n):=block([res:rhs(eq),k:n+1],
 assume(cos(%alpha)\ensuremath{>}0),
 for i:1 thru n do res:subst(a[i],'diff(Y,X,i),res),
 a[k]:expand(ratsimp(trigsimp(res)))
)$
```

Die neue Differentialgleichung wird berechnet (Doppelapostroph) und mit der alten neuer Koeffizient berechnet - siehe obiges Schema!

```
(%i11) calcCoeff(till):=block(
 for j:2 thru till do (
 eq[j]:ratsimp('diff(eq[j-1],X,1)),
 substn(eq[j-1],j)
)
)$
```

Berechnung bis  $a_{16}$

```
(%i12) calcCoeff(15);
```

Die short-time Näherung - die brauchen wir zum Vergleich später!

```
(%i14) Y_st:=%e^(2*X)/(4*V_X0^2)+X/(2*V_X0^2)+tan(%alpha)*X+1/(4*V_X0^2);
```

$$-\frac{e^{2X}}{4V_{X0}^2} + \frac{X}{2V_{X0}^2} + \tan(\alpha)X + \frac{1}{4V_{X0}^2} \quad (Y\_st)$$

Die Koeffizienten “unserer” Potenzreihe im Vergleich mit der short-time Näherung

```
(%i15) displayDiff(till):=block(
 b:make_array(hash,30),
 Y_st_taylor: taylor(Y_st,X,0,20),
 print("Coeff of X^i", " ", "short-time", " ", " ", "power-series"),
 for i thru till do (
 b[i]: coeff(Y_st_taylor,X,i),
 print("i =",i," ",b[i]," ", " ", expand(ratsimp(a[i]/i!)))
)
)$
```

Jetzt die Ausführung: Bis  $\mathcal{O}(2)$  herrscht Übereinstimmung

**(%i16)** `displayDiff(4);`

| Coeff of $X^i$ | short-time             | power-series                                                          |
|----------------|------------------------|-----------------------------------------------------------------------|
| i=1            | $\tan(\alpha)$         | $\tan(\alpha)$                                                        |
| i=2            | $-\frac{1}{2V_{X0}^2}$ | $-\frac{1}{2V_{X0}^2}$                                                |
| i=3            | $-\frac{1}{3V_{X0}^2}$ | $-\frac{1}{3\cos(\alpha)V_{X0}^2}$                                    |
| i=4            | $-\frac{1}{6V_{X0}^2}$ | $\frac{\sin(\alpha)}{12V_{X0}^4} - \frac{1}{6\cos(\alpha)^2V_{X0}^2}$ |

Wir machen eine Liste mit den einzelnen Termen und ...

**(%i17)** `powerList:makelist(ratsimp((a[i]/i!)*X^i),i,0,16)$`

... reduzieren obige Liste mit "+" - bilden also die Summe!

**(%i18)** `powerSeries:lreduce("+",powerList)$`

Jetzt legen wir die Werte fest: Abschusswinkel  $30^\circ$ ,  $V_{X0} = 1$  und die Simulationsweite (durch Probieren)

**(%i21)** `%alpha:%pi/6$V_X0:1$simWidth:0.75$`

Wir legen obige Potenzreihe als Funktion  $Y(X)$  für den Plotbefehl fest

**(%i22)** `define(Y(X),float(powerSeries))$`

Nur wegen der Neugier schauen wir uns die Potenzreihe auch mal an!

**(%i23)** `float(Y(X));`

$$\begin{aligned}
 & -0.0025X^{16} + 0.0029X^{15} + 0.0063X^{14} + 0.0072X^{13} + 0.003X^{12} - 0.0087X^{11} - 0.026X^{10} - 0.041X^9 \\
 & - 0.038X^8 - 0.014X^7 + 0.0049X^6 - 0.036X^5 - 0.18X^4 - 0.38X^3 - 0.5X^2 + 0.58X \quad (19.56)
 \end{aligned}$$

Nun das RUNGE-KUTTA Verfahren für die ursprüngliche Diffglg. dritter Ordnung - gleich wie bei short-time Näherung

**(%i24)** `sol30:rk([z_2,z_3, 2*z_3*sqrt(1+z_2^2)], [z_1,z_2,z_3], [0,tan(%alpha),-1/V_X0^2], [x,0,simWidth,0.01])$`

Für den Plotbefehl benötigen wir nur  $(x, z_1)$  - das sind jeweils die ersten 2 Komponenten der Lösungsliste

**(%i25)** `plotPoints30:map(lambda([x],firstn(x,2)),sol30)$`

Jetzt sehen wir uns die 3 Funktionen an: Runge-Kutta, Potenzreihe und short-time-Näherung

## 19. Der schiefe Wurf

```
(%i26) plot2d([[discrete,plotPoints30],Y(X),Y\st],[X,0,simWidth],
[legend,"rk-30Grad","PowerSeries(16)","short-time"],
[gnuplot_preamble,"set key bottom left;set xtics font \",20\";
set ytics font \",20\";set key font \",20\""]);
```

Wie man bei 19.56 (das Näherungspolynom) erkennen kann, zwingt die Fakultät im Nenner keineswegs die Koeffizienten gegen Null, d.h. eine schnelle Konvergenz kann man nur bei



$$X < 1 \Rightarrow x < \ell = \frac{1}{k} = \frac{2m}{\rho_M c_W A}$$

erwarten. Also das Verhältnis der reibungsbestimmenden Größen (Dichte des Mediums, Querschnittfläche, Widerstandsbeiwert) zur Masse des Körpers sollte groß sein.

Die Flugbahn eines Papierblatts bleibt weiterhin unberechenbar!

Zur Demonstration hier noch die Flugbahnberechnungen für  $X > 1$ . Man sieht dass unsere Potenzreihennäherung den Einfluss der Luftreibung unterschätzt (erst recht natürlich die short-time-Näherung). Die Bewegung geht im Grenzfall zu einem reinen vertikalen Fall mit konstanter Geschwindigkeit über!

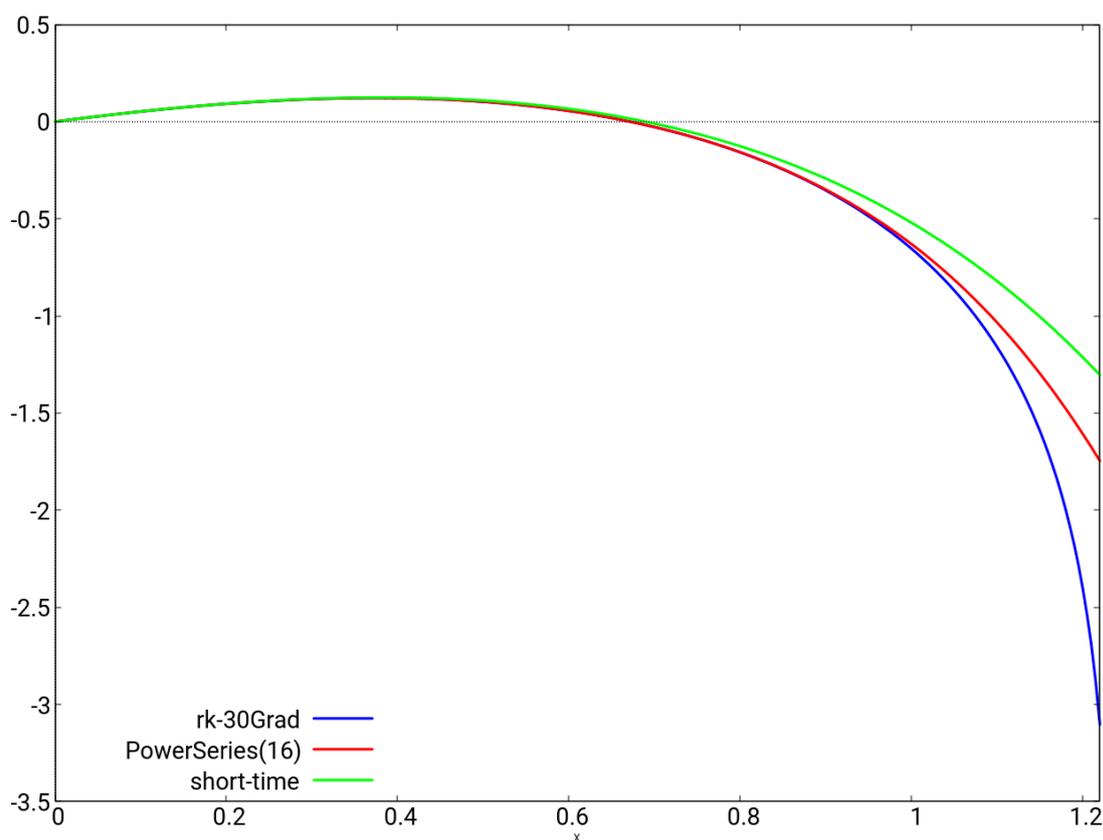


Abb.200 : Übergang in reine Fallbewegung

Damit ist unser Ausflug zum “Werfen” und “Schießen” in Luft zu Ende. Folgende Quellen

wurden verwendet:

*G.W. Parker: "Projectile motion with air resistance quadratic in the speed", North Carolina, 1977*

*Riccardo Borghi: "Trajectory of a body in a resistant medium: an elementary derivation",  
2013 Eur. J. Phys. 34 359*

### 19.3.11 ANHANG: dimensionslose Variablen

Um das Prinzip zu erklären, nehmen wir zu Beginn eine einfache Gleichung:

$$s = s_0 - v t_0 \quad (19.57)$$

$s_0$  und  $t_0$  seien Konstanten aus  $\mathbb{R}$  mit den Einheiten "Weg" bzw. "Zeit". Die Variablen  $s$  und  $v$  besitzen die Einheiten "Weg" bzw. Geschwindigkeit, sodass  $v t_0$  ebenfalls die Einheit eines "Weges" hat - sonst könnte man die Subtraktion ja nicht ausführen ("Zwetschkenknödel minus Äpfel"?). Um auf dimensionslose Einheiten zu kommen, dividieren wir 19.57 durch  $s_0$ :

$$\frac{s}{s_0} = 1 - v \frac{t_0}{s_0} \quad (19.58)$$

Wir können jetzt eine dimensionslose "Wegeinheit"  $S := \frac{s}{s_0}$  und "Geschwindigkeitseinheit"  $V := v \frac{t_0}{s_0}$  festlegen, sodass 19.58 sich vereinfacht zu

$$S = 1 - V \quad (19.59)$$

Durch diese Einheitenumstellung bleibt auch die Zeit nicht unberührt, denn es gilt:

$$\frac{S}{V} = T = \frac{s s_0}{v s_0 t_0} = \frac{t}{t_0} \quad (19.60)$$

Was ist dadurch gewonnen? Wie fast immer gibt es pros und cons:

- Da die Konstanten verschwunden sind, lässt sich 19.59 sicher leichter und universeller (für alle Konstanten) lösen als 19.57
- Nachdem 19.59 gelöst ist, muss man die Lösungen ins SI-System zurückverwandeln - das ist mit den Transformationsgleichungen nicht schwer aber doch Arbeit.
- Es kommt also darauf an: Ist die Ausgangsgleichung schwer zu knacken (denken Sie an eine heikle Differentialgleichung) wird die Vereinfachung zu dimensionslosen Einheiten vermutlich schwerer wiegen, als die zusätzliche Arbeit anschließend für die Umwandlung.



Je schwieriger die Ausgangsgleichung, umso mehr spricht für dimensionslose Größen!

Nehmen Sie die Pendelgleichung:



$$m\ddot{x} = -m g \sin \theta \quad x(\theta) = L \theta \Rightarrow \ddot{x} = \frac{d}{dt} \frac{dx}{d\theta} \frac{d\theta}{dt} = L \frac{d^2\theta}{dt^2}$$

$$\frac{d^2\theta}{dt^2} - \frac{g}{L} \sin \theta = 0 \Leftrightarrow \frac{d^2\theta}{dT^2} - \sin \theta = 0$$

### 19.3.12 ANHANG: asymptotische Taylorreihe

Gibt man für den Entwicklungspunkt einer Funktion “ $\infty$ ” in *wxMaxima* ein, gibt es keine Fehlermeldung, sondern einen Term, der auch negative Exponenten enthält. Für  $x \gg 1$  liefert das eine ziemlich gute Näherung - man spricht von **asymptotischer Näherung** durch eine Taylorreihe. Aber was macht *wxMaxima* da genau?

```
(%i3) f(x):=sqrt(1+x^2)$
```

```
(%i9) exptdispflag:false$
```

```
(%i10) define(t(x),taylor(f(x),x,inf,4)); t(x) := x + \frac{x^{-1}}{2} - \frac{x^{-3}}{8} + ... (%o10)/T
```

```
(%i17) float(f(100)); 100.0049998750062 (%o17)
```

```
(%i18) float(t(100)); 100.004999875 (%o18)
```

Es passiert folgendes:

- In  $f(x)$  wird  $x$  durch  $\xi^{-1}$  ersetzt (Beachte:  $x \rightarrow \infty \Leftrightarrow \xi \rightarrow 0$ )

$$f\left(\frac{1}{\xi}\right) = \sqrt{1 + \xi^{-2}}$$

- Diese Funktion wird jetzt mit Entwicklungspunkt 0 “taylorisiert”:

```
define(g(%xi),taylor(f(1/%xi),%xi,0,4))
```

$$\xi^{-1} + \frac{\xi}{2} - \frac{\xi^3}{8} + \dots$$

- In  $g$  wird jetzt wieder  $\xi$  durch  $x^{-1}$  ersetzt:

```
expand(g(1/x))
```

$$x + \frac{x^{-1}}{2} - \frac{x^{-3}}{8}$$

Also halten wir fest:

$$f(x) \approx g\left(\frac{1}{x}\right) \quad \text{für } x \gg 1$$

mit  $g(x) := \text{taylor}(f(1/x), x, 0, n)$

### 19.3.13 ANHANG: Invertierung einer Funktion mit Runge-Kutta

Angenommen wir hätten eine Funktion  $y(w)$  mit

$$y(w) := w_0 + \int_{w_0}^w f(x) dx \quad \text{mit } w_0 \in \mathbb{R} \quad \Leftrightarrow y = F(w) \quad (19.61)$$

- wir suchen aber die Umkehrfunktion  $w = F^{-1}(y)$ . Bei etwas komplexeren  $f$  müssen wir froh sein, wenn wir das Integral “knacken” können - von der anschließenden Invertierung von  $F$  ganz zu schweigen. Wenn wir allerdings mit einer Tabelle zufrieden sind, die “jedem”  $w$  ein  $y$  zuordnet, dann können wir es numerisch schaffen!

Zuerst verwandeln wir obige Funktion  $y$  in ein Anfangswertproblem:

$$\frac{dy}{dw} = f(w) \quad y(w_0) = w_0$$

Mit `rk(f(w),y,w_0,[w,w_0,2,0.1])` erstellen wir eine Liste und vertauschen die Komponenten. Das implementieren wir jetzt in *wxMaxima*:

(%i1) `m(x):=x; /* die 1. Mediane als Bezugsgerade */` m(x) := x (%o1)

Das “Versuchskaninchen” - einfach genug, um  $y^{-1}$  zu bestimmen

(%i2) `f(x):=x^2;` f(x) := x<sup>2</sup> (%o2)

(%i7) `w_0:1$ /* Setzen irgendeinen Wert */`

(%i3) `define(y(w),w_0+integrate(f(x),x,1,w));`  $y(w) := \frac{w^3}{3} + \frac{2}{3}$  (%o3)

(%i4) `eq:y(w)=x;`  $\frac{w^3}{3} + \frac{2}{3} = x$  (eq)

Nur Lösung [3] ist reell und unsere Umkehrfkt.

(%i5) `sol:solve(eq,w);`  $[w = \frac{(\sqrt{3}i - 1)(3x - 2)^{\frac{1}{3}}}{2}, w = -\frac{(\sqrt{3}i + 1)(3x - 2)^{\frac{1}{3}}}{2}, w = (3x - 2)^{\frac{1}{3}}]$

(%i6) `y_inv(x):=rhs(sol[3])$`

Jetzt die Tabelle  $y \leftrightarrow w$

(%i8) `plotList:rk(f(w),y,w_0,[w,w_0,2,0.1])$`

Vertauschen der Komponenten einer Liste aus Paaren

(%i9) `invert(l):=map(lambda([x],[second(x),first(x)]),l)$`

```
(%i10) plot2d([[discrete,plotList],[discrete,invert(plotList)],y(x),y_inv(x),m(x)],
 [x,1,2],[y,1,2.5], [same_xy, true], [legend, " R-
 K"," R-K-invers",
 " y - analytisch", " y^{}{-
 1} - analytisch"," Mediane"],
 [gnuplot_preamble, "set key top left; set xtics font \", 10\";
 set ytics font \", 10\"; set key font \", 15\" "],
 [style,[lines,6,1,2], [lines,6,2,2],[lines,3,3,2],[lines,3,6,2], [lines,1,5,2]])$
```

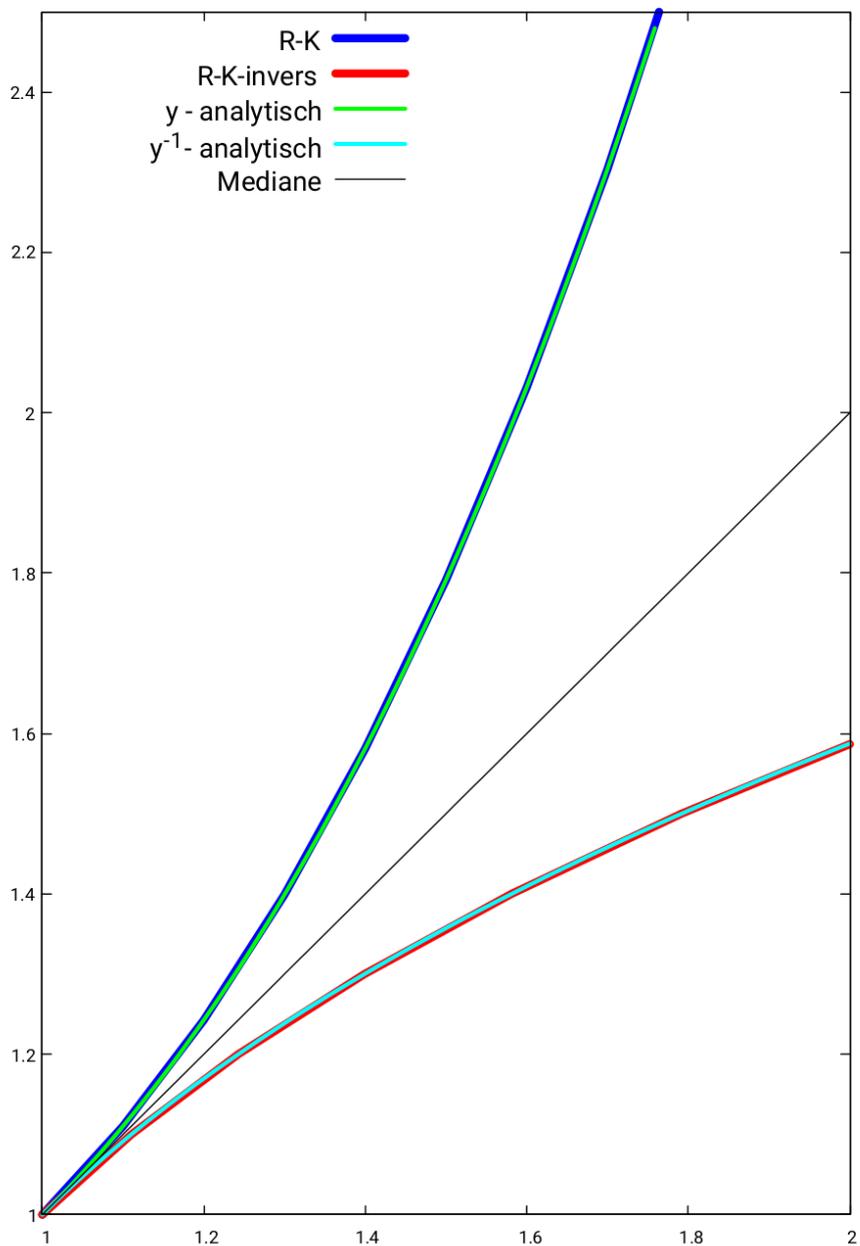


Abb.201 : Invertierung der Funktionstabelle vs. analytische Methode



Runge-Kutta Verfahren und anschl. Vertauschung liefert tabellarisch denselben Graph wie die analytische Methode (die nur bei "einfachen" Funktionen  $f$  gelingt). Außerdem können wir das Runge-Kutta Verfahren auch dazu verwenden (wie das bei den *wxMaxima*-Programmen geschehen ist), um tabellarisch die Integralgleichung 19.61 zu lösen!

### 19.3.14 ANHANG: Lineare Differentialgleichung erster Ordnung

$$\dot{u} = a(t)u \qquad \dot{u} = a(t)u + s(t) \qquad (19.62)$$

Differentialgleichungen obiger Struktur heißen *homogene bzw. inhomogene lineare Differentialgleichungen erster Ordnung*. Die zweite Gleichung von 19.35 ist von dieser Gestalt mit

$$a(t) := -V_x(t) \quad \text{und} \quad s(t) := -1$$

#### Die homogene Gleichung

Wie meist führt die Lösung der inhomogenen Gleichung über die Lösung der homogenen. Also knöpfen wir uns diese vor:

Wenn  $a(t)$  stetig ist besitzt sie auch eine Stammfunktion z.B.

$$A(t) := \int_{t_0}^t a(x) dx \quad \text{bzw. irgendeine Stammfkt.} \quad \int a(t) dt$$

Dann sind alle Lösungen der homogenen linearen Gleichung von der Gestalt

$y(t) := C \exp\left(\int a(x) dx\right)$

mit  $C \in \mathbb{R}$  Anpassungsfaktor für Anfangswertproblem

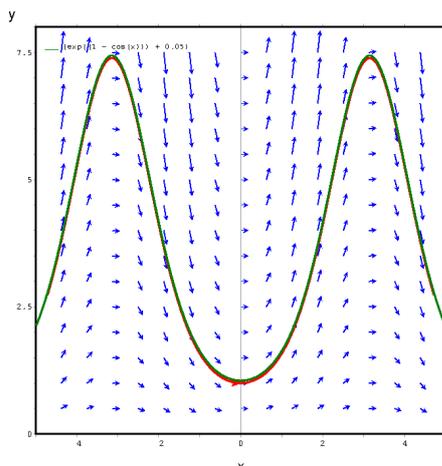
Sei  $z(t)$  irgendeine andere Lösung der homogenen Gleichung 19.62, dann gilt

$$\frac{d}{dt} \frac{y}{z} = \frac{\dot{y}z - y\dot{z}}{z^2} = \frac{ayz - yaz}{z^2} = 0 \Rightarrow y = Cz \quad \text{mit } C \in \mathbb{R}$$

#### Beispiel 19.1



$$\dot{u} = \sin(t) \cdot u \qquad u(0) = 1 \Rightarrow u(t) = e^{1-\cos(t)}$$



```
plotdf(sin(x)*y, [trajectory_at,0,1],
[xfun,"exp(1-cos(x))+0.05"],
[y,0,8], [x,-5,5]);
```

Die Funktion  $\exp(1-\cos(x))+0.05$  wurde um 0.05 Einheiten nach oben verschoben - damit sie die andere Bahn nicht überdeckt!

Abb.202 : Direction-Field mit Trajektorie

### Die inhomogene Gleichung

Die allgemeine Lösung des Anfangswertproblems

$$\dot{u}(t) = a(t) \cdot u(t) \quad u(0) = u_0 \quad (19.63)$$

lautet also

$$u_g(t) = u_0 \exp\left(\int_{t_0}^t a(x) dx\right)$$

Das inhomogene Anfangswertproblem lautet

$$\dot{u}(t) = a(t) \cdot u(t) + s(t) \quad u(0) = u_0 \quad (19.64)$$

### Theorem 19.2

$$\left\{ \begin{array}{l} \text{Sei } u_p(t) \text{ eine beliebige partikuläre Lösung von 19.64} \\ u(t) \text{ sei eine beliebige Lösung von 19.64} \end{array} \right\} \Rightarrow (u - u_p) \text{ ist Lösung von 19.63}$$

*Beweis:*

$$\begin{aligned} \frac{d}{dt}(u - u_p) &= \dot{u} - \dot{u}_p = a(t) \cdot u + s(t) - (a(t) \cdot u_p + s(t)) = a(t) \cdot (u - u_p) \\ \Rightarrow \quad &\boxed{u = u_g + u_p} \end{aligned}$$

□

Wir brauchen also die allgemeine Lösung des homogenen Anfangswertproblems (AWP) und eine partikuläre Lösung des inhomogenen Anfangswertproblems! Man bekommt diese mit der Methode *Variation der Konstanten*:

$$u_p(t) = C(t) \exp\left(\int a(x) dx\right)$$

Die Konstante  $C$  wird so lange variiert, bis sie die inhomogene Differentialgleichung erfüllt:

$$\frac{d}{dt} \left[ C(t) \exp\left(\int a(x) dx\right) \right] = a(t) \left[ C(t) \exp\left(\int a(x) dx\right) \right] + s(t) \Rightarrow \dot{C} = s(t) \exp\left(-\int a(x) dx\right)$$

Mit  $\boxed{C(t) = \int s(t) \exp\left(-\int a(x) dx\right) dt}$  haben wir  $C$  und damit  $u_p$  gefunden

Dies Ergebnis benutzen wir jetzt um mit *wxMaxima* die *short-time*-Näherung zu finden:

Zuerst lösen wir die Gleichung für  $V_x$

$$\text{\%i1) diffEq1:'diff(V_x,T)=-V_x^2;}$$

$$\frac{d}{dT}V_x = -V_x^2 \quad (\text{diffEq1})$$

$$\text{\%i2) gSol1:ode2(diffEq1,V_x,T);}$$

$$\frac{1}{V_x} = T + \%c \quad (\text{gSol1})$$

$$\text{\%i3) spSol1:ic1(gSol1,T=0,V_x=V_x0);}$$

$$\frac{1}{V_x} = \frac{T V_{x0} + 1}{V_{x0}} \quad (\text{spSol1})$$

Jetzt haben wir die Lösung für  $V_x(T)$

$$\text{\%i4) define(V_x(T),rhs(linsolve(spSol1,V_x)[1]));}$$

$$V_x(T) := \frac{V_{x0}}{T V_{x0} + 1} \quad (\text{\%o4})$$

Bei unserer zweiten inhomogenen linearen Dfglg ist  $a := -V_x$ ,  $s = -1$

$$\text{\%i5) a(t):=-V_x(t);}$$

$$a(t) := -V_x(t) \quad (\text{\%o5})$$

$$\text{\%i6) assume(T>0,V_x0>0);}$$

$$[T>0, V_{x0}>0] \quad (\text{\%o6})$$

$$\text{\%i7) define(d(t),exp(integrate(a(t),t)));}$$

$$d(t) := \frac{1}{V_{x0}t + 1} \quad (\text{\%o7})$$

Wir haben jetzt die allgemeine(“general”) Lösung der homogenen Gleichung:

$$\text{\%i8) define(V_yg(T),V_y0*d(T));}$$

$$V_{yg}(T) := \frac{V_{y0}}{T V_{x0} + 1} \quad (\text{\%o8})$$

$$\text{\%i9) define(b(t),exp(integrate(-a(t),t)));}$$

$$b(t) := V_{x0}t + 1 \quad (\text{\%o9})$$

Jetzt bestimmen wir unsere Konstante  $C$

$$\text{\%i10) define(C(t),integrate((-1)*b(t),t));}$$

$$C(t) := -\frac{V_{x0}t^2}{2} - t \quad (\text{\%o10})$$

Damit haben wir jetzt die partikuläre Lösung für  $V_y$

$$\text{\%i11) define(V_yp(t),ratsimp(C(t)*d(t)));}$$

$$V_{yp}(t) := -\frac{V_{x0}t^2 + 2t}{2V_{x0}t + 2} \quad (\text{\%o11})$$

Die Summe der allgemeinen Lösung der homogenen und partikulären der inhomogenen Gleichung

$$\text{\%i13) define(V_y(T),ratsimp(V_yg(T)+V_yp(T)));}$$

$$V_y(T) := \frac{2V_{y0} - T^2 V_{x0} - 2T}{2T V_{x0} + 2} \quad (\text{\%o13})$$



Obiges Ergebnis ist eine Bestätigung von 19.38!



# 20 | Gradientenabstieg

## 20.1 Partielles Differenzieren

Wir haben eine Funktionen von mehreren Variablen (wir beschränken uns hier auf 2 - wie zum Beispiel die Summe zweier Zahlen) und die Funktion liefert eine reelle Zahl, dann gilt folgende Festlegung

### Definition 20.1 partielle Ableitung

Ist  $f : \mathbb{R}^2 \rightarrow \mathbb{R}$   
 $(x, y) \mapsto f(x, y)$  eine reellwertige Funktion mit 2 Argumenten  
- dann wird festgelegt

$$\frac{\partial f}{\partial x}(x_0, y_0) = \lim_{h \rightarrow 0} \frac{f(x_0 + h, y_0) - f(x_0, y_0)}{h} \quad (20.1)$$

20.1 heißt partielle Ableitung von  $f$  nach  $x$  and der Stelle  $(x_0, y_0)$ .

Es wird also nach einer(1) bestimmten Variablen abgeleitet und die andere(n) wird (werden) als konstant angenommen:

### Beispiel 20.2



$$f(x, y) := x y^2 \Rightarrow \frac{\partial f}{\partial x}(a, b) = b^2 \quad \frac{\partial f}{\partial y}(a, b) = 2 a b$$

### Definition 20.3 Gradient

Der Spaltenvektor  $\left( \frac{\partial f}{\partial x}(x_0, y_0), \frac{\partial f}{\partial y}(x_0, y_0) \right)^T =: \nabla f(x_0, y_0)$  heißt **Gradient** von  $f$  an der Stelle  $(x_0, y_0)$

### 20.2 Eigenschaften des Gradienten

Dazu visualisieren wir uns eine Funktion  $f(x, y)$  in Geogebra3d:

Wir nehmen eine radialsymmetrische Funktion mit Minimum im Ursprung z.B.

$$f(r) = -A \cos(r) + d \Rightarrow f(x, y) = -2 \cos(\sqrt{x^2 + y^2}) + d \quad d \text{ als Schieberegler}$$

also los geht's mit *Geogebra*:

- Zuerst obige Funktion festlegen:  
 $f(x, y) = -2 * \cos(\text{sqrt}(x^2 + y^2)) + d$      $d$  als Slider sichtbar machen
- Jetzt die partiellen Ableitungen :  
 $f_x = \text{Derivative}(f, x)$  und  $f_y = \text{Derivative}(f, y)$
- Als "Zucker" legen wir die Kurve fest, wo die  $x$ - $y$ -Ebene geschnitten wird:  
 $g: f(x, y) = 0$     dient eigentlich nur zur "Orientierung"
- Nun ein Punkt  $A$  auf der Fläche, die durch  $f$  festgelegt ist:  
 $A: \text{PointIn}(f)$     - er ist beliebig verschiebbar!
- Jetzt der eigentliche Gradientenvektor  $\vec{u}$ :  
 $u = \text{Vector}((f_x(x(A), y(A)), f_y(x(A), y(A))))$  !
- Wir tragen den negativen (wegen der besseren Sichtbarkeit) Gradienten bei  $A$  ein:  
 $\text{grad} = \text{Vector}(A, A - u)$     - der "eigentliche" Gradient zeigt in die andere Richtung

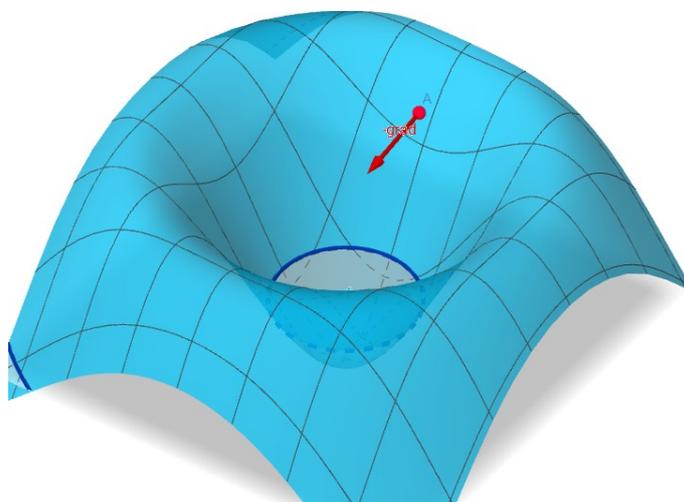


Abb.203 : Funktionsfläche mit eingezeichnetem negativen Gradientenvektor

Man kann sich leicht durch Verschieben des Punktes  $A$  überzeugen:

Der negative Gradientenvektor zeigt immer in Richtung des lokalen Minimums von  $f$

Ist das nur Zufall bei unserem Beispiel? Wir zeigen: NEIN - das hat System!

**Definition 20.4 Richtungsableitung**

Sei  $\vec{u}$  ein Einheitsvektor im  $\mathbb{R}^2$  mit den Koordinaten  $(a, b)$ ,  $f$  unsere obige Funktion vom  $\mathbb{R}^2$  in  $\mathbb{R}$ , dann heißt

$$D_{\vec{u}}f(x_0, y_0) := \lim_{h \rightarrow 0} \frac{f(x_0 + ah, y_0 + bh) - f(x_0, y_0)}{h}$$

Richtungsableitung von  $f$  in Richtung  $\vec{u}$  an der Stelle  $(x_0, y_0)$

**Theorem 20.5 Richtungsableitung mit Hilfe des Gradienten**

$$D_{\vec{u}}f(x_0, y_0) = \nabla f(x_0, y_0) \cdot \vec{u}$$

Also anstatt obigen Grenzwert zu berechnen, kann man einfach den Gradienten skalar mit  $\vec{u}$  multiplizieren!

*Beweis:*

$$\begin{aligned} D_{\vec{u}}f(x_0, y_0) &= \lim_{h \rightarrow 0} \frac{f(x_0 + ah, y_0 + bh) - f(x_0, y_0)}{h} = \\ &= \lim_{h \rightarrow 0} \left[ \frac{f(x_0 + ah, y_0 + bh) - f(x_0, y_0 + bh)}{h} + \frac{f(x_0, y_0 + bh) - f(x_0, y_0)}{h} \right] = \\ &= \lim_{h \rightarrow 0} \left[ \frac{f(x_0 + ah, y_0 + bh) - f(x_0, y_0 + bh)}{ah} a + \frac{f(x_0, y_0 + bh) - f(x_0, y_0)}{bh} b \right] = \\ &= \frac{\partial f}{\partial x}(x_0, y_0) a + \frac{\partial f}{\partial y}(x_0, y_0) b = \nabla f(x_0, y_0) \cdot \vec{u} \end{aligned}$$

□

**Theorem 20.6**

$\nabla f(x_0, y_0)$  ist die Richtung des stärksten Anstiegs von  $f$  bei  $(x_0, y_0)$

*Beweis:* Wir wissen  $\vec{a} \cdot \vec{b} = \|\vec{a}\| \|\vec{b}\| \cos \theta$  also  $\nabla f(x_0, y_0) \cdot \vec{u} = \|\nabla f(x_0, y_0)\| \cdot 1 \cdot \cos \theta$   
 $\nabla f(x_0, y_0) \cdot \vec{u}$  ist also am größten, wenn  $\theta = 0$  - es ist am kleinsten wenn  $\theta = \pi$

□

$-\nabla f(x_0, y_0)$  zeigt also in Richtung des größten Abstiegs!

## 20.3 Die Kettenregel

## Beispiel 20.7



$$f(x, y) = \sqrt{x} y^2 \text{ wobei } x = x(t) := \cos t \text{ und } y = y(t) := \sin t$$

$$\frac{d}{dt} f(x(t), y(t)) = ?$$

$$\begin{aligned} \frac{d}{dt} f(x(t), y(t)) &= \frac{d}{dt} \sqrt{\cos t} \sin^2 t = 1/2(\cos t)^{-0.5} (-\sin t) \sin^2 t + \sqrt{\cos t} 2 \sin t \cos t = \\ &= \frac{\partial f}{\partial x} \cdot \frac{d}{dt} x(t) + \frac{\partial f}{\partial y} \cdot \frac{d}{dt} y(t) \end{aligned}$$

Mit der Bezeichnung  $\vec{x} = (x(t), y(t))$  ergibt sich also die Kettenregel

$$\frac{d}{dt} f(\vec{x}(t)) = \nabla f(\vec{x}(t)) \cdot \frac{d}{dt} \vec{x}(t) \quad (20.2)$$

## 20.4 Höhenlinien und Gradient

**Definition 20.8 Höhenlinien**

$f$  sei eine Funktion vom  $\mathbb{R}^2$  in  $\mathbb{R}$ , dann heißt die Kurve  $(x, y)$  im  $\mathbb{R}^2$  für die gilt

$$f(x, y) = c \in \mathbb{R} \quad \text{Höhenlinie von } f \text{ für } c$$

**Theorem 20.9 Gradient ist eine Normale der Höhenlinie**

*Beweis:* Sei  $\vec{x}(t)$  eine parametrisierte Kurve im  $\mathbb{R}^2$  für die gilt  $f(\vec{x}(t)) = c$

$$f(\vec{x}(t)) = c \quad \left| \frac{d}{dt} \xrightarrow{20.2} \nabla f(\vec{x}(t)) \cdot \frac{d}{dt} \vec{x}(t) = 0 \right.$$

Der Gradient steht also senkrecht auf der Tangente der Höhenlinie. Ein Ball rollt also senkrecht zu den Höhenlinien "ins Tal". Je dichter die Höhenlinien liegen, umso steiler ist es dort - auf weniger Distanz wird mehr Höhe gewonnen (verloren)!

□

Im Licht obiger Erkenntnisse schauen wir uns ein Geogebra-Arbeitsblatt auf Geogebra.org von Tatsuyoshi Hamada an:

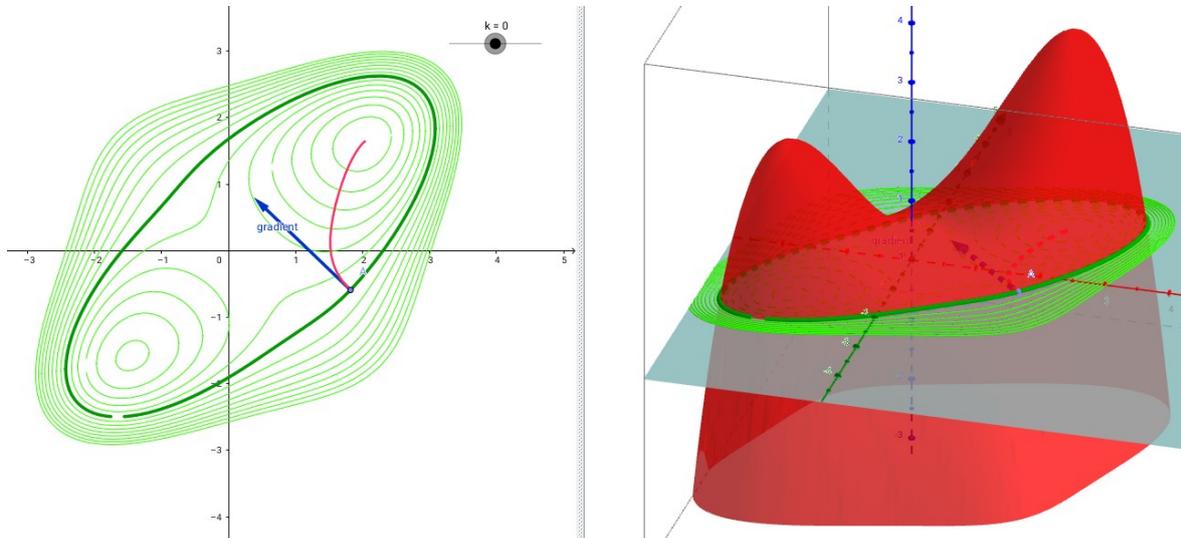


Abb.204 : Höhenlinien mit Gradientenvektor samt "Rollkurve" eines Balls

Arbeitsblattanalyse:

- Zuerst Eingabe der Funktion (woher sie Herr Hamada hat, wird uns wohl für immer verschlossen bleiben):  
 $f(x,y) = (-x^2(x+1)(x-2)/4 - (y^4 - 2y^2 + y + 2)/3 + 2xy)/2 + 1$
- Bestimmung des Gradienten:  $f_x: \text{Derivative}(f,x)$  und  $f_y: \text{Derivative}(f,y)$
- Schieberegler  $k$  für Ebene  $z = k$ :  $k=1$  und  $e: z=k$
- Kurve  $g$  (dunkelgrün), wo der Graph von  $f$  die Ebene  $e$  schneidet:  $g: f(x,y) - k = 0$
- Beliebiger verschiebbarer Punkt  $A$  auf  $g$  - an dem wir den Gradienten einzeichnen:  
 $A = \text{PointIn}(g)$
- Gradient in  $A$ :  $\text{grad}: \text{Vector}(A, A + (f_x(x(A), y(A)), f_y(x(A), y(A))) ) )$
- Eine Schar von Höhenlinien (wie man sie von Landkarten kennt):  
 $\text{list}: \text{Sequence}(f(x,y) - k = 0, k, -4, 4, 0.5)$
- Und schließlich die rote Kurve - es ist die "Rollkurve eines Balls" vom "Gipfel" durch den Punkt  $A$ . Gesucht ist als jene parametrisierte Kurve  $\vec{x}(t)$  für die gilt:

$$\text{Tangente} = \text{Gradient: } \frac{d}{dt} \vec{x}(t) = \nabla f(\vec{x}(t)) \quad \text{mit Randbedingung } \vec{x}(t_0) = A \quad (20.3)$$

## 20. Gradientenabstieg

Jetzt kann *Geogebra* folgende Differentialgleichung lösen:

$$\frac{dy}{dx} = \frac{h(x,y)}{g(x,y)} \quad (20.4)$$

Und zwar mit dem Befehl `SolveODE(h(x,y), g(x,y), x(t_0), y(t_0), t, Delta_t)` dabei ist  $t$  der Maximalwert des internen Parameters  $t$  für die die Kurve  $\vec{x}(t)$ ,  $\Delta t$  die Schrittweite

Wir können 20.3 umformen auf obige Form, es gilt ja

$$\frac{dx}{dt} = \frac{\partial f}{\partial x}(x(t), y(t)) =: f_x(x, y) \quad \text{bzw.} \quad \frac{dy}{dt} = \frac{\partial f}{\partial y}(x(t), y(t)) =: f_y(x, y)$$

also

$$\frac{dy}{dx} = \frac{dy}{dt} \frac{dt}{dx} = f_y(x, y) [f_x(x, y)]^{-1} = \frac{f_y(x, y)}{f_x(x, y)}$$

Der Befehl lautet also bei uns

`SolveODE(f_y, f_x, x(A), y(A), 4, 0.1)`

dabei wurde der Wert "4" bzw. "0.1" experimentell bestimmt ("glatte Kurve")

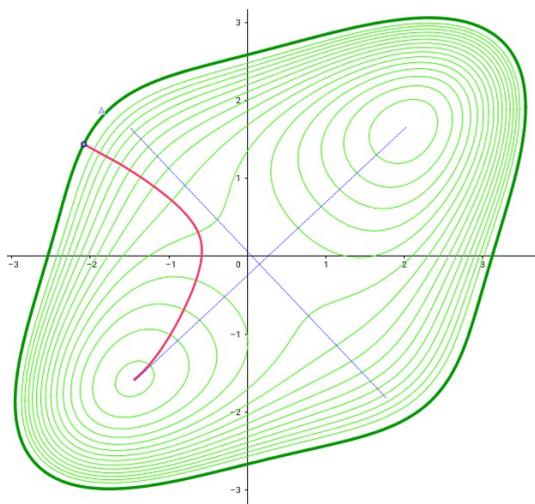


Abb.205 : Rollkurve 1

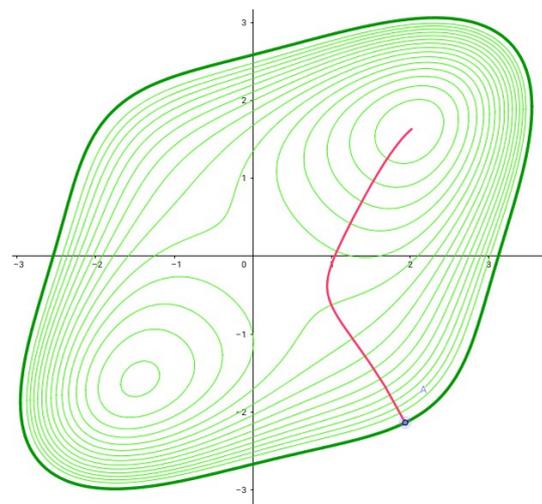


Abb.206 : Rollkurve 2

Klar kann man erkennen, dass die "Rollkurven" (dessen Tangenten in Richtung des Gradienten weisen) immer senkrecht zu den Höhenlinien stehen. Wobei der Sattel zwischen den beiden "Gipfeln" "entscheidet", ob die Aufstiegskurve auf den oberen oder unteren Gipfel führt.

Auf den Maxima (Gipfeln) und dem Sattel verschwindet der Gradient, deren Lage finden wir mit `wxMaxima` schnell heraus.

Komponenten des Gradienten verschwinden!

```
(% i1) (f_x:-4*x^3+3*x^2+4*x+8*y=0, f_y:-4*y^3+6*x+4*y-1=0)$
```

Wir holen uns aus der zweiten Gleichung einen Ausdruck für  $x$  und substituieren in 1

```
(% i2) sx:rhs(solve(f_y,x)[1]);
```

$$\frac{4y^3 - 4y + 1}{6} \quad (\text{sx})$$

Es entsteht ein Polynom neunten Grades

```
(% i3) define(f(y),lhs(ratsimp(subst(sx,x,f_x)*108)));
```

$$f(y) := -128y^9 + 384y^7 + 48y^6 - 384y^5 - 96y^4 + 464y^3 + 48y^2 + 528y + 79$$

Punktliste initialisieren, Nullstellen des obigen Polynoms mit 3 Ausgabeziffern

```
(% i4) (p_s:[], fpprintprec:3,y_s:float(realroots(f(y))));
```

$$[y = -1.59, y = -0.149, y = 1.64]$$

Obige Lösungen in  $sx$  einsetzen und Punktliste erstellen (2 Maxima, 1 Sattel dazwischen)

```
(% i5) (for i thru length(y_s) do p_s: cons([subst(rhs(y_s[i]),y,sx), rhs(y_s[i]), p_s), p_s);
```

$$[[2.04, 1.64], [0.264, -0.149], [-1.45, -1.59]]$$

```
(% i6) plot2d(f(x),[x,-2,2]);
```

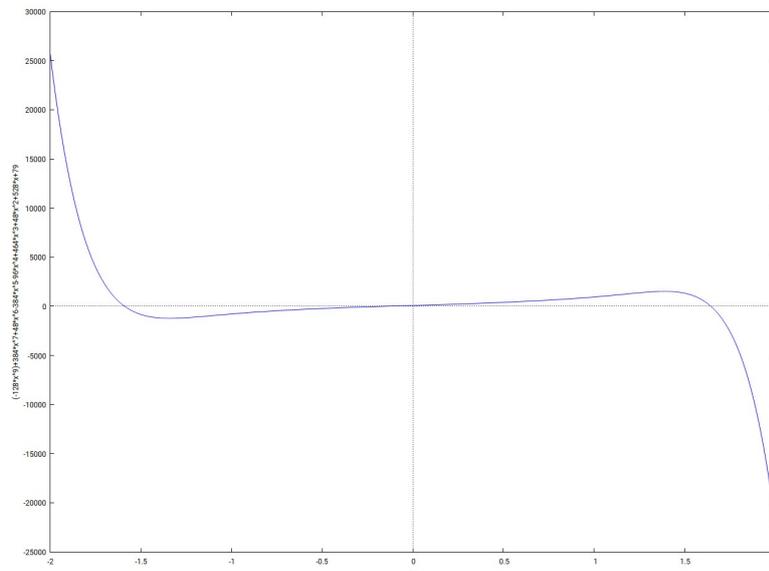


Abb.207 : Nullstellen des Gradienten: 2 Maxima und 1 Sattelpunkt

Wenn wir dem negativen Gradienten folgen (im Definitionsraum der Funktion  $f$ ) sinkt der Funktionswert von  $f$  bis wir bei einem (lokalen) Minimum landen - haben wir Pech ist es ein Sattelpunkt (aber da geht es "daneben" weiter nach unten)

## 20.5 Gradientenabstieg - gradient descent

Der Algorithmus zum Auffinden des Minimums einer Funktion  $f(\vec{x})$  ist jetzt klar:

Von einem zufällig gewählten Anfangspunkt  $\vec{x}_0$  folgen wir in “kleinen” Schritten dem negativen Gradienten:

$$\vec{x}_{k+1} = \vec{x}_k - \gamma_k \cdot \nabla f(\vec{x}_k) \quad (20.5)$$

$\gamma_k$  sollte groß genug sein, um zügig vorwärts zu kommen - aber nicht über das Ziel hinausschießen.



Der Gradient sollte keine Sprungstellen haben

$f$  sollte nur 1 Minimum haben - sonst besteht die Gefahr, dass man ein lokales Minimum “ansteuert”

Wir sehen uns das Verfahren an einem Beispiel an:

### Beispiel 20.10



Wir versuchen mit der einem Polynom dritten Grades die Sinusfunktion im Intervall  $[0, \pi/2]$  anzunähern, also

$$f(x) = d + cx + bx^2 + ax^3 \approx \sin(x) \quad \text{für } x \in [0, \pi/2]$$

$$f(0) = \sin(0) \quad \text{und} \quad f'(0) = \sin'(0) \quad \text{führt uns zu} \quad d = 0 \quad \text{und} \quad c = 1$$

Wir teilen nun  $[0, \pi/2]$  in  $N$  Teilintervalle mit den Stützstellen  $x_0, x_1, \dots, x_N$  und berechnen dort die Summe des quadratischen Fehler von  $f$  zum exakten Wert:

$$err(a, b) := \sum_{i=0}^N [x_i + bx_i^2 + ax_i^3 - \sin(x_i)]^2 \quad (20.6)$$

$err(a, b)$  sollte ein Minimum werden. Mit dem Gradientenabstiegsverfahren können wir dieses suchen!

Wir verschaffen uns einen Überblick über unsere Error-Funktion mit *GNU-Octave*- ob eine Chance für ein Minimum besteht. Wir benutzen zur Veranschaulichung die die Funktion *surf* (surface) - als Input benötigt sie Stützpunkte auf der  $x$ -Achse ( $tx$ ), Stützpunkte auf der  $y$ -Achse und die  $z$ -Werte an den Gitterpunkten als Matrix. Also dem Gitterpunkt bestehend aus dem  $i$ -ten Stützpunkt der  $x$ -Achse und dem  $j$ -ten Stützpunkt der  $y$ -Achse wird der Wert  $z(i, j)$  zugeordnet.

Als “Zucker” verwenden wir *surf* - da bekommt man die Contourlinien “gratis”.

Hier berechnen wir die  $z$ -Werte zuerst als Vektor und formen erst zum Schluss mit *reshape* zur Matrix um.

Beachten Sie auch in der Funktion  $f$  die “Punkte” vor den Operationszeichen! (die Operationen werden komponentenweise durchgeführt)

```

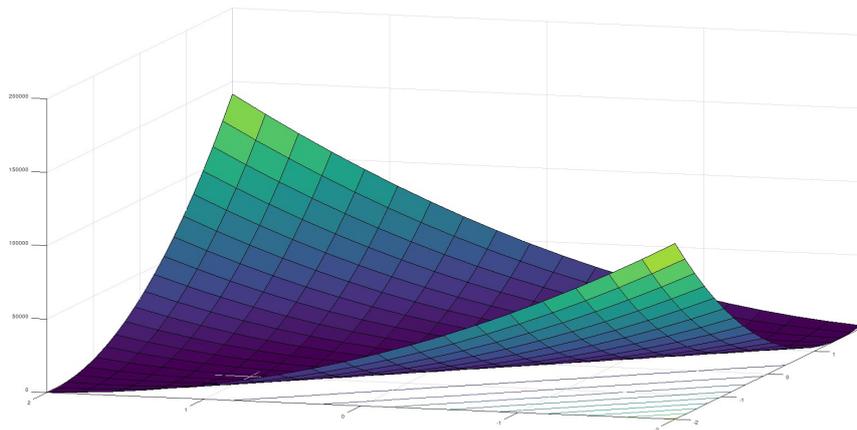
x_i = linspace(0, pi/2, 100);
2 tx = ty = linspace(-2, 2, 20);

4 function val_f = f(b,a,x_i)
 val_f = sum((x_i + b*x_i.^2 + a*x_i.^3) - sin(x_i)).^2;
6 endfunction

8 function z = err(tx, ty, x_i)
 z = [];
10 for a=tx
 for b=ty
12 z=[z, f(b,a,x_i)];
14 endfor
 endfor
16 endfunction

z_values = reshape(err(tx, ty, x_i),length(tx),length(ty));
18 surfc(tx,ty,z_values);

```

Abb.208 : err(a,b) im Überblick mit *GNU-Octave*

Der Graph lässt befürchten (Contourlinien), dass es womöglich kein eindeutiges Minimum gibt. Wir schauen uns den Gradientenabstieg mit *wxMaxima* an:

Berechnung der Stützpunkte und Ausgabeparameter - anschl. Definition der Error-Funktion

```
(% i1) (d:float((%pi/2))/N, x[j]:=j*d, fpprintprec:5, ratprint:false, globalsolve:true)$
```

```
(% i2) depends(err,[a,b])$
```

```
(% i3) err:sum((x[j] + b*x[j]^2 + a*x[j]^3 - sin(x[j]))^2, j, 1, N);
```

$$\sum_{j=1}^N \left( -\sin\left(\frac{1.5708j}{N}\right) + \frac{3.8758aj^3}{N^3} + \frac{2.4674bj^2}{N^2} + \frac{1.5708j}{N} \right)^2 \quad (\text{err})$$

```
(% i4) N:100$
```

## 20. Gradientenabstieg

---

Setzen  $N=100$ , multiplizieren aus und berechnen die Summe (“nouns” evaluiert)

```
(% i5) err_simp:ev(err, expand,nouns); 124.83b2 + 328.41ab + 49.857b + 222.18a2 + 67.222a + 5.0871
```

Jetzt wird der Gradient berechnet - als Vektor bzw. Liste

```
(% i6) grad:[diff(err_simp,a), diff(err_simp,b)]; [328.4b + 444.3a + 67.22, 249.6b + 328.4a + 49.85]
```

Wir schreiben *err\_simp* für Vektoreingabe um und legen die Error-Funktion *ferr(v)* fest

```
(% i7) err_v: subst(v[2],b,subst(v[1],a,err_simp))$
```

```
(% i8) define(ferr(v),err_v)$
```

Dasselbe machen wir für den Gradienten

```
(% i9) grad_v: subst(v[2],b,subst(v[1],a,grad))$
```

```
(% i10) define(fgrad(v), grad_v)$
```

Wir gehen vom Ausgangspunkt (*v*) in Richtung des negativen Gradienten - wir verkürzen den “Gehweg” solange bis die Error-Funktion abnimmt (wir brechen aus der Schleife aus)  
eigentlich könnte das Verfahren hier scheitern (der Ausgangspunkt ist das Minimum (unwahrscheinlich))  
- aus Gründen der Übersichtlichkeit verzichten wir auf solche Überprüfungen!

```
(% i11) getLineSearchOrder(v):=block([v_n:0],
 for j:0 thru 40 do (
 v_n: v - 10-(j)*fgrad(v),
 if ferr(v_n) < ferr(v) then return()
),
 v_n
)$
```

Hier das eigentliche Verfahren - falls der Betrag des Gradienten an einer Stelle unter eine gewisse Schwelle gesunken ist, brechen wir ab

```
(% i12) grad_descent(startPoint):=block ([currentP:startPoint],
 for i thru 1000 do (
 currentP: getLineSearchOrder(currentP),
 if (fgrad(currentP) . fgrad(currentP) < 10-(8)) then return()
),
 currentP
)$
```

Aufruf des Algorithmus

```
(% i13) grad_descent([random(10), random(10)]);
```

|                         |
|-------------------------|
| $[-0.13262, -0.025251]$ |
|-------------------------|

Wenn wir obigen Vektor in die Gradienten-Funktion einsetzen, sehen wir, dass er “beinahe” verschwindet.

Da bei dieser Aufgabe der Gradient linear ist in den Variablen *a* und *b*, lässt sich diese Aufgabe auch mit einem linearen Gleichungssystem lösen - und unser Gradientenverfahren wird einem Kontrolltest unterzogen:

... bis hierher wie oben

```
(% i5) err_simp:ev(err, expand, simp, nouns)$
```

Wir ersetzen  $a$  und  $b$  durch  $x$  und  $y$  für die impliziten Kurven

```
(% i6) err_xy: subst(y,b,subst(x,a,err_simp))$
```

```
(% i7) grad:[diff(err_simp,a), diff(err_simp,b)]$
```

Wo verschwindet der Gradient? Wir lösen das lineare Gleichungssystem  
`linsolve(grad,[a,b]),nu-`

```
(% i8) mer; [a: -0.1326, b: -0.025267]
```

```
(% i9) gr1:implicit(err_xy=0.06,x,-0.2,-0.08,y,-0.08,0.05)$
```

```
(% i10) gr2:implicit(err_xy=0.04,x,-0.2,-0.08,y,-0.08,0.05)$
```

```
(% i11) gr3:implicit(err_xy=0.02,x,-0.2,-0.08,y, -0.08,0.05)$
```

```
(% i14) gr4:implicit(err_xy=0.005,x,-0.2,-0.08,y, -0.08,0.05)$
```

```
(% i15) draw2d(color=red,gr1, color=blue,gr2, color=green,gr3, color=orange,gr4,
color=violet, point_type = filled_circle, point_size = 2, points([[a,b]])
)$
```

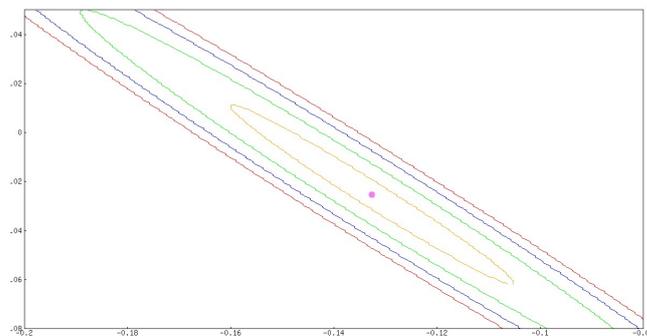


Abb.209 : Contour-Plot beim Minimum(siehe "Punkt")

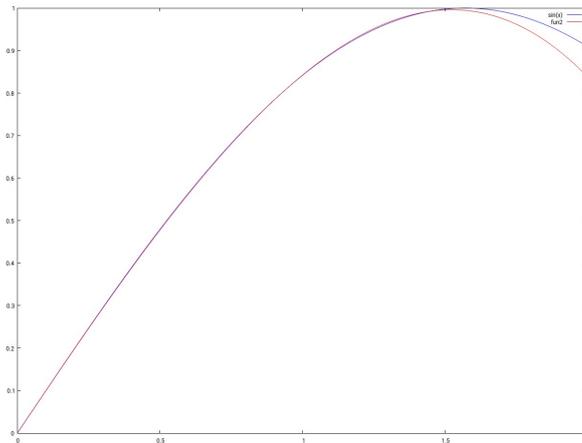


Abb.210 : Unsere Näherung ist nicht schlecht in  $[0, \pi/2]$

### Beispiel 20.11



Wir haben bei unserem letzten Beispiel bei der Näherungsfunktion  $f(x) = d + cx + bx^2 + ax^3$   $d = 0$  und  $c = 1$  gesetzt - wir haben damit (da nur 2 Variablen bleiben) eine sehr anschauliche Funktion erhalten. Rein mathematisch können wir das Gradientenabstiegsverfahren leicht auf alle 4 Variablen erweitern!

Hier die geringfügigen Änderungen:

```
(% i1) (d1:float((%pi/2))/N, x[j]:=j*d1, fprintfprec:5, ratprint:false, globalsolve:true)$
(% i2) depends(err,[a,b,c,d])$
(% i3) err:sum((d+c*x[j] + b*x[j]^2 + a*x[j]^3 - sin(x[j]))^2, j, 1, N)$
(% i4) N:100$
(% i5) err_simp:ev(err, expand,nouns)$
(% i6) grad:[diff(err_simp,a), diff(err_simp,b), diff(err_simp,c),diff(err_simp,d)]$
(% i7) err_v: subst(v[4],d,subst(v[3],c,subst(v[2],b,subst(v[1],a,err_simp))))$
(% i8) define(ferr(v),err_v)$
(% i9) grad_v: subst(v[4],d,subst(v[3],c,subst(v[2],b,subst(v[1],a,grad))))$
(% i10) define(fgrad(v), grad_v)$
(% i11) getLineSearchOrder(v):=...
(% i12) grad_descent(startPoint):=...

(% i13) linsolve(grad,[a,b,c,d]),numer; [a:-0.11297, b:-0.072043, c:1.0286, d:-0.0025019]
(% i14) min:grad_descent([random(10), random(10),random(10),random(10)]);
 [-0.11292, -0.072184, 1.0287, -0.002516]

(% i15) plot2d([sin(x), d+c*x+b*x^2+a*x^3],[x,0,2], [legend,"sin", "f"])$
```

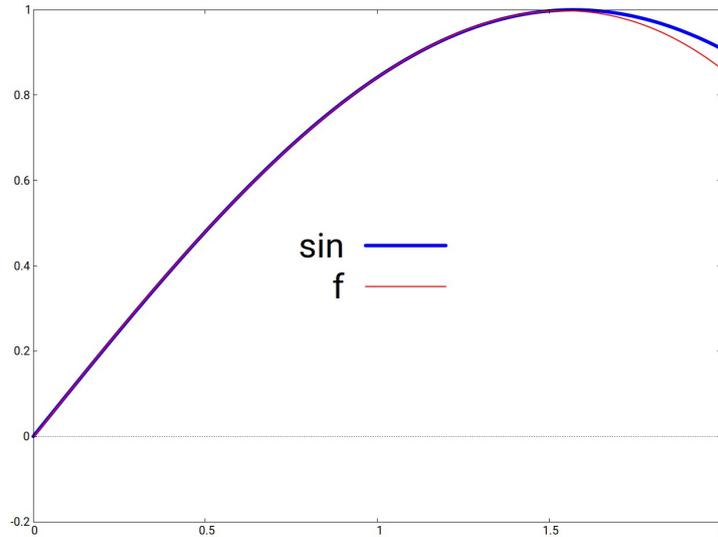


Abb.211 : Näherung in  $[0, \pi/2]$  kaum vom “Original” unterscheidbar

### 20.5.1 Erweiterte Kettenregel

Seien  $f$  und  $\vec{x}$  zwei Funktionen mit

$$f : \mathbb{R}^2 \longrightarrow \mathbb{R} \qquad \vec{x} : \mathbb{R}^2 \longrightarrow \mathbb{R}^2$$

$$(x_1, x_2) \mapsto f(x_1, x_2) := x_1^2 x_2 + x_1 x_2^2 \qquad (t_1, t_2) \mapsto \vec{x}(t_1, t_2) = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} := \begin{pmatrix} 2t_1 + t_2 \\ t_1 - 2t_2 \end{pmatrix}$$

$h := (f \circ \vec{x})$  ist also eine Funktion von  $\mathbb{R}^2 \longrightarrow \mathbb{R}$ .

Wie berechnet man den Gradienten von  $h$  bei  $(t_1, t_2) = \nabla h(t_1, t_2)$ ? Wir zeigen an Hand obigen Beispiels wie es geht, nicht warum es so geht!

☛ Zuerst die “konservative” Methode mit Substituieren:

$$h(t_1, t_2) = [x_1(t_1, t_2)]^2 x_2(t_1, t_2) + x_1(t_1, t_2) [x_2(t_1, t_2)]^2 = 2t_2^3 - 3t_1 t_2^2 - 11t_1^2 t_2 + 6t_1^3$$

$$\nabla h(t_1, t_2) = \begin{pmatrix} -3t_2^2 - 22t_1 t_2 + 18t_1^2 \\ 6t_2^2 - 6t_1 t_2 - 11t_1^2 \end{pmatrix} \qquad (20.7)$$

☛ Kettenregel in Tensorschreibweise:

$$\underbrace{\frac{\partial h(\vec{t})}{\partial t_i}}_1 = \underbrace{\frac{\partial f(\vec{x})}{\partial x_j}}_2 \underbrace{\frac{\partial x_j}{\partial t_i}}_3 \qquad (20.8)$$

## 20. Gradientenabstieg

Ausdruck 1 hat einen freien Index  $i \in \{1, 2\}$  und ist daher ein Vektor (1 dim. Tensor)  
 Ausdruck 2 hat einen freien Index  $j \in \{1, 2\}$  und ist daher ein Vektor (1 dim. Tensor)  
 Ausdruck 3 hat 2 freie Indices  $i, j$  und ist daher eine Matrix (2 dimensionaler Tensor)  
 Kommt ein Index in einem multiplikativen Ausdruck mehr als einmal vor, dann wird darüber summiert.  
 Ausdruck 2 und 3 steht also für

$$\sum_{j=1}^2 \frac{\partial f(\vec{x})}{\partial x_j} \frac{\partial x_j}{\partial t_i} \quad j \text{ ist ein "dummy" Index (jedes andere Symbol möglich)} \quad (20.9)$$

Obige Summe nennt man Kontraktion zweier Tensoren über den Index  $j$   
 2 und 3 zusammen haben also nur 1 freien Index  $i$  - bilden also einen Vektor (womit das Gleichheitszeichen in 20.8 Sinn ergibt!)

Die Matrix  $J = a_{ji} := \frac{\partial x_j}{\partial t_i}$  wird als *JACOBI-Matrix* (engl. *Jacobian*) bezeichnet.

Sei  $\vec{\partial} f$  der Zeilenvektor  $\frac{\partial f(\vec{x})}{\partial x_j}$ , dann kann man 20.9 schreiben als  $\vec{\partial} f \cdot J = J^t \cdot \vec{\partial} f^t$

(der letzte Ausdruck (mit der transponierten Matrix) gilt für Spaltenvektoren)

So jetzt an die Arbeit:

### Beispiel 20.12



Die Jacobi-Matrix  $J$  ist bei uns:  $J = \begin{pmatrix} 2 & 1 \\ 1 & -2 \end{pmatrix}$

$$\begin{aligned} \vec{\partial} f \cdot J &= (x_2^2 + 2x_1x_2 \quad 2x_1x_2 + x_1^2) \cdot \begin{pmatrix} 2 & 1 \\ 1 & -2 \end{pmatrix} = \\ &= (2x_2^2 + 6x_1x_2 + x_1^2 \quad x_2^2 - 2x_1x_2 - 2x_1^2) = (-3t_2^2 - 22t_1t_2 + 18t_1^2 \quad 6t_2^2 - 6t_1t_2 - 11t_1^2) \end{aligned}$$

oder mit Spaltenvektoren:

$$\begin{pmatrix} 2 & 1 \\ 1 & -2 \end{pmatrix}^t \cdot \begin{pmatrix} 2x_2^2 + 6x_1x_2 + x_1^2 \\ x_2^2 - 2x_1x_2 - 2x_1^2 \end{pmatrix} = \begin{pmatrix} -3t_2^2 - 22t_1t_2 + 18t_1^2 \\ 6t_2^2 - 6t_1t_2 - 11t_1^2 \end{pmatrix}$$

### 20.5.2 Erweiterte Kettenregel im Einsatz

Zurück zu unserem ursprünglichen Beispiel 20.11 - jetzt im Lichte der neuen Nomenklatur:

$$err(a, b, c, d) := \sum_{i=0}^N [d + cx_i + bx_i^2 + ax_i^3 - \sin(x_i)]^2 \quad \text{schreiben wir um zu}$$

$$\ell(\vec{v}) = \underbrace{(v_j x_i^j - y_i)}_{z_i} \cdot \underbrace{(v_j x_i^j - y_i)}_{h(z_i)} \quad j \in \{0, 1, 2, 3\} \quad i \in \{0, 1, \dots, n\} \quad (20.10)$$

•  $err \rightarrow \ell$  (für *loss* Verlustfunktion (eher üblich));  $\vec{v} = v[0], v[1] \dots v[3] := d, c \dots a$

•  $y_i := \sin(x_i)$  (Vektor);  $v_j x_i^j$  (Vektor) Summation über  $j$

•  $(v_j x_i^j - y_i) =: z_i$  (Vektor) freier Index  $i$ ;  $z_i z_i$  (Skalar) Summation über  $i$

•  $\ell(\vec{v}) = h(\vec{z}(\vec{v}))$ ;  $\ell: \mathbb{R}^4 \rightarrow \mathbb{R}$ ;  $z: \mathbb{R}^4 \rightarrow \mathbb{R}^{n+1}$ ;  $h: \mathbb{R}^{n+1} \rightarrow \mathbb{R}$

• Kronecker Delta  $\delta_{ik} = \begin{cases} 1 & \text{für } i = k \\ 0 & \text{sonst} \end{cases} \quad (\Rightarrow a_i \delta_{ik} = a_k)$

• Gradient von 20.10 mit 20.8:

$$\begin{aligned} \nabla \ell &= \frac{\partial \ell(\vec{v})}{\partial v_j} = \frac{\partial h(\vec{z})}{\partial z_i} \frac{\partial z_i(\vec{v})}{\partial v_j} = \frac{\partial}{\partial z_i} (z_k z_k) \overbrace{\frac{\partial}{\partial v_j} (v_\ell x_i^\ell - y_i)}^{\text{Jacobian}} = \left[ \frac{\partial z_k}{\partial z_i} z_k + z_k \frac{\partial z_k}{\partial z_i} \right] x_i^\ell \frac{\partial v_\ell}{\partial v_j} = \\ &= [\delta_{ki} z_k + z_k \delta_{ki}] x_i^\ell \delta_{\ell j} = 2 z_i \underbrace{x_i^j}_J \end{aligned}$$

Zusammenfassung:

1. Die Jacobi-Matrix  $J = \frac{\partial z_i}{\partial v_k} = J_{ik} = x_i^k$  -  $(n + 1)$  Zeilen, 4 Spalten
2.  $\nabla \ell = 2 \vec{z} J = J^t (2 \vec{z}^t)$
3.  $\ell = \vec{z} \cdot \vec{z}$

## 20.6 Implementation in Python

In *Python* wird die Matrixmultiplikation im Modul *numpy* mit `np.matmul(A,B)` ausgeführt. Ist  $A$  ein 1d-array wird in der Dimension eine 1 vorangestellt - aus (4) wird also (1,4) (Zeilenvektor) und das Ergebnis ist wieder ein 1d-array.

Ist  $B$  ein 1d-array wird 1 nachgestellt (es entsteht ein Spaltenvektor) und das Ergebnis ist wieder ein 1d-array.

Sehen wir uns dies an einem Beispiel an (Ausgaben in Kommentar):

## 20. Gradientenabstieg

```
1 import numpy as np
3 a=np.arange(8).reshape((2,4)) #(2x4)-Matrix
 print(f"a= {a} ", a.shape)
5 # a= [[0 1 2 3] [4 5 6 7]] (2, 4)
 b=2*np.ones(2)
7 # print(f"b= {b} ", b.shape)
 # b= [2. 2.] (2,) -> 1d array
9 # print("b*a= ", np.matmul(b, a)) #(1,2)x(2,4)
 # b*a= [8. 12. 16. 20.]
11 c=np.ones(4)
 # print(f"c= {c} ", c.shape)
13 # c= [1. 1. 1. 1.] (4,) -> 1d array
 # print("a*c= ", np.matmul(a, c)) #(2,4)x(4,1)
15 # a*c= [6. 22.] -> 1d array
 # print("c*a= ", np.matmul(c, a))
17 c.reshape(1,4) # no effect
 # print(f"c= {c} ", c.shape)
19 # c= [1. 1. 1. 1.] (4,)
 c=c[:, np.newaxis]
21 # print(f"after newaxis c= {c} ", c.shape)
 # after newaxis c= [[1.] [1.] [1.] [1.]] (4, 1) -> Spaltenvektor
23 # print("a*c= ", np.matmul(a, c))
 # a*c= [[6.] [22.]]
25 c_t = c.T
 # print(f"c_t= {c_t} ", c_t.shape)
27 # c_t= [[1. 1. 1. 1.]] (1, 4) -> Zeilenvektor
 # print("a*c_t= ", np.matmul(a, c_t))
29 #ValueError: matmul: Input operand 1 has a mismatch in its core dimension
```

Jetzt zurück zum Beispiel  $\ell(\vec{v}) = (v_j x_i^j - y_i)^2 \rightarrow \min$

```
1 import math
 import numpy as np
3 import matplotlib.pyplot as plt
 import timeit
5
 def main():
7 start = timeit.default_timer()
 x = np.linspace(0, math.pi/2, 101, dtype=np.longdouble)
9 y = np.sin(x)
 v = 10*np.array(np.random.randn(4), dtype=np.longdouble)
11 Jac = jacTransp(len(v), x)
13
 for _ in range(50000):
 v_new = getLineSearchOrder(v, y, Jac)
15 if v_new is None:
 print("No more precision possible cause of eps!")
17 break
 v = v_new
19
 loss_now = loss(v, y, Jac)
21 grad_now = gradLoss(Jac, zVec(Jac, v, y))
```

```

23 print()
 out = "***** Polynom: y ="
 for i in range(len(v)):
25 out = f'{out} {v[i]:+}*x^{i}'
 print(out)
27 print()
 print(f'loss_now: {loss_now}')
29 print(f'grad_now: {grad_now}')
 stop = timeit.default_timer()
31 print('Time: ', stop - start)
 plot(x,y,v,Jac)
33
def jacTransp(rows,x): # eigentlich J^t (Transponierte)
35 J = np.ones((rows, len(x)))
 for j in range(1,rows):
37 J[j,0:] = J[j-1,0:]*x
 return J

```

Zuerst natürlich die Modulimporte: *math* für *pi*, *numpy* für Matrizenrechnung, *matplotlib.pyplot* für Zeichnung und *timeit* für die Zeitmessung.

Im Hauptprogramm werden zuerst die 1d-arrays *x* und *y* festgelegt, der Wert der Variablen *v* wird zufällig gewählt. In Zeile 12 wird die transponierte Jacobi-Matrix berechnet, indem die Matrix zuerst mit 1 besetzt wird, die 2-te Zeile ist die erste komponentenweise mit  $\vec{x}$  multipliziert, die *j*-te Zeile ist die *j* - 1-te Zeile komponentenweise mit  $\vec{x}$  multipliziert. (Python nummeriert von 0 aus - kleinster Wert den *j* - 1 annehmen kann.)

Dann kommt die bekannte Schleife, wo in *getLineSearchOrder(v,y,J)* der Gradient  $\nabla\ell$  bestimmt wird und ein neuer Vektor  $\vec{v}_n = \vec{v} - \eta\nabla\ell$  mit  $\eta \in \{10^0, 10^{-1}, 10^{-2}, \dots, 10^{-99}\}$  bis  $\ell(\vec{v}_n) < \ell(\vec{v})$  gilt. Falls irgendeine Komponente von  $\eta\nabla\ell$  (im Programm *fg*) kleiner als die maximale Genauigkeit (*eps*) ist, wird abgebrochen.

```

42 def getLineSearchOrder(v,y,J):
 fg = gradLoss(J, zVec(J,v,y))
 for _ in range(100):
44 v_n = v - fg
 if loss(v_n,y,J) < loss(v,y,J):
46 return v_n
 fg = fg/10.0
48 if fg.any() <= np.finfo(np.longdouble).eps:
 return None
50
def zVec(Jac,v,y_exact): # vector z in text
52 return np.matmul(v,Jac)-y_exact

54 def gradLoss(J,zVector):
 return np.matmul(J,2*zVector)
56
def loss(v,y_exact,J):
58 loss_vec = zVec(J,v,y_exact)
 return np.inner(loss_vec, loss_vec) # inner product for quadratic sum
60
def plot(x,y,v,J):
62 _, ax = plt.subplots() # Create a figure containing a single axis.

```

## 20. Gradientenabstieg

```
ax.plot(x,y,linewidth=6.0, color="greenyellow")
64 ax.plot(x,zVec(J,v,np.zeros(J.shape[1])), 'r') # zVec(J,v,0) = y_p (predicted)
66 plt.title('Sinus(green) and polynomial approximation(red)',fontsize=30,pad=40)
68 plt.xlabel('x-Werte')
70 plt.ylabel('y-Werte')
plt.show()

main()
```

Die Ausgabe des Programms unterscheidet sich natürlich in “Kleinigkeiten”, da ja der Anfangsvektor  $\vec{v}$  zufällig gewählt wird. Wichtig ist, dass der Gradient beinahe verschwindet (wir sind in einem Minimum) und der Verlust klein ist. Wie “gut” unsere Lösung ist sieht man dann an der Zeichnung! Mit einer einzigen Änderung in Zeile 10 ( $4 \rightarrow n$ ) geht auf eine Näherung mit einem Polynom  $(n - 1)$ -ten Grades. Bei  $n > 4$  schnellen allerdings die nötigen Iterationen und daher die Rechenzeit beachtlich in die Höhe!

\*\*\*\*\*

No more precision possible cause of eps!

\*\*\*\*\* Polynom:  $y = -0.0021336139822648276*x^0 + 1.0268251609353407*x^1 - 0.06982693301185583*x^2 - 0.11379244484292975*x^3$

loss\_now: 7.56119764554385e-05

grad\_now: [-2.02042396e-10 -5.31290597e-10 7.40364258e-11 -5.93766768e-10]

Time: 0. Sinus(green) and polynomial approximation(red)

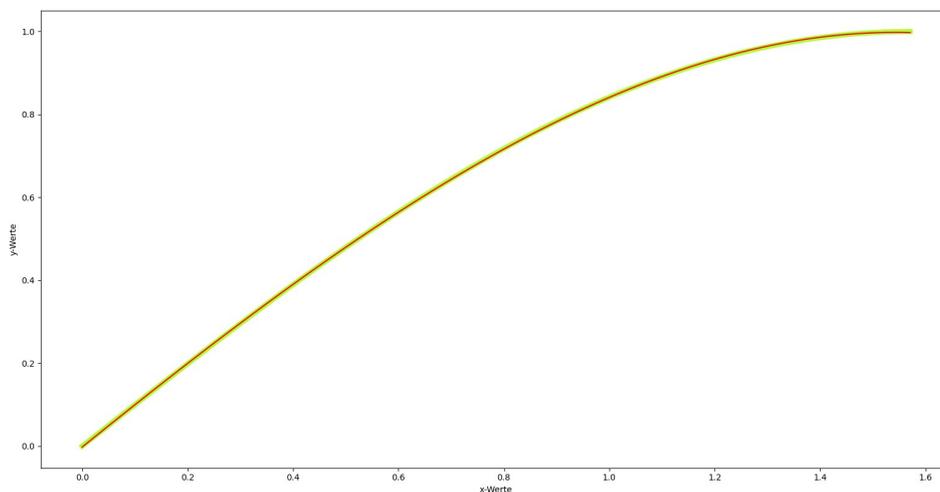


Abb.212 : Ausgabe mit *matplotlib.pyplot*)

Man kann im Python-Modul *numpy* aber die Analogie zur Tensor-Theorie noch weitertreiben, nämlich mit  $np.einsum('ij,jk',a,b) \Leftrightarrow a_{ij} b_{jk}$

$$\frac{\partial z_i}{\partial v_j} = J_{ij} = x_i^j \quad (\text{Jacobian})$$

```

34 def jacobian(x, cols):
35 J = np.ones((len(x), cols))
36 for j in range(1, cols):
37 J[0:, j] = J[0:, j-1]*x
38 return J

```

$$z_i = v_j x_i^j - y_i = v_j J_{ij} - y_i$$

```

50 def zVec(Jac, v, y_exact):
51 return np.einsum('j, ij', v, Jac) - y_exact

```

$$\nabla \ell = 2z_i x_i^j = 2z_i J_{ij}$$

```

54 def gradLoss(J, zVector):
55 return 2*np.einsum('i, ij', zVector, J)

```

$$\ell = z_i z_i$$

```

56 def loss(v, y_exact, J):
57 zVector = zVec(J, v, y_exact)
58 return np.einsum('i, i', zVector, zVector)

```

$$y_{approx} = v_j x_i^j = v_j J_{ij}$$

```

60 def plot(x, y, v, J):
61 y_approx = np.einsum('j, ij', v, J)
62 ax.plot(x, y_approx, 'r')

```

Ausgabe: **Time: 0.7723** - Tribut der “Bequemlichkeit” (50% mehr)

Ausgiebig Gebrauch vom Gradientenabstieg macht man in “Deep Learning”, wo mit Neuronalen Netzen verschiedenster Architektur das Eingangsmuster (z.B. Buchstabe) einem Ausgangsmuster (z.B. ASCII-Code) angepasst wird - der Gradientenabstieg heißt dort *Backpropagation*.



# 21 | Hough-Transformation

## 21.1 Problembeschreibung

Man hat eine Punktmenge in einer Ebene (Bild). Welche Punkte liegen auf einer Geraden d.h. bilden eine Kante? Denken Sie an Leitlinien auf der Straße (Stichwort “autonomes Fahren”) oder Nummertafelerkennung auf Mautstraßen ....

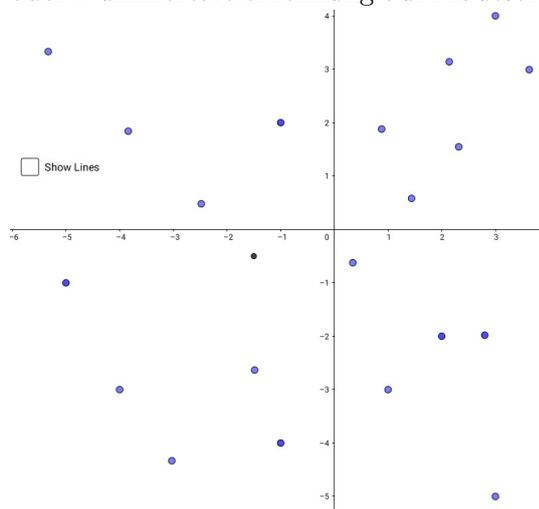


Abb.213 : Punktemenge

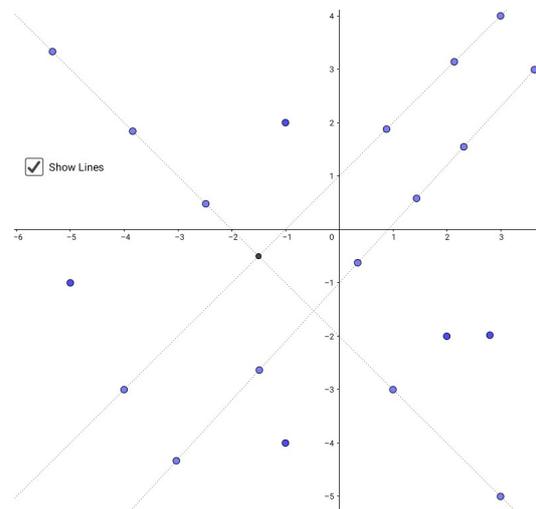


Abb.214 : eingezeichnete Geraden

Dem geschulten Auge des Lesers erscheint diese Aufgabe relativ leicht lösbar - aber wie sagt man das einem Computer?

Eine Möglichkeit:

Wir ordnen die  $L(=len(p))$  Punkte nach ihrer x-Koordinate:

$P[0]$  ist also ganz links,  $P[L-1]$  ganz rechts.

Als Datenstruktur wählen wir eine Matrix  $M[i, j] := [i, j, k_1, k_2, \dots, k_n]$  = Liste aller Punkte, die auf der Geraden durch  $P[i]$  und  $P[j]$  liegen.

Damit könnte der Pseudocode für den Algorithmus so ausschauen:

Beachte, dass die Matrix symmetrisch ist ( wir bearbeiten nur die rechte Seite (R-Matrix)) - die Diagonale ist sinnlos!  $\rightarrow$  in **Zeile 2** läuft  $j$  erst von  $(i + 1)$  weg!

**Zeile 3:** Auch  $k > j$ , weil Gerade  $g_1 : (i = 2, j = 5, k = 3) \Leftrightarrow$  Gerade  $g_2 : (i = 2, j = 3, k = 5)$

## 21. Hough-Transformation

---

```
1 for i=0 bis L-1 do
2 for j=i+1 bis L-1 do
3 for k=j+1 bis L-1 do
4 falls Abstand(Gerade(P[i], P[j]), P[k]) nahe Null, dann
5 Füge bei matrix[i][j] k zur Liste hinzu
```

Die vielen Schleifen lassen sich schwer in *Geogebra* umsetzen, daher habe ich mich für *Python* entschieden. Obiger Algorithmus ist in der Routine `common_points` umgesetzt - alle anderen Unterprogramme sind nur Hilfsroutinen zum Plotten oder für die Ausgabe.

In Zeile 37 wurde die wichtige Routine `already_listed` vorerst auskommentiert - wir werden noch auf sie zu sprechen kommen.

```
import matplotlib.pyplot as plt
2 import numpy as np
from typing import List, Tuple

4 Matrix = List[List[List[int]]] # for annotation only
6 PointLst = List[Tuple[float]] # for annotation only

8 # for plotting the points we need x- and y coordinates as separated lists
def get_xy(pts:PointLst)->List[List]:
10 x_c = [x[0] for x in pts]; y_c = [x[1] for x in pts]
 return [x_c,y_c]

12
14 # calculate distance of point p frome line(lp1,lp2) with formula:
|a x b| = d . |b| with a = lp2-lp1, b=p-lp1
def get_dist(lp1:tuple, lp2:tuple, p:tuple)->float:
16 lp1=np.asarray(lp1); lp2=np.asarray(lp2); p=np.asarray(p)
 d=np.linalg.norm(np.cross(lp2-lp1,p-lp1))/np.linalg.norm(p-lp1)
18 return d

20 # checks if the indices i,j are contained in previous lists
def already_listed(i:int, j:int, m:Matrix)->bool:
22 for r in range(i+1):
 for c in range(r+1,j):
24 if (i in m[r][c]) and (j in m[r][c]):
 return True
26 return False

28 # main algorithm as described above in text
def common_points(_p:PointLst, m:Matrix)->Matrix:
30 for i in range(len(_p)):
 for j in range(i+1, len(_p)):
32 for k in range(j+1,len(_p)):
 d=get_dist(_p[i],_p[j],_p[k])
34 # exist list with i,j in former lists?
 ### uncomment the following line for full functional program
36 # if already_listed(i,j,m):break
 if d < 0.01:
38 m[i][j].append(k)
 return m
40
```

```

output the index-list if length > threshold
42 def show_idx_mat(m:Matrix, th:int)->None:
 p_lst = []
44 for i in range(len(p)):
 for j in range(len(p)):
46 if len(m[i][j])>th:
 print(m[i][j], " ", end="")
48 print()

50 # converts index-lists to points
52 def idx2points(_mat:Matrix, _p:PointLst)->List[List[Tuple]]:
 all_lines = []
54 for i in range(len(_mat)):
 for j in range(len(_mat[0])):
56 if len(_mat[i][j]) > threshold:
 cp =[_p[i],_p[j]] #all points on a common line
58 cp += [_p[k] for k in _mat[i][j]]
 all_lines.append(cp)
60 return all_lines

62 ##### MAIN #####
p = [(-5.33, 3.33), (-2.48, 0.48), (-3.84, 1.84), (1, -3), (3, -5), (-4, -3),
64 (-1.5, -0.5),(2.14, 3.14), (3, 4), (3.63, 2.99), (2.32, 1.55), (0.34, -0.62),
(-3.03, -4.33), (-1.49, -2.63), (-1, 2), (-5, -1), (2, -2), (-1, -4),
66 (2.8, -1.98), (1.44, 0.58), (0.88, 1.88)]

68 # we sort by x coordinate, first point is on the left, last point has
position on the right side
70 p.sort(key=lambda tup: tup[0])
p = [(-5.33, 3.33), (-5, -1), (-4, -3), (-3.84, 1.84), (-3.03, -4.33), ...
72

74 # mat[2,3] = [2,3,5,9] --> line through point with index 2,3,5,9
type->Matrix: List[List[List[int]]]
76 # initialization of matrix: each line consists at least of 2 points
mat = [[[j,i] for i in range(len(p))] for j in range(len(p))]
78 # search for lines with number of points **greater** than
#threshold (=1 -> all possible lines)
80 threshold = 2
mat = common_points(p, mat)
82 show_idx_mat(mat, threshold)
all_lines = idx2points(mat, p)
84 x_c,y_c = get_xy(p)
plt.subplot()
86 for line in all_lines:
 lx,ly=get_xy(line)
88 plt.plot(lx,ly)
plt.scatter(x_c,y_c,s=100)# size of points increased
90 for i in range(len(x_c)):
 plt.annotate(str(i),(x_c[i]-0.02, y_c[i]+0.15)) # numbers of points added
92 plt.show()

```

## 21. Hough-Transformation

---

Die Routine `print_idx_mat(mat, threshold)` im Hauptprogramm liefert folgende Ausgabe:

```
[0, 3, 5, 6, 12, 18] [0, 5, 6, 12, 18] [0, 6, 12, 18] [0, 11, 16] [0, 12, 18]

[2, 6, 11, 15, 19] [2, 7, 17] [2, 11, 15, 19] [2, 15, 19]
[3, 5, 6, 12, 18] [3, 6, 12, 18] [3, 12, 18]
[4, 7, 10, 13, 16, 20] [4, 10, 13, 16, 20] [4, 13, 16, 20] [4, 16, 20]
[5, 6, 12, 18] [5, 12, 18]
[6, 11, 15, 19] [6, 12, 18] [6, 15, 19]
[7, 10, 13, 16, 20] [7, 13, 16, 20] [7, 16, 20]

[10, 13, 16, 20] [10, 16, 20]
[11, 15, 19]

[13, 16, 20]
```

Wir sehen:

Die Gerade durch die Punkte `[0,5,6,12,18]` ist bereits durch `[0,3,5,6,12,18]` festgelegt. Es ergeben sich viele weitere Redundanzen.

Wir müssen also z.B. bei `[3, 5, 6, 12, 18]` überlegen, existiert bereits:

`[0,... 3,... 5...]` oder `[1,... 3,... 5...]` oder `[2,... 3,... 5...]` oder `[3,... 5...]`

Genau diese Überprüfung leistet `already_listed`. Wir binden sie also in `common_points` ein und erzielen folgende Ausgabe:

```
[0, 3, 5, 6, 12, 18] [0, 11, 16]
[2, 6, 11, 15, 19] [2, 7, 17]
[4, 7, 10, 13, 16, 20]
```

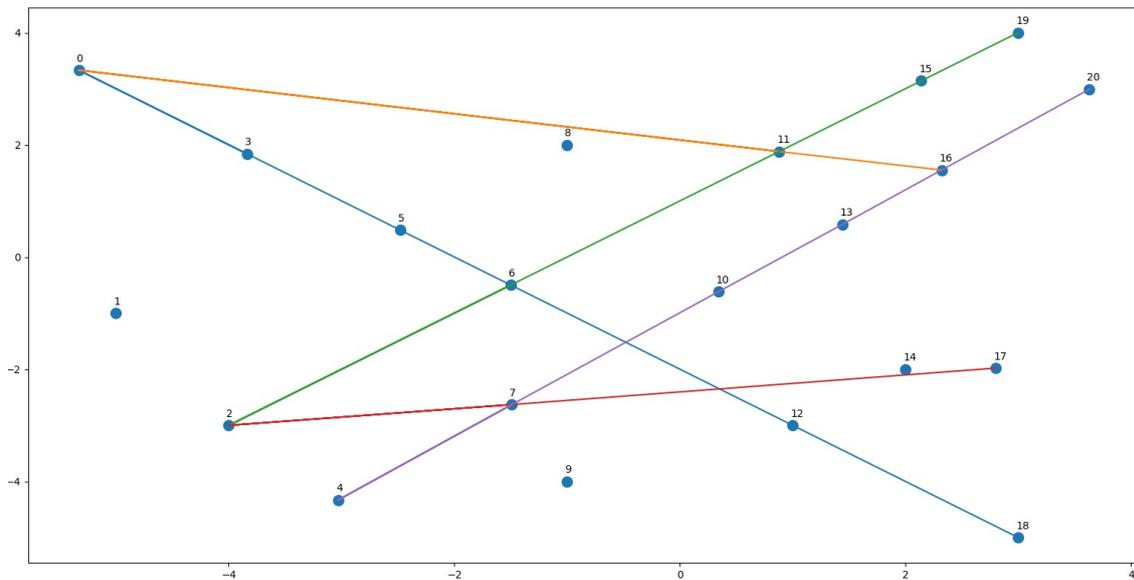


Abb.215 : Punkte auf einer Geraden

Der Algorithmus der bei einigen Punkten ganz gut funktioniert, erleidet bei größeren Punktmengen ein zeitliches Fiasko - bei Punktmengen wie sie in Bildern vorkommen erfordert die Matrix eine Größe, die den normalen Speicher eines PC (2022 ca. 16GB) überlaufen lässt.

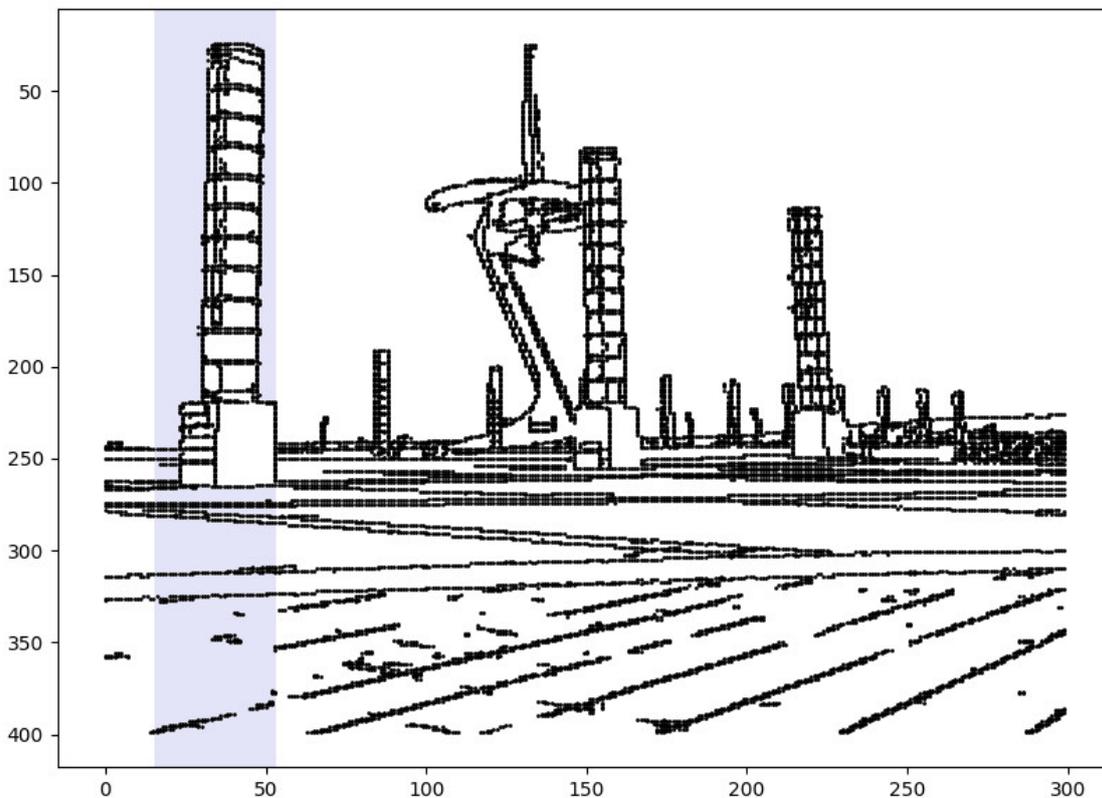


Abb.216 : Schwarz-Weiß Bild - Barcelona, Olympic Park

Allein nur den blau unterlegten Bildteil des oberen Schwarzweißbildes mit unserem Algorithmus zu analysieren benötigt auf einem Ryzen 5600G mehrere Tage - so geht's also nicht!

## 21.2 Hough Transformation

### 21.2.1 Spezielle Geradendarstellung

Sei  $g_A$  eine Gerade im  $\mathbb{R}^2$  durch den Punkt  $A$  ( $\vec{x}_A$  der Ortsvektor zu  $A$ ),  $\vec{x}$  ein Ortsvektor zu einem beliebigen Geradenpunkt und  $\vec{a}$  ein bestimmter Vektor  $\vec{x} - \vec{x}_A$  (Richtungsvektor):

$$g_A : \vec{x} = \vec{x}_A + \lambda \vec{a} \quad \lambda \in \mathbb{R}$$

Sei  $\vec{n} = \begin{pmatrix} \cos \theta \\ \sin \theta \end{pmatrix} \perp \vec{a}$  (siehe Abb. 217) und bezeichne  $d(g, P)$  den Minimalabstand  $P$  zu  $g$  :

$$\vec{x} = \vec{x}_A + \lambda \vec{a} \mid \cdot \vec{n} \Rightarrow \vec{n} \cdot \vec{x} = \vec{n} \cdot \vec{x}_A + 0 \Rightarrow \boxed{\rho = \cos \theta x_A + \sin \theta y_A} \quad \rho = d(g_A, O)$$

Halten wir fest:

**Definition 21.1 Hough-Raum**

$$\forall \rho \in \mathbb{R}^+ \quad \forall \theta \in [0, 2\pi[ : \rho = | \cos \theta x + \sin \theta y | \quad (21.1)$$

ist eine Gerade durch den Punkt  $P(x|y)$  mit  $\rho$  als Normalabstand vom Ursprung und Polarwinkel  $\theta$ .

$(\rho, \theta) \in (\mathbb{R}^+ \times [0, 2\pi[)$  bezeichnet man als Hough-Raum.

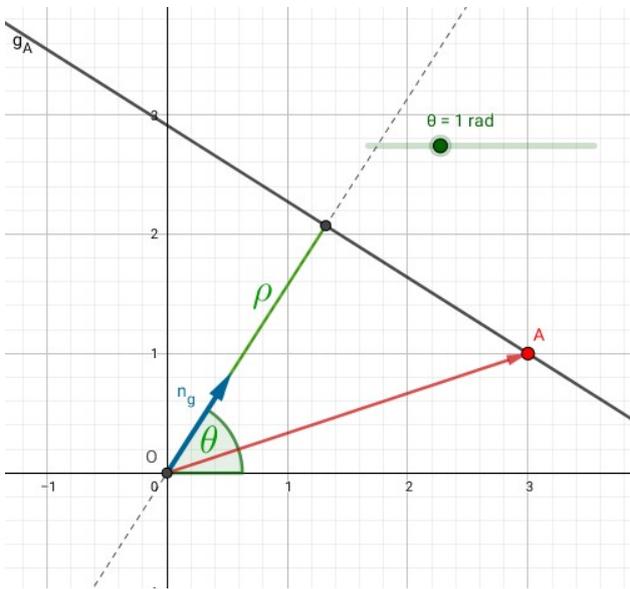


Abb.217 : Geradendarstellung in Polarkoordinaten

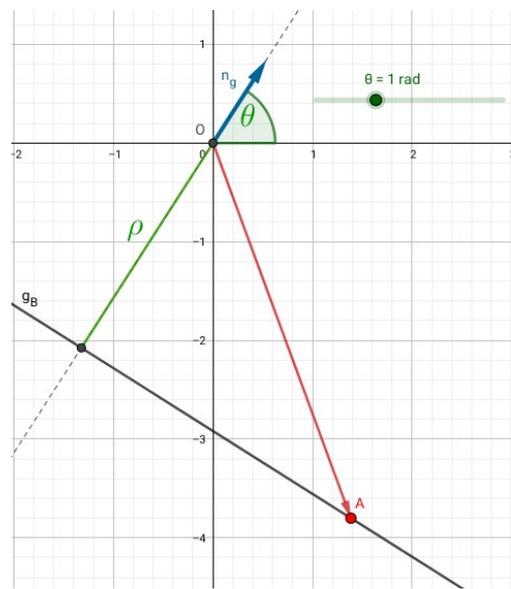


Abb.218 :  $\theta \in [0, \pi[$  möglich mit negativem  $\rho$

Lassen wir auch negative  $\rho$  zu, können wir  $\theta$  und damit den Hough-Raum einschränken:

$(\rho, \theta) \in \mathbb{R} \times [0, \pi[$  - wir lassen in Gleichung 21.1 das Betragszeichen weg!

Abb.217/218: Die beiden Geraden können bei **gleichem**  $\theta$  (statt  $\theta + \pi$ ) durch das Vorzeichen von  $\rho$  unterschieden werden!

**Beispiel 21.2**



Eine Gerade mit  $\theta = \pi/6$  geht durch den Punkt  $P(x = 2/\sqrt{3}, y = 2)$ .

Im Hough-Raum wird diese Gerade durch den Punkt

$$H(\rho, \theta) = H(x \cdot \cos \theta + y \cdot \sin \theta, \theta) = H(2, \pi/6) \text{ repräsentiert!}$$

In welcher Größenordnung bewegt sich  $\rho$ ? Als Minimalabstand ist er kleiner/gleich  $|\vec{OA}|$ . Das

## 21. Hough-Transformation

größte  $\rho$  ist als erreicht, wenn der Normalvektor  $\vec{n}_g$  in Richtung  $A$  zeigt (Abb. 217):

$$\text{dann gilt} \quad \tan \theta = \frac{y(A)}{x(A)} \quad \text{bzw.} \quad \rho_{max} = \sqrt{x(A)^2 + y(A)^2} \quad (21.2)$$

### Beispiel 21.3



Wir legen durch die Punkte  $A(3, 1)$ ,  $B(1, -1)$  und  $C(6, 4)$  alle möglichen Geraden, d.h. wir variieren über  $\theta$ (Slider).

$$\rho_X = \begin{pmatrix} \cos \theta \\ \sin \theta \end{pmatrix} \cdot \overrightarrow{OX} \quad X \in \{A, B, C\} \quad H_X = (\theta, \rho_X)$$

Für die Ortskurven vom  $H_X$  ergeben sich untenstehende ‘Sinuskurven’ nach Gleichung 21.1(ohne Betragszeichen)

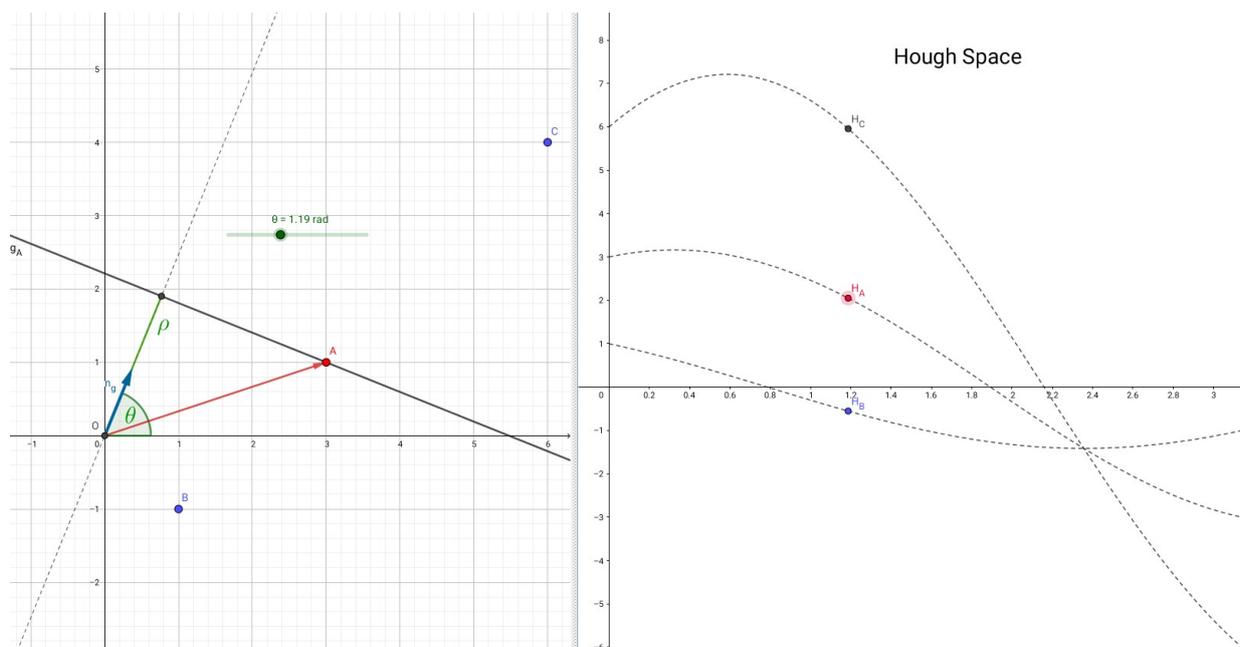


Abb.219 : ‘Alle’ Geraden durch  $A, B$  und  $C$  im Hough-Raum

Dort wo sich die Kurven schneiden, gibt es eine Gerade durch die Punkte  $A, B, C$ . Man vergewissert sich durch bewegen des Sliders für  $\theta$ .

Nun liegt ein Algorithmus für das Aufspüren von gemeinsamen Geraden vor uns:

Haben wir ein Schwarzweißbild der Höhe  $H$  und Breite  $B$  vor uns mit Koordinatenursprung links unten, dann ist laut Formel 21.2  $\rho_{max} = \sqrt{H^2 + B^2}$

Damit ist die Dimension des Hough-Space(Akkumulators):  $\theta \in [0, \pi[$  und  $\rho \in [-\rho_{max}, \rho_{max}]$  - diesen ‘quantisieren’ wir in ‘Pixel’ der Größe  $[\Delta\theta, \Delta\rho]$

Die Dimension der Akkumulatormatrix ist daher  $m := \lceil \pi / \Delta\theta \rceil \times \lceil 2\rho_{max} / \Delta\rho \rceil$

```
 $\rho_{max} := \text{sqrt}(H^2 + B^2)$
```

```
rows:=2* ρ_{max} / $\Delta\rho$
```

```
cols:= $\pi/\Delta\theta$
```

```
houghSpace[rows,cols]:=0
```

```
foreach pixel in image !=0 do
```

```
 for θ :=0 to int($\pi/\Delta\theta$) do
```

```
 ρ := int[(pixel_x*cos θ +pixel_y*sin θ + ρ_{max})/ $\Delta\rho$]
```

```
 houghSpace[ρ , θ]++
```

### 21.3 NACHTRAG

Wie entstand das Schwarzweißbild in Abb. 216:  
Ausgangsbild war eine Aufnahme des Olympic Park in Barcelona



Abb.220 : Barcelona, Olympic Park

Obiges Bild wurde mit der opencv-python Bibliothek [cv2](#) und dessen Canny-Edge Filter zu einem Schwarzweißbild verarbeitet. Dieser Filter ermittelt die Farbgradientenänderungen im Bild - man erhält also die Kanten. Die Theorie dazu wäre ein eigenes Kapitel - wir lassen das hier einmal als “blackbox” so stehen!

Den Kontrast erhöhen wir indem wir alle Werte unter 21 auf 0 reduzieren, alle über 20 auf 1. Anschließend speichern wir die Bildmatrix mit Namen  $x$  im File `pic_mat1.npz`.

Dies ist notwendig, da die Module `cv2` und `matplotlib` nicht kompatibel sind, und daher nicht gleichzeitig importiert werden können (benutzen verschiedene `gtk` Versionen)

```

1 import cv2
2 import numpy as np
3
4 def create_edge_pic():
5 img = cv2.imread("barcelona.jpg") # Read image
6 t_lower = 60 # Lower Threshold
7 t_upper = 180 # Upper threshold
8 aperture_size = 3 # Aperture size
9 # Applying the Canny Edge filter
10 edge = cv2.Canny(img, t_lower, t_upper,
11 apertureSize=aperture_size)
12 scale_percent = 50 # percent of original size
13 width = int(img.shape[1] * scale_percent / 100)
14 height = int(img.shape[0] * scale_percent / 100)
15 dim = (width, height)
16 # resize image
17 resized = cv2.resize(edge, dim, interpolation = cv2.INTER_AREA)
18 cv2.imshow('original', img)
19 cv2.imshow('edge-resized', resized)
20 cv2.waitKey(1000) # wait 1 second
21 cv2.destroyAllWindows()
22 # we save matrix: less than 21 -->0, greater than 20 --> 1
23 pic=np.asanyarray(resized)
24 pic[pic < 21] = 0; pic[pic > 20] = 1
25 np.savez("pic_mat1",x=pic)
26
27 create_edge_pic()
28

```

```

1 with np.load('pic_mat1.npz') as data:
2 pic_mat = data['x']
3 x_c = []; y_c = []
4 for c in range(15,55): # we only focus on the "blue" column in the picture
5 for r in range(len(pic_mat)):
6 if pic_mat[r][c]>0:
7 x_c.append(c)
8 y_c.append(r)
9
10 plt.subplot()
11 plt.gca().invert_yaxis()
12 plt.scatter(x_c,y_c,s=1, color="black")
13 plt.show()
14
15 start = time.time()
16 p = create_pointlist(x_c, y_c)
17 p.sort(key=lambda tup: tup[0])
18 mat = [[[j,i] for i in range(len(p))] for j in range(len(p))]
19 mat = common_points(p, mat)
20 ende = time.time()
21 print_idx_mat(mat, threshold)
22 print(f'Laufzeit: {ende - start} Sekunden')

```



# 22 | Spezielle Relativitätstheorie (SRT)

Das Kapitel wird hier behandelt, weil sich mit den Minkowskidiagrammen eine Veranschaulichung mit *Geogebra* dieser Theorie ergibt.

## 22.1 Postulate der Speziellen Relativitätstheorie

• **Relativitätsprinzip (RP):**

Die physikalischen Gesetze haben in allen Inertialsystemen dieselbe mathematische Form

• **Konstanz der Lichtgeschwindigkeit:**

Die Lichtgeschwindigkeit im Vakuum hat in allen Inertialsystemen denselben Wert

Das RP geht bereits auf Galilei zurück - er stellte bereits fest, dass im Innern von Schiffen - die sich bei ruhigen Wellengang bewegen - dieselben physikalischen Gesetze gelten wie an Land. Geriet dann allerdings mit Newtons "absoluten Raum" als Bühne der Physik etwas in den Hintergrund und erst mit Maxwells "Äthertheorie" flammte die Diskussion neu auf - Maxwells Theorie sagte die Lichtgeschwindigkeit im Vakuum voraus - Geschwindigkeit in Bezug worauf? Das Prinzip der Konstanz der Lichtgeschwindigkeit war dann aber erst eine Folge der Erkenntnisse der Experimentalphysik des ausgehenden 19. Jahrhunderts (z.B. Michelson-Moreley)

3 Beispiele von der Macht des Relativitätsprinzips:

1. Die Geschwindigkeitsvektoren 2-er Billiardkugeln nach dem Stoß bilden einen rechten Winkel!

Tischsystem: bewegte Kugel  $2\vec{v}$  und ruhende Kugel  
 Koordinatensystem  $K'$  bewegt sich mit  $\vec{v}$  auf Tisch zu  
 Wir wenden in  $K'$  die Erhaltungssätze an:

$$m\vec{v} + m(-\vec{v}) = \vec{0} = m(\vec{u}_1 + \vec{u}_2) \Rightarrow \vec{u}_1 = \vec{u} = -\vec{u}_2$$

$$2m v^2 = m u^2 + m (-u)^2 \Rightarrow |\vec{u}| = |\vec{v}|$$

$\vec{v} + \vec{u}_1, \vec{v} + \vec{u}_2$  sind die Geschw. im Tischsystem

$$v^2 - u^2 = 0 = (\vec{v} + \vec{u}) \cdot (\vec{v} - \vec{u}) \quad \text{siehe Abb. 221}$$

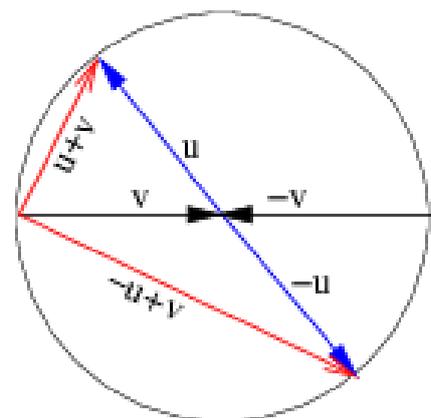


Abb.221 : Relativitätsprinzip



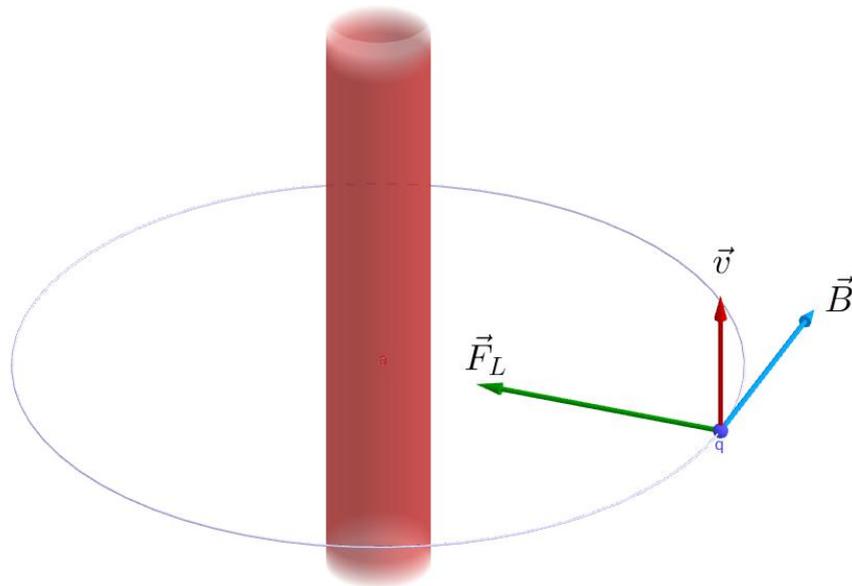


Abb.223 : Lorentzkraft  $F_L$  auf Ladung  $q$  eines stromdurchflossenen Leiters

Wir beziehen uns natürlich unwillkürlich auf ein Bezugssystem in dem der Leiter ruht und der Sachverhalt erscheint klar:

$$\vec{F}_L = q(\vec{E} + \vec{v} \times \vec{B}) \quad \begin{matrix} E = 0 \wedge \phi = 0 \\ \Downarrow \\ \Rightarrow \end{matrix} \quad F_L = qvB = qv \frac{\mu_0 I}{2\pi \rho} \quad \text{in negative } x\text{-Richtung}$$

**Was sieht ein Bezugssystem, das sich mit der Ladung  $q$  mitbewegt?**

$v$  von  $q$  ist jetzt verschwunden und damit die Lorentzkraft! Stattdessen bewegt sich der Leiter und in ihm die Ladungsträger!

Eine exakte Analyse würde hier zu weit gehen, aber die Stoßrichtung ist folgende: Im Leiter bewegen sich die positiven Atomrümpfe und die Elektronen mit verschiedenen Geschwindigkeiten an  $q$  vorbei (und damit an dessen Ruhesystem). Dies führt zu verschiedenen Längenkontraktionen, d.h. die positive und negative Ladungsdichte (Ladung pro Länge) unterscheiden sich für  $q$  - der Leiter erscheint für  $q$  geladen. Es existiert also ein elektrisches Feld  $\vec{E}$ . Rechnet man dies mit der SRT durch, ergibt sich ein  $\vec{E}$ -Feld genau mit der "richtigen" Kraft! Im Term für die Lorentzkraft ist also der "Wirkanteil" vom magnetischen auf das elektrische Feld übergegangen!



Nimmt man das RP ernst, sind das elektrische und das magnetische Feld nur Teile einer einzigen Entität ("Dings"), wir nennen es **elektromagnetisches Feld** !

In der Theorie der Elektrodynamik (Maxwell'schen Gleichungen) ist also bereits die SRT "eingearbeitet". Kein Wunder, dass diese Gleichungen **lorentzinvariant** sind.

## 22. Spezielle Relativitätstheorie (SRT)

### 22.2 Neue Einheiten

Wir benutzen hier ein “neues” Einheitensystem bei dem die Lichtgeschwindigkeit neu mit  $c_n = 1$  (dimensionslos) festgelegt wird. Dies erreichen indem wir die Geschwindigkeit  $v_n$  neu definieren

$$v_n = \frac{v_a}{c_a} \quad \text{Index } a \text{ bedeutet "alte" Einheiten}$$

Bei der Zeit bleibt alles beim alten:  $t_n = t_a$  (Einheiten Sekunden, Stunden, ...Jahre)

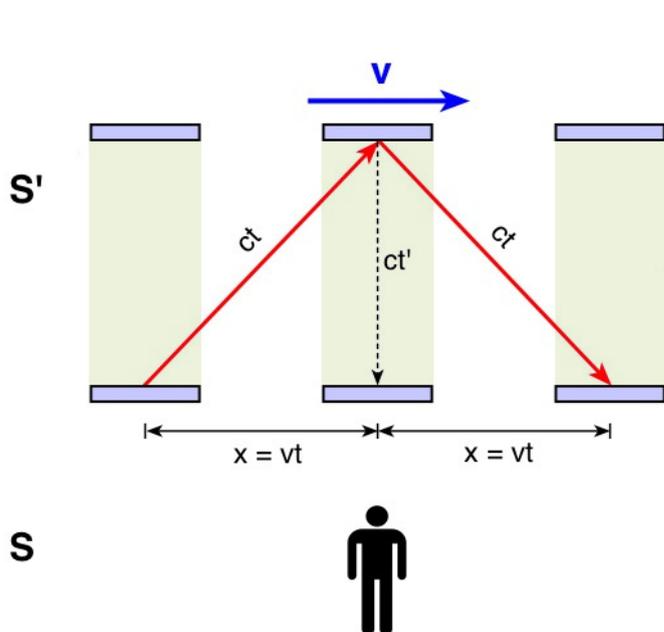
Entfernungen hängen mit obigen Größen zusammen:

$$x_a = v_a \cdot t_a = c_a \cdot v_n \cdot t_n \Rightarrow \underbrace{\frac{x_a}{c_a}}_{x_n} = v_n \cdot t_n \Rightarrow x_n = \frac{x_a}{c_a}$$

Die neuen Längeneinheiten haben also die Dimension der Zeit! (typisch für eine Raum-Zeit)

### 22.3 Lichtuhr

Wir konstruieren eine Uhr indem wir einen  $30 \text{ cm} (= d)$  langen Zylinder innen an beiden Enden verspiegeln und Photonen hin- und herschicken. Bei jeder Spiegelung wird ein Zähler inkrementiert.  $d$  in unseren neuen Einheiten ist  $10^{-9} \text{ s}$  also  $1 \text{ ns}$ . Im System  $S'$ , in welchem die Uhr ruht, wird  $t'$  gemessen ( $1 \text{ ns}$  pro “Tick”).  $S'$  bewegt sich relativ zu  $S$  mit  $v$ . In  $S$  ist der zurückgelegte Weg des Lichts länger als für eine gleichartige in  $S$  ruhende Uhr.  $t'$  ist also gedehnt - dilatiert. Diese Zeitdehnung einer “bewegten” Uhr heißt **Zeitdilatation**.



$$c = 1 \quad \gamma := \frac{1}{\sqrt{1 - v^2}} > 1$$

Pythagoräischer Lehrsatz:

$$t'^2 + (vt)^2 = t^2$$

$$t'^2 = t^2(1 - v^2)$$

$$t' \gamma = t \quad (22.1)$$

“Eine ruhende Uhr geht am schnellsten” - bewegt man sie wird sie langsamer! (Für diese Messung benötigt man mindestens 2 “synchronisierte” Uhren)

Abb.224 : Einstein'sche Lichtuhr



Was hat unsere Lichtuhr mit der "richtigen Zeit" zu tun? Es ist eine Uhr wie jede andere, die auf Elektrodynamik (Materie) basiert. Ein anderer Hinweis stammt von Myonen, die in der oberen Atmosphäre durch kosmische Strahlung entstehen und deren Halbwertszeit  $1,52 \mu s$  beträgt. Ihre mittlere "Reichweite" selbst bei Lichtgeschwindigkeit wäre also  $3 \cdot 10^{-5} km/s \cdot 1,5 \cdot 10^{-6} s \approx 450 m$ . Die Zeitdilatation liefert einen Grund, warum eine so große Anzahl von ihnen noch auf Meeresebene nachzuweisen sind!

Wie kann nun ein Beobachter in  $S$  die Zeitdilatation messen? (Eine Uhr mitbewegen ist ja wohl keine Lösung)

**Er benötigt an jedem Ort eine Uhr und diese laufen synchron!**

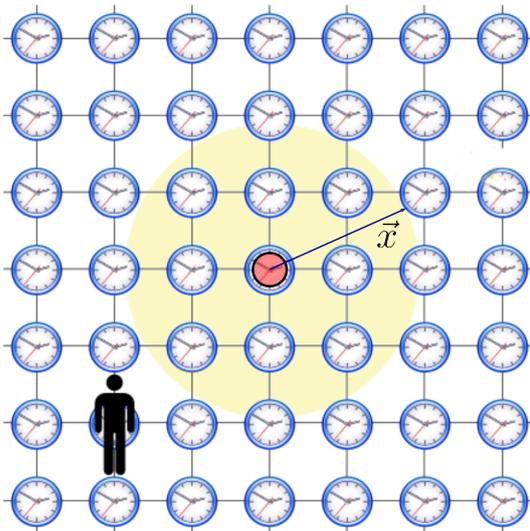


Abb.225 : synchronisierte Uhren überall

Wie kann man einen solchen Satz synchronisierter Uhren erreichen?

Man wählt einen Koordinatenursprung und nimmt die Uhr dort  $C_0$  mit der Zeit  $t_0$  als Referenzuhr. Man sendet nun vom Ursprung einen kugelförmigen Lichtblitz aus - die Laufzeit des Lichts für die Uhr an Position  $\vec{x}$  ist dann  $|\vec{x}|$ . (Beachte:  $c = 1$  in neuen Einheiten!)

Die Uhr in  $\vec{x}$  wird auf  $t_0 + |\vec{x}|$  gestellt! Der Einfachheit wählen wir  $t_0 = 0$  dann gilt in unseren System

$$t = |\vec{x}| \Leftrightarrow t^2 - x^2 = 0 \quad (22.2)$$

In einem Inertialsystem  $K'$  (Relativgeschwindigkeit  $V$ ) werden die Uhren genau so synchronisiert, sodass sich dort ergibt

$$t'^2 - x'^2 = 0 \quad \stackrel{(t'_0 = 0) \wedge (x'_0 = 0)}{\Leftrightarrow} \quad (t' - t'_0)^2 - (x' - x'_0)^2 = \Delta t'^2 - \Delta x'^2 = 0$$

es ergibt sich also der wichtige Zusammenhang zwischen beiden Inertialsystemen

$$t^2 - x^2 = t'^2 - x'^2 = 0 \quad \stackrel{(x_0, t_0) = (x'_0, t'_0) = (0, 0)}{\Leftrightarrow} \quad \Delta t^2 - \Delta x^2 = \Delta t'^2 - \Delta x'^2 = 0 \quad (22.3)$$



Die Uhrensynchronisation von  $K$  ist vom Standpunkt von  $K'$  "unsinnig" - die Uhren von  $K$  (in dem sie ruhen) bewegen sich von  $K'$  aus gesehen mit  $V$  auf das Synchronisationslicht zu (bzw. weg je nach Vorzeichen von  $V$ ) und zeigen daher eine "falsche" Zeit an. Das ist eben die Crux - eine Synchronisation, die für alle Inertialsysteme gilt, gibt es nicht! Dennoch gibt es Zusammenhänge - 22.3 ist einer davon!

## 22.4 Zweidimensionale Raumzeit

Vereinfachend stellen wir die Bewegung eines Körpers (oder die Bewegung eines anderen Koordinatensystems) in einem zweidimensionalen Raum-Zeit Koordinatensystem dar, wobei zweite und dritte Ortskoordinate unterdrückt werden, jeder Punkt  $E_1 = (x, t)$  darin stellt also ein Ereignis  $E_1$  dar mit der Ortskoordinate  $x$  und der Zeitkoordinate  $t$ . (Manchmal unterdrückt man nur 1 Ortskoordinate - veranschaulicht die Bewegung also in einem 3-dimensionalen Koordinatensystem. In der Mathematik ist man natürlich nicht auf Veranschaulichung angewiesen und schreibt  $E_1 = (\vec{x}, t)$  - 4 dimensionale Raumzeit).

### 22.4.1 Zeitdilatation mit neuer Nomenklatur

2 Koordinatensysteme  $K$  und  $K'$  - ihre Achsen stimmen zum Zeitpunkt  $t_0 = t_0' = 0$  überein und  $K'$ , in dem im Ursprung die Uhr  $C_0'$  ruht, bewegt sich in die positive  $x$ -Richtung mit der gleichförmigen Geschwindigkeit  $V$ . Die Uhren  $C_0$  und  $C_1$  von  $K$  sind nach obigem Verfahren synchronisiert.

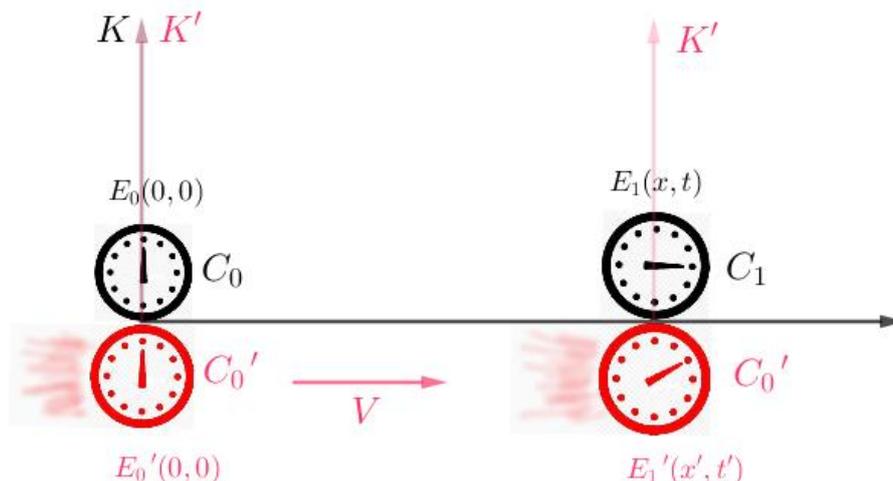


Abb.226 : 3 Uhren:Eigenzeit ist immer kleiner als Koordinatenzeit

Wegen  $E_0 = E_0' = (0, 0)$  ist die Voraussetzung für 22.3 erfüllt und gilt

$$E_1(x, t) = E_1(V \cdot t, t) = E_1'(x', t') = E_1'(0, t') \Rightarrow \Delta x = V \Delta t \text{ und } \Delta x' = 0$$

damit wird 22.3 zu

$$\Delta t^2 - (V \Delta t)^2 = \Delta t'^2 - 0 \Rightarrow \boxed{\Delta t = \gamma \Delta t'} \quad (22.4)$$

Die Zeit  $\Delta t'$ , die in einem Inertialsystem angezeigt wird, in dem die Uhr ruht, heißt **Eigenzeit (proper time interval)** - Uhr erfährt keine Ortsveränderung!

Die Zeit  $\Delta t$  wird zwischen 2 synchronisierten Uhren an verschiedenen Orten abgelesen, sie heißt **Koordinatenzeit (coordinate time interval)**

### 22.4.2 Weltlinien

Nicht jede Kurve in einem Raum-Zeit-Diagramm ist physikalisch zulässig. Wir wissen aus der Experimentalphysik, dass die Lichtgeschwindigkeit im Vakuum (in unseren Einheiten  $c = 1$ ) die maximale Geschwindigkeit eines Körpers ist, d.h. die Weltlinie eines Körpers muss sich jederzeit innerhalb eines "Lichtkegels" - der diese maximale Geschwindigkeit repräsentiert - befinden.

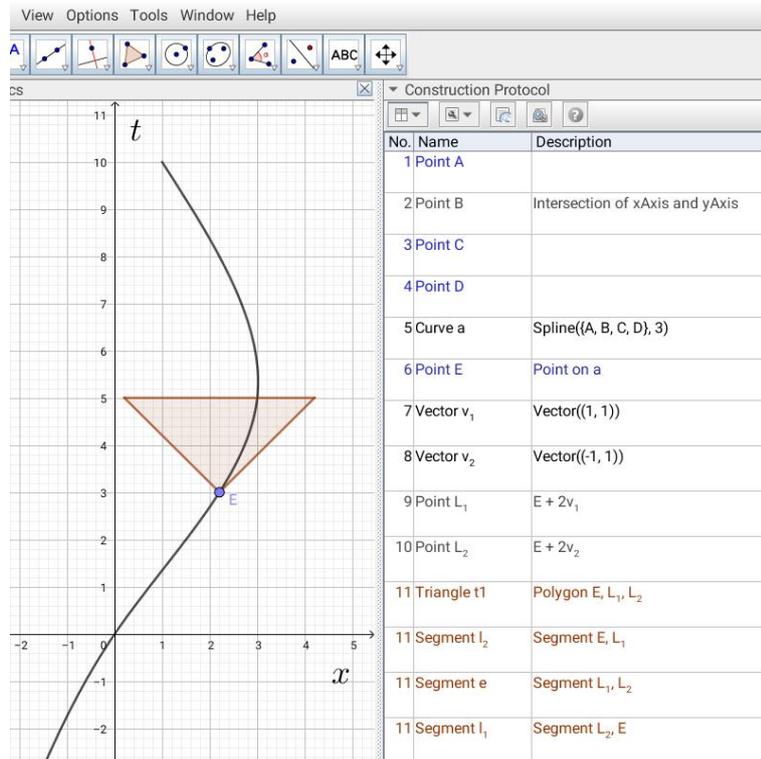


Abb.227 : Weltlinie eines "realen" Körpers

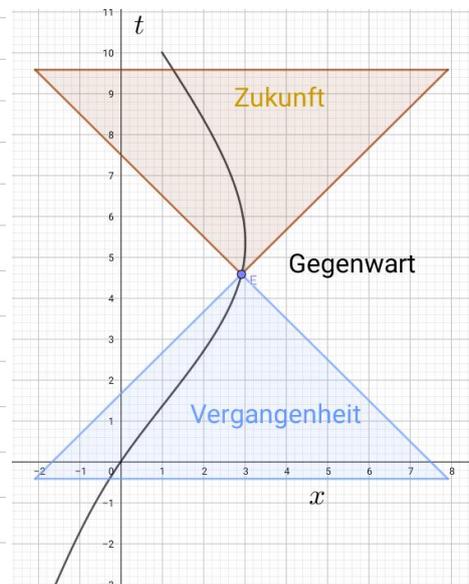


Abb.228 : 3 Bereiche in Bezug auf E

Verschiebt man den Massenpunkt  $E$  entlang seiner Weltlinie, muss die Kurve (falls sie eine Weltlinie von  $E$  sein soll) innerhalb des braunen Lichtkegels liegen. Bei uns ist dieser Kegel ein Dreieck, weil wir 2 Dimensionen unterdrückt haben - sonst ist es ein 4-dimensionaler Kegel. Außerdem können nur Ereignisse, die im blauen Kegel liegen  $E$  beeinflussen - dieser stellt also die mögliche Vergangenheit von  $E$  dar - es ist klar, dass die Weltlinie auch innerhalb dieses Kegels liegen muss.

$E$  teilt also die Raumzeit in 3 Bereiche: Zukunft, Vergangenheit und Gegenwart - wobei eigentlich das Wort "mögliche" beigefügt werden müsste!

Wieso die Gegenwart nicht nur aus einer waagrechten Linie (bzw. Ebene bzw. Hyperebene) durch  $E$  besteht wird noch diskutiert werden!



Lassen Sie sich nicht täuschen - die Gegenwart zerfällt nicht in 2 unzusammenhängende Bereiche - sondern sie ist das Gebiet "rund" um die Kegel, die man sich wiederum unbegrenzt fortgesetzt denken muss!

## 22. Spezielle Relativitätstheorie (SRT)

### 22.4.3 Lorentztransformation

Ist unser Koordinatensystem  $K$  ein Inertialsystem (also unbeschleunigt) muss die Weltlinie (Historie) eines anderen Inertialsystems  $K'$  eine Gerade sein. Eine Transformation  $\Lambda$  von  $K$  nach  $K'$  muss also linear sein! ( $\vec{x} = (x, t)$ )

$$\text{Gerade } \vec{x} = \vec{p} + \lambda \vec{r} \Rightarrow \vec{x}' = \Lambda(\vec{x}) = \Lambda(\vec{p} + \lambda \vec{r}) = \Lambda(\vec{p}) + \lambda \Lambda(\vec{r}) = \vec{p}' + \lambda \vec{r}'$$

Wir wissen aus Theorem 5.3, dass jede lineare Transformation als Matrix darstellbar ist. Es muss also gelten

$$\begin{pmatrix} x \\ t \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x' \\ t' \end{pmatrix} = \begin{pmatrix} a x' + b t' \\ c x' + d t' \end{pmatrix}$$

Nehmen wir uns die erste Zeile für die korrespondierenden Ereignisse

$E_1(V t, t) \leftrightarrow E'_1(0, t')$  und  $E_2(0, t) \leftrightarrow E'_2(-V t', t')$  vor - aus der ersten Gleichheit ergibt sich

$$t' = t/\gamma$$

$$V t = a \cdot 0 + b t' \quad \stackrel{!}{\Rightarrow} \quad b = \gamma V \quad \text{und mit } E_2$$

$$0 = a(-V t') + b t' \Rightarrow (\gamma V - a V) t' = 0 \Rightarrow a = \gamma \quad \text{also}$$

$$x = \gamma(x' + V t')$$

Da in beiden Systemen (zunächst für positive Ortskoordinaten)  $t = x$  bzw.  $x' = t'$  wegen der Uhrensynchronisation gilt, wird durch Einsetzen

$$t = \gamma(t' + V x')$$

also zusammengefasst

#### Definition 22.1 Lorentztransformation

2 Koordinatensysteme  $K$  und  $K'$  - wobei ihre Achsen zum Zeitpunkt  $t_0 = t_0' = 0$  übereinstimmen und  $K'$  bewegt sich in die positive  $x$ -Richtung mit der gleichförmigen Geschwindigkeit  $V$ . Ihre Uhren sind nach obigem Verfahren synchronisiert - dann transformieren ihre Koordinaten mit folgender Matrix  $\Lambda$ :

$$\left. \begin{array}{l} x = \gamma(x' + V t') \\ t = \gamma(t' + V x') \end{array} \right\} \Leftrightarrow \begin{pmatrix} x \\ t \end{pmatrix} = \Lambda \begin{pmatrix} x' \\ t' \end{pmatrix} = \gamma \begin{pmatrix} 1 & V \\ V & 1 \end{pmatrix} \begin{pmatrix} x' \\ t' \end{pmatrix}$$

$$\begin{pmatrix} x' \\ t' \end{pmatrix} = \Lambda^{-1} \begin{pmatrix} x \\ t \end{pmatrix} = \gamma \begin{pmatrix} 1 & -V \\ -V & 1 \end{pmatrix} \begin{pmatrix} x \\ t \end{pmatrix} \quad (22.5)$$

Auch die Gültigkeit von 22.3 lässt sich leicht nachweisen!  
Algebraische Untersuchung von  $\Lambda$  siehe Anhang!

### 22.4.4 Veranschaulichung

Die Lorentztransformation lässt sich gut mit geometrischen Mitteln veranschaulichen und dazu ist *Geogebra* ein tolles Programm - dies ist der eigentliche Grund, warum die SRT in diesem Buch gelandet ist.

Dazu schreiben wir 22.5 zur Linearkombination um (siehe auch 5.1):

$$\begin{pmatrix} x \\ t \end{pmatrix} = \gamma \begin{pmatrix} 1 & V \\ V & 1 \end{pmatrix} \begin{pmatrix} x' \\ t' \end{pmatrix} = \gamma \left[ x' \begin{pmatrix} 1 \\ V \end{pmatrix} + t' \begin{pmatrix} V \\ 1 \end{pmatrix} \right] = x' \underbrace{\gamma \begin{pmatrix} 1 \\ V \end{pmatrix}}_{\vec{v}_1} + t' \underbrace{\gamma \begin{pmatrix} V \\ 1 \end{pmatrix}}_{\vec{v}_2}$$

wir können also schreiben

$$x' \vec{v}_1 + t' \vec{v}_2 = \begin{pmatrix} x \\ t \end{pmatrix} = \vec{x} \quad (22.6)$$

$\vec{x}$  wird also in die Komponenten  $x' \vec{v}_1$  und  $t' \vec{v}_2$  zerlegt, d.h.  $x'$  und  $t'$  sind die Koordinaten von  $\vec{x}$  bezüglich der Basis  $B = \{\vec{v}_1, \vec{v}_2\}$



$\vec{v}_1$  und  $\vec{v}_2$  sind weder Einheitsvektoren noch orthogonal - allerdings sind beide gleich lang und liegen symmetrisch um die 1. Mediane

22.6 ist eine Handlungsanweisung für eine zeichnerische Darstellung:

- Zeichne  $\vec{x}$  zu einem beliebigen Punkt der Raumzeit
- Zeichne  $\vec{v}_1$  und  $\vec{v}_2$  und
- bestimme deren "Anteil" bei der Addition zu  $\vec{x}$

Dieses Programm setzen wir jetzt in *Geogebra* um.

Die Eckpunkte des Konstruktionsprotokolls:

- Slider für  $V \in [0, 1]$ ,  $O = (0, 0)$ ,  $P_1 = (1, V)$ ,  $P_2 = (V, 1)$  und Mediane  $s : y = x$
- $S'$ -Achsen:  $x_s : \text{Line}(O, P_1)$  bzw.  $t_s : \text{Line}(O, P_2)$
- Um  $\vec{v}_1$  zu erhalten, gilt es  $\overrightarrow{OP_1}$  um  $\gamma$  zu verlängern:

- (a)  $H_3$ : Schnitt des Einheitskreises mit waagrechter Geraden  $g_5$  durch  $P_1$
- (b)  $H_1 : (x(H_3), 0)$  wobei  $x(H_3) = \sqrt{1 - V^2} = 1/\gamma$  und  $H_2 : (1, 0)$
- (c)  $g_1 : \text{Line}(H_1, P_1)$  und  $g_2 \parallel g_1 \wedge H_2 \in g_2$ ,  $V_1 : g_2 \cap x_s$ , es gilt

$$\frac{\overrightarrow{OP_1}}{\overrightarrow{OH_1}} = \frac{\overrightarrow{OV_1}}{\overrightarrow{OH_2}} \Rightarrow \overrightarrow{OV_1} = \gamma \overrightarrow{OP_1} \Rightarrow \overrightarrow{OV_1} = \vec{v}_1$$

## 22. Spezielle Relativitätstheorie (SRT)

- $v_1 : Vector(V_1), V_2 : Reflect(V_1, s)$  und  $v_2 : Vector(V_2)$
- Ein beliebiger Punkt  $P$  ( $\vec{x}$  im Text - frei beweglich)
- Wir ermitteln jetzt die Koordinaten  $x_{P'}$  und  $t_{P'}$  bezüglich der Basis  $\vec{v}_1$  und  $\vec{v}_2$ :
  - (a)  $g_3 : P \in g_3 \wedge g_3 \parallel t_s$  und  $g_4 : P \in g_4 \wedge g_4 \parallel x_s$ ;
  - (b) Punkte  $X_P = g_3 \cap x_s \wedge T_P = g_4 \cap t_s$  - wir zeigen die *Caption* der beiden Punkte mit  $x_{P'}$  und  $t_{P'}$  an - was natürlich nicht stimmt,  $x_{P'}$  ist ja eine Zahl - kein Punkt.
  - (c) Es gilt

$$x_{P'} \overrightarrow{OV_1} = \overrightarrow{OT_P} \Rightarrow x\text{-Koordinate reicht: } x_{P'} = \frac{x(X_P)}{x(V_1)} \quad \text{analog für } t_{P'}$$

- Wir überprüfen mit Rechnung (22.5):

$$\begin{pmatrix} x' \\ t' \end{pmatrix} = \gamma \begin{pmatrix} 1 & -V \\ -V & 1 \end{pmatrix} \begin{pmatrix} x \\ t \end{pmatrix} = \frac{5}{4} \begin{pmatrix} 1 & -3/5 \\ -3/5 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 6/5 \end{pmatrix} = \begin{pmatrix} 7/20 \\ 3/4 \end{pmatrix}$$

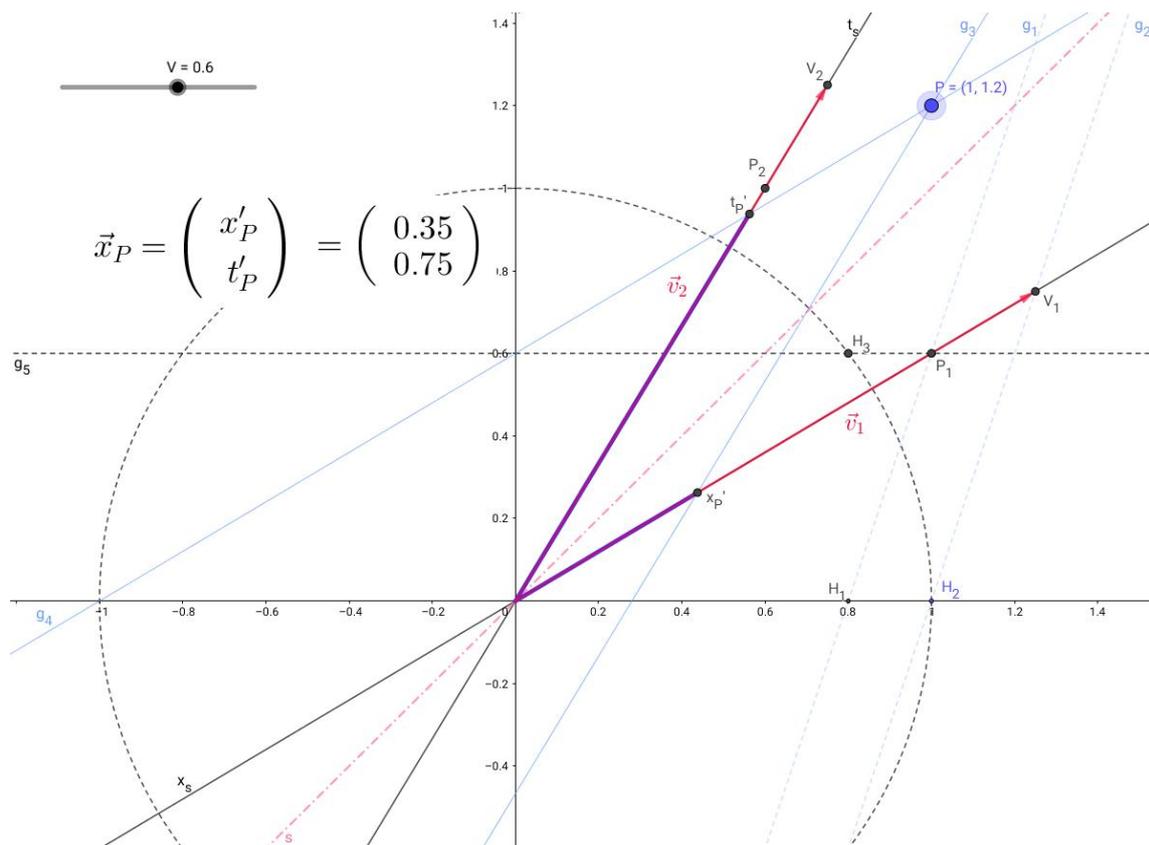


Abb.229 : Veranschaulichung der Lorentztransformation

### 22.4.5 Einheiten in S'

In Abb.229 haben wir den Basisvektor  $\vec{v}_1 = \overrightarrow{OV_1}$  relativ "aufwendig" bzw. umständlich konstruiert. Mit der Lorentztransformation (22.5) ginge das schneller:

Wir suchen jene Punkte der  $x'$ -Achse mit Abstand  $n$  vom Ursprung, und zwar in den Koordinaten von  $S - (x, t)$  :

$$\begin{pmatrix} x \\ t \end{pmatrix} = \gamma \begin{pmatrix} 1 & V \\ V & 1 \end{pmatrix} \begin{pmatrix} x' \\ t' \end{pmatrix} = \gamma \begin{pmatrix} 1 & V \\ V & 1 \end{pmatrix} \begin{pmatrix} n \\ 0 \end{pmatrix} \Rightarrow \begin{pmatrix} x \\ t \end{pmatrix} = n \begin{pmatrix} \gamma \\ \gamma V \end{pmatrix} \quad (22.7)$$

Die letzte Gleichung stellt eine durch  $V$  parametrisierte Kurve dar - wir bilden die quadratische Differenz der Koordinaten, sodass der Parameter verschwindet

$$x^2 - t^2 = n^2(\gamma^2 - \gamma^2 V^2) = n^2 \Leftrightarrow \frac{x^2}{n^2} - \frac{t^2}{n^2} = 1 \quad \text{eine gleichseitige Hyperbel}$$

Die "Eichhyperbeln" für die  $t'$ -Achse sind natürlich die Spiegelungen um die 1. Mediane:

$$\frac{t^2}{n^2} - \frac{x^2}{n^2} = 1$$

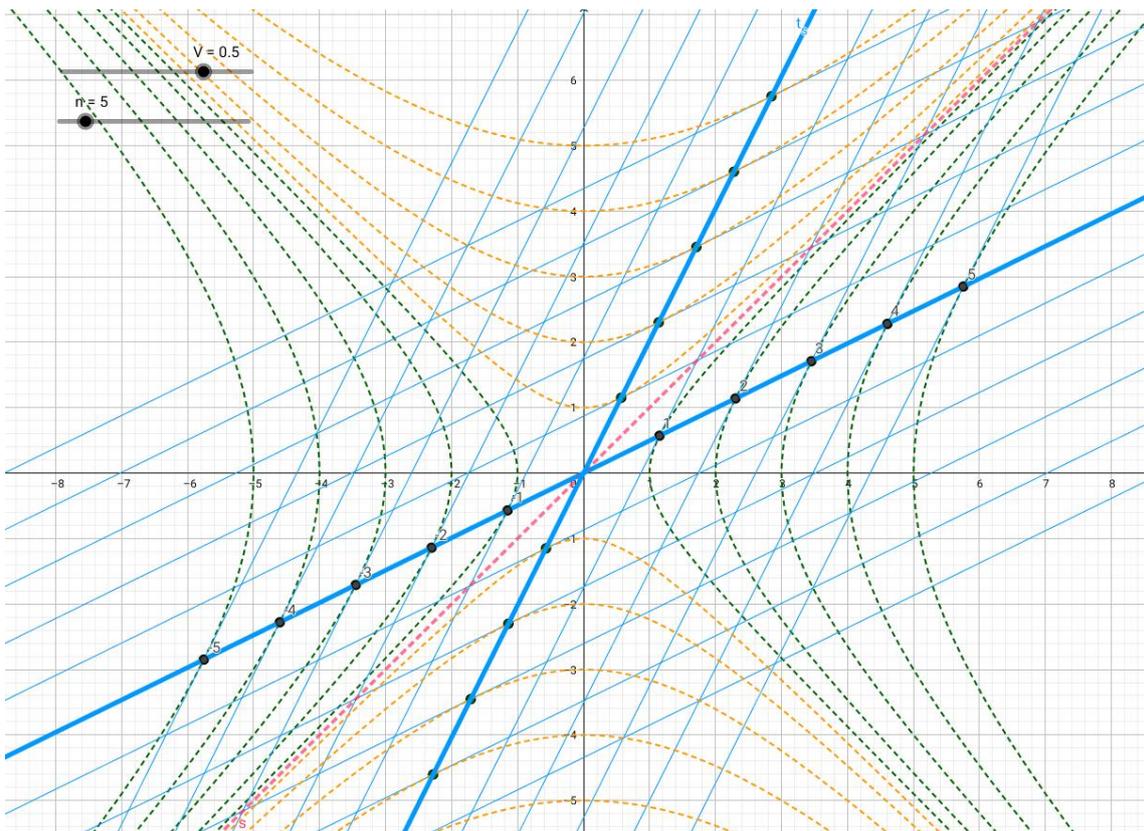


Abb.230 : Einheiten in S'

## 22. Spezielle Relativitätstheorie (SRT)

- Slider für  $V$ , Mediane  $s$ ,  $x_s$ - und  $t_s$ -Achse wie vorhin
- Integer-Slider für  $n$  - Anzahl der "Eichhyperbeln"
- Wir zeichnen die  $n$  Hyperbeln für  $x_s$  und  $t_s$ : `hypListx:Sequence(x^2-y^2=k^2,k,1,n)` und `hypListt:Sequence(Reflect(Element(hypListx,k),s),k,1,Length(hypListx))`
- Schneiden der Hyperbeln mit der  $x_s$ -Achse:  
`gridx:Sort(Flatten(Sequence({Intersect(hypListx(k), x_s)}, k, 1, n)))`  
Beachte die geschwungene Klammer vor `Intersect` (Tip vom Forum!). `Sequence` verlangt 1 Ausdruck - auch wenn er eine Liste ist - dafür bekommen wir eine Liste von Listen (2 Schnittpunkte), die wir mit `Flatten` wieder vereinfachen und nach den  $x$ -Werten sortieren.
- Einheiten auf der  $t_s$ -Achse: `gridt:Sort(Sequence(Reflect(gridx(k),s),k,1,2n))`
- Das ganzzahlige Liniengitter in  $S'$ : `l1:Sequence(Line(gridx(k), t_s),k,1,2n)` und `l2:Sequence(Line(gridt(k),x_s),k,1,2n)`
- Die Beschriftung der Punkte lässt sich nur für "individuelle" Punkte skripten - also müssen wir sie erzeugen. Ich habe das in Javascript für `Update von gridx` erledigt. Die Punkte bezeichnen wir mit  $X_1, X_2, \dots, X_{2n}$ . Wir speichern das vorige  $2n$  (= höchster Index) in `upIndex`. Falls  $n$  dekrementiert wird, müssen die oberen Punkte gelöscht werden!

```
var points; //Javascript array for gridx - start with 0!!
2
function evCmd(cmd){
4 ggbApplet.evalCommand(cmd);
}
6
function importPointList(){
8 var pL = ""+ggbApplet.getValueString("gridx");
 pL=pL.split("=")[1]; // split off "gridx"
10 pL=pL.replace(/\{/g, "["); // convert to Javascript syntax
 pL=pL.replace(/\}/g, "]");
12 pL=pL.replace(/\\/g, "[");
 pL=pL.replace(/\\/g, "]");
14 return eval(pL);
}
16
//j current Index, l length of array
18 function createPoint(j,l){
 evCmd("X_{"+(j+1)+"}:Point({"+ points[j][0]+", "+points[j][1]+ "})");
20 // if j<l Caption is negative
 evCmd("SetCaption(X_{"+(j+1)+"}, \""+ +(j-1)+"\)");
22 }
24 function deletePoints(low,up){
 for (var j=low; j < up+1; j++) ggbApplet.deleteObject("X_{"+j+"}");
26 }
```

```

// main function
2 function labelPoints(){
 points=importPointList(); // javascript array created
4 var n=ggbApplet.getValue("n");
 var upDel=ggbApplet.getValue("upIndex");
6 deletePoints(n+1,upDel); //delete superfluous points
 for (var i=0; i < n; i++) createPoint(i,n); // negative labels
8 for (var i=n; i < 2*n; i++) createPoint(i,n-1); // positive labels
 ggbApplet.setValue("upIndex",2*n); // store for next run
10 }
12 labelPoints();

```

An der Zeichnung sieht man, dass die Gittergeraden durch die Schnittpunkte der Hyperbeln mit den Achsen auch die Tangenten an die Hyperbeln sind - das kann doch kein Zufall sein:

$$\text{hyp}_n : x^2 - t^2 = n^2 \quad x_s : t = V \cdot x \Rightarrow \text{hyp}_n \cap x_s : x^2 - V^2 x^2 = n^2 \Rightarrow x = \pm \gamma n$$

Schnittpunktkoordinaten im System S:  $S_n(\pm \gamma n, \pm \gamma V n)$

$$\text{Ableitung von } \text{hyp}_n : 2x - 2t \frac{dt}{dx} = 0 \Rightarrow \frac{dt}{dx} = \frac{x}{t} = \frac{\pm \gamma n}{\pm \gamma V n} = \frac{1}{V} = \frac{dt_s}{dx}$$

Die Tangenten in den Schnittpunkten haben also dieselbe Steigung wie die  $t_s$ -Achse!



Tangenten vom linken Hyperbelast stehen für negative  $x'$ -Koordinaten und Tangenten vom unteren Hyperbelast für negative  $t'$ -Koordinaten.

Damit ergibt sich eine Möglichkeit die Koordinaten von  $P(x,t)$  in  $S'$  zu bestimmen, ohne vorher deren Koordinatenachsen zu eichen:

- ☛ Man zeichnet die "Messhyperbel" für die  $x'$ -Achse mit  $s_x \in \mathbb{R}^+$  als *Slider* ausgeführt:  
hyp\_x:  $x^2 - y^2 = s_x^2$
- ☛ Wir schneiden mit der  $x'$ -Achse:  $\text{hyp}_x \cap x_s = \{A, B\}$ :  
Intersect(hyp\_x, x\_s)
- ☛ Wir legen in A und B die Tangenten an  $\text{hyp}_x$ :  
t\_A: Tangent(A, hyp\_x) und t\_B: Tangent(B, hyp\_x)
- ☛ Jetzt wir der *Slider*  $s_x$  so verändert bis die Tangente  $t_A$  bzw.  $t_B$  den Punkt  $P$  schneidet (mit Zoomen und kleiner Schrittweite von  $s_x$  erzielt man eine hohe Genauigkeit)!
- ☛ Dieselbe Prozedur wendet man für die Ermittlung von  $t'$  an, also  
hyp\_y:  $y^2 - x^2 = s_y^2$  usw.
- ☛ Zur "Probe" eichen wir die  $x'$ -Achse:  
Aus 22.7 wissen wir  $(x = \gamma) \cap x_s = \{E_1\}$  ist der Einheitspunkt
- ☛ Die Skalierung der  $x'$ -Achse ergibt sich dann mit  
grid\_x: Sequence(0 + n Vector(E\_1), n, -10, 10)



### 22.4.6 Zeitdilatation im Minkowski-Diagramm

#### Der alte ‘‘Anfängerirrtum’’

Oft bekommt man von ‘‘Novizen der SRT’’ zu horen, dass es doch nicht moglich sein konne, dass ein Beobachter von  $S$  behauptet, die Uhren in  $S'$  gehen langsamer **und** ein Beobachter von  $S'$  behauptet dasselbe fur die Uhren von  $S$ . Immer wieder muss man darauf hinweisen, dass die Gleichzeitigkeit, d.h. die Synchronizitat der Uhren ein relatives Phanomen (darum Relativitatstheorie) darstellen - es sind dabei mindestens 2 synchronisierte Uhren  $C_1, C_2$  (mit einem gewissen Abstand) beteiligt, die relativ zueinander in Ruhe sind, und 1 Uhr  $U_1$ , die sich zu diesem synchronisierten Uhrensatz mit konstantem  $V$  bewegt. Wer sich bewegt ist physikalisch nicht feststellbar (Relativitatsprinzip), deshalb konnte ein Beobachter neben  $U_1$  in einem Abstand eine Uhr  $U_2$  aufstellen und beide Uhren synchronisieren - nur fur einen Beobachter im System, in dem  $C_1, C_2$  ruhen, ist dies **keine** Synchronisation - fur ihn zeigen  $U_1$  und  $U_2$  ‘‘Hausnummern’’ an und er vermisst sie einzeln beim ‘‘vorbeifliegen’’ von  $C_1$  und  $C_2$ . Das Umgekehrte gilt naturlich genauso! Sehen wir uns das im Minkowskidiagramm an:

#### Ablesen im Minkowski-Diagramm

Wir zoomen uns in Abb.231 rein. Wegen 22.7 kennen wir die Koordinaten von  $E_1$  in  $S$  und  $S'$  und daher auch von  $E_2 \rightarrow E_2(x = \gamma V, t = \gamma) = E_2(x' = 0, t' = 1)$

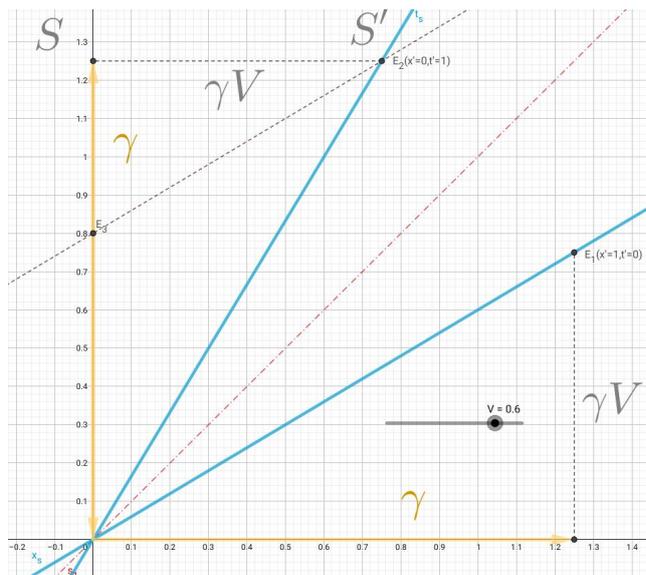


Abb.232 : Zeitdilatation im Minkowski-Diagramm

Ein Beobachter  $B$  in  $S$  hat 2 Uhren  $C_1$  und  $C_2$  aufgestellt, eine im Ursprung eine bei  $x = \gamma$  und beide synchronisiert.

$B$  vermisst eine Uhr  $U_1$ , die mit  $V$  durch seinen Ursprung fliegt. Beide Uhren sind zu diesem Zeitpunkt auf Null gestellt.

Messung bei  $E_2 : t = \gamma, t' = 1 \Rightarrow t = \gamma t'$

**Die bewegte Uhr geht langsamer!**

Ein Beobachter  $B'$  in  $S'$  hat auch sein System mit synchronisierten Uhren zugeflastert. Als  $C_1$  durch  $E_3$  fliegt, liest er diese ab - sie zeigt 0.8, seine zeigt 1!

Also auch er kommt zum Schluss:

**Die bewegte Uhr geht langsamer!**

Der Dilatationsfaktor ist jeweils  $\frac{1}{\gamma}$

Berechnung der  $t$ -Koordinate von  $E_3$ :

$$g \parallel x_s \wedge E_2 \in g \Rightarrow g : y(E_2) = V \cdot x(E_2) + t \Rightarrow t = \gamma - \gamma V^2 = \frac{1}{\gamma}$$

Dem ‘‘Paradoxon’’ zugrunde liegt die **Relativitat der Gleichzeitigkeit**

## 22. Spezielle Relativitätstheorie (SRT)

### 22.4.7 Zeitdilatation andersrum - Längenkontraktion

#### Ein Gedankenexperiment (noch)

Ein Raumschiff steht zu seinem Zielstern (10 Lichtjahre entfernt) in Ruhe. Es beschleunigt innerhalb eines halben Tages auf  $V = \sqrt{99/100}$ . Erreicht sein Ziel und bremst innerhalb eines halben Tages wieder auf Relativgeschwindigkeit zum Ziel  $V = 0$  ab. Wir vernachlässigen den Tag der Beschleunigungsphase (in dem das Raumschiff kein Inertialsystem darstellt) und sehen uns den Rest an - in dem man das Raumschiff als ruhendes Inertialsystem betrachten kann:

$\gamma = 10$ , d.h. statt annähernd 10 Jahre benötigt das Raumschiff eine Eigenzeit von ungefähr

$$\tau = \frac{t}{\gamma} \approx \frac{10}{10} = 1 \text{ Jahr.}$$

Im Raumschiff ist man also der berechtigten Meinung:

Entweder ist man mit 10-facher Lichtgeschwindigkeit geflogen oder die Entfernung hat sich um  $\gamma$  verkürzt. Erstere Hypothese kann die Raumschiffbesatzung wegen Geschwindigkeitsmessungen verwerfen.

Bezeichnet man die Strecke Raumschiff-Zielstern als "Maßstab", so kommt man zum Schluss:

Bewegte Maßstäbe  $L'$  werden kürzer (gemessen):  $L' = \frac{L_0}{\gamma}$       $L_0$  ist Länge im Ruhesystem

#### Längenkontraktion im Minkowskidiagramm

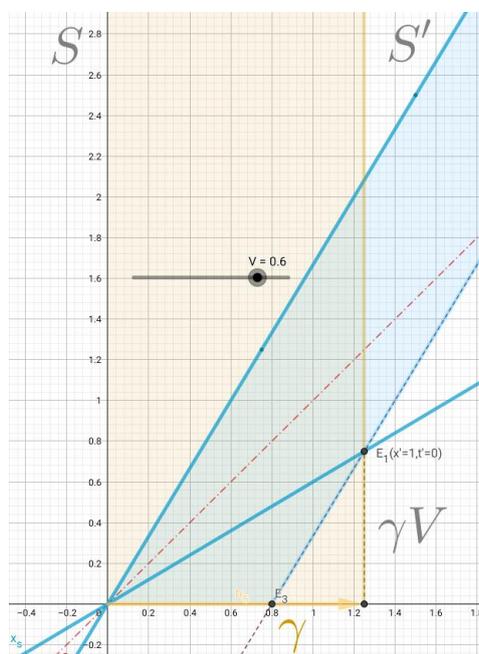


Abb.233 : Längenkontraktion

In nebenstehender Abbildung sind die Weltflächen(=Weltlinien aller Massenpunkte) zweier "Maßstäbe" eingezeichnet - einer ruht in  $S$  (gelb), der andere in  $S'$  (blau).

Um die Länge eines bewegten Maßstabs festzulegen, muß man **gleichzeitig** Anfangs- und Endkoordinate bestimmen:

Der blaue Maßstab hat in seinem Ruhesystem  $S'$  die Länge 1 (Ereignisse  $O(x' = 0)$ ,  $E_1(x' = 1)$ ,  $t' = 0$ ) Im System  $S$  (in dem er sich mit  $V$  bewegt) hat er die Länge  $1/\gamma$  (mit der Punktsteigungsform):

$$y = 0 \quad \Rightarrow \quad y - \gamma V = \frac{1}{V}(x - \gamma) \quad \Rightarrow \quad x = \frac{1}{\gamma}$$

Der gelbe Maßstab hat in seinem Ruhesystem  $S$  die Länge  $\gamma$ , im System  $S'$  (in dem er sich mit  $V$  bewegt) werden gleichzeitig seine Endpunkte in  $O$  und  $E_1$  gemessen: seine Länge 1. Also von  $\gamma$  auf 1 verkürzt.

## 22.5 Visualisierung des Zwillingsparadoxons

Zwilling  $A$  bleibt in einem Inertialsystem (seine Uhren sind stets synchron). Sein Zwilling  $B$  fliegt mit einer Rakete von  $A$  weg auf ein Ziel zu und kehrt zu  $A$  zurück.

Der Einfachheit halber verwenden wir folgenden Modell:

Die Beschleunigung von erfolge sprunghaft und zwar alle  $\Delta t = 0,4$  Jahre (von  $A$  aus gesehen) mit den Schritten (beachte den letzten Schritt:  $v = 1$  ist nicht möglich - wieso eigentlich? Wir führen später das Szenario der **gleichmäßigen Beschleunigung** in  $S$  ad absurdum!):

$$v_i = 0.1, 0.2, 0.3 \dots 0.8, 0.9, 0.95$$



Beachte: Nach jedem "Geschwindigkeitssprung" sind die Uhren von "B" nicht mehr synchron, er ist also nur zwischen den Sprüngen ein Inertialsystem - sonst nicht! Die Symmetrie  $A$  bewegt sich von  $B$  weg, weil auch  $B$  ein Inertialsystem darstellt, lässt sich nicht aufrecht erhalten - nur  $A$  ist ein Inertialsystem!

Wir stellen diese Situation in *Geogebra* dar, wobei wir berücksichtigen, dass die bewegte Uhr langsamer geht - die Zeit, in dem diese Uhr ruht (*Eigenzeit*) ist "weniger", als die Uhren in dem System anzeigen, indem sich diese bewegt:

$$\tau = \frac{1}{\gamma} t = \sqrt{1 - v^2} t \quad \tau \text{ ist Eigenzeit, } t \text{ ist Zeit im "Dauerinertialsystem"}$$

- Etwas realitätsfern gehen wir davon aus, dass jeweils nach 0,4 Jahren - von  $A$  aus gesehen - ein Geschwindigkeitssprung von  $B$  stattfindet, wir definieren  $\Delta t: 0.4$
- Wir legen die Relativ-Geschwindigkeitsliste fest:  
`v_i = {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95}`
- Die Raumzeitpunkte bei denen die Geschwindigkeitssprünge stattfinden:

$$P(x_{i+1}, t_{i+1}, i + 1) \quad \text{mit} \quad x_{i+1} = x_i + v_i \cdot \Delta t \quad t_{i+1} = t_i + \Delta t \quad i = i + 1$$

$\ell 1$ : `IterationList((x(A)+Δt*v_i(z(A)), y(A)+Δt, z(A)+1), A, {(0, 0, 1)}, 10)`

Wir benötigen nur Raum- und Zeitkoordinaten:  $\ell 2$ : `Zip((x(1), y(1)), 1, ℓ1)`

- Liste der  $\sqrt{1 - v_i^2}$  und dazugörigen Eigenzeiten von  $B$ :  
`invGamma = sqrt(1 - v_i^2) \quad \tau_i = invGamma * Δt`
- Berechnung der gesamten Eigenzeit unter der Voraussetzung, dass Bremsvorgang und Rückflug symmetrisch zur "Beschleunigungsphase" verlaufen:  
`τ = Sum(τ_i) * 4 = 12.1177` als Formel ( $v(t)$  ist eine Treppenfunktion):

$$\tau = \sum_i \sqrt{1 - v_i^2} \Delta t \cdot 4 = 4 \cdot \int_0^4 \sqrt{1 - v(t)^2} dt \quad (22.8)$$

- Die Weltlinie ergibt aus den gespiegelten Weltpunkten:  
`worldLine = Polyline(spaceTimePoints)`

## 22. Spezielle Relativitätstheorie (SRT)

Die gesamte Flugbahn ergibt sich durch Spiegelungen:

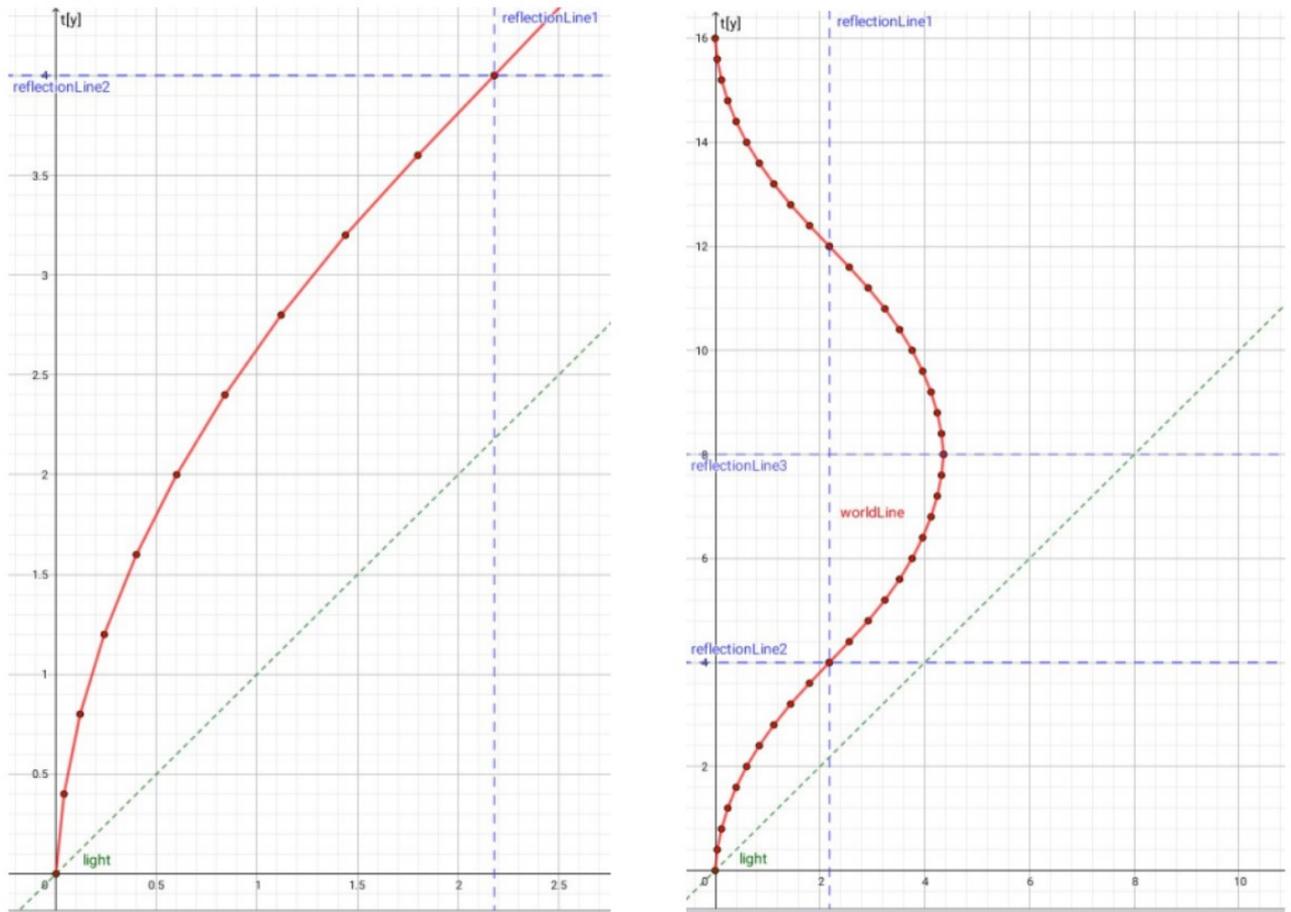


Abb.234 : Beschleunigungsteil der Weltlinie vs. gesamte Weltlinie

Die Eigenzeit von  $B$  bei einer beliebigen *stetigen* Geschwindigkeit  $v(t)$  ist nach Formel 22.8:

$$\tau_B = \int_0^{t_A} \sqrt{1 - v(t)^2} dt \quad \text{beide Uhren zu Beginn auf 0 gestellt, } t \text{ in } A \text{ gemessen} \quad (22.9)$$

- $\forall t > 0 : v(t) = 0 \Rightarrow \tau_B = t_A$   
Keine Relativgeschwindigkeit, dann bleiben die Uhren synchron
- $\exists t > 0 : v(t) \neq 0 \Rightarrow \tau_B < t_A$

Wir schauen uns das Integral 22.9 in Grafik2 des obigen Arbeitsblatts an:

- Zuerst erstellen wir die Punkte von 22.9 in der Beschleunigungsphase  
`(5=Join({(0, 1)}, Sequence((i * Δt, invGamma(i)), i, 1, Length(v_i))))`

- Wir spiegeln um den letzten Punkt ("Bremsphase") - diese "Fleißaufgabe" müssen wir machen, damit die spätere Splinefunktion auch symmetrisch wird:

```
reflectionLine4: x = x(Element(15, Length(15)))
```

```
ℓ5' : Reflect(15, reflectionLine4)
```

```
invGammaPoints = Sort(Unique(Join({ℓ5, ℓ5'})))
```

*Unique* - um doppelten Punkt (Reflectionpoint) zu eliminieren

*Sort* - weil Spline-Tool Punkte mit zunehmenden  $x$ -Wert verlangt

- Wir benutzen aus Kapitel 16 das Werkzeug  $csplineLU(<List>)$  um eine stetige Geschwindigkeit zu konstruieren:

```
csplineLU(invGammaPoints) → v(x)
```

- Berechnet man von  $v(x)$  die 20-stellige Untersumme (mal 2) ergibt das dann wieder die "unstetige" Eigenzeit von  $B$  für die Reise (oben  $\tau$  genannt):

```
 $\tau_{low} = LowerSum(v, 0.000001, 8, 20)*2$
```

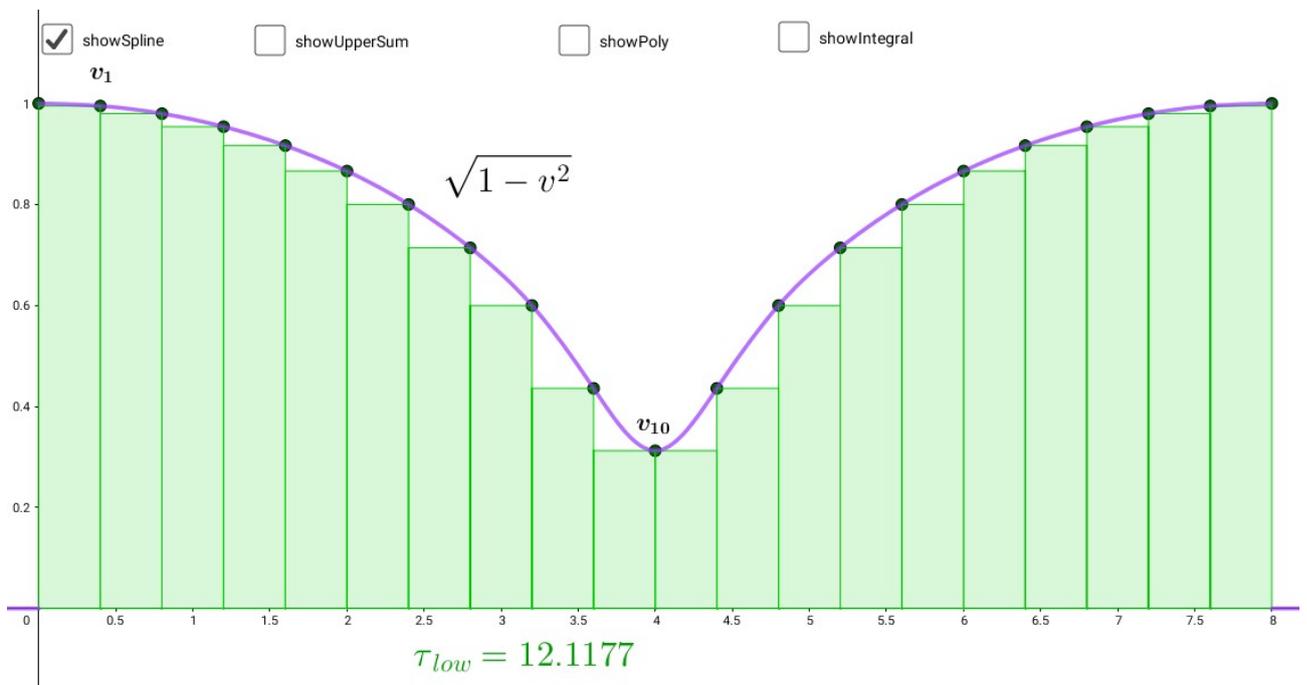


Abb.235 : stetige Geschwindigkeit vs. Treppenfunktion

Der Vorteil der Splinefunktion für  $v$ , dass sie nicht nur stetig ist, sondern stetig differenzierbar, d.h. auch die Beschleunigung ist noch stetig (macht also keine Sprünge). Man kann sich allerdings einen Antrieb vorstellen, der nur aus- bzw. eingeschaltet wird. Jedenfalls ist jetzt die Eigenzeit durch ein Integral berechenbar.

Die Näherungen (Untersumme, Obersumme, Trapeznäherung) und das Integral werden durch Checkboxes angezeigt.

## 22. Spezielle Relativitätstheorie (SRT)

Wie sieht diese Reise in  $S'$  für  $B$  aus - dessen Uhren gehen ja anders!

Wir erkunden dies mit einigen *GNU-Octave*-scripts:

```
1 # we must not start with "function" - so we define
2 # speed-vector and time-intervall in frame S
3 v=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95];
4 v=[v, flip(v)];
5 # ΔT=0.4 →u0394Δ(); Parser does NOT accept Greek characters, so we write
 DeltaT=0.4;
6
7 # input: [1,2], [3,4] → output: [1,3,2,4]
8 # 1 2
9 # | | (:) → [1;3;2;4] 1-D indexing result in column-vector
10 # 3 4 transposition "'" yields the result
11 ## btw: utf8-code for floor and ceiling is u2308-u230b and you can
12 ## write it in vim by Ctrl+v and afterwards type u2308
13 ## you can write such character in latex-listings-package with "literate"
14 function result=interweave(ar1, ar2=ar1)
15 result=[ar1;ar2](:)';
16 endfunction
17
18 function fh=linFunc(k,d,xvec)
19 fh=@(xvec) k*xvec+d; #we return a function handle with argument 'xvec' only
20 endfunction
21
22 # we construct the "stair"
23 x=interweave(0:0.4:8); v=[0,interweave(v),0];
24 myColors=colormap(ocean(100)); # 10 types of blue
25 plot(x,v,'b-'); area(x,v,"FaceColor",myColors(95,:));
26 ylabel("velocity"); xlabel("time"); hold on;
27
28 #now we plot the linear approximation to the step-function (transpose is needed)
29 f_1=linFunc(1/4,0.05); f_2=linFunc(1/8,0.475);
30 x11=linspace(-0.2,3.4,2); y11=f_1(x11); x=[x11',(8-x11)'];
31 plot(x,y11,'r-',"linewidth",2.5,[3.8,4.2],[0.95,0.95],'r-',"linewidth",2.5);
32 hold on;
33
34 x12=linspace(3.4,3.8,2); y12=f_2(x12); x=[x12',(8-x12)'];
35 plot(x,y12,'r-',"linewidth",2.5); hold on;
36
37 # translation-vector; lower 2 triangles
38 moveXY=[0.4,0.1]; x=[-0.2 0 0; 8.2 8 8]; y=[0 0 0.05; 0 0 0.05];
39 for i=1:8
40 # we move triangle upward and reflect it afterwards at x=4
41 x=[x; x(end-1,:)+moveXY(1)]; y=[y; y(end-1,:)+moveXY(2)];
42 xh=x(end,:); x=[x; [8-xh(1),8-xh(2),8-xh(3)]];
43 y=[y;y(end,:)]; # y-values do not change
44 endfor
45
46 #we add the 2 upper triangles
47 x=[x; [3.4 3.6 3.6; 8-3.4 8-3.6 8-3.6]];
48 y=[y; [f_2(3.4), f_2(3.4), f_2(3.6)]]; y=[y;y(end,:)];
49 fill(x',y',"r"); set(gca,'xtick',-0.4:0.4:8.4,'ytick',0:0.1:1);
50 axis([-0.4 8.4 0 1]); title("Velocity Diagram");
```

Zuerst “drehen” wir etwas am Geschwindigkeits-Diagramm:

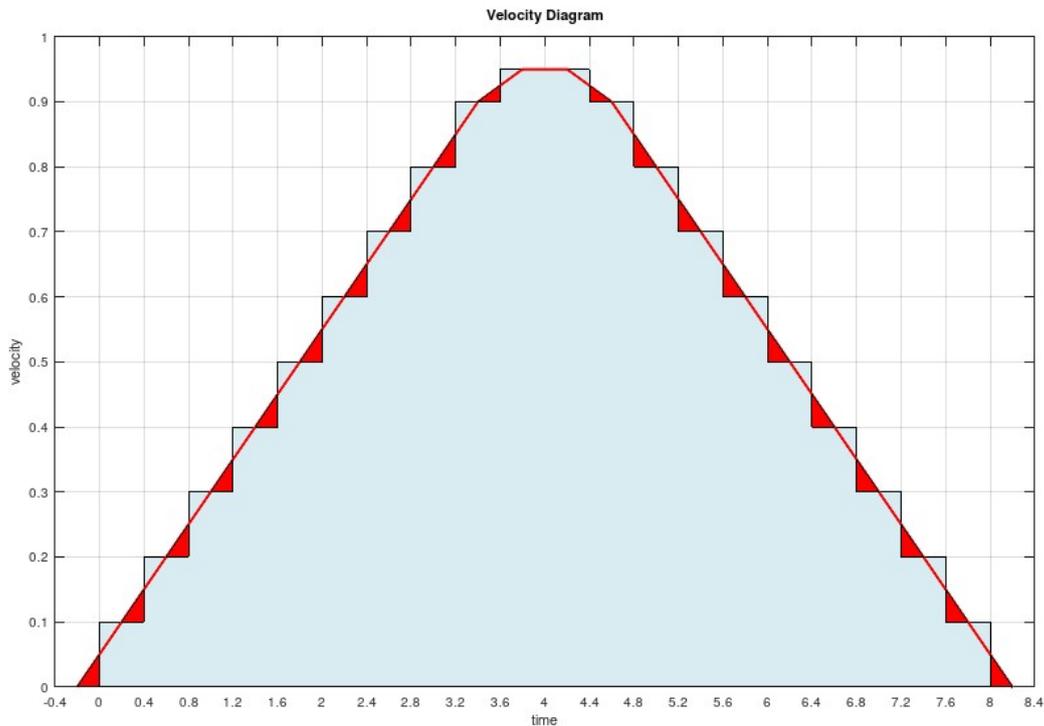


Abb.236 : Approximation der Treppenfunktion

Wir führen für unsere Treppenfunktion von  $v_s(t)$  (*step*) eine lineare Näherung ein (rot) - im vorhergehenden Listing in Zeile 29 (`f_1`, `f_2`) bzw. 30 (Intervallgrenzen) ersichtlich.

$$v_s(t) \approx v_\ell(t) = \begin{cases} \frac{1}{4}x + 0.05 & \text{falls } -0.2 \leq x \leq 3.4 \\ \frac{1}{8}x + 0.475 & \text{falls } 3.4 \leq x \leq 3.8 \\ 0.95 & \text{falls } 3.8 \leq x \leq 4 \end{cases} \quad (22.10)$$

Die Geschwindigkeit des Raumschiffs ist jetzt stetig - das ist realitätsnäher. Da das Integral über die Geschwindigkeit den zurückgelegte Weg ergibt, ergeben sich gleiche Wegstrecken - allerdings beginnt die Reise etwas früher und endet etwas später. Außerdem vermeiden wir die “unendlichen” Beschleunigungen an den Sprungstellen. Der zurückgelegte Weg  $s$  ist daher:

$$s = 2 \sum_{i=1}^{10} v_i = 2 \int_{-0.2}^4 v_\ell(t) dt$$

```

v=[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95];
s1=2*0.4*sum(v);
printf("Weg mit Summe berechnet: %d Lichtjahre \n",s1);
s2=2*(quad(f_1, -0.2,3.4)+ quad(f_2,3.4,3.8))+0.4*0.95;
printf("Weg mit Integral berechnet: %d Lichtjahre \n",s2);

```

4.36 Lichtjahre in beiden Fällen

## 22.6 ANHANG1: Algebraische Eigenschaften von $\Lambda$

Die 2 dimensionale Lorentztransformation lautet:

$$\begin{pmatrix} x \\ t \end{pmatrix} = \Lambda \begin{pmatrix} x' \\ t' \end{pmatrix} = \gamma \begin{pmatrix} 1 & V \\ V & 1 \end{pmatrix} \begin{pmatrix} x' \\ t' \end{pmatrix} \quad \text{also} \quad \Lambda = \gamma \begin{pmatrix} 1 & V \\ V & 1 \end{pmatrix}$$

$\Lambda$  ist symmetrisch und daher diagonalisierbar. Wir suchen eine orthogonale Basis mit den Eigenvektoren. Aber zuerst die Eigenwerte  $\lambda_1, \lambda_2$ :

$$\Lambda \vec{x} = \lambda \vec{x} \Rightarrow |\Lambda - \lambda I| = 0 \Rightarrow \lambda_1, \lambda_2 = \gamma(1 \pm V)$$

also

$$\lambda_1 = \frac{\sqrt{1+V}}{\sqrt{1-V}} \quad \lambda_2 = \frac{\sqrt{1-V}}{\sqrt{1+V}} \Rightarrow \boxed{\lambda_1 \cdot \lambda_2 = 1} \quad (22.11)$$

Jetzt die Suche nach den Eigenvektoren:

$$(\Lambda - \lambda_i I)\vec{w}_i = \vec{0} \Rightarrow \vec{w}_1 = 1/\gamma \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad \text{bzw.} \quad \vec{w}_2 = 1/\gamma \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

Die Länge der Eigenvektoren spielt keine Rolle, also bilden die Einheitsvektoren

$$B = \{\vec{v}_1, \vec{v}_2\} = \left\{ \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix} \right\} \quad \text{eine Basis die } \Lambda \text{ diagonalisiert}$$

Sei

$$\vec{x} = \begin{pmatrix} x \\ t \end{pmatrix}, \vec{x}' = \begin{pmatrix} x' \\ t' \end{pmatrix}, C := \begin{pmatrix} \vdots & \vdots \\ \vec{v}_1 & \vec{v}_2 \\ \vdots & \vdots \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \Rightarrow C^2 = I \Rightarrow \boxed{C^{-1} = C}$$

Sei  $[\vec{x}]_B$  die Basisdarstellung des geometrischen Objekts  $\vec{x}$  zur Basis  $B$ , dann gilt nach 5.2 für die Transformation zwischen Standardbasis und  $B$ :  $\vec{x} = C [\vec{x}]_B$

Wir formen die Lorentztransformation um auf Basisdarstellung von  $B$

$$\vec{x} = \Lambda \vec{x}' \Rightarrow C [\vec{x}]_B = \Lambda C [\vec{x}']_B \Big| C^{-1}. \quad \Rightarrow \quad \boxed{[\vec{x}]_B = C^{-1} \Lambda C [\vec{x}']_B} \quad \text{wobei gilt}$$

$$C^{-1} \Lambda C = \frac{\gamma}{2} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & V \\ V & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}$$

also lautet die Lorentztransformation in der Basisdarstellung  $B$ :

$$[\vec{x}]_B = \begin{pmatrix} \frac{\sqrt{1+V}}{\sqrt{1-V}} & 0 \\ 0 & \frac{\sqrt{1-V}}{\sqrt{1+V}} \end{pmatrix} [\vec{x}']_B \quad (22.12)$$

Ersetzt man in 22.12  $[\vec{x}]_B$  durch  $C^{-1}\vec{x}$  ergibt sich

$$\begin{aligned} I \quad x + t &= \lambda_1 (x' + t') \\ II \quad x - t &= \lambda_2 (x' - t') \end{aligned} \tag{22.13}$$

Multipliziert man  $I$  mit  $II$  ergibt sich mit 22.11

$$x^2 - t^2 = x'^2 - t'^2$$

Mit der Synchronisationsbedingung  $|x| = t$  folgt, dass obiger Term in beiden Systemen verschwindet!

**Alladi Ramakrishnan** kommt in seinem Beitrag: "Further Unnoticed Symmetries in Special Relativity" (1983) im JOURNAL OF MATHEMATICAL ANALYSIS AND APPLICATIONS auf anderem Weg zu 22.13:

Dieser Weg sei hier kurz skizziert:

Sei  $\mathcal{O}$  ein Beobachter, der in seinem Bezugssystem  $S$  ruht. Bei 2 Ereignissen  $\mathcal{A} = (x_A, t_A)$  bzw.  $\mathcal{B} = (x_B, t_B)$  mit positiver  $x$ -Koordinate gehen Lichtblitze aus - mit welcher Zeitdifferenz erreichen sie  $\mathcal{O}$ ?

Die räumliche Trennung der beiden Ereignisse sei  $\Delta x$ , die zeitliche  $\Delta t$ .

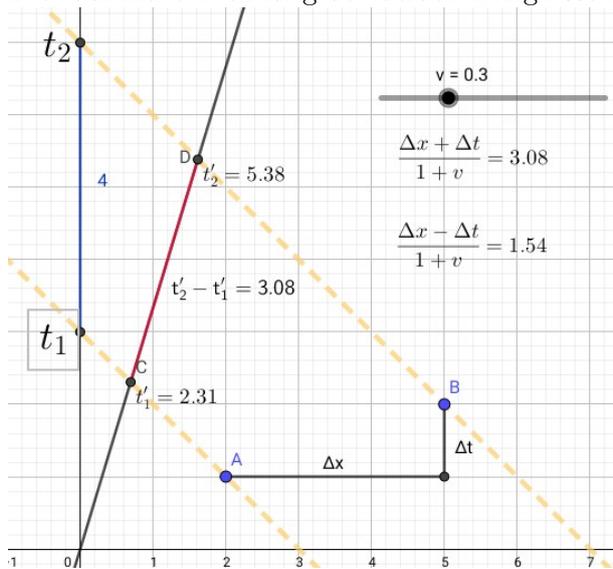
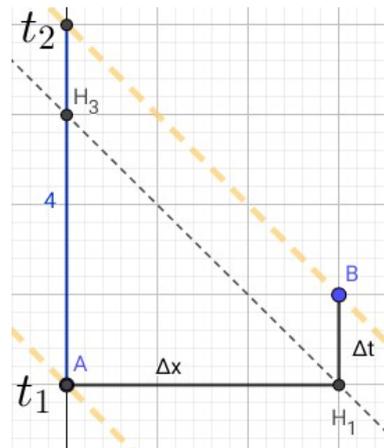


Abb.237 : Lichtblitze von A bzw. B



Mit  $c = 1$  ist  $\triangle AH_1H_3$  gleichschenkelig, also  $t_2 - t_1 = \Delta t + \Delta x$

$\mathcal{O}'$  sei ein Beobachter, der sich gegenüber  $\mathcal{O}$  mit  $v$  bewegt. Welchen Zeiterschied berechnet  $\mathcal{O}$  für  $\mathcal{O}'$  für die eintreffenden Lichtblitze?

Weltlinie von  $\mathcal{O}'$  im Ruhesystem von  $\mathcal{O}$  :  $t = (1/v)x$  (siehe *wxMaxima* im Anschluss)

$$\begin{aligned} t - t_A &= -(x - x_A) && \text{Lichtblitz von } \mathcal{A} \\ t - t_B &= -(x - x_B) && \text{Lichtblitz von } \mathcal{B} \end{aligned}$$

Wann treffen sie auf  $tv = x$ ?

## 22. Spezielle Relativitätstheorie (SRT)

Die Gleichung von  $\mathcal{O}'$  in Variablen  $t_1$  und  $t_2$  und die beiden Lichtblitze

```
(% i1) (eq1:x_1=v*(t_1), eq2:x_2=v*(t_2),globalsolve:true)$
```

```
(% i2) (eq3: -(x_1-x_A)=t_1-t_A, eq4: -(x_2-x_B)=t_2-t_B)$
```

Lösen die beiden linearen Gleichungssysteme für  $t_1, x_1, t_2$  und  $x_2$

```
(% i3) (linsolve([eq1,eq3],[t_1,x_1]), linsolve([eq2,eq4],[t_2,x_2]),s:ratsimp(t_2-t_1))$
```

Ersetzen die Differenzen durch neue Symbole

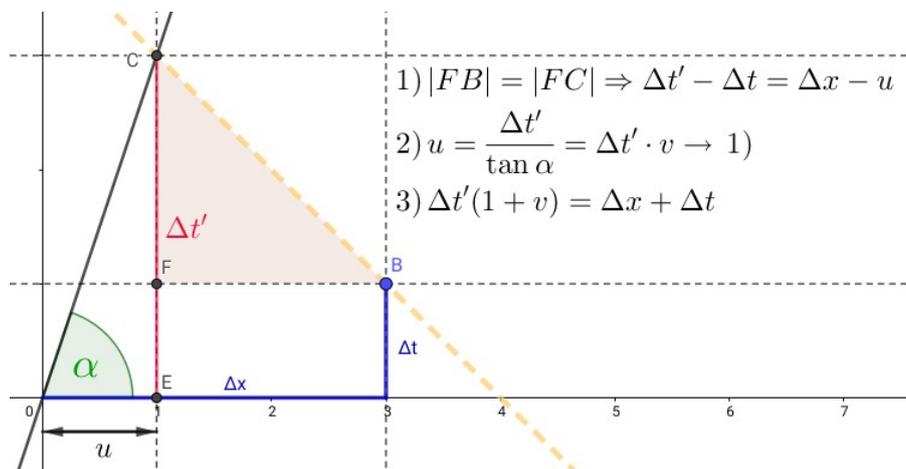
```
(% i6) s2:ratsubst(Δt,t_B-t_A, ratsubst(Δx, x_B-x_A,s));
```

$$\frac{x+t}{v+1}$$

```
(kill(t_2,t_1),print(t_2,"'-",t_1,"'=",s2))$
```

$$t_2' - t_1' = \frac{x+t}{v+1} \quad (22.14)$$

Wir kontrollieren das Ergebnis 22.14 mit geometrischen Überlegungen:



- 1)  $|FB| = |FC| \Rightarrow \Delta t' - \Delta t = \Delta x - u$
- 2)  $u = \frac{\Delta t'}{\tan \alpha} = \Delta t' \cdot v \rightarrow 1)$
- 3)  $\Delta t'(1 + v) = \Delta x + \Delta t$

Abb.238 :  $\tan \alpha = 1/v$  und Lichtblitz  $BC$  hat Steigung -1



$\Delta t'$  ist die Ordinate von  $C$  nicht dessen Abstand vom Ursprung

$\mathcal{O}'$  misst für obige Zeitdifferenz 22.14 diesen Wert allerdings multipliziert mit Bondi's k Faktor (siehe Wikipedia ), also

$$t' + x' = k[(t + x)/(1 + v)] \quad (\text{wir lassen das Präfix "Δ" jetzt weg}) \quad (22.15)$$

Wir vertauschen die Rollen von  $\mathcal{O}$  und  $\mathcal{O}'$ , dadurch wird  $v$  durch  $-v$  ersetzt und die gestrichenen und ungestrichenen Größen vertauschen:

$$t + x = k[(t' + x')/(1 - v)] \quad (22.16)$$

Multiplikation von 22.15 mit 22.16 ergibt  $k^2 = 1 - v^2$ . Damit wird 22.16 zu 22.13-I. Sind  $\mathcal{A}$  und  $\mathcal{B}$  "links" von  $\mathcal{O}$  ergibt sich II

## 22.7 ANHANG3: Vektoren

- Was macht ein geometrisches Objekt aus? (was ist unabhängig von der Wahl des KS?)  
z.B.: Würfel mit Seitenlänge 2

- in einem Koordinatensystem mit Achsen parallel zu Würfelkanten
- Rotation um  $R_x(30^\circ)$  und Translation um 1 in pos. x-Achse wobei

$$(x, y, z) \cdot R = (x', y', z') \quad \text{und}$$

$$R_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & \sin \theta \\ 0 & -\sin \theta & \cos \theta \end{pmatrix} \quad R_y = \begin{pmatrix} \cos \theta & 0 & -\sin \theta \\ 0 & 1 & 0 \\ \sin \theta & 0 & \cos \theta \end{pmatrix} \quad R_z = \begin{pmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Man kann zeigen, dass sich aus diesen 3 Rotationen jede beliebige zusammensetzen lässt! (Subscript ist Rotationsachse!)

Übrigens lassen sich obige Rotationsmatrizen auf 2 Weisen deuten:

- als Rotation eines Punktes P um Winkel  $\theta \rightarrow P'$
- als die Koordinaten, die sich für P ergeben, wenn man K um den Winkel  $-\theta$  rotiert. Wenn man K im positiven Sinn dreht, muss man also oben  $-\theta$  einsetzen! (wird weiter unten gebraucht!)

- die Koordinaten machen den Würfel nicht aus (sie sind für jeden Beobachter – je nach gewähltem Koordinatensystem anders), sondern die **Beziehungen zwischen den Koordinatenpunkten**. (Welche sind das für einen Würfel?)

1. **Vektoren** sind Zahlentupel, die beim Übergang von einem Koordinatensystem K in ein KS K' ( das zu K verschoben und/oder rotiert ist), auf die **gleiche** Weise transformieren, sodass die Beziehungen zwischen den Vektoren **erhalten** bleiben.

Bei unserem Beispiel:

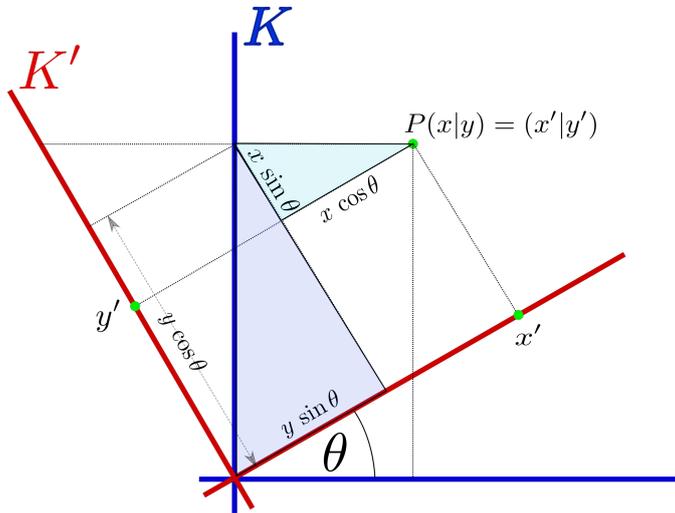
- (a) Länge der Kanten AB bzw. A'B'
  - (b) Parallelität der Kanten AB und DC (nicht die Vektoren selbst bleiben unverändert, sondern ihre Beziehung (Parallelität))
2. Was in der Geometrie die geometrischen Objekte sind, sind in der Physik die **physikalischen Gesetze**. Wie kann man also physikal. Gesetze formulieren, damit sie in allen(?wie darf ihre Beziehung sein, können sie beschleunigt zueinander sein?) Bezugssystemen gelten? S. 160 Feynman Lectures (Bd. I)

Wir haben in der Mechanik für die Bewegung eines Teilchens einen Satz von 3 Gleichungen kennengelernt:

$$F_x = m \frac{d^2x}{dt^2} \quad F_y = m \frac{d^2y}{dt^2} \quad F_z = m \frac{d^2z}{dt^2}$$

## 22. Spezielle Relativitätstheorie (SRT)

Was passiert nun mit diesen 3 Gleichungen, wenn man vom KS  $K(x,y,z)$  zu einem KS  $K'(x',y',z')$  wechselt, welches gegenüber  $K$  um den Winkel  $\theta$  um die  $z$ -Achse verdreht ist? (Zeichnen Sie sich die Situation für einen Punkt  $P(x|y)$  und einen Kraftvektor  $\vec{F}$  auf! Das Ergebnis der Überlegungen:



$$\begin{aligned}x' &= x \cos \theta + y \sin \theta \\y' &= y \cos \theta - x \sin \theta \\z' &= z\end{aligned}$$

zum Punkt  $P$  denke man sich einen Kraftvektor  $\vec{F}$  mit den Koordinaten  $F_x, F_y$  und  $F_z$  in  $K$ . Seine Koordinaten in  $K'$  sind dann

$$\begin{aligned}F_{x'} &= F_x \cos \theta + F_y \sin \theta \\F_{y'} &= F_y \cos \theta - F_x \sin \theta \\F_{z'} &= F_z\end{aligned}$$

Abb.239 : Punkt-Transformation

Übrigens erhält man dieses Ergebnis auch mit der Rotationsmatrix  $R_z$  unter Beachtung der Bemerkung für Rotation der KS!

Wir überzeugen uns von der Gültigkeit von

$$F_{x'} = m \frac{d^2 x'}{dt^2} \quad F_{y'} = m \frac{d^2 y'}{dt^2} \quad F_{z'} = m \frac{d^2 z'}{dt^2}$$

indem wir die Transformationsgleichungen (1) – (6) einsetzen und differenzieren! Unser physikal. Gesetz ist also **invariant** gegenüber Rotationen ( damit ist **nicht** gemeint, dass KS  $K'$  gleichmäßig rotiert!!) um einen best. Winkel!

### 3. Was sind nun Vektoren in der Physik?

Nicht nur die Newton'schen Gesetze, sondern auch viele andere physikal. Gesetze besitzen die Eigenschaft, die wir **Invarianz oder Symmetrie** unter Translation und Drehung der Achsen nennen.

Transformieren 3 skalare Größen genauso wie  $(x, y, z) = \vec{r}$ , dann bilden sie einen Vektor!

Was bringt das nun? Nehmen wir einmal an in einem best. KS gilt:  $\vec{F} = \vec{r}$  **und** wir können zeigen, dass die 3 Zahlen  $F_x, F_y$  und  $F_z$  einen Vektor bilden (wie in der Schreibweise schon angedeutet), dann gilt diese Gleichung in **jedem** KS.

Aber es kommt noch besser:

Meistens müssen wir nicht einmal zeigen, wie sich die 3 Zahlen transformieren, denn die Eigenschaft „Vektor sein“ bleibt nämlich unter gewissen mathematischen Operationen erhalten:

Addition, Subtraktion, Differenzieren(?wonach?), Multiplikation mit einem Skalar, usw. (dies ist Gegenstand der Vektoranalysis und -algebra)

Mit diesem „Hammer“ ausgerüstet ist jetzt klar, dass

$$\vec{F} = m \frac{d^2 \vec{x}}{dt^2}$$

ein vektorielles Gesetz sein muss und daher in allen KS gilt! Toll oder?