5. Implementation der Algorithmen

5.1. Überblick

Alle Herleitungen, die in dieser Arbeit gemacht wurden, erfordern mit steigender Ordnung in z und e einen exponentiell wachsenden Rechenaufwand. Der Einsatz von Computeralgebrasystemen ist gefragt. Grundsätzlich eignet sich dafür jede symbolische Programmiersprache, die listenorientiert arbeitet. Die Idee der objektorientierten und symbolischen Programmiersprache ist keineswegs etwas Neues, so hat die Mutter aller Programmiersprachen LISP bereits alle Fähigkeiten, die hierfür ausreichen: Listen, Funktionen und Mustererkennung. Jede algebraische Umformung und mathematische Funktion kann dann im Prinzip als eine Transformation der Struktur einer Liste auf eine andere implementiert werden. Viele grundlegende Algorithmen, wie das Differenzieren und Integrieren eines Ausdrucks, stellt bereits das Package REDUCE bereit. Darauf aufbauend kann in direkter Weise ein Algorithmus implementiert werden, der Ausdrücke in Taylorreihen entwickelt. Ebenso sollte die Fertigkeit Gleichungssysteme zu lösen vorhanden sein, sie wird in den Herleitungen unzählige Male dazu verwendet, unbestimmte Größen, die aus den Ansätzen her rühren, zu bestimmen. Es werden nur grundlegende mathematische Funktionen angewandt, wobei oft die Beziehungen der trigonometrischen Äquivalenzen verwendet werden. Um die auftretenden Differentialgleichungen zu lösen, wurden explizite Algorithmen entwickelt. Es ist daher nicht notwendig, vorgefertigte Integratoren hierfür zu verwenden. Alle Berechnungen mit Zahlen werden durch exakte Ganzzahlarithmetik durchgeführt. Dies ist im Prinzip nicht notwendig, ermöglicht aber eine bessere Kontrolle des numerischen Fehlers, der durch Division und Wurzelziehen von sehr kleinen und großen Zahlen zu Stande kommt.

Der Markt an Software, die symbolische Programmierung implementiert, ist groß: So Derive (Texas Instruments), Waterloo Maple (Maple Soft) oder Reduce (Lisp). Die für diese Arbeit verwendete Software ist *Mathematica* von Wolfram Research. Es handelt sich hierbei um ein sehr vielseitiges Softwareprodukt, das dennoch einheitlich in Form und Sprache bleibt und grundlegenden Prinzipien folgt. Für jedes mathematische, physikalische oder technische Problem findet sich, wenn nicht in den tausenden Standardfunktionen, so in einem der vielen Packages, ein oder mehrere Lösungsvorschläge. Ist kein entsprechender Algorithmus für die Lösung eines Problems vorhanden, so kann dieser in sehr direkter Art implementiert werden.

Das Problem, das bei einer solch großen Allzwecksoftware unweigerlich auftritt, ist, dass man die Übersicht über die Ergebnisse leicht verlieren kann. Vor allem dann, wenn Algorithmen verwendet werden, die man nicht kennt und deren Ergebnisse nicht nachvollziehbar sind. So ist es heutzutage kein Problem mehr, auf einem GHz Prozessor mit entsprechender Software Integrale oder Differentialgleichungen von fast beliebiger Gestalt zu lösen. Ein anderes Problem schleicht sich ein: Was soll ich mit einem 10 Seiten - Ausdruck anfangen, der unleserlich, unstrukturiert und oft nicht reproduzierbar ist? Um dies zu verhindern ist es

notwendig, die Ergebnisse auf ihre Nutzbarkeit für weitere Überlegungen immer wieder zu überprüfen. Die Bewältigung eines Problems verlagert sich ein wenig mehr auf die Seite, Ergebnisse zu prüfen anstatt sie zu erzeugen. Dies geschieht ebenfalls aufgrund der Menge an Information, die verarbeitet werden soll, oft automatisiert. Es muss daher nicht mehr die Aufgabe sein, alle Rechnungen per Hand durchzuführen, vielmehr ihren Verlauf zu organisieren und zu automatisieren. So wurden die Herleitungen dieser Arbeit nach wie vor für kleine Ordnungen mit der Hand durchgeführt, um den dahinter stehenden Algorithmus zu begreifen. Im nächsten Schritt wurde dieser dann in eine geeignete Form gebracht, um ihn in die Programmiersprache zu übersetzen. Hierbei wurde auf zwei Merkmale großes Augenmerk gelegt: Geschwindigkeit aber auch Lesbarkeit. Das zweite auf Kosten des ersteren vollkommen aufzugeben, wäre nicht sinnvoll gewesen. Es handelt sich daher nicht um die schnellsten Implementationen, jedoch sicher um lesbare und für den Leser reproduzierbare.

5.2. Beschreibung der Algorithmen - Lineare Analyse

Hier werden die Funktionen, die anhand der Algorithmen aus dem 2. Kapitel entstanden sind, definiert, kurz beschrieben und auf ihre Verwendbarkeit analysiert.

■ 5.2.1. PrimariesDistance

PrimariesDistance[n] entwickelt die Lösung des Keplerproblems $r'' = \frac{1-e^2}{16\,r^3} - \frac{1}{8\,r^2}$ mit $r(t) = \frac{1}{2}\,(1 - e\cos(u(t)))$ gegeben in seiner transzendenten Form $u(t) = t + e\sin(u(t))$ nach dem Parameter e bis zur n-ten Ordnung.

■ Mathematische Beschreibung

Da es nicht möglich ist, die Keplergleichung u(t) in geschlossener Form anzugeben, muss diese approximiert werden. Die einfachste Art, diese numerisch zu lösen, ist durch einfache Iteration. Um einen analytischen Ausdruck zu bekommen, wird diese n-fach iteriert und dann nach dem Parameter e bis zur Ordnung n entwickelt. Dies geschieht durch einfache Taylorreihenentwicklung. Das Problem reduziert sich im Wesentlichen darauf, die i-ten Ableitungen der iterierten Form zu bilden, sowie die hierbei auftretenden trigonometrischen Ausdrücke wiederholt durch Reduktion bezüglich ihrer Argumente zu vereinfachen.

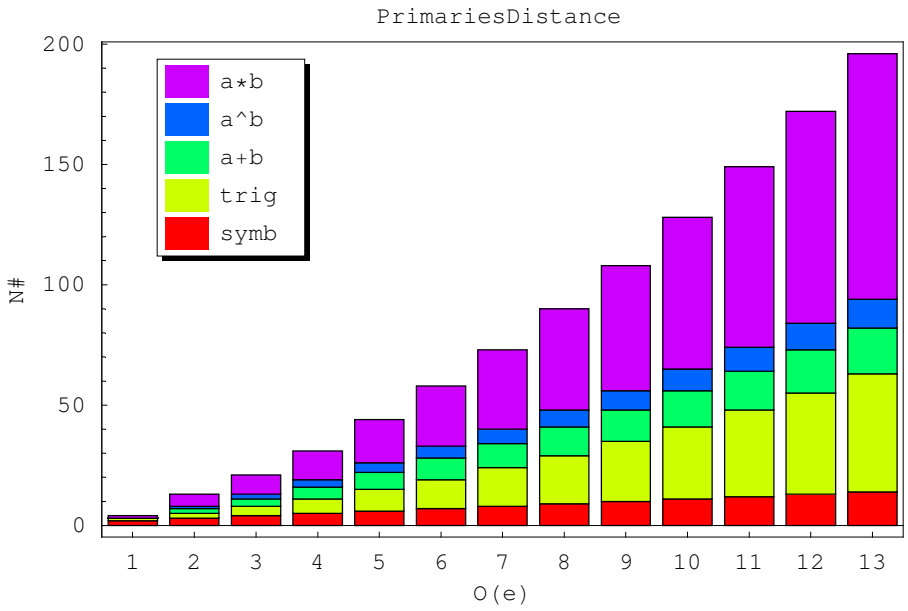
■ Implementation in Mathematica

Voraussetzung : *KeplerSolution*

```
PrimariesDistance[n_Integer] := Module[{x}, $Derivation =
    {x = Normal[KeplerSolution[n]],
     x = Series[x, {e, 0, n}],
     x = TrigReduce[x],
     PrimariesDistance[n] = (1 + x)/2}][[-1]]
```

- **Analyse**

  Der Algorithmus arbeitet gut und sehr schnell bei kleinen Ordnungen und liefert seine Ergebnisse in einer sehr kompakten Form.



Grafik 40: Überblick über die Anzahl der notwendigen Operationen, die PrimariesDistance in Abhängigkeit der Ordnung erzeugt.

Die Anzahl der Terme, die während den Ableitungen entstehen, wächst nicht linear mit der Ordnung. Geht man zu Ordnungen jenseits der 10., so muss sehr viel Zeit dafür aufgewendet werden, die einzelnen Terme zu vereinfachen, was sich direkt in der Berechnungszeit niederschlägt. Dies führt dazu, dass die verwendete Prozessorzeit exponentiell mit wachsender Ordnung ansteigt. Dem Algorithmus ist daher eine obere Schranke aufgrund dieser Tatsache gesetzt.

- **Vorkommen**

  Kap. 1.3. , 2.1.

5.2.2. LinearCoefficient

LinearCoefficient[n] entwickelt den linearen Koeffizienten der polynomischen Bewegungsgleichung des Sitnikovproblems nach dem Störparameter e bis zur Ordnung n.

- **Mathematische Beschreibung**

Da nur kleine Abstände des dritten Körpers von der Ebene der Primärkörper betrachtet werden, wird seine Bewegungsgleichung nach der Amplitude z entwickelt. Der lineare Koeffizient ergibt sich durch Entwicklung von $1/r(t, e)^3$ nach der Exzentrizität e, wobei r der Abstand der Primärkörper vom Baryzentrum ist.

- **Implementation in Mathematica**

Voraussetzung : *PrimariesDistance*

```
LinearCoefficient[n_Integer] := Module[{x}, $Derivation =
    {x = 1/PrimariesDistance[n]^3,
     LinearCoefficient[n] = TrigReduce[x]}][[-1]]
```

- **Analyse**

-

- **Vorkommen**

Kap. 2.1.

■ 5.2.3. AmplitudeFunctionSystem

AmplitudeFunctionSystem[n] erstellt das resultierende Gleichungssytem n-ter Ordnung für den linearisierten Fall der Bewegungsgleichung des Sitnikovproblems. Das Resultat ist ein System aus (n+1) Differentialgleichungen 2. Ordnung und wird mittels der Funktion AmplitudeFunction gelöst.

- **Mathematische Beschreibung**

Aus der Floquet - Theorie folgt, dass die Differentialgleichung für die Amplitudenfunktion w: $w'' + g(t)\,w - w^{-3} = 0$ lautet. Da in unserem Fall der linearisierten Bewegungsgleichung des Sitnikovproblems $g = g(t, e)$ ist, wird ein Störansatz der Amplitudenfunktion um den Parameter e versucht: $w = w_0 + w_1\,e + ... + O(e^n)$. Setzt man diesen in die Differentialgleichung ein, so ergibt sich durch Vergleich der Koeffizienten in e ein Gleichungssytem, bestehend aus (n+1) Differentialgleichungen. Dieses kann dann rekursiv in den Funktionen gelöst werden.

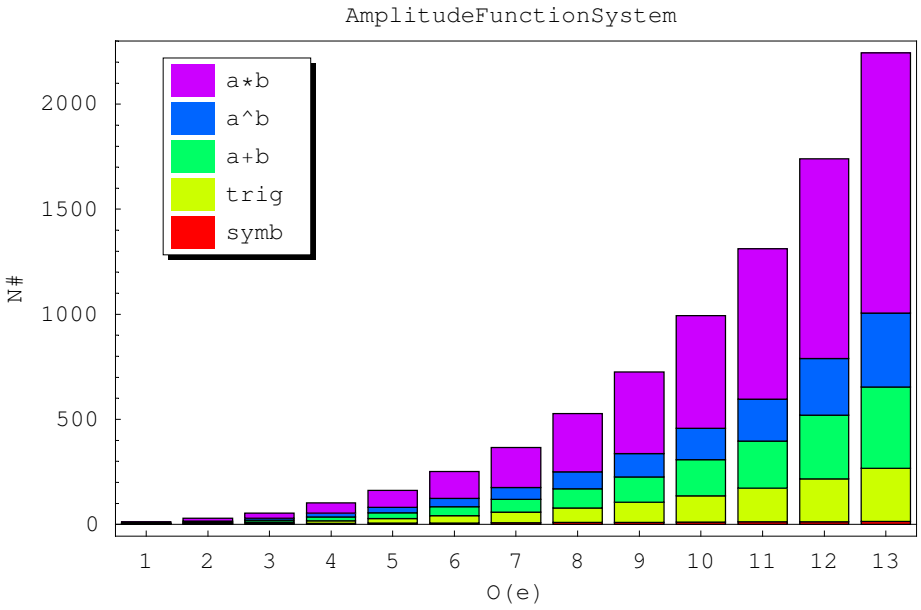
- **Implementation in Mathematica**

Voraussetzung : *LinearCoefficient*

```
AmplitudeFunctionSystem[n_Integer] :=
 Module[{x},
   $Derivation =
    {x = SeriesData[e, 0, Table[Subscript[w, i][t], {i, 0, n}],
       0, n + 1, 1],
     x = ∂{t,2} x + LinearCoefficient[n] x - 1/x^3,
     x = Simplify[x],
     AmplitudeFunctionSystem[n] = Replace[x[[3]], lhs_ → lhs == 0, 1]}
  ][[-1]]
```

■ Analyse

Die Termanzahl sowie die Menge der auszuführenden Operationen wachsen mit zunehmender Ordnung exponentiell an. Da es sich um ein rekursiv zu lösendes System handelt, ist es erst dann möglich, Vereinfachungen vorzunehmen, wenn alle Lösungen der darunter liegenden Ordnungen vorliegen. Dies stellt eine schöne Herausforderung an den Integrator des Systems dar.



Grafik 41: Übersicht über die Anzahl der Terme und Operationen, die von der Funktion AmplitudeFunctionSystem in Abhängigkeit der Ordnung in e erzeugt werden.

Da eine durchgehende Struktur im Gleichungssystem zum Zeitpunkt der Arbeit nicht bekannt war, musste der Befehl *Simplify* verwendet werden, um das Aussehen der Gleichungen zu vereinfachen. Dieser benötigt viel Zeit und Speicher, da er fast alle bekannten Umformungen des Systems ausprobieren muss, um auf eine einfache Form zu kommen.

■ **Vorkommen**

Kap. 2.3.

■ 5.2.4. AmplitudeFunction

AmplitudeFunction[n] berechnet die Amplitudenfunktion n-ter Ordnung in e durch iteratives Lösen des Differentialgleichungsystems AmplitudeFunctionSystem[n].

■ **Mathematische Beschreibung**

Das aus dem Störansatz um den Parameter e resultierende System AmplitudeFunctionSystem[n] besteht aus (n+1) Differentialgleichungen der Form $w_i'' + 8\, w_i = F_i(w_0, \ldots, w_j, t)$ mit j < i. Die allgemeine Lösung einer solchen Differentialgleichung ergibt sich zu:

$$w_i(t) = c_1 \cos\left(2\sqrt{2}\right) + c_2 \sin\left(2\sqrt{2}\, t\right) - w_{i,\text{part}}$$

Die Lösungen müssen gemäß Floquet - Theorie die 2π Periodizität erfüllen, die Konstanten c_i können daher in diesem Fall o.B.d.A. null gesetzt werden. Übrig bleibt es, die partikuläre Lösung mit 2π - Periodizität zu finden. Nun ist die einzige periodische Lösung, die die Gleichung nullter Ordnung ($w_0 + 8\, w_0 - w^{-3} = 0$) erfüllt, die konstante Funktion. Wählt man die nicht komplexe, nicht negative Lösung, so erhält man für $w_0 = 1/2^{3/4}$. Setzt man diese in die Gleichung der nächsten Ordnung ein, so erhält man wiederum eine Gleichung vom Typus, wie oben beschrieben. Gelöst werden diese durch den Algorithmus, *ParticularDSolve*, wie unten beschrieben.

■ **Implementation in Mathematica**

Voraussetzung : *AmplitudeFunctionSystem, ParticularDSolve*

```
AmplitudeFunction[n_Integer] :=
 Module[{x, Y, Z = {Subscript[w, 0][t] → 1/2^(3/4)}},
   $Derivation =
    {Y = AmplitudeFunctionSystem[n],
     Y = Y /. Join[Z, {Subscript[w, 0]''[t] → 0}],
     Do[{Y = Rest[Y], x = ParticularDSolve[Y[[1]], Subscript[w, i][t], t],
       Z = Flatten[{Z, x}],
       x = Flatten[Table[∂{t,i} x, {i, 0, 2}]], Y = Y /. x}, {i, 1, n}],
     x = SeriesData[e, 0, Table[Subscript[w, i][t], {i, 0, n}],
        0, n + 1, 1] /. Z,
     AmplitudeFunction[n] = x}][[-1]]
```

Abgesehen vom Kern des Algorithmus *ParticularDSolve*, der die auftretenden Differentialgleichungen löst, übernimmt die Funktion die Aufgaben, die gewonnenen Lösungen und deren Ableitungen in das System einzusetzen, die Differentialgleichung

bereitzustellen und aus den einzelnen Lösungen die Gesamtlösung in Form einer Taylorreihe aufzustellen.

```
ParticularDSolve[equ_, func_, var_] :=
 Module[{ansatz, x, xx, k = 0},
   $Derivation =
    {x = InhomogenPart[equ, func],
     x = TrigReduce[x],
     x = Cases[x, Sin[_] | Cos[_], ∞],
     xx = Prepend[Union[x], 1],
     ansatz = x = Plus @@ (#1 A[k++] &) /@ xx,
     x = Table[∂{var,i} (func → x), {i, 0, DifferentialOrder[equ]}],
     x = Rewrite[equ /. x],
     x = TrigReduce[x[[1]]],
     x = Collect[x, Sin[_] | Cos[_]],
     xx = Flatten[Table[Cases[{x}, A_ xx[[i]], ∞] /. xx[[i]] → 1,
        {i, 2, Length[xx]}]],
     x = x /. Table[xx[[i]] → 0, {i, 1, Length[xx]}],
     x = Thread[Prepend[xx, x] == 0],
     x = Solve[x, Table[A[n], {n, 0, k - 1}]],
     x = ansatz /. Flatten[x],
     func → x}][[-1]]
```

Die Funktion *ParticularDSolve* löst eine Differentialgleichung, indem sie einen trigonometrischen Ansatz der gleichen Form, wie die des inhomogenen Teils der Gleichung aufstellt. Hierzu werden alle trigonometrischen Funktionen der Form (cos(n t) und sin(n t)) aus dem inhomogenen Teil extrahiert und mit freien Koeffizienten versehen. Der Ansatz und seine Ableitungen werden in die Ausgangsgleichung eingesetzt und ein daraus resultierendes Gleichungssystem aufgestellt. Dieses wird gelöst, das Ergebnis hat die Form des Ansatzes. Die 2π-Periodizität ist auf jeden Fall erfüllt, da der Ausgangspunkt für den Ansatz, der inhomogene Teil der Gleichung, ebenfalls 2π periodisch war.

```
InhomogenPart[equ_, y_[t_]] :=
 Module[{x},
   $Derivation = {x = DifferentialOrder[equ],
     x = Table[∂{t,i} (y[t] → 0), {i, 0, x}], x = equ /. x, x = Rewrite[x],
     x[[1]]}][[-1]]
```

Der inhomogene Anteil der Gleichung wird durch *InhomogenPart* bestimmt:

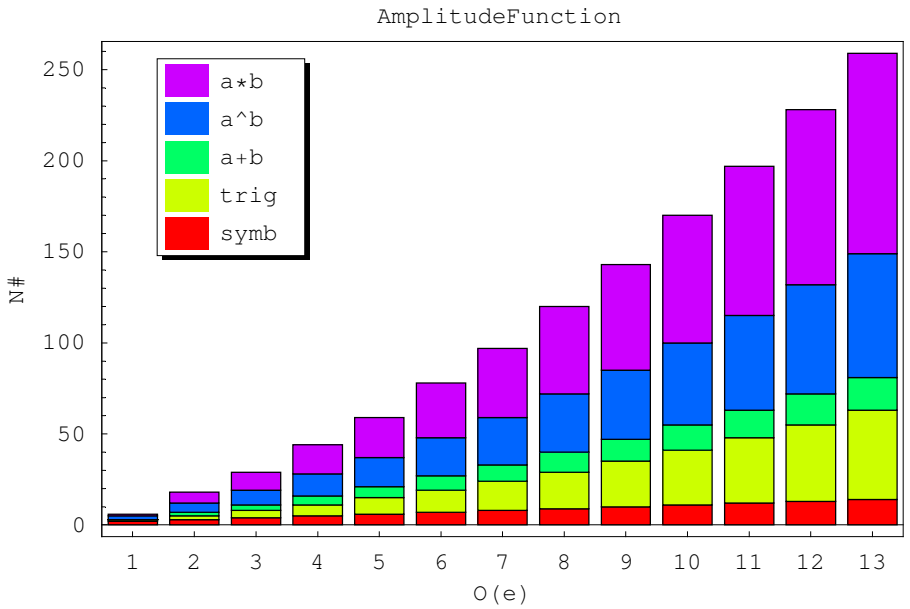
```
Rewrite[equ_Equal] := equ[[1]] - equ[[2]] == 0

DifferentialOrder[exp_] :=
 Module[{tmp = exp, i = 1},
   $Derivation =
    {While[MemberQ[tmp, _^(i)[_], ∞] || MemberQ[tmp, _^(_)[_], ∞],
      {tmp = tmp /. _^(i)[_] → NULL, i++}],
     $Result = i - 1}][[-1]]
```

DifferentialOrder bestimmt die höchste Ableitung der Gleichung. In unserem Fall ist diese zwei. Um die Funktionen jedoch auch für spätere Zwecke einsetzen zu können, wurden diese, soweit es möglich war, für einen allgemeineren Zweck geschrieben. Der Algorithmus ist somit auch bei Gleichungen mit höheren Ableitungen einsetzbar.

- **Analyse**

Der Algorithmus arbeitet sehr schnell, wenn man bedenkt, dass die Gleichungen, die gelöst werden sollen, aus einer Unzahl an Termen besteht. Zudem wächst die Anzahl der Operationen mit zunehmender Ordnung exponentiell an, was sich direkt auf die Komplexität des inhomogenen Teiles der Gleichungen auswirkt. Zudem können aufgrund des rekursiven Charakters des Systems erst während der Laufzeit der Funktion eventuelle Vereinfachungen durchgeführt werden. Diese beschränken sich jedoch zum Glück auf das Vereinfachen von trigonometrischen Termen. Mit jeder Ordnung, daher jeder weiteren Gleichung im System, gegeben durch *AmplitudeFunctionSystem,* kommen aufgrund der Produkte aus den vorherigen Lösungen neue Terme mit weiteren trigonometrischen Argumenten hinzu.

Die Anzahl der Terme und Operationen nimmt nur schwach exponentiell mit wachsender Ordnung zu. Es ist daher ohne weiteres möglich, für genauere Analysen größere Ordnungen zu verwenden.



Grafik 42: Anzahl der von der Funktion AmplitudeFunction erzeugten Terme und Operationen, in Abhängigkeit der Ordnung in e.

Die Komplexität der Aufgabe schlägt sich direkt in der Berechnungszeit nieder. Ableitungen bis zur 17. Ordnung konnten trotzdem in annehmbarer Zeit durchgeführt werden.

■ **Vorkommen**

Kap. 2.3.

■ 5.2.5. PhaseFunction

PhaseFunction[n] berechnet die Phasenfunktion des linearisierten Sitnikovproblems gemäß der Floquet Theorie. Hierzu wird das Integral über das Quadrat der Amplitudenfunktion gebildet.

■ **Mathematische Beschreibung**

Aus dem Ansatz der Floquet - Theorie für die Lösung einer linearen gewöhnlichen Differentialgleichung zweiter Ordnung mit periodischer Koeffizientenfunktion folgen zwei Bestimmungsgleichungen für die zeitabhängige Amplituden- und Phasenfunktion. Ist die Gleichung für die Amplitudenfunktion w gelöst, so ergibt sich die Phasenfunktion ψ aus der Beziehung.

$$\psi(t) = \int_0^t \frac{1}{w(s)^2}\, ds$$

Die Funktion *PhaseFunction* entwickelt den Integrand nach e und integriert ihn dann nach der Zeit. Das Ergebnis wird nach den trigonometrischen Ausdrücken sortiert.

■ **Implementation in Mathematica**

Voraussetzung : *AmplitudeFunction*

```
PhaseFunction[n_Integer] := Module[{x},
   $Derivation =
    {x = 1/AmplitudeFunction[n]^2,
     x = ∫ x dt,
     x = TrigReduce[n],
     PhaseFunction[n] = Collect[x, Sin[_] | Cos[_]]}][[-1]]
```

■ **Analyse**

Die wesentlichen Aufgaben des Algorithmus wurden bereits in der mathematischen Beschreibung aufgezeigt. Die Anforderungen an die Funktion beschränken sich auf die Entwicklung einer Störreihe, das Integrieren nach der Zeit von trigonometrischen Funktionen und das Sortieren eines Ausdrucks nach trigonometrischen Mustern.

Die Term- und Operationsanzahl entspricht im Wesentlichen der von *AmplitudeFunction*.

▪ Vorkommen

Kap. 2.4.

■ 5.2.6. LinearSolution

LinearSolution[n] gibt die Lösung der linearisierten Bewegungsgleichung des Sitnikovproblems bis zur n-ten Ordnung bezüglich des Parameters e aus.

▪ Mathematische Beschreibung

Die Lösung einer Differentialgleichung vom Hill´schen Typus lautet gemäß Floquet - Theorie.

$$z(t) = \frac{z_0}{w_0}\cos(\psi(t)) + w_0\, v_0 \sin(\psi(t))$$

Hierbei sind z_0 und v_0 die Anfangsamplitude bzw. Anfangsgeschwindigkeit und w_0 die Amplitudenfunktion zum Zeitpunkt t = 0.

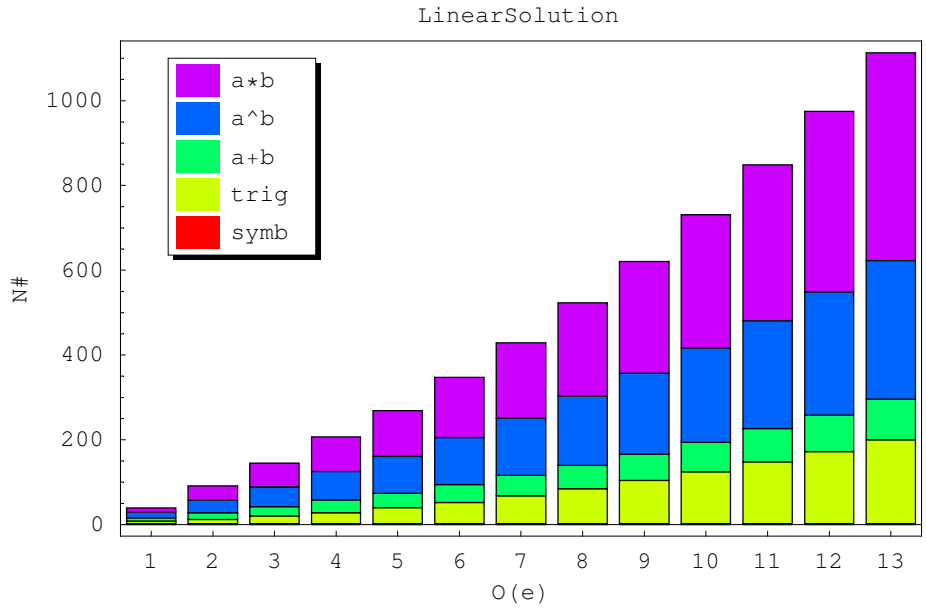
▪ Implementation in Mathematica

Voraussetzung : *AmplitudeFunction, PhaseFunction*

```
LinearSolution[n_Integer] := Module[{w, ψ}, $Derivation =
    {w[t] = Normal[AmplitudeFunction[n]],
     w[0] = Normal[AmplitudeFunction[n]] /. t → 0,
     ψ[t] = Normal[PhaseFunction[n]],
     x = Cos[ψ[t]] Subscript[z, 0] w[t] / w[0] +
       Sin[ψ[t]] w[0] w[t] Subscript[v, 0], LinearSolution[n] = x}][[-1]]
```

▪ Analyse

Die Funktion stellt nur die Ergebnisse der vorangegangenen Algorithmen zusammen. Die Anzahl der Terme und Operationen ergibt sich zu der doppelten Summe von jenen aus *AmpltiudeFunction* und *PhaseFunction.*



Grafik 43: Anzahl der durchzuführenden Operationen, die von der Näherungslösung LinearSolution in Abhängigkeit der Ordnung e erzeugt werden.

Die Anzahl der Terme der Lösung der linearisierten Bewegungsgleichung des Sitnikov Problems wächst schwach exponentiell an und erfordert bei 13. Ordnung in e über 1000 durchzuführende Operationen, um einen Lösungswert zu berechnen.

■ Vorkommen

Kap. 2.4.

■ 5.2.7. PhaseShift

PhaseShift[n] gibt die Phasenverschiebung im linearisierten Sitnikovproblem bis zur n-ten Ordnung in e wieder. Sie entspricht dem linear mit der Zeit anwachsenden Koeffizienten von PhaseFunction.

■ Mathematische Beschreibung

Die Phasenverschiebung der Lösung des linearisierten Sitnikovproblems setzt sich aus säkularen und oszillierenden Teilen zusammen: $\phi_{\text{ges}} = \phi_{\text{lin}} * t + \phi_{\text{osz}}$. Die linear mit der Zeit anwachsende Größe hängt direkt mit der Stabilität der Lösung zusammen.

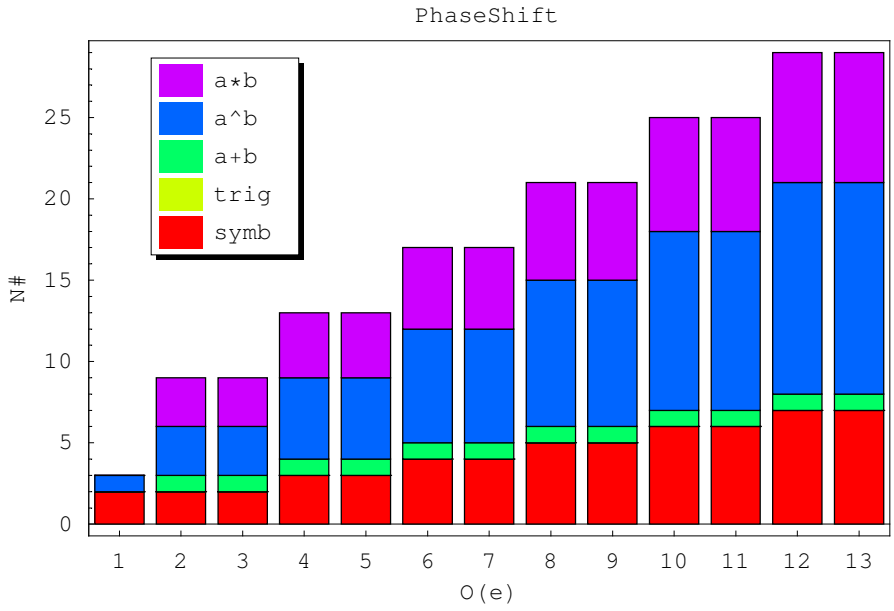
■ Implementation in Mathematica

Voraussetzung : *PhaseFunction*

```
PhaseShift[n_Integer] := Module[{x},
   $Derivation =
    {x = PhaseFunction[n],
     PhaseShift[n] = Coefficient[x, t, 1] + O[e]^(n+1)}][[-1]]
```

■ **Analyse**

Die Anzahl der Terme und Operationen wächst mit steigender Ordnung linear an. Die Ordnungen (n,n+1) entsprechen sich. (n = 2,3,4,...)



Grafik 49: Anzahl der Terme in Abhängigkeit der Ordnung, die von der Funktion PhaseShift bestimmt wird.

Auf die Bedeutung der Phasenverschiebung und ihr Zusammenhang mit der Stabilität wird in Kapitel 2 eingegangen.

■ **Vorkommen**

Kap. 2.5.

■ 5.2.8. TransferMatrix

TransferMatrix[n] leitet die Abbildungsvorschrift R in $\vec{x}_{n+1} = R^n \vec{x}_n$ über eine Umlaufperiode der Primärkörper bis zur n-ten Ordnung in e ab. R ist hierbei eine (2×2) Matrix, zeitunabhängig mit det R = 1.

■ Mathematische Beschreibung

Die Lösung einer Gleichung vom Hill´schen Typus ist z(t) = a·w(t)+cos(ψ(t)+b) oder in anderer Schreibweise $z(t) = k_1 \cdot \cos(\psi(t)) + k_2 \cdot \sin(\psi(t))$. Die Transfermatrix ergibt sich gemäß Floquet - Theorie aus der kosinus- (c) und sinusartigen (s) Lösung der Gleichung und ist durch sie und deren Ableitungen folgendermaßen definiert:

$$R = \begin{pmatrix} c(T) & s(T) \\ c'(T) & s'(T) \end{pmatrix}$$

Hier ist T = 2π die gleiche Periode wie in der Koeffizientenfunktion f(t+T) = f(t) in $z'' + f(t)\,z = 0$. Um von der allgemeinen Lösung auf die sinus- und kosinusartige Lösung zu kommen, werden die Anfangsbedingungen derart gewählt, dass c(0) = 1, $z'(0) = 0$ und s(0) = 0 sowie $s'(0) = 1$ sind. Bestimmt man mit Hilfe dieser Anfangsbedingungen die Konstanten k_1 und k_2, so folgen c(t) und s(t) direkt aus diesen.

Die Bedeutung der Matrix R ist neben der Tatsache, dass sie eine Abbildungsvorschrift im Phasenraum definiert, in ihrer Spur zu suchen. Ist diese |Sp R| < 2, so bleibt die Lösung an das System gebunden, ansonsten ungebunden. Diese Tatsache ermöglicht die Durchführung einer einfachen Stabilitätsanalyse.

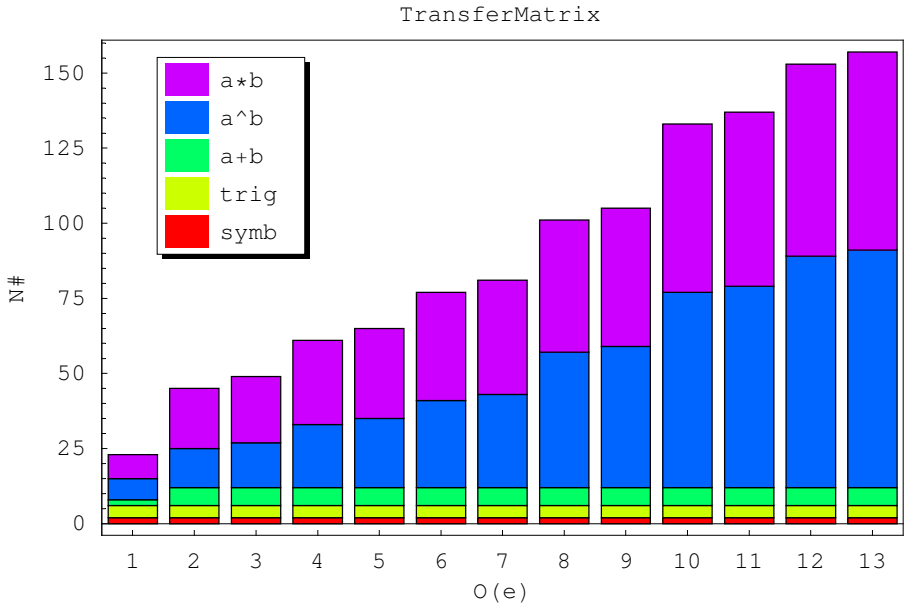
■ Implementation in Mathematica

Voraussetzung : *AmplitudeFunction, PhaseFunction*

```
TransferMatrix[n_Integer] :=
 Module[{x, z, k, c, s},
   $Derivation =
    {z[t_] = k[1] Normal[AmplitudeFunction[n]]
        Cos[Normal[PhaseFunction[n]]] +
       k[2] Normal[AmplitudeFunction[n]] Sin[Normal[PhaseFunction[n]]],
     x = {z[t] == 1 /. t → 0, ∂t z[t] == 0} /. t → 0, x = Solve[x, {k[1], k[2]}],
     c[t_] = z[t] /. x[[1]], x = {z[t] == 0 /. t → 0, ∂t z[t] == 1} /. t → 0,
     x = Solve[x, {k[1], k[2]}], s[t_] = z[t] /. x[[1]],
     x = {{c[t], s[t]}, {∂t c[t], ∂t s[t]}} /. t → 2 π,
     x = Collect[x, {Sin[_], Cos[_], e}],
     TransferMatrix[n] = Simplify[x]}][[-1]]
```

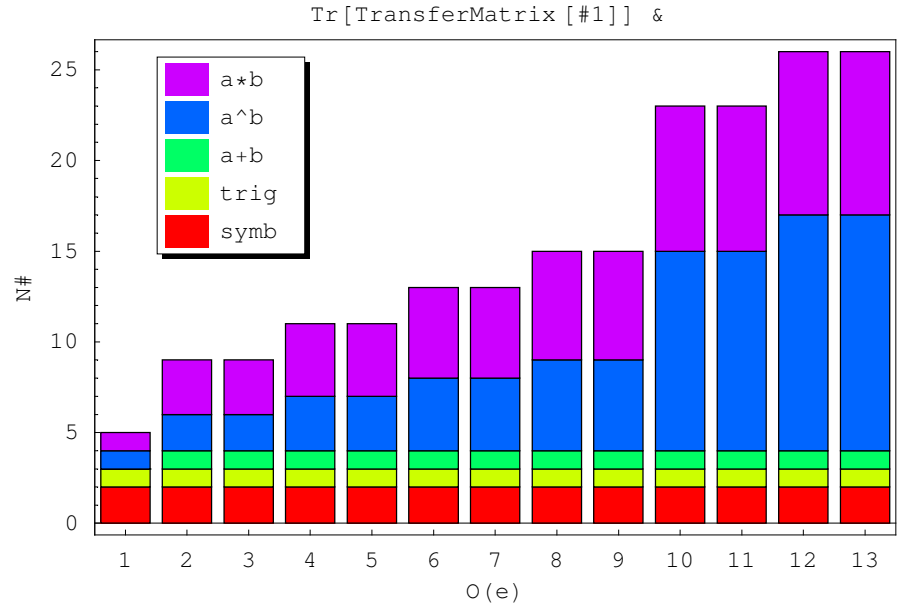
■ Analyse

Die Funktion implementiert das Verfahren, die Koeffizienten der kosinus- sowie sinusartigen Lösung aufzufinden und daraus die Transfermatrix R aufzustellen. Folgende Darstellung veranschaulicht die Anzahl der auftretenden Operationen in Abhängigkeit mit wachsender Ordnung:



Grafik 50: Ergebnis der Funktion TransferMatrix in Abhängigkeit der Ordnung des Störparameters e.

Die Spur der Matrix ergibt sich über die Funktion Tr[R] und wurde in Kapitel 2, Abschnitt 2.5. bis zur 17. Ordnung entwickelt. Die Anzahl der wachsenden Terme bleibt hier angenehmerweise klein in Abhängigkeit mit steigender Ordnung .



Grafik 51: Übersicht über die Termanzahl, die vom Algorithmus in Abhängigkeit der Ordnung in e erzeugt werden.

- **Vorkommen**

  Kap. 2.5.

5.3. Beschreibung der Algorithmen - Nichtlineare Analyse

Hier werden die Funktionen, die anhand der Algorithmen aus dem 3. Kapitel entstanden sind, definiert, kurz beschrieben und auf ihre Verwendbarkeit analysiert.

■ 5.3.1. ApproximateEquation

ApproximateEquation[n] gibt die Bewegungsgleichung des Sitnikovproblems mit r(t) - entwickelt um die Exzentrizität e bis zur n-ten Ordnung - wieder.

- **Mathematische Beschreibung**

  Die Bewegungsgleichung des Sitnikovproblems lautet:

$$z'' + \frac{z}{(r(t)^2 + z^2)^{3/2}} = 0$$

  Der Abstand der Primärkörper r(t) ist über die transzendente Keplergleichung u(t) = t+e*sin(u(t)) durch $r(t) = \frac{1}{2}$(1-e*cos(u(t))) gegeben. Entwickelt man u(t) nach dem Störparameter e, so erhält man einen Ausdruck für r(t) in Form einer Taylorreihe.

- **Implementation in Mathematica**

  Voraussetzung : *PrimariesDistance*

```
ApproximateEquation[n_Integer] :=
 Module[{x},
   $Derivation =
    {x = PrimariesDistance[n],
     x = x^2,
     x = TrigReduce[x],
     ApproximateEquation[n] = z''[t] + z[t] / (Normal[x] + z[t]^2)^(3/2) == 0}][[-1]]
```

- **Analyse**

  Die Gleichung ApproximateEquation[13] wurde für die Bestimmung der numerischen Lösungen in dieser Arbeit verwendet.

- **Vorkommen**

  Kap. 2.4., 3.4.

■ 5.3.2. PolynomialEquation

PolynomialEquation[n] entwickelt die Bewegungsgleichung des Sitnikovproblems nach e und z derart, dass in den Produkten $e^l z^m$ l+m ≤ n bleibt.

■ Mathematische Beschreibung

Da nur Lösungen mit kleinen Exzentrizitäten und Amplituden in dieser Arbeit untersucht wurden, wird die Differentialgleichung nach e und z derart entwickelt, dass in den auftretenden Termen $e^l z^m$ l+m ≤ n bleibt. Die Lösungen sind somit auch nur in jenen Parametergrenzen konvergent, wo sowohl der Absolutbetrag der Exzentrizität |e| als auch das Verhältnis |z(t)/r(t)| kleiner 1 bleiben. Die Entwicklung nach dem Parameter e resultiert aus der Approximation von r(t), die Differentialgleichung kann daher durch die Form:

$$z'' + \sum_{k=0}^{n} g_i(t)\, z^{2k+1} = 0$$

beschrieben werden.

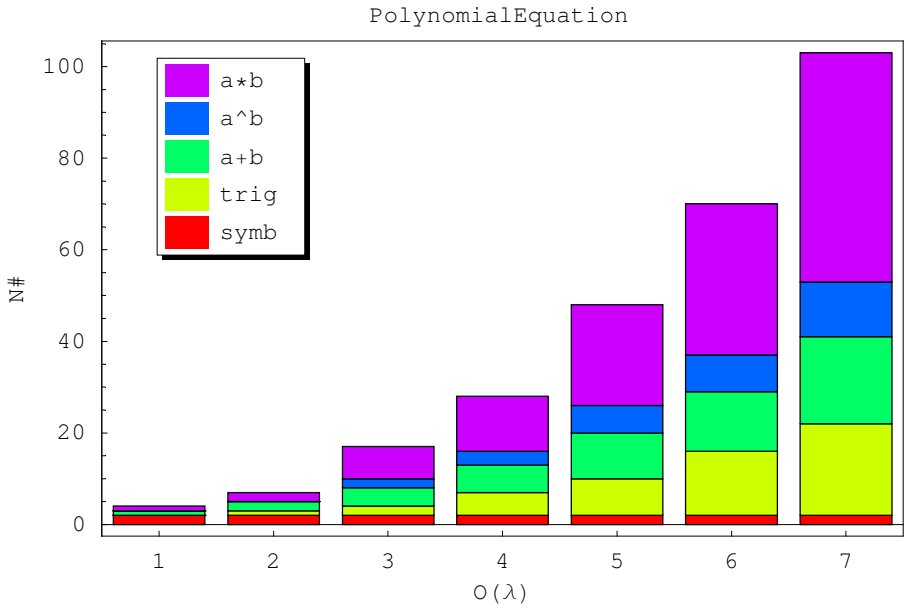
■ Implementation in Mathematica

Voraussetzung : *PrimariesDistance*

```
PolynomialEquation[n_Integer] := Module[{x}, $Derivation =
  {x = z''[t] + Sum[Binomial[-3/2, k] z[t]^(2 k+1) / r[t]^(2 k+3), {k, 0, (n-1)/2}],
   x = x /. r[t] → PrimariesDistance[n],
   x = Normal[x],
   x = Expand[x],
   x = x /. _ z[t]^j_ e^k_ :→ 0 /; j + k > n,
   x = x /. _ z[t]^j_ e | _ z[t] e^j_ :→ 0 /; j ≥ n,
   x = x /. _ z[t] e :→ 0 /; n == 1,
   x = TrigReduce[x],
   x = Collect[x, {z[t], e}],
   PolynomialEquation[n] = x == 0}][[-1]]
```

■ Analyse

Die Funktion approximiert die Bewegungsgleichung, indem sie den Abstand der Primärkörper in die polynome Darstellung einsetzt und nach den kleinen Parametern entwickelt. Mit Hilfe von Ersetzungsregeln wird die Bedingung $e^l z^m$, l+m ≤ n erfüllt. Die Anzahl der durchzuführenden Operationen wächst exponentiell mit steigender Ordnung an, wie aus Grafik 52 ersichtlich ist:



Grafik 52: Übersicht über die Termanzahl, die vom Algorithmus in Abhängigkeit der Ordnung in λ erzeugt wird.

Die Genauigkeit der Approximation wird in Kapitel 3 diskutiert.

- **Vorkommen**

Kap. 1.3., 3.1.

5.3.3. CourantSnyderTransformedA

CourantSnyderTransformedA[n] führt die Courant & Snyder - Transformation an der polynomischen Bewegungsgleichung des Sitnikovproblems n-ter Ordnung mit unbekannter Amplitudenfunktion w durch. Die darin vorkommende Variable t hängt nun von ψ, der linearisierten Phasenfunktion der Lösung ab.

- **Mathematische Beschreibung**

Die Courant & Snyder - Transformation baut auf den Ergebnissen der Floquet - Theorie auf. Nimmt man eine polynomische Differentialgleichung:

$$z'' + g(t)\, z + F(z, t) = 0$$

und führt an ihr die Transformationen y = z/w und $\psi = \int_0^t 1/w^2 \, ds$ durch, wobei w die Amplitudenfunktion und ψ die Phasenfunktion des linearisierten Falls kennzeichnen, so erhält man eine Differentialgleichung der Form:

$$y'' + y + F(y, t(\psi)) = 0$$

Der lineare Teil der Gleichung kann somit durch die Courant & Snyder - Transformation auf die Form des gestörten harmonischen Oszillators gebracht werden.

■ Implementation in Mathematica

Voraussetzung : *PolynomialEquation, CourantSnyderTransform*

```
CourantSnyderTransformedA[n_Integer] :=
 Module[{x},
   $Derivation =
    {x = PolynomialEquation[n],
     x = CourantSnyderTransform[x, w, {z, t}, {y, ψ}], x = Expand[x[[1]]],
     x = Expand[ x / (Coefficient[x, y''[ψ], 1]) ], x = Collect[x, {y[ψ], w, e}],
     CourantSnyderTransformedA[n] = x == 0}][[-1]]
```

Die Funktion führt die Courant & Snyder - Transformation automatisiert an der polynomischen Bewegungsgleichung durch. Herzstück des Algorithmus ist der Befehl *CourantSnyderTransform*:

```
CourantSnyderTransform[exp_, af_, {x_, t_}, {y_, ψ_}] :=
 Module[{w, x},
   $Derivation =
    {x = TransformDependentVar[exp, x[t], y[t] == x[t]/w[t]],
     x = TransformIndependentVar[x, y[t], ψ[t] == ∫ 1/w[t]^2 dt],
     x = x /. w''[t] → 1/w[t]^3 - Coefficient[exp[[1]], x[t], 1] w[t],
     x = FullSimplify[x, w[t] > 0, Trig → False],
     $Result = x[[1]] - x[[2]] == 0 /. w[t] → af}][[-1]]
```

CourantSnyderTransform[exp, af, {x,t}, {y,ψ}] führt die Courant & Snyder Transformation am Ausdruck *exp* mit der Amplitudenfunktion *af* von den Variablen *x* von *t* auf die Variablen *y* von *ψ* durch.

Die Transformation der abhängigen Variable wird hierbei durch die Funktion *TransformDependentVar* durchgeführt:

```
TransformDependentVar[exp_, fun_, rel_Equal] :=
 Module[{x, o},
   $Derivation =
    {o = DifferentialOrder[exp],
     x = Table[∂_{fun[[1]],i} rel, {i, 0, o}],
     x = Solve[x, Table[∂_{fun[[1]],i} fun, {i, 0, o}]],
     x = Flatten[x],
     $Result = exp /. x}][[-1]]
```

Es wird im Ausdruck *exp* die abhängige Variable definiert durch die Funktion *fun* über die Beziehung *rel* transformiert.

Die Transformation der unabhängigen Variablen wird über die Funktion *TransformIndependentVar* vollzogen:

```
TransformIndependentVar[exp_, fun_, rel_] :=
 Module[{x, y, o},
   $Derivation =
    {o = DifferentialOrder[exp],
     y = Head[fun],
     x = Table[∂{rel⟦1,1⟧,i} (y[rel⟦1⟧] == y[rel⟦1, 1⟧]), {i, 0, o}],
     x = x /. rel⟦1⟧ → Head[rel⟦1⟧],
     x = Solve[x, Table[∂{fun⟦1⟧,i} fun, {i, 0, o}]],
     x = Flatten[x], y = rel /. Equal → Rule,
     y = Table[∂{y⟦1,1⟧,i} y, {i, 0, o}],
     $Result = exp /. x /. y}]⟦-1⟧
```

Die Übergabe an die Funktion entspricht der von *TransformDependentVar*. Für die Festlegung der maximal auftretenden Ableitung wird die Funktion *DifferentialOrder* (s.o.) benötigt. Die Objekte *TransformDependentVar* und *TransformIndependentVar* sind ebenfalls für beliebige Differentialgleichungen einsetzbar.

- **Vorkommen**

Kap. 3.1.

■ 5.3.4. CourantSnyderTransformedB

CourantSnyderTransformedB[n] führt die Courant & Snyder - Transformation an der polynomischen Bewegungsgleichung des Sitnikovproblems n-ter Ordnung durch. Die darin vorkommende Variable t hängt nun im Unterschied zu *CourantSnyderTransformedA* von ψ ab.

- **Mathematische Beschreibung**

Siehe *CourantSnyderTransformedA*.

- **Implementation in Mathematica**

Voraussetzung : *CourantSnyderTransformedA, AmplitudeFunction*

```
CourantSnyderTransformedB[n_Integer] :=
 Module[{x, xx},
   $Derivation = {x = CourantSnyderTransformedA[n]⟦1⟧,
     xx = AmplitudeFunction[n], x = x /. w → xx, x = Expand[Normal[x]],
     x = x /. _ e^i_ y[ψ]^j_ :↦ 0 /; i + j > n,
     x = x /. _ e y[ψ]^j_ | _ e^j_ y[ψ] :↦ 0 /; j == n, x = TrigReduce[x],
     CourantSnyderTransformedB[n] = Collect[x, {y[ψ], e}] == 0}]⟦-1⟧
```

Die Funktion setzt die Amplitudenfunktion in die von *CourantSnyderTransformedA* aufgestellte Differentialgleichung ein und reorganisiert die Gleichung.

- **Vorkommen**

Kap. 3.1.

■ 5.3.5. InversePhaseFunction

InversePhaseFunction[n] berechnet zu der Phasenfunktion der Floquet - Theorie die Umkehrtransformation bis zur n-ten Ordnung.

- **Mathematische Beschreibung**

Die Phasenfunktion stellt eine Beziehung $\psi = \psi(t,e)$ dar. Gesucht sei deren Umkehrung, daher $t = t(\psi,e)$. Die Phasenfunktion ist in Form einer Störreihe bekannt. Im ersten Schritt setzt man für deren Umkehrung eine Störreihe mit unbekannten Koeffizientenfunktionen an und in die ursprüngliche Beziehung ein. Entwickelt und ordnet man dieses nach dem Parameter e, so erhält man ein Gleichungssystem, welches die Bedingungen für die Koeffizientenfunktionen bestimmt. Dieses kann induktiv gelöst werden, die Störreihe der Umkehrfunktion ist nun bekannt.

- **Implementation in Mathematica**

Voraussetzung : *PerturbativeSeries, PhaseFunction*

```
InversePhaseFunction[n_Integer] :=
 Module[{x, xx, X},
   $Derivation =
    {x = Normal[PhaseFunction[n]],
     xx = Normal[PerturbativeSeries[t[ψ], e, n]], x = x /. t → xx,
     x = Series[x, {e, 0, n}] == ψ,
     X = x[[1, 3]],
     X = {X[[1]] == ψ, Thread[Rest[X] == 0]},
     x = Solve[X[[1]], Subscript[t, 0][ψ]],
     X = Flatten[X /. x],
     X = Solve[X, Table[Subscript[t, i][ψ], {i, 1, n}]],
     X = Flatten[Append[X, x]],
     x = TrigReduce[xx /. X],
     x = Series[x, {e, 0, n}],
     InversePhaseFunction[n] =
      SeriesData[e, 0, Collect[x[[3]], {Sin[_] | Cos[_], ψ}], 0, n + 1, 1]}][[
  -1]]
```

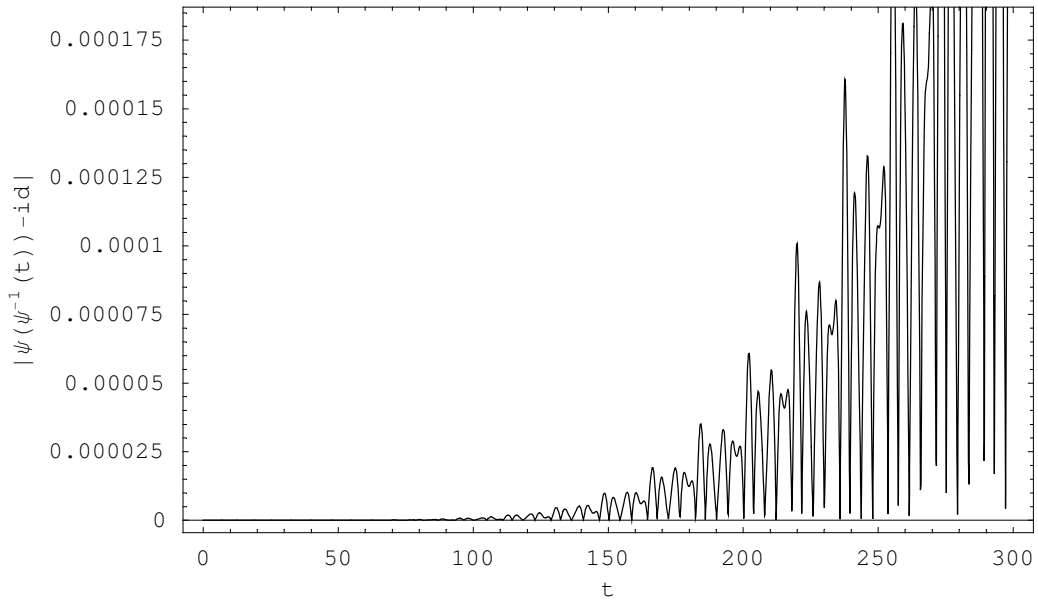
- **Analyse**

Die Hauptaufgaben des Algorithmus bestehen darin, einen Ausdruck nach einem kleinen Parameter zu entwickeln, das Gleichungssystem, das die unbestimmten Koeffizientenfunktionen bestimmt, aufzustellen, und dieses zu lösen. Die Anzahl der Terme, die bei der Prozedur entstehen, entspricht jener der Amplitudenfunktion.

Das Problem eine Umkehrfunktion zu einer gegebenen Störreihe zu bilden, tritt in dieser Arbeit zweimal auf. Einmal muss sie, wie hier zur Phasenfunktion, ein weiteres Mal zu der von der Poincaré - Lindstedt - Methode resultierenden *SigmaFunction* (Abbildung von $\sigma \to \tau$) berechnet werden. Es ist ein leichtes einen Algorithmus zu finden, der diese Aufgabe für jede beliebige Störreihe löst. In *Mathematica* lässt sich dies auf die folgende Art und Weise realisieren:

```
InversePerturbativeSeries[ser_SeriesData, ψ_[t_], n_] :=
 Module[{e = ser[[1]], x, xx, X},
   $Derivation =
    {x = Normal[ser], xx = Normal[PerturbativeSeries[t[ψ], e, n]],
     x = x /. t → xx,
     x = Series[x, {e, 0, n}] == ψ,
     X = x[[1, 3]],
     X = {X[[1]] == ψ, Thread[Rest[X] == 0]},
     x = Solve[X[[1]], Subscript[t, 0][ψ]], X = Flatten[X /. x],
     X = Solve[X, Table[Subscript[t, i][ψ], {i, 1, n}]],
     X = Flatten[Append[X, x]],
     x = xx /. X,
     $Result = Series[x, {e, 0, n}]}][[-1]]
```

Zu einer Relation $\psi(t) = \psi_0(t) + \psi_1(t)\, e + \psi_2(t)\, e^2 + ... O(e^n)$ wird eine weitere $t(\psi) = t_0(\psi) + t_1(\psi)\, e + t_2(\psi)\, e^2 + O(e^n)$ erzeugt, welche die Bedingung $\psi(\mathrm{t}(\psi)$ und $\mathrm{t}(\psi(\mathrm{t}))$ gleich id + $O(e^n)$ erfüllt. Es zeigt sich, dass die Inverse ein anderes Konvergenzverhalten aufweisen kann.



Grafik 53: Abweichung von der Identitätsfunktion. Die Exzentrizität wurde für die Darstellung e = 0.2 gewählt. Der Effekt wird mit zunehmenden e größer.

Deutlich ist die Abweichung von der 1. Mediane zu sehen. Die Exzentrizität wurde für die Grafik e = 0.2 gewählt. Die Differenz steigt mit wachsendem e. Vermutungen liegen nahe, dass der größte Beitrag für den Fehler in der Approximation durch die Umkehrtransformation zu Stande kommt. Es zeigt sich im Vergleich mit den Differenzplots aus dem vorherigen Abschnitt, dass ungefähr an der selben Stelle die Abweichung am größten wird. Abhilfe könnte dadurch geschaffen werden, dass man das Problem gänzlich von einer anderen Seite aufrollt. (Ist es zum Beispiel möglich, die Differentialgleichung für die Amplitudenfunktion derart umzuformen, so dass aus ihr direkt die Umkehrfunktion mit Hilfe der Störungsrechnung abgeleitet werden könnte?) Nähere Untersuchungen in diese Richtung, werden im Anschluss an diese Arbeit durchgeführt werden.

- **Vorkommen**

Kap. 3.2.

■ 5.3.6. TauEquation

TauEquation[n] transformiert die unabhängige Variable der Courant & Snyder transformierten polynomischen Bewegungsgleichung n-ter Ordnung über die Beziehung: $\tau = \psi / 2\sqrt{2}$

- **Mathematische Beschreibung**

-

■ **Implementation in Mathematica**

Voraussetzung : *CourantSnyderTransformedB, TransformIndependentVar*

```
TauEquation[n_Integer] := Module[{x}, $Derivation =
    {x = CourantSnyderTransformedB[n][[1]],
     x = TransformIndependentVar[x, y[ψ], τ[ψ] == ψ/√8],
     x = Expand[x 8],
     TauTransformed[n] = Collect[x, {y[τ], λ, e}] == 0}][[-1]]
```

Es handelt sich um eine standardmäßige Herleitung, wie sie von der Funktion *TransformIndependentVar* mehrfach durchgeführt wird.

■ **Vorkommen**

Kap. 3.1., 3.3.

■ 5.3.7. TauTransformedA

TauTransformedA[n] ersetzt die vorkommende Variable t durch t($\psi(\tau)$) und entwickelt die trigonometrischen Terme nach dem Störparameter e. Hierfür wird die inverse Phasenfunktion ψ^{-1} bis zur n-ten Ordnung verwendet.

■ **Mathematische Beschreibung**

Die Variable t in cos(n t) und sin(n t) wird über die Beziehung $t = \psi^{-1}$(t,e) und $\psi = 2\sqrt{2}\,\tau$ ersetzt. Die trigonometrischen Terme werden nach dem kleinen Parameter e entwickelt. Terme $e^j\, y^k$ mit j+k > n werden hierbei vernachlässigt.

■ **Implementation in Mathematica**

Voraussetzung: *TauEquation, InversePhaseFunction*

```
TauTransformedA[n_Integer] := Module[{x}, $Derivation =
    {x = TauEquation[n], x = x /. t → Normal[InversePhaseFunction[n]],
     x = x /. ψ → √8 τ,
     x = x /. Cos[arg_] :↦ Normal[Series[Cos[arg], {e, 0, n}]] /;
         MemberQ[arg, e, ∞],
     x = TrigReduce[x[[1]]] /. _ y[τ]^i_ e^j_ :↦ 0 /; i + j > n,
     TauTransformedA[n] = Collect[x, {y[τ], e, Sin[_] | Cos[_]}] == 0}][[-1]]
```

Die Termanzahl weicht nicht wesentlich von jener der Ausgangsgleichung ab.

■ **Vorkommen**

Kap. 3.3.

■ 5.3.8. TauTransformedB

TauTransformedB[n] liefert die Bewegungsgleichung des Sitnikovproblems in den Variablen y und τ entwickelt bis zur Ordnung n, wobei Terme multipliziert mit $e^j y^k$ mit j+k > n vernachlässigt werden. Im Unterschied zu TauTransformedA[n] wird ein künstlicher Störparameter λ eingeführt, der Terme gleicher Ordnungen zusammenhält.

■ **Mathematische Beschreibung**

Der künstliche Störparameter λ dient als Ordnungsparameter. Hierbei gilt λ^n hält alle Terme $e^i y^k$ mit i+j = n multiplikativ zusammen. Diese Form der Darstellung bereitet die Gleichung auf die Methode von Poincaré - Lindstedt vor.

■ **Implementation in Mathematica**

Voraussetzung :*TauTransformedA*

```
TauTransformedB[n_Integer] := Module[{x}, $Derivation =
    {x = TauTransformedA[n][[1]],
     x = Expand[x],
     x = x /. y[τ]^i_ :> (λ y[τ])^i /; i ≥ 3,
     x = x /. e :> e λ,
     TauTransformedB[n] = Collect[x, {λ, y[τ], e}] == 0}][[-1]]
```

Der Einführung des künstlichen Störparameters λ fällt eine spezielle Bedeutung zu. Wird die Methode der gedehnten Koordinaten an der Gleichung in einer anderen Form durchgeführt, so erzeugt sie nicht konvergente Reihenglieder. Es ist daher wichtig, gleiche Potenzen auf einmal zu verarbeiten. Das Problem der multidimensionalen Störungsrechnung in den Störparametern erfordert viel Erfahrung. Wenig konnte hierzu in der Literatur gefunden werden.

■ **Vorkommen**

Kap. 3.3.

■ 5.3.9. MPLExpansion

MPLExpansion[n] entwickelt eine Störungslösung der nichtlinearen Bewegungsgleichung des Sitnikovproblems n-ter Ordnung (TauTransformedB[n]) bis zur n-ten Ordnung in λ. Die Lösung wird dem Objekt NonlinearSolution[m,n] zugewiesen, wobei vorgabemäßig m = n ist. Die durch die Methode von Poincaré-Lindstedt erzeugten Zeitdehnungskoeffizienten werden in SigmaFunction[m,n] gespeichert. Das aus dem Störansatz resultierende Gleichungssystem wird dem Objekt SigmaSystem[m,n] übergeben. Die Funktion implementiert das Poincaré-Lindstedt - Verfahren bis zu beliebiger Ordnung in λ. Der Algorithmus schreibt Zwischenresultate in die Dateien "X.pr", "Y.pr", "T.pr". Diese werden in dem Ordner, in dem die Funktion ausgeführt wird gespeichert, und nach Vollendigung der Herleitung wieder gelöscht.

■ Mathematische Beschreibung

Die Poincaré - Lindstedt Methode ist ein klassisches Verfahren der Störungstheorie, welches zusätzlich zu einem klassischen Ansatz der abhängigen Variable $y(\tau) = y_0(\tau) + y_1(\tau)\, e + y_2(\tau)\, e^2 + ... + O(e^n)$ einen Ansatz in der unabhängigen Variablen $\tau = \tau_0(\sigma) + \tau_1(\sigma)\, e + \tau_2(\sigma)\, e^2 + ... + O(e^n)$ durchführt. Dadurch können Säkularterme, die aufgrund der Methode auftreten können, durch geeignete Wahl der τ_i vermieden werden. Diese werden während der Durchführung des Algorithmus bestimmt. Ergebnis ist eine Störungslösung y(τ,e) sowie eine Zeitdehnungstransformation $\tau(\sigma$,e). Setzt man die beiden Ansätze in die Ausgangsgleichung ein, so erhält man das nach den Potenzen in e geordnete Gleichungssystem, um die einzelnen Größen zu bestimmen. Eine detaillierte Beschreibung der Methode kann in Kapitel 3 gefunden werden.

■ Implementation in Mathematica

Voraussetzungen:
PerturbativeSeries,
TransformDependentVar,
TransformIndependentVar,
TauTransformedB

```
MPLExpansion[n_Integer, opts___] :=
 Module[{time = SessionTime[], j = 1, a, m, Int, T = {T0[σ] → σ},
    X, Xcopy, Y, x, xx, form},

   Options[MPLExpansion] = {TraceFormat → Identity, StartingOrder → 1,
     Order → n};

   form = TraceFormat /. {opts} /. Options[MPLExpansion];
   a = StartingOrder /. {opts} /. Options[MPLExpansion];
   m = Order /. {opts} /. Options[MPLExpansion];
```

```
Int[exp_] := Module[{c, s},
  s = TrigReduce[Sin[2 √2 σ] * exp];
  c = TrigReduce[Cos[2 √2 σ] * exp];
  s = Integrate[s, σ];
  c = Integrate[c, σ];
  Cos[2 √2 σ] * s/(2 √2) - c/(2 √2) Sin[2 √2 σ]];

$Derivation =
 {x = TauTransformedB[n],
  ansatz[τ] = PerturbativeSeries[T[σ], λ, m] /. T0[σ] → σ,
  ansatz[y] = PerturbativeSeries[y[σ], λ, m],
  x = TransformIndependentVar[x, y[τ], τ[σ] == ansatz[τ]],
  x = TransformDependentVar[Normal[x], y[σ], y[σ] == ansatz[y]],

  Print["Provisional Results written to : ", "X.pr, Y.pr, T.pr"],

  Print[">>> System to solve for... "],
  X = Xcopy = Thread[x[[1, 3]] == 0] /. τ → σ,
  Print[form[X], "  (time constrained: ", SessionTime[] - time,
   ")"],

  Print[">> Unperturbed solution..."],
  x = Y = {y0[σ] → C[1] Cos[2 √2 σ] + C[2] Sin[2 √2 σ]},
  Print[form[x], "  (time constrained: ", SessionTime[] - time,
   ")"],

  Table[{
    x = Flatten[Table[D[x, {σ, i}], {i, 0, 2}]], Put[X = X /. x, "X.pr"],

    Print[">> Resulting equation for order...", k],
    x = Expand[X[[k + 1, 1]]],
    x = TrigReduce[x],
    x = Collect[x, Sin[_] | Cos[_]],
    Print[form[x == 0], "  (time constrained: ",
     SessionTime[] - time, ")"],

    x = x /. yk[σ] | yk''[σ] :→ 0,

    Print["> Coefficient of secular term..."],
    xx =
     (Cases[x, _ * Sin[2 √2 σ], ∞] /. Sin[2 √2 σ] → 1)[[1]] == 0,
    Print[form[xx], "  (time constrained: ", SessionTime[] - time,
     ")"],

    Print["> Solving for ", Tk[σ], "..."],
    xx = DSolve[xx, Tk[σ], σ, GeneratedParameters → $] /. $[_] → 0,
    Print[form[xx], "  (time constrained: ", SessionTime[] - time,
```

```
    ")"],

   Put[T = Flatten[Append[T, xx]], "T.pr"],

   Print[">> Resulting equation for order... ", k],
   xx = Flatten[Table[D[xx, {σ, i}], {i, 0, 2}]],
   X = Expand[X /. xx], x = Expand[x /. xx],
   x = x /. amp_ * Sin[arg_] :> Simplify[amp] * Sin[arg],
   x = x /. amp_ * Cos[arg_] :> Simplify[amp] * Cos[arg],
   Print[form[x + 8 y_k[σ] + y_k''[σ] == 0], "  (time constrained: ",
    SessionTime[] - time, ")"],

   Print[">> Solving for ", y_k[σ], " (with k[1]=k[2]=0)..."],
   x = Int[x],
   xx = TrigReduce[x],
   xx = xx /. amp_ * Sin[arg_] :> amp * Sin[Expand[arg]],
   xx = xx /. amp_ * Cos[arg_] :> amp * Cos[Expand[arg]],
   Print[form[xx], "  (time constrained: ", SessionTime[] - time,
    ")"],

   Put[x = Y = Append[Y, y_k[σ] → xx], "Y.pr"]},
  {k, a, m}],

 Print["Results written to > SigmaSystem, > SigmaFunction, >
    NonlinearSolution (", m, " ", n, ")"];

 SigmaSystem[n, m] = Xcopy;
 SigmaFunction[n, m] = (ansatz[τ] /. T);

 x = ansatz[y] /. Y,
 x = Collect[x[[3]], Sin[_] | Cos[_]],
 x = x /. amp_ * Sin[arg_] :> FullSimplify[amp] * Sin[arg],
 x = x /. amp_ * Cos[arg_] :> FullSimplify[amp] * Cos[arg],
 NonlinearSolution[n, m] = SeriesData[λ, 0, x, 0, m + 1, 1],

 DeleteFile[{"X.pr", "Y.pr", "T.pr"}],

 $Result =
  {SigmaSystem[n, m],
   τ[σ] → SigmaFunction[n, m],
   y[σ] → NonlinearSolution[n, m]}

}][[-1]]
```

- **Beschreibung**

Der wesentliche Parameter, der an die Funktion übergeben werden muss, ist die Ordnung der Ausgangsgleichung. Implizit wird daraus gefolgert, dass eine Entwicklung bis zu dieser

Ordnung gemacht werden soll. Folgende zusätzliche Optionen können der Funktion übergeben werden.

TraceFormat→format: Format in welchem, die Zwischenergebnisse auf dem Bildschirm ausgegeben werden sollen.

StartingOrder→n: Ordnung, mit der die Berechnungen begonnen werden sollen. Wird diese Option gesetzt, so erwartet der Befehl die Existenz der drei Dateien, die Zwischenergebnisse bis zur Ordnung (n-1) enthalten.

Order→m: Soll die Lösung zu einer anderen Ordnung entwickelt werden, als implizit durch die Ausgangsgleichung vorgesehen, so kann diese hier gesetzt werden.

- **Vorkommen**

Kap. 3.3.

■ 5.3.10. SigmaSystem, SigmaFuntion, NonlinearSolution

SigmaSystem[m,n], SigmaFunction[m,n] und NonlinearSolution[m,n] werden während der Ausführung des Algorithmus MPLExpansion[n,opt] erzeugt und beinhalten die Resultate von diesem. Um eines der Objekte zu erzeugen, muss die Funktion MPLExpansion[n,opts] aufgerufen werden.

- **Mathematische Beschreibung**

s.o.

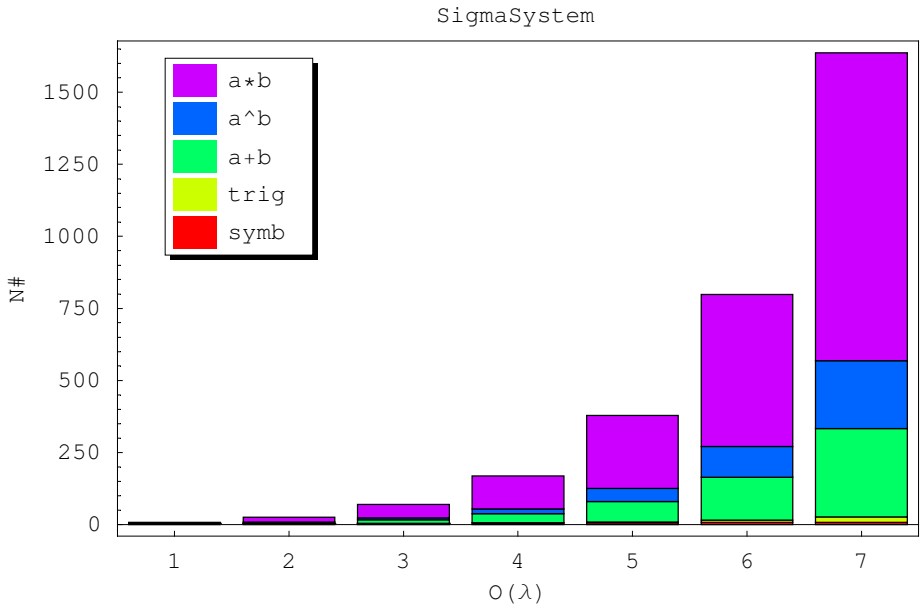
- **Implementation in Mathematica**

Voraussetzung : *MPLExpansion*

–

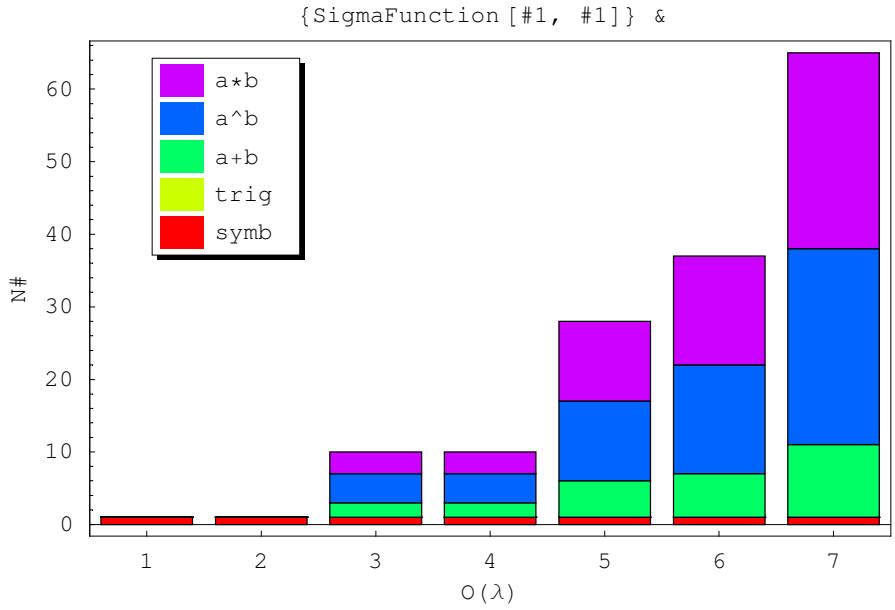
- **Analyse**

Interessant ist es, die Anzahl der Terme und Operationen, die bei der Methode von Poincaré-Lindstedt erzeugt werden, zu untersuchen. In allen folgenden Grafiken ist mit Ordnung gemeint, dass von der Gleichung, definiert durch die Funktion *TauEquation[n],* ausgegangen wird, und diese durch einen Störansatz bis zur n-ten Ordnung entwickelt wurde.

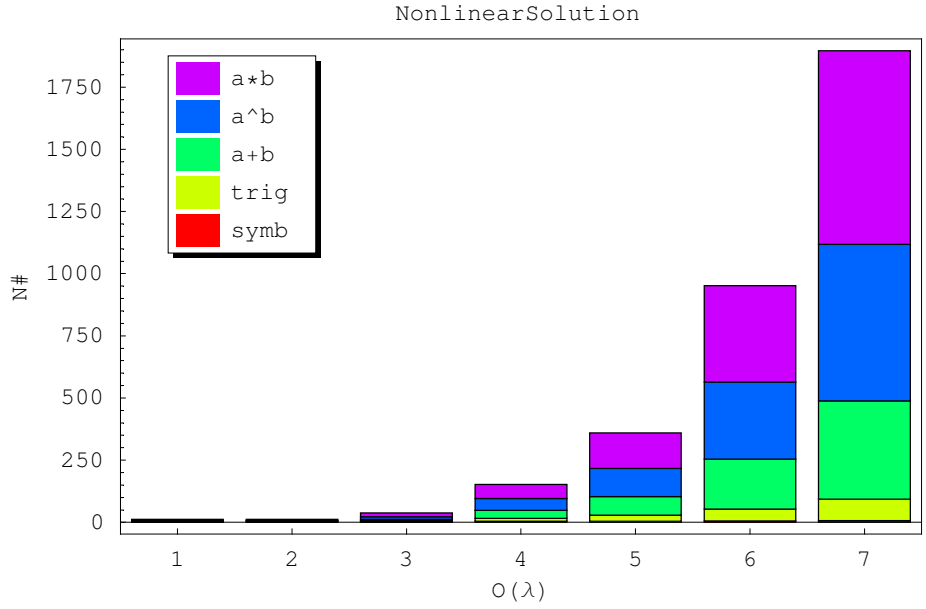


Grafik 54: Größe des Gleichungssystems in Abhängigkeit des Störparameters λ, dass mit Hilfe der Methode von Poincaré - Lindstedt gelöst werden muss.

Das Differentialgleichungssystem, das durch den Algorithmus gelöst werden muss, ist für die 7. Ordnung in λ durch über 1500 Operationen definiert. Es ist eine Herausforderung, dieses auf automatisierten Wege zu lösen. Die Anzahl der Terme und Operationen wächst mit steigender Ordnung stark exponentiell an, eine obere Grenze bezüglich der Berechenbarkeit ist offensichtlich. Dennoch handelt es sich bei den Differentialgleichungen um einfache inhomogene Gleichungen, die im Wesentlichen nichtlinear gestörte Oszillatoren sind. Die Gleichungen können durch Auffinden von Integralen dennoch gelöst werden, wie im Abschnitt 3.3. beschrieben wird.



Grafik 55: Anzahl der Operationen, die von der Funktion MPLExpansion in Abhängigkeit des Störparameters λ erzeugt werden.



Grafik 56: Anzahl der Operationen, die eine Lösung des Sitnikovproblems in Abhängigkeit des Systemparameters λ darstellen.

Eine Beschreibung der Funktion *MPLExpansio*n, die diese Objekte erzeugt, kann in 5.3.9. gefunden werden.

- **Vorkommen**

  Kap. 3.3.

■ 5.3.11. InverseSigmaFunction

InverseSigmaFunction[n,m] berechnet die Umkehrtransformation zu der anhand der Poincaré-Lindstedt Methode gefundenen Zeitdehnungsfunktion SigmaFunction[n,m].

■ Mathematische Beschreibung

Die Poincaré-Lindstedt Methode oder Methode der gedehnten Koordinaten führt an einer Differentialgleichung zwei Störansätze durch. Einen für die abhängige, einen für die unabhängige Variable. Nach der Durchführung ist es notwendig, eine Umkehrtransformation zu der Zeitdehnungsfunktion resultierend von dem Ansatz zu finden, um die Lösung in der alten Zeit darstellen zu können. Hierfür ist es notwendig, eine Störreihe zu invertieren, was nach einem einfachen Schema durchgeführt werden kann.

Die Umkehrung einer Störreihe erfolgt, indem die inverse Transformation als Störreihe angesetzt und in diese eingesetzt wird. Unter Vernachlässigung höherer Ordnungen wird das daraus resultierende Gleichungssystem für die einzelnen Ordnungen des Störparameters sukzessive gelöst.

■ Implementation in Mathematica

Voraussetzung : *PerturbativeSeries, SigmaFunction*

```
InverseSigmaFunction[n_Integer, m_Integer] :=
 Module[{x},
   $Derivation = {x = SigmaFunction[n, m],
     If[n == 0, InverseSigmaFunction[0, 0] = τ,
      InverseSigmaFunction[n, m] =
       InversePerturbativeSeries[x, τ[σ], m]}][[-1]]
```

Die Funktion *InverseSigmaFunction* dient als Verwaltungsfunktion und ruft als eigentlichen Algorithmus die Funktion *PerturbativeSeries* auf, die auf jede beliebige Störreihe angewandt werden kann.

```
InversePerturbativeSeries[ser_SeriesData, ψ_[t_], n_] :=
 Module[{e = ser[[1]], x, xx, X},
   $Derivation = {x = Normal[ser],
     xx = Normal[PerturbativeSeries[t[ψ], e, n]], x = x /. t → xx,
     x = Series[x, {e, 0, n}] == ψ, X = x[[1, 3]],
     X = {X[[1]] == ψ, Thread[Rest[X] == 0]},
     x = Solve[X[[1]], "Subscript"[t, 0][ψ]], X = Flatten[X /. x],
     X = Solve[X, Table["Subscript"[t, i][ψ], {i, 1, n}]],
     X = Flatten[Append[X, x]], x = xx /. X,
     $Result = Series[x, {e, 0, n}]}][[-1]]
```

InversePerturbativeSeries[*ser,ψ[t],n*] bestimmt die Inverse einer Störreihe *ser* der Form $\psi_0[t] + \psi_1[t]\,\lambda + \psi_2[t]\,\lambda^2 + ...$, mit bekannten Funktionen ψ_i, bis zur Ordnung $O(\lambda^n)$. Das

Resultat entspricht daher einer Reihenentwicklung um den selben Parameter λ der Form $t_0[\psi] + t_1[\psi]\,\lambda + t_2[\psi]\,\lambda^2 + ... + O[\lambda]^n$. Die unbekannten Koeffizientenfunktionen t_i werden bestimmt und ausgegeben. Die Funktion greift auf das Objekt *PerturbativeSeries* zu, welches nicht anderes als eine Störreihe mit unbekannten Koeffizienten um einen Störparameter bis zu einer bestimmten Ordnung definiert:

```
PerturbativeSeries[f_[x_], {e_, eo_}, {a_, o_}] :=
 SeriesData[e, eo, Table["Subscript"[f, i][x], {i, a, o + 1}], a, o + 1, 1]

PerturbativeSeries[f_[x_], {e_, eo_}, o_] :=
 PerturbativeSeries[f[x], {e, eo}, {0, o}]

PerturbativeSeries[f_[x_], e_, {a_, o_}] :=
 PerturbativeSeries[f[x], {e, 0}, {a, o}]

PerturbativeSeries[f_[x_], e_, o_] :=
 PerturbativeSeries[f[x], {e, 0}, {0, o}]
```

Der Algorithmus funktioniert für beliebige Störreihen, die Berechnungszeit und Genauigkeit hängt von der Komplexität der Koeffizientenfunktionen der Eingabe ab.

- **Analyse**

Die Bestimmung der Inversen t(ψ) einer gegebenen Störreihe ψ(t) führt im Allgemeinen zu dem Problem, dass für große Werte für die unabhängige Variable die Identitätsfunktion id = t(ψ(t)) = ψ(t(ψ) um die erste Meridiane zu oszillieren beginnt. Die Genauigkeit ist daher durch den Abbruchfehler stark eingeschränkt, das Ergebnis sollte sorgfältig auf die Anwendbarkeit überprüft werden.

- **Vorkommen**

Kap. 3.4.1.

■ 5.3.12. Sigma

Sigma[n] berechnet die nichtlineare Phasenfunktion der Lösung der Bewegungsgleichung des Sitnikov Problems bis zur Ordnung $O(\lambda^n)$. Die dort auftretende unabhängige Variable σ wird über die Funktion in die Zeit t rücktransformiert.

- **Mathematische Beschreibung**

Über die inverse Zeitdehnungsfunktion, die von der Methode der gedehnten Koordinaten her rührt, und der linearisierten Phasenfunktion, die mit Hilfe der Floquet Theorie hergeleitet wurde, wird die nicht lineare Phasenfunktion der Lösung des Sitnikov Problems aufgestellt und nach den Einfluss der einzelnen Terme auf die Lösung geordnet.

■ **Implementation in Mathematica**

Voraussetzung : *InverseSigmaFunction, PhaseFunction*

```
Sigma[n_] :=
 Module[{x},
   $Derivation =
    {x =
      ExpandAll[Normal[InverseSigmaFunction[n, n]
           /. τ → ψ/(2 √2) /. ψ → Normal[PhaseFunction[n]]]] /. λ → 1;
     x = x /. {C[1] :> C[1] λ, C[2] :> C[2] λ, e :> e λ}; x = x /. λ^m_ :> 0 /; m > n;
     x = x /. λ → 1; x = Collect[x, {t, Sin[_] | Cos[_], e}];
     x = x //. a_ C[1]^p_ + a_ C[2]^p_ + b___ :> a (C[1]^p + C[2]^p) + b,
     Sigma[n] = x //. a_ C[1]^p1_ C[2]^p2_ + a_ C[1]^p2_ C[2]^p1_ + b___ :>
       a (C[1]^p1 C[2]^p2 + C[1]^p2 C[2]^p1) + b}][[-1]]
```

Es handelt sich hier im Wesentlichen darum, Muster zu erkennen und zu ersetzen. Die Berechnungszeit, um diese Muster zu identifizieren, hängt vor allem von dem Verfahren ab, welches von dem CAS verwendet wird.

■ **Vorkommen**

Kap. 3.4.

■ 5.3.13. SolutionOfLhotka

SolutionOfLhotka[n] berechnet die Lösung der Bewegungsgleichung des Sitnikov Problems bis zur Ordnung $O(\lambda^n)$. Die unabhängige Variable σ wird über die Funktion Sigma[n] in die alte Zeit t transformiert.

■ **Mathematische Beschreibung**

Über die Floquet Theorie wird die linearisierte Amplituden- und Phasenfunktion der Lösung bestimmt und dazu verwendet, die Bewegungsgleichung anhand der Courant & Snyder Transformation in einen nichtlinear gestörten Oszillator überzuführen. Diese wird mit der Methode von Poincaré-Lindstedt gelöst, die Terme geordnet und in eine physikalisch relevante Form gebracht.

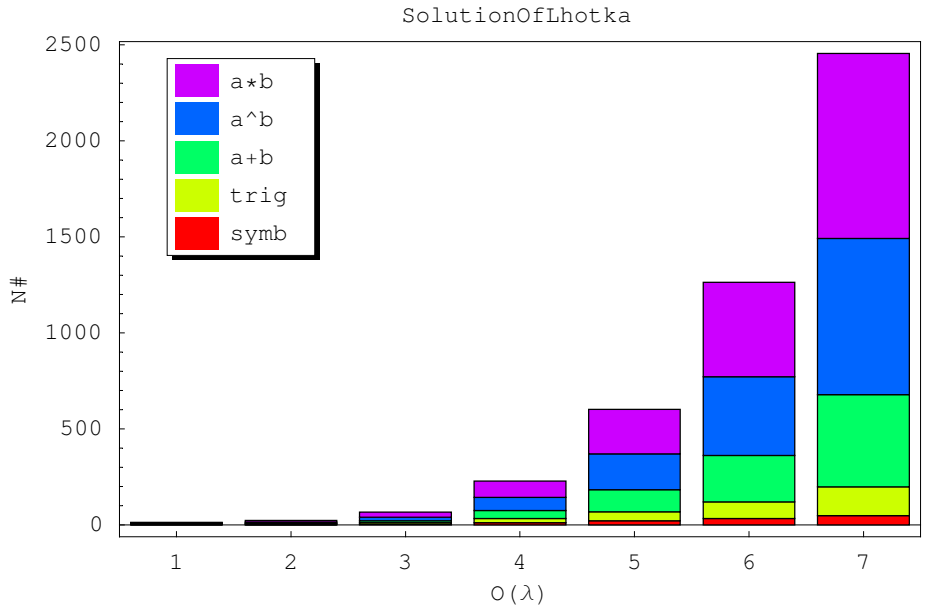
■ **Implementation in Mathematica**

Voraussetzung : *AmplitudeFunction, NonlinearSolution*

```
SolutionOfLhotka[n_Integer] :=
 Module[{x},
   $Derivation =
    {x = ExpandAll[
       Normal[AmplitudeFunction[n]] Normal[NonlinearSolution[n, n]] /.
        λ → 1]; x = x /. {C[i_] :↦ C[i] λ, e :↦ e λ}; x = ExpandAll[x];
     x = x /. λ^m_ :↦ 0 /; m > n;
     x = Collect[x, {Cos[_ σ], Sin[_ σ], Cos[_ + _ σ], Sin[_ + _ σ], λ}];
     x = x /. a_ λ :↦ FullSimplify[a] λ; x = x /. a_ λ^i_ :↦ FullSimplify[a] λ^i;
     x = x /. amp_ Cos[arg_] :↦ amp Cos[Expand[arg]];
     x = x /. amp_ Sin[arg_] :↦ amp Sin[Expand[arg]];
     SolutionOfLhotka[n] = x /. λ → 1}][[-1]]
```

■ Analyse

Die Funktion stellt die Ergebnisse der nicht linearen Analyse zusammen und gibt die Lösung des Sitnikov Problems aus. Es werden ein letztes Mal die Terme nach ihrem Einfluss auf die Lösung geordnet und der künstliche Störparameter λ durch die Identität ersetzt.



Grafik 57: Anzahl der von SolutionOfLhotka erzeugten Terme und Operationen, die notwendig sind, um einen Funktionswert zu berechnen. Die Anzahl der Operationen wächst exponentiell an, der Lösungsmethode ist daher eine obere Schranke aufgrund der Berechenbarkeit gesetzt.

Schon für die 7. Ordnung in λ sind ~2500 Operationen notwendig, um einen Punkt der Bahn des Sitnikov Problems zu lösen. Der Sinn der verwendeten Störungstheorie liegt weniger darin, einen Zahlenwert zu einem Zeitpunkt t auszugeben, als vielmehr die Abhängigkeiten der Systemparameter zu studieren, wie es in Kapitel 4 auch ansatzweise getan wird. Auch wenn es theoretisch möglich wäre, die in dieser Arbeit entstandenen Algorithmen dazu zu verwenden, Lösungen höherer Ordnungen zu erzeugen, scheint dies in Angesicht von Grafik 57 nicht

sinnvoll. Zu viele Terme werden produziert und der Zuwachs in der Genauigkeit läge außerhalb des Bereichs, in dem die polynomische Approximation funktioniert.

- **Vorkommen**

  Kap. 3.4.

5.4. Weitere Objekte des SDS

Die Definitionen der weiteren Objekte des *Sitnikov Derivation System* finden sich auf der beiliegenden CD wieder. Ihre Verwendung sollte anhand der Namensgebung ersichtlich sein, im Folgenden werden die wichtigsten von ihnen noch einmal zusammengestellt und eine kurze Beschreibung angeführt. Diese kann auch über den in *Mathematica* zur Verfügung gestellten Operator *?name* abgerufen werden.

AmplitudeSet

AmplitudeSet[N] erstellt die Menge der nichtlinearen Amplitudenfunktionen $\{A_k\}$, k = 1..N, wobei die Indizierung mit jener aus Abschnitt 3.4. konform geht.

Coefficient

Coefficient[e, z_0, v_0] bestimmt den numerischen Wert der Konstanten C[1] und C[2] anhand der Systemkonfiguration, definiert durch die Exzentrizität e, der Anfangsamplitude z_0 und der Anfangsgeschwindigkeit v_0 des dritten Körpers.

CourantSnyderTransform

CourantSnyderTransform wird in 5.5.3. beschrieben. Die Funktion führt die Courant & Snyder Transformation an allgemeinen Differentialgleichungen durch.

CtoKLM

CtoKLM[i,j] ersetzt die auftretenden Konstanten C[1] und C[2] durch jene, wie sie in Kapitel 3.4.2. anhand (131.1-131.5) definiert sind. Für den Fall A_k sind i = 1 und j = 2, für B_k ist i = 2 und j = 1 zu setzen.

DifferentialOrder

DifferentialOrder[expr] wird in vielen Funktionen des SDS verwendet und bestimmt die höchste auftretende Ableitung in einem Ausdruck *expr*.

FrequencySet

FrequencySet[N] erstellt die Menge der nichtlinearen Phasenfunktionen $\{\psi_k\}$, k = 1..N, wobei die Indizierung mit jener aus Abschnitt 3.4. konform geht.

GetResults

GetResults lädt die vorberechneten Ergebnisse dieser Arbeit in den Arbeitsspeicher und weist sie den entsprechenden Objekten zu.

InhomogenPart

InhomogenPart wird für mehrere Funktionen verwendet und extrahiert den inhomogenen Anteil einer Differentialgleichung.

InversePerturbativeSeries

InversePerturbativeSeries[ser,y[x],n] erstellt zu einer Störreihe ser der Form $y_0[x] + y_1[x]\,\lambda + ... + O[\lambda]^m$ die Umkehrtransformation x[y] der Ordnung n in Form einer Störreihe $x_0[y] + x_1[y]\,\lambda + ... O[\lambda]^n$.

KeplerSolution

KeplerSolution[n] gibt die iterierte Form der Lösung der transzendenten Keplergleichung wieder.

KLMtoC

KLMtoC[i,j] ersetzt die in Abschnitt 3.4.2. definierten Konstanten K_p, L_p , M_p durch jene, in *Mathematica* verwendeten C[1] und C[2]. Wird die Funktion für A_k angewendet, so sind i = 1 und j = 2 zu setzen. Für die Amplitudefunktionen B_k umgekehrt (i = 2, j = 1).

LinearEquation

LinearEquation[N] gibt die linearisierte Bewegungsgleichung des Sitnikov Problems der Ordnung $O(e^N)$ wieder.

ParticularDSolve

ParticularDSolve ist der Kern des Algorithmus AmplitudeFunction und wird auch dort (5.4.4.) beschrieben.

PerturbativeSeries

PerturbativeSeries[f[x],λ,N] erstellt das Objekt $f_0[x] + f_1[x]\,\lambda + ... O[\lambda]^{N+1}$.

Rewrite

Rewrite[equ] setzt eine Gleichung gleich Null.

SignSet

SignSet[N] erstellt die Menge der Vorzeichenfunktionen {sgn[k]}, k = 1,..,N, wobei die Indizierung mit jener aus Abschnitt 3.4. konform geht.

Solution´A

Solution´A[k] gibt A_k, definiert in Abschnitt 3.4. wieder.

Solution´B

Solution´B[k] gibt B_k, definiert in Abschnitt 3.4. wieder.

Solution´Psi

Solution´Psi[k] gibt ψ_k, definiert in Abschnitt 3.4. wieder.

Solution´z

Solution´z[N] gibt die Lösung der Bewegungsgleichung des Sitnikov Problems in den Variablen z und t mit noch nicht bestimmten Konstanten C[1] und C[2] der Ordnung N wieder.

TauTransformed

TauTransformed[N] gibt die τ-transformierte Bewegungsgleichung des Sitnikov Problems bis zur Ordnung $O(\lambda^{N+1})$ wieder. Im Unterschied zu TauEquation[N] erfolgt die Sortierung nach dem Parameter λ.

TransformDependentVar

TransformDependentVar[equ,f[t],f[t]==F[f[t]]] transformiert eine Differentialgleichung equ der Funktion f[t] über die Relation f[t] = F[g[t]] oder $g[t] == G^{-1}[f[t]]$.

TransformIndependentVar

TransformDependentVar[equn,f[t],t[s]==F[s]] transformiert eine Differentialgleichung equ der Funktion f[t] über die Relation t[s]==F[s] oder $s[t] == F^{-1}[t]$.

$Derivation

Die Variable $Derivation beinhaltet die Zwischenschritte der letzten Berechnung, ausgeführt mit einem Objekt des SDS.

$Objects

Die Variable $Objects beinhaltet alle aktiven Objekte des SDS, die in den Hauptspeicher geladen wurden.

$Result

Die Variable $Results beinhaltet das letzte Ergebnis, ausgeführt mit einem Objekt des SDS, das keine automatische Zuweisung besitzt.

Abschließende Bemerkung

Alle Objekte des SDS wurden an den Ergebnissen dieser Arbeit und darüber hinaus getestet. Viele Funktionen (*TransformDependentVar, DifferentialOrder*, etc...) wurden abstrakter geschrieben, als für diese Arbeit notwendig, um sie für spätere Zwecke einsetzen zu können. Sie wurden nicht auf alle möglichen Fälle getestet, ihre Ergebnisse müssen für jede nicht in dieser Arbeit durchgeführte Anwendung auf Konsistenz und Richtigkeit getestet und möglicherweise modifiziert werden! Der Autor übernimmt keine Verantwortung für eventuell auftretende Fehler in Berechnungen und daraus folgende Schäden.

Das Package wurde für *Mathematica* V5 geschrieben und auch getestet. Für Adaption auf niedrigere Versionen siehe *Incompatible Changes* (*Mathematica* Help Browser / *Mathematica* Reference).

Updates, Diskussionsbeiträge und Verbesserungsvorschläge können über e-mail Kontakt mit dem Autor über die Adresse - lhotka@astro.univie.ac.at - bezogen bzw. eingebracht werden.