Das Kettenkarussell mit GiNaC

Thomas Orgis

23.01.2005

1 Einleitung

Ich möchte im Folgenden die Behandlung eines hamiltonschen Systems unter Benutzung der Open Source C++ Algebrabibiothek GiNaC[1] demonstrieren. GiNaC führt dabei die Ableitungen der Bewegungsgleichungen sowie Berechnungen aus und schreibt auch Formeln für LaTeX. Für die Plots der ermittelten Funktionen und errechneten Daten wird Gnuplot bemüht.

Anlass dieser Ausführungen ist übrigens das Computerpraktikum zur Einführung in die Nichtlinearen Dynamik WS2004/5 an der Universität Potsdam. Die Aufgabenstellung bestand in der computergestützten Betrachtung eines einfachen Modells für ein Kettenkarussel, die in Phasenportraits münden sollte.
Unter computergestützt ist im Kontext des Praktikums üblicherweise die Verwendung von Matlab oder Mathematica zu verstehen; jedoch zeigte sich Dr. Udo Schwarz meiner tendenziellen Abneigung gegen zu große monolithische Programme im Allgemeinen und der Syntax und Bedienung der beiden Ms im Speziellen gegenüber durchaus aufgeschlossen, so dass ich mich also mit der hier vorgestellten Alternative befassen konnte.

Es ist erklärtes Ziel, aufzuzeigen, dass man solche Aufgaben auch gut ohne ein monolithisches Algebra-Numerik-Datenanalyse-Paket (folgend "AND-Paket") erledigen und dabei dem guten alten UNIX-Prinzip der geteilten Aufgaben (ein Programm macht im Wesentlichen nur eine Sache aber diese dafür gut) treu bleiben – oder es neu entdecken – kann.

Eine Übersicht der Komponenten des genutzten "*des*integrierten" AND-Paketes:

- GNU C++ Compiler und Anhang (Standardbibliotheken, make)
- Bibliothek CLN[2], optional mit GMP[3] darunter
- GiNaC C++ Algebrabibliothek für ... nunja, Algebra - oder auch "symbolische Manipulationen"
- Gnuplot[4] für graphische Darstellungen
- LaTeX für ordentliche Dokumentation
- PERL-Interpreter und ein paar Skripte als verbindender "Klebstoff"

[1] GiNaC is Not a CAS; siehe http://www.ginac.de
[2] Class Library for Numbers; http://www.ginac.de/CLN
[3] GNU Multiple Precision Arithmetic Library; http://www.swox.com/gmp/
[4] http://www.gnuplot.info

Dazu gehört in meinem Fall noch das UNIX-artige Betriebssystem, welches aber keine zwingende Vorraussetzung ist, da diese Komponenten grundsätzlich plattformübergreifend verfügbar sind. Das begleitende Programmpaket (C++ Code, Perl-Skripte, Makefile, ...) ist freilich unter GNU/Linux entwickelt und auch nur in dieser Umgebung getestet worden. Es wurden erfolgreich die GCC-Versionen 3.2 und 3.3.3 eingesetzt.
Dazu sei allerdings noch angemerkt, dass ein Optimierungslevel (Option -O2 oder auch -O3) für die mathematischen Bibliotheken mitunter zum erfolgreichen Kompilieren notwendig ist. Weiterhin wurde von mir eine aktuelle CVS-Verison von Gnuplot eingesetzt (auf dem Weg zu Version 4.1), welche nun doch endlich eine direkte Farbwahl für Punkte und Linien zulässt.
Abgesehen vom CVS-Gnuplot könnte Quantian[5] zum Ausprobieren des folgend Beschriebenen ohne zu viel Kopfschmerzen hilfreich sein. Es handelt sich dabei um eine Knoppix-Variante, die um mathematisch-wissentschaftliche Software (u.a. eben GiNaC) erweitert wurde. Dies bedeutet: DVD / CD brennen, davon booten und ohne aufwendige Installation mit einem kompletten Linux-System arbeiten.

2 Crash-Kurs GiNaC

2.1 Einleitung

GiNaC ist eine C++-Bibliothek für symbolische Mathematik. Sie definiert keine eigene Sprache wie die "normalen" Computeralgebrasysteme (CAS), sondern implementiert entsprechende Datentypen und Methoden in Form einer Klassenbibliothek in der etablierten Programmiersprache C++. Je nach Geschmack ist sie eine Komponente, um nun tatsächlich ein CAS zu bauen[6] oder ein direkteres Werkzeug als Ersatz für ein solches.

Anhand des zur Lösung des Karusselproblems geschriebenen Programmes werde ich nun einige Aspekte des Einsatzes von GiNaC demonstrieren. Es mag für den Praktiker hilfreich oder auch schon ausreichend sein, sich einfach die Hauptdatei `rechner.cpp` sowie die Header in `include/` und die Quellen in `parts/` (diese Angaben beziehen sich auf das Komplettpaket, von dem diesen Dokument ein Teil ist) direkt anzusehen.
Die wesentlichen Quellen tauchen auch in diesem Text auf und werden hoffentlich zu Genüge erklärt, wo dies angemessen erscheint.
Der geneigte Leser sollte allerdings schon grundsätzlich wissen, wie man ein C++-Programm schreibt. Ich werde hier nicht erklären, was `cout << "Hallo Welt!" << endl;` bedeutet...

Die Arbeit mit den mathematischen Objekten wird dadurch vereinfacht, dass nach Möglichkeit die C-Operatoren wie `+`, `-`, und `==` sinngemäß für GiNaC-Objekte überladen sind. Ausnahme ist der Potenzoperator, welcher durch die Funktion `pow()` umgesetzt wird. Es bleibt also prinzipiell dabei, dass man die Basismathematik "normal" niederschreibt – wie man es von Rechnungen in C kennt.

2.2 Der Weg zum Anfang

Aller Anfang eines C++-Programmes ist die Einbindung der Schnittstellen (Header), mit denen man arbeiten möchte. Hier ist die Datei `common.h`, in der die GiNaC-relevanten Stellen klar ersichtlich sind:

```
//fuer mkdir
#include <sys/stat.h>
#include <sys/types.h>
```

[5] http://dirk.eddelbuettel.com/quantian.html
[6] Octave-forge (http://octave.sourceforge.net) beinhaltet u.a. eine Einbindung von GiNaC in das ansonsten rein numerische Octave (http://www.octave.org)

```
#include <stdlib.h>

//Ein-/Ausgabe
#include <iostream>
#include <sstream>
#include <fstream>

//GiNaC
#include <ginac/ginac.h>

//zur Bequemlichkeit
using namespace std;
using namespace GiNaC;

//global gueltige symbole, definiert in common.cpp
//algebraische Symbole (Variablen und Parameter)
extern symbol alpha;
extern symbol p;
extern symbol Omega;
extern symbol onull;
extern symbol l;
extern symbol m;
extern symbol A;
```

Die ersten 6 `#include`s sollten niemanden verwirren, der bis hier weiter gelesen hat: Ich möchte nur Verzeichnisse anlegen bzw. überhaupt etwas anrichten können und auch etwas bequeme Dateioperationen betreiben.
Darunter sieht man schnell: Wenig überraschend stellt `ginac/ginac.h` das GiNaC-Interface zur Verfügung. Zu beachten ist allerdings, dass alle GiNaC-Symbole erstmal im gleichnamigen Namensraum leben. Nachdem dieser betreten ist, liegt alles etwas näher. Ansonsten müsste vor die GiNaC-spezifischen Datentypen und sonstigen Symbole der Präfix "`GiNaC::`".

2.3 Drei Typen und andere im Hintergrund

Es gibt viele Klassen, darunter auch so schöne Dinge wie Matrizen oder auch Integrale... für den Anfang reichen aber auch wenige grundlegende: **Symbole**, **Ausdrücke** mit diesen und **Relationen** zwischen Ausdrücken.
Nebenbei gibt es dann noch tatsächlich Nummern für den Fall, dass es am Ende eine Zahl geben soll, und Konstanten, die hier lediglich in Form von `Pi` vorkommen und nicht viel Aufregendes tun, außer sich selbst darzustellen.

2.3.1 Symbole

In `common.h` ist dann auch schon der erste der beiden wesentlichen Datentypen zu sehen: Symbole, genannt `symbol`. Diese stellen genau das dar, was man sich darunter vorstellt: einzelne Variablen.
Diese sind aber im Detail deutlich zu unterscheiden von simplen C-Variablen der Typen `int` oder `double`. Man behandelt sie zwar praktisch sehr ähnlich, muss sich aber der Tatsache bewusst sein, dass sie tatsächlich strukturierte Objekte sind. In der Struktur steckt die eigentliche Identifizierung eines Symbols; diese wird auch in einen Ausdruck aufgenommen – nicht die C-Variable selbst.

Bevor ich weiter Unsicherheit verbreite, sehen wir uns aber erstmal ein paar Deklarationen in `common.cpp` an:

```
#include "../include/common.h"

//algebraische Symbole (Variablen und Parameter)
symbol alpha("alpha");
symbol p("p_a","p_\\alpha");
symbol Omega("Omega");
symbol onull("omega_0","\\omega_0");
symbol l("l");
symbol m("m");
symbol A("A");
```

Die einfachste sinnvolle Deklaration eines `symbol`s der Form `symbol varname(''Name'')` enthält den Namen der C-Variable, `varname`, und als Konstruktorparameter eine textuelle Repräsentation dieser Variablen. Diese wird vor allem in textuellen Aus- und Eingaben (und sei es die Konsole) verwendet, um das Symbol zu bezeichnen.

Als zweite sinnvolle Variante tritt der Konstruktor mit zwei Parametern auf, wobei der zweite eine Repräsentation in LaTeX-Code beinhaltet, falls diese sich von der normalen Text-Variante unterscheidet. Praktischerweise sorgt eine Automatik dafür, dass griechische Buchstaben auch meist in LaTeXals solche zu erkennen sind: Dass die Variable `Omega` geTEXt

```
\Omega
```

ist, brauche ich nicht extra zu erwähnen. Wie man nun an eine LaTeX-Ausgabe kommt, ist in 2.4.3 zu sehen.

In der wirklich einfachsten Deklarationsform gibt man gar keine Parameter an und bekommt dann eine automatisch generierte Textrepräsentation der Form `symbol123` bei Ausgaben vorgesetzt. Das will man selten.

Achtung! Symbole mit gleichem Namen und gleicher Repräsentation müssen nicht gleich sein!
Die C-Variablennamen verschwinden beim Kompilieren und die textuellen Repräsentationen zeigen sich nur im ... nunja... Text. Tatsächlich unterschieden werden die Symbole anhand interner Bezeichnungen und die Identität von Symbolen gehorcht der C-Logik. Wenn ich also lokal in einer Funktion ein `symbol x(''x'')` definiere und in einen Ausdruck packe, welcher einer anderen Funktion (also einem anderen Kontext) übergeben wird, die ebenfalls ein solches Symbol `x` kennt, dann sehen sie zwar in einer Textausgabe gleich aus, sind aber verschiedene Dinge.

Einfache Methoden zur Vermeidung solcher Zweideutigkeiten sind:

- **globale Variablen:** Für kleinere Projekte kann man auch mal ein paar globale Variablen definieren - auch wenn diese oftmals als prinzipiell böse Sache dargestellt werden. Im überschaubaren Rahmen finde ich sie durchaus zweckmäßig – weshalb ich hier auch diesen Weg gewählt habe.

  Dazu sind aber die `extern`-Schlüsselwörter in `common.h` und die tatsächliche Definition in `common.cpp` lebenswichtig. Ohne diese würde der C-Compiler für jede Kompiliereinheit, die diese Header-Datei einbindet, eine *eigene und unabhängige* Instanz eines jeden Symbols erzeugen.

- **Durchreichen der Symbole als Funktionsargumente:** Dadurch erhält die aufgerufene Funktion eine exakte Kopie oder besser per Referenz das gleiche Objekt. Dies ist schön sauber aber mitunter unbequem. Für größere Projekte empfiehlt sich der objektorientierte Anzatz auch für die Anwendung, wo dann Symbole im Objekt gespeichert sind und so für alle Objektmethoden in natürlicher Weise eindeutig zur Verfügung stehen.

Symbole sind ganz einfach und friedlich zu verwenden, solange man sicherstellt, dass man wirklich mit den richtigen arbeitet.

2.3.2 Ausdrücke

Ausdrücke `ex` stellen allgemeine mathematische Terme dar - Verknüpfungen von Symbolen und anderen Ausdrücken über Funktionen und Operatoren (welche natürlich Funktionen darstellen). Einfachste Ausdrücke bestehen nur aus einem Symbol (ein `symbol` ist also kompatibel mit einem `ex`), während kompliziertere in naheliegender Weise konstruiert werden.

Als Beispiel diene die Definition des hamiltonschen Karussells mit dem Potential $V(\alpha)$ und der passenden Hamiltonfunktion in den kanonischen Variablen α (Auslenkungswinkel) und p_α (dem zugehörigen Winkelimpuls):

$$V(\alpha) = ml^2\omega_0^2\left[(1-\cos(\alpha)) - \Omega\left(A + \frac{1}{2}\sin\alpha\right)\sin\alpha\right] \quad (1)$$

$$H(\alpha, p_\alpha) = \frac{p_\alpha^2}{2ml^2} + V(\alpha) \quad (2)$$

Implementiert habe ich diese über Funktionen, die Ausdrücke für das Potential und die Hamiltonfunktion zurückgeben:

```
#include "../include/common.h"

//das Potential
ex poti()
{
   return m * pow(l,2) * pow(onull,2) *
   ( (1-cos(alpha)) - Omega * (A + (ex)1/2 * sin(alpha)) * sin(alpha) );
}

//die Hamilton-Funktion
ex hamilton()
{
   return pow(p,2) / (2*m*pow(l,2)) + poti();
}
```

Ich kann also einfach mit `ex ham = hamilton();` einen Ausdruck definieren und mit der Hamilton-Funktion belegen. Die Konstruktion mit den arithmetischen C-Operatoren bedarf aber auch etwas Umsicht: Man beachte beim Potential den expliziten Typecast `(ex)1/2` - ohne diesen würde hier eine Null stehen, da C dann zuerst im Ganzzahl-Kontext 1 durch 2 teilt und dann das Ergebnis 0 in die GiNaC-Welt übertritt.

Eine andere Möglichkeit zu Vermeidung dieses Problems wäre auch die Änderung der Reihenfolge zu `sin(alpha)*1/2`, was C zum impliziten Cast bewegt, da nun eindeutig `(ex)`-Kontext vorliegt (die Multiplikation wird zuerst ausgeführt). Wen das überrascht, den möchte ich sanft auf das C/C++-Buch seiner Wahl verweisen.

Ausdrücke sind die Objekte, mit denen man tatsächlich die meiste Zeit arbeitet. Es gibt vielerlei Methoden für sie, die all die mathematischen Operationen repräsentieren, die das Leben so lebenswert machen – einige werden demnächst noch auftauchen.

2.3.3 Exakte oder zumindest sehr genaue Nummern

In Zusammenarbeit mit CLN verfügt GiNaC über die Klasse `numeric`, welche im Idealfall eine exakte Nummer repräsentiert (z.B. rationale Zahlen als Brüche von "großen" Ganzzahlen). Diese Klasse begegnet einem spätestens bei der numerischen Evaluierung von Ausdrücken. so verwende ich dieses Konstrukt zum Erhalt von Fließkommazahlen mit (für mich) ausreichender Genauigkeit:

```
double wert = ex_to<numeric>(ausdruck.evalf()).to_double();
```

Darin ist zum Einen die Fließkommaevaluierung von nicht exakt Anzugebendem wie der Zahl π mittels `ausdruck.evalf()` (`evalf()` ist also eine Methode der Klasse `ex`, siehe 2.4) enthalten und zum Anderen die Wandlung des Ergebnisses zu einem `numeric`-Objekt mittels `ex_to<numeric>` und folgende Stauchung auf eine `double`-Zahl.

Man sieht: Einfache Zahlen sind nicht das Einfachste mit GiNaC!

2.3.4 Relationen

Ein `relational` repräsentiert eine Relation zwischen zwei Ausdrücken – einfachste Form ist eine Wertzuweisung zu einer Variablen als Gleichung: `relational belegung = x == 4;`.
Relationen kommen hier lediglich in ihrer Funktion als Argument der `subs`-Methode zum Einsatz, siehe 2.4.2.

2.4 Grundlegende Methoden für Ausdrücke

Operationen auf mathematische Ausdrücken stellen sich mit GiNaC als Methoden von entsprechenden Objekten dar - im Beispipel nur Instanzen von `ex`. Nach dem objektorientierten Denkmodell sagen wir also einem Ausdruck, er soll sich doch bitte in eine Reihe entwickeln - wie das geht, weiß er per Definition.

Der Umgang von Erst/Zweit/Drittsemester-Physikstudenten mit Mathematik-Klausuren wäre deutlich entspannter, verhielten sich papiergebundene Ausdrücke dort wie die digitalen hier;-)

Es ist allerdings recht zweckmäßig, dass die (zumindest mir begegneten) Methoden von GiNaC-Ausdrücken diese nicht direkt verändern, sondern den resultierenden Ausdruck als Wert zurückgeben. Wenn wir also vom Ausdruck eine Reihenentwicklung erwarten, dann wollen wir, dass er uns eine herausgibt und sich nicht zu einer macht.

2.4.1 Differenzieren

Ist man mit dem objektorientierten Weltbild vertraut, so überrascht die Funktion des folgenden Quelltextes nicht.

```
#include "../include/common.h"

ex poisson(ex F, ex H)
{
    return F.diff(alpha) * H.diff(p) - F.diff(p) * H.diff(alpha);
}
```

Der Physiker erkennt darin hoffentlich die Poissonklammer

$$\{F, H\} = \frac{\partial F}{\partial \alpha} \cdot \frac{\partial H}{\partial p_\alpha} - \frac{\partial F}{\partial p_\alpha} \cdot \frac{\partial H}{\partial \alpha} \tag{3}$$

mit den hier relevanten kanonischen Variablen α (Ort) und p_α (Impuls).

`y.diff(x)` ist also die erste Ableitung von `y` nach `x`. Allgemein bedeutet `y.diff(x,n)` die n-te Ableitung nach `x`. Gibt es da noch etwas zu sagen?

2.4.2 Substituieren

Des öfteren möchte man Ausdrücke in andere einsetzten bzw, Teile dieser Ausdrücke durch speziellere Angaben ersetzen. Dies erreicht die Methode `subs`, die eine Relation oder eine Liste dieser in dem Konstrukt `lst(rel1,rel2,...)` als Argumente verarbeitet. Die zweite Variante ist in der Einsetzungsfunktion in `subs.cpp` zu sehen:

```
#include "../include/common.h"

ex subs(ex &in, double Omegaval, double Aval)
{
    return
    in.subs(lst(A == Aval, Omega == Omegaval, m == 1, l == 1, onull == 1));
}
```

Hier wird ein neuer Ausdruck konstruiert, welcher durch Einssetzung der Masse und Einheiten sowie der gewünschten Werte für Ω und A aus `in` hervorgeht.

2.4.3 Formatierte Ausgabe

Ich sprach früher davon, dass Symbole wissen, wie sie in LaTeX auszusehen haben. Um in den Genuss dieser Repräsentation zu kommen, benötigt man einen Ausgabestream, der mittels `<< latex` in den LaTeX-Modus geschalten wurde – dies zeigt `totex.cpp`:

```
#include "../include/common.h"

//Einen Ausdruck in eine LaTeX-Datei schreiben
```

```
void totex(ex expr, const char * name)
{
   ofstream lf(name); //Datei oeffnen
   //Mit dem Befehl << latex wird die Ausgabe in den LaTeX Modus geschalten.
   //Eine ex weiß, wie sie in LaTeX auszusehen hat!
   lf << latex << expr << endl;
   lf.close();
}
```

3 Die Arbeit

3.1 Hamilton und Potentiale

So, nun sind die Werkzeuge da und es wurde auch schon das System dem Programm vorgestellt. Nun sei es aber noch einmal ausformuliert:

Gegenstand der Betrachtung ist ein idealisiertes reibungsloses (konservatives) Kettenkarussel[7] mit den normierten (dimensionslosen) Parametern Auslegerlänge $A = \frac{a}{l}$ und Drehfrequenz $\Omega = \frac{\omega}{\omega_0}$ sowie einer Kindmasse m (jemand sollte ja mit dem Karussel fahren, damit die Rechnung auch realitätsnah ist;-).

Die ursprünglichen Gleichungen (Potential, Hamilton) dafür haben wir schon in 2.3.2 gesehen. Hier erscheinen sie in der Form, wie sie GiNaC sieht:

$$V(\alpha) = \omega_0^2(1 - \Omega\sin(\alpha)(A + \frac{1}{2}\sin(\alpha)) - \cos(\alpha))ml^2 \quad (4)$$

$$H(\alpha, p_\alpha) = \omega_0^2(1 - \Omega\sin(\alpha)(A + \frac{1}{2}\sin(\alpha)) - \cos(\alpha))ml^2 + \frac{1}{2}\frac{p_\alpha^2}{ml^2} \quad (5)$$

Man merkt: GiNaC hat sie umgeformt. Das passiert mit jedem eingegebenen Ausdruck; er wird umgeformt (üblicherweise vereinfacht), um optimal verarbeitet werden zu können. Man muss nun nicht streiten, welches die beste Form ist; wichtig ist, dass sie algebraisch äquivalent bleiben und das ist hier der Fall[8].

Es ist ein Leichtes, nun für ein paar Parameter die Potentiale zu berechnen – siehe Abb. 1. Wenn wir das Potential eines konservativen hamiltonischen Systems sehen, dann bildet sich im Hinterkopf ja auch gleich ein Phasenportrait: Von stabilen Oszillationen (Ellipsen) innerhalb der Töpfe (Minima des Potentials = stabile Fixpunkte) über Separatritzen, welche die Maxima (instabile Fixpunkte) austesten, bis hin zu den "Überfliegern" mit noch mehr kinetischer Energie; beim gewählten Beispiel wäre das ein an der Kette um die Aufängung rotierender Passagier - wenn dies denn baulich möglich ist.

Für die qualitativen Betrachtungen wurden die Konstanten m, l und ω_0 gleich 1 gesetzt - was man auch mit der Wahl von Einheiten umschreiben könnte (diese Praxis sollte schon in der Funktion `subs()` aufgefallen sein).

[7] Mit Kette ist amgesichts des Potentials ein starrer Stab gemeint, der wie eine Kette *aussieht*. Nur wenn er auch noch anfängt zu rasseln, dann ist das Modell in Schwierigkeiten.

[8] GiNaC führt Umformungen aus, die fast überall (außer auf Nullmengen) algebraisch korrekt sind. $\frac{x}{x}$ wird zu 1, was eigentlich Sinn macht aber prinzipiell bei $x = 0$ nicht so einfach ist.

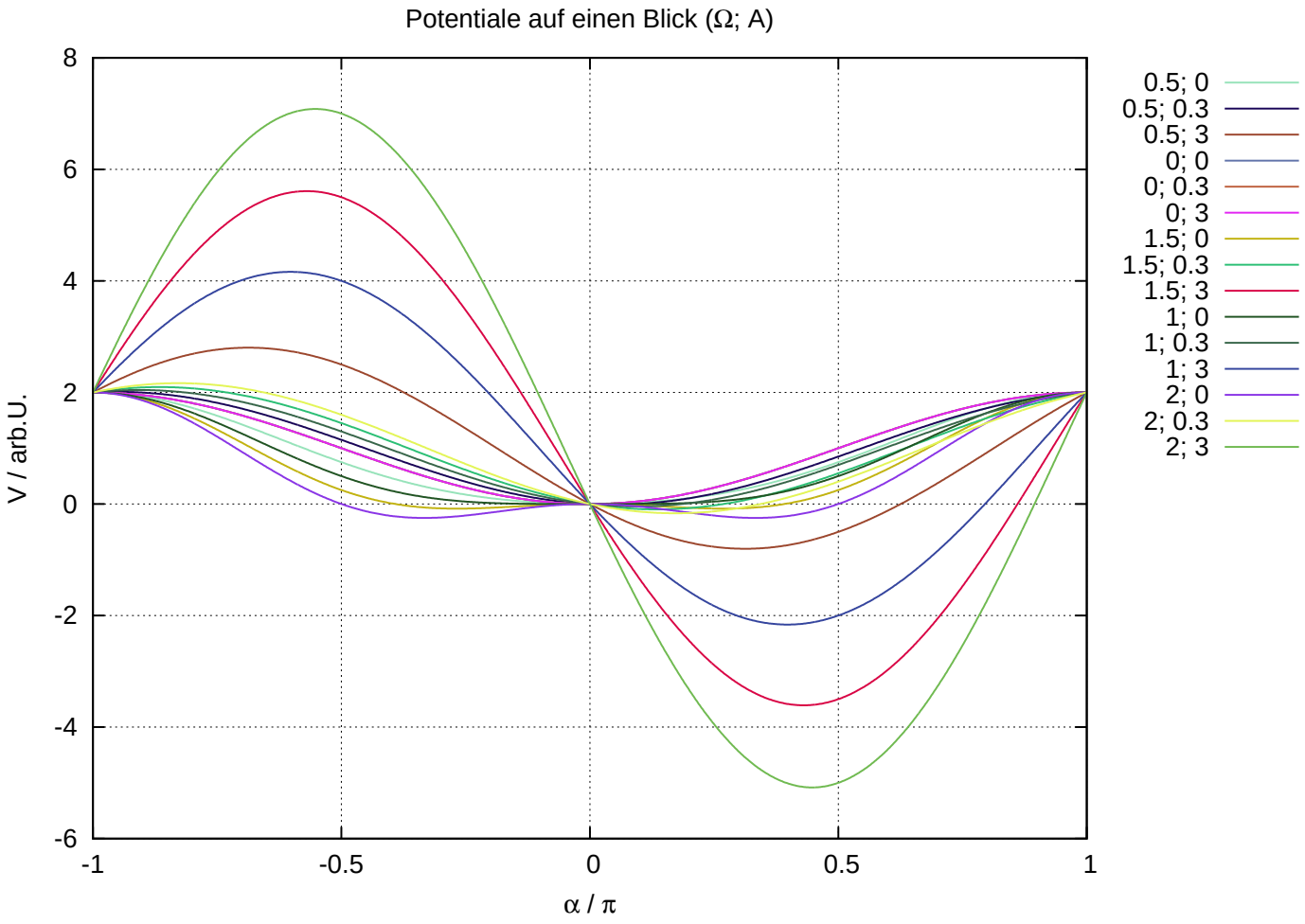

Abbildung 1: Ein Sammelblick auf einige Potentiale

3.2 Bewegungsgleichungen und Integration

Die ersten Zeilen von `main()` in `rechner.cpp` – auch zu sehen in Abschnitt 5 – produzieren über die Poissonklammern die Bewegungsgleichungen:

$$\begin{aligned}
\dot{\alpha} &= \{\alpha, H\} && (6)\\
&= \frac{p_\alpha}{ml^2} && (7)\\
\dot{p}_\alpha &= \{p_\alpha, H\} && (8)\\
&= -\omega_0^2 ml^2(-\frac{1}{2}\Omega\cos(\alpha)\sin(\alpha) + \sin(\alpha) - \Omega\cos(\alpha)(A + \frac{1}{2}\sin(\alpha))) && (9)
\end{aligned}$$

Das ist ein schön gekoppeltes System, welches folgend numerisch behandelt werden soll, um einen Eindruck vom Phasenraum zu erhalten. Da hier der Gedanke und nicht maximale Effizienz und Genauigkeit zählt, werden dazu direkt die GiNaC-Ausdrücke benutzt und ein einfacher Runge-Kutta-Integrator implementiert (`integrate.cpp`):

```
#include "../include/common.h"

void integrate(ostream &of, ex &ap, ex &pp,
               double av, double pv, double step, long num)
{
    //ap = f(alpha,p)
    //pp = f(alpha,p)

    ex ka[4];
    ex kp[4];
    ex nexta;
    ex nextp;

    //Uebersichtlicherweise wird der Winkel in Vielfachen von Pi angegeben.
    //a = b*pi -> apunkt = bpunkt*pi -> bpunkt = apunkt/pi ...
    ex pi_ap = ap.subs(alpha == alpha*Pi) / Pi;
    ex pi_pp = pp.subs(alpha == alpha*Pi);

    //Dateikopf und Startpunkt
    of << "#alpha/pi  p_alpha" << endl;
    of << av << "  " << pv << endl;

    //und nun num Punkte dazu
    for(int i = 0; i < num; ++i)
    {

        /*
          Berechnung mit GiNaC-Ausdruecken.
          Natuerlich is dies alles reichlich ineffizient, wenn wir an der hohen
           Genauigkeit gar nicht interessiert sind. Einfach double-Nummern werden
           sicherlich schneller mit einer "einfachen" C-Funktion erzeugt, jedoch
```

```
        muesste man dafuer diese Funktion erzeugen, was automatisch nicht so
        einfach ist.
        Somit ist es fuer den Einstieg schneller, die Operationen
        brav symbolisch anzulassen.
    */

    ka[0] = (ex)step * pi_ap.subs(lst( alpha == av, p == pv ));
    kp[0] = (ex)step * pi_pp.subs(lst( alpha == av, p == pv ));

    ka[1] = (ex)step *
            pi_ap.subs(lst( alpha == av + ka[0] / 2, p == pv + kp[0] / 2 ));
    kp[1] = (ex)step *
            pi_pp.subs(lst( alpha == av + ka[0] / 2, p == pv + kp[0] / 2 ));

    ka[2] = (ex)step *
            pi_ap.subs(lst( alpha == av + ka[1] / 2, p == pv + kp[1] / 2 ));
    kp[2] = (ex)step *
            pi_pp.subs(lst( alpha == av + ka[1] / 2, p == pv + kp[1] / 2 ));

    ka[3] = (ex)step * pi_ap.subs(lst( alpha == av + ka[2], p == pv + kp[2] ));
    kp[3] = (ex)step * pi_pp.subs(lst( alpha == av + ka[2], p == pv + kp[2] ));

    nexta = av + ( ka[0] + 2*ka[1] + 2*ka[2] + ka[3]) / 6;
    nextp = pv + ( kp[0] + 2*kp[1] + 2*kp[2] + kp[3]) / 6;

    /*
        Ein evtl. enthaltenes Pi wird mittels evalf() neutralisiert, bevor eine
        numeric-Nummer daraus erzeugt werden kann!
        Dann einfache Auswertung als double.
        Hier werfen wir einen großen Haufen Genauigkeit weg!
    */
    av = ex_to<numeric>(nexta.evalf()).to_double();
    pv = ex_to<numeric>(nextp.evalf()).to_double();

    /*
        Ebenfalls gehe ich von einem Winkel alpha aus, in dem das System auch
        periodisch ist.
        Soll heißen: -Pi bis +Pi reicht mir.
    */
    of << ( av > 1 ?
            av - (long)((av+1)/2)*2 :
            ( av < -1 ? av - (long)((av-1)/2)*2 : av)
          ) << "   " << pv << endl;
  }
}
```

Man beachte die Winkeltransformation: α wird zum einen in Einheiten von π betrachtet und dann auch in der Ausgabe auf das Intervall $[-1;1]$ beschränkt mit der Feststellung, dass Winkel, die sich um geradzahlige Vielfache von 2π unterscheiden, äquivalent sind.
Somit versuche ich es nach einem Blick auf die Potentiale mit Startimpulsen $p_\alpha \in [-10;10]$ und Startwinkel $\alpha_0 = 0$. Die Hoffnung besteht, dass einige der produzierten Trajektorien ein qualitatives Phasenportrait erahnen lassen.

3.3 Phasenportraits

3.3.1 Entstehung

Schlusseendlich gibt es etwas zu sehen, nachdem die Kommandozeile für das in `rechner.cpp` stehende Hauptprogramm (siehe Abschnitt 5 oder direkt den Quelltext)

```
cd data; ../rechner 0 0.01 1000 20
```

für etwas Rechnerbeschäftigung gesorgt hat und in den Daten zu vielerlei Phasenportraits resultierte, welche folgend zu bewundern aber auch argwöhnisch zu beäugen sind.

Zur Graphenerstellung dient zum einen das PERL-Skript `plot.pl`, welches sich grob selbst erklärt (Parameter `-h`), in Verbindung mit dem `Makefile`, welches wiederum das Gnuplot-Frontend `gplot`[9] bemüht.
Ach ja – das aktuelle Gnuplot macht natürlich die eigentliche Arbeit; ebenso Ghostscript und die `netpbm`-Programme zur Erstellung der Bitmaps aus den mitunter recht großen und langwierig zu rendernden Postscript-Bildern.
Genaueren Informationen zur Verwendung dieser Komponenten suche man bitte im `Makefile` und in der Datei `LIESMICH`

3.3.2 Betrachtungen

Es ist langsam an der Zeit, sich anzusehen, was der `rechner` so produziert hat. In der als Inspiration dienenden Aufgabe war die Betrachtung von Phasenportraits für verschiedene Drehfrequenzen $\Omega \in \{0; 0.5; 1; 2\}$ und verschwindend kleinem ($A = 0$), kurzem ($A < 1$) sowie langem ($A > 1$) Ausleger gefordert. Dem folge ich hier und habe für A die Werte 0, 0.3 und 3 gewählt.
In den Bildern sind jeweils einige Trajektorien (startend bei $\alpha = 0$ mit verschiedenen Impulsen) durch die schwarzen Punkte dargestellt und in blau das entsprechende Potential darüber gelegt. Die Einheiten stellen sich so dar, dass man potentielle und kinetische Energie direkt vergleichen kann, indem man die Zahlen für das Potential und den Impuls vergleicht: 5 Impuls sind 5 Energie.

[9] Ich meine hier das im Quellpaket enthaltene von mir, nicht zu verwechseln mit dem unter http://gplot.sourceforge.net!

Portrait für $\Omega = 0; A = 3$**.** Dies ist stellvertretend für alle weiteren Situationen ohne Drehung, da die Auslegerlänge ohne Zentrifugalkraft belanglos ist.
Das Karussell stellt in einer solchen Konfiguration ein mathematisches Pendel dar, welches – genug Schwung vorausgesetzt – auch um die Aufhängung rotieren kann.
Wie gut zu sehen ist, sind die Schwingungen um den Nullpunkt zentriert.

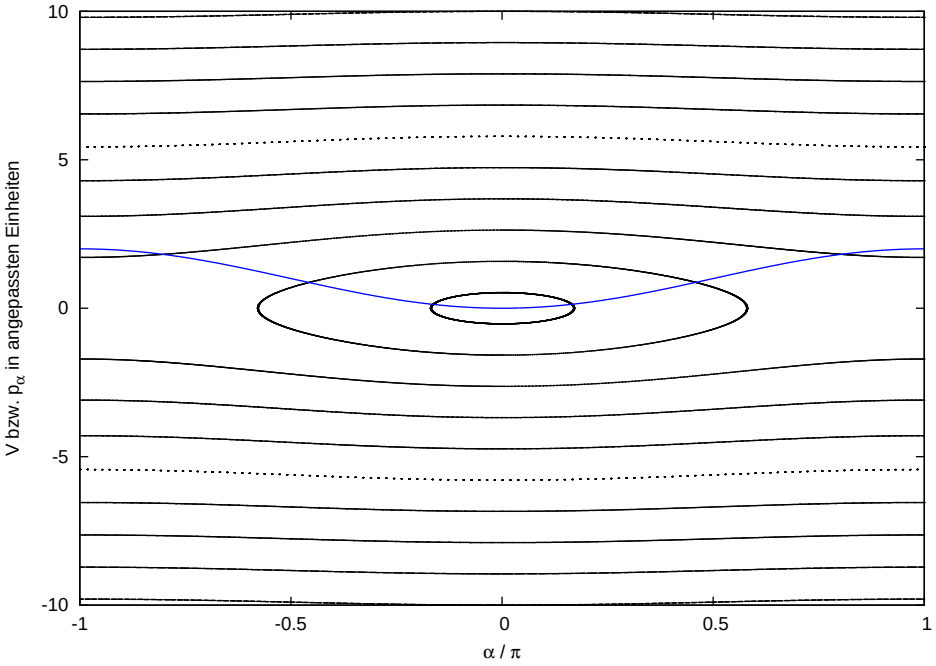

Abbildung 2: Portrait für $\Omega = 0; A = 3$.

Portrait für $\Omega = 0.5; A = 0$**.** Nun herrscht langsame Drehung ohne Ausleger. Was man bisher erkennt, ist, dass das Potential und die Ellipsen breiter werden.

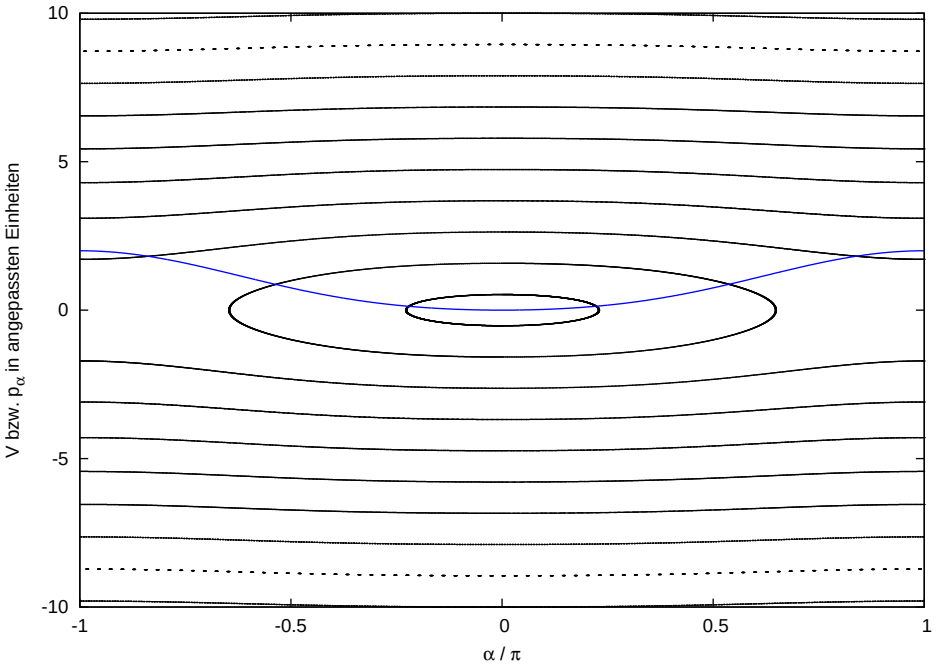

Abbildung 3: Portrait für $\Omega = 0.5; A = 0$.

Portrait für $\Omega = 1; A = 0$**.** Es geht weiter in die Breite...

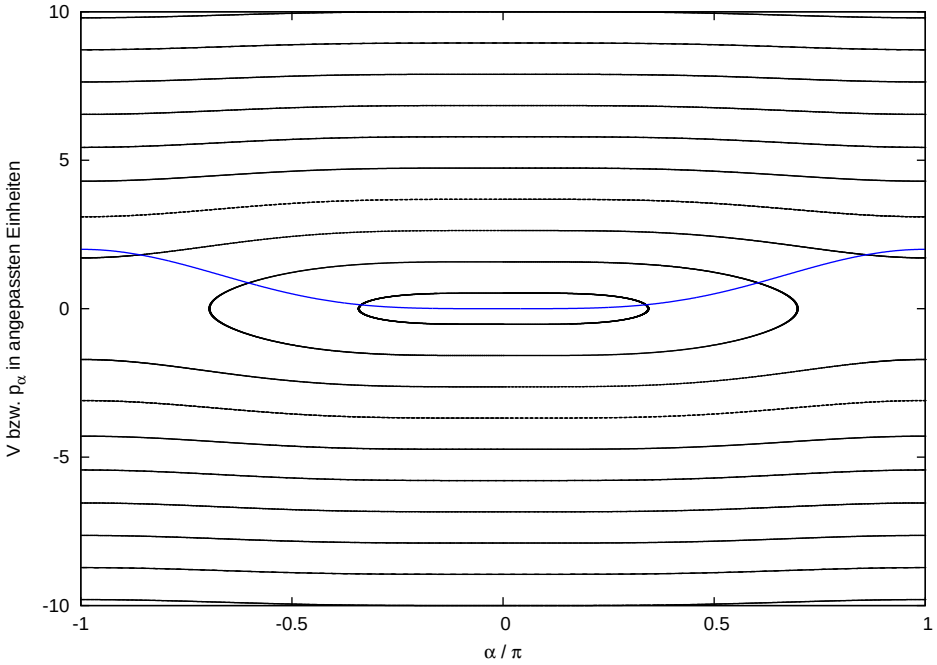

Abbildung 4: Portrait für $\Omega = 1; A = 0$.

Portrait für $\Omega = 2; A = 0$**.** Hier wird es deutlich: Es bilden sich zwei Potentialminima und folglich auch zwei Schwingungszentren heraus heraus.
Da noch kein Ausleger existiert, gibt es keinen Grund, die Symmentrie zwischen positiven und negativen Winkeln zu brechen. Dass sich stabile Gebiete (ortsgebundene Schwingungen anstatt wilde Rotation) bei Winkeln $\neq 0$ einstellen, sollte mit etwas Zentrifugaldenken klar sein. Anhand des Potentials zu erahnen sind weiterhin die instabilen Fixpunkte bei $(0; 0)$ und $(\pm 1; 0)$

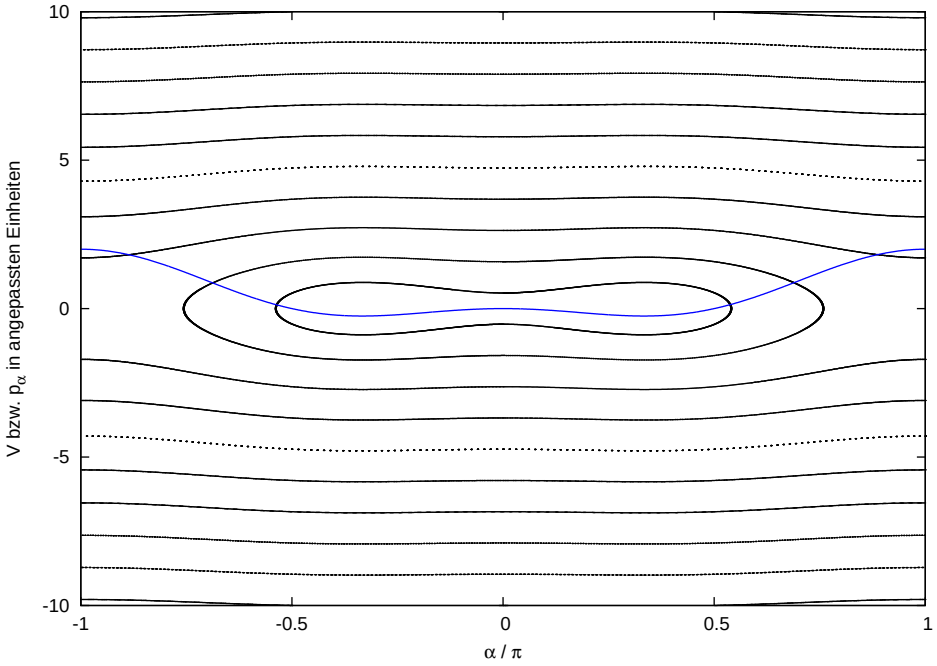

Abbildung 5: Portrait für $\Omega = 2; A = 0$.

Portrait für $\Omega = 0.5; A = 0.3$**.** Jetzt wird wieder langsam gedreht, aber als neues Element ist ein kurzer Ausleger angebaut. Hier ist noch nicht so viel zu erkennen. Aber es deutet sich an.

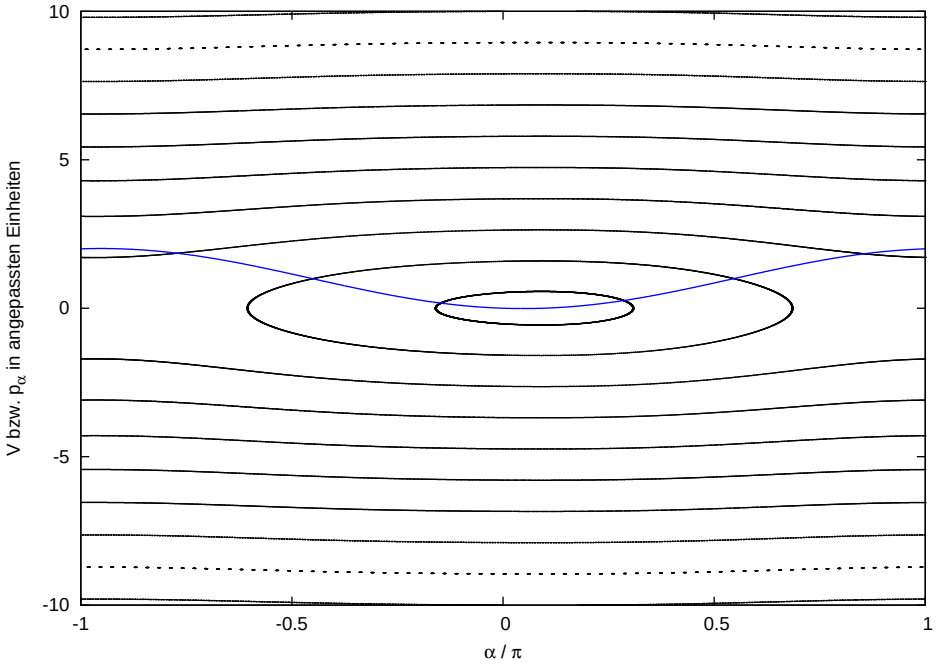

Abbildung 6: Portrait für $\Omega = 0.5; A = 0.3$.

Portrait für $\Omega = 1; A = 0.3$**.** Ja, der Ausleger macht natürlich Schluss mit der positiver-negativer-Winkel-Symmetrie (harmonisch ist das Potential natürlich auch nur ohne Drehung); das nun einsame stabile Gebiet verlagert sich zu positiven Winkeln.

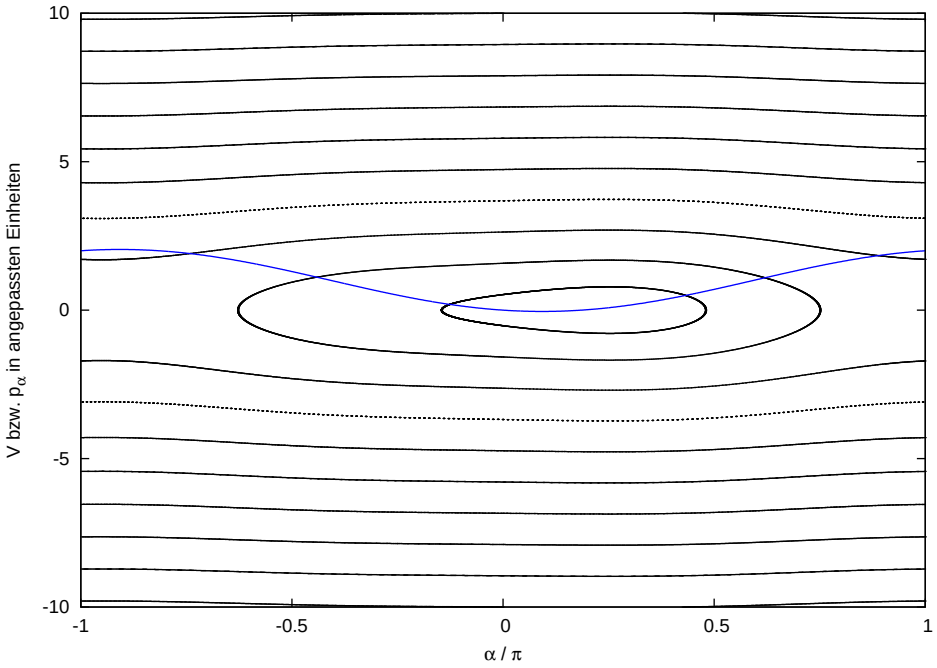

Abbildung 7: Portrait für $\Omega = 1; A = 0.3$.

Portrait für $\Omega = 2; A = 0.3$. Das sind nun wirklich keine harmonischen Ellipsen mehr – auch wenn mir die Reaktion der Trajektorien etwas extrem erscheint (wie sicher und genau ist denn nun die gewählte Integration?), ist die Tendenz klar.

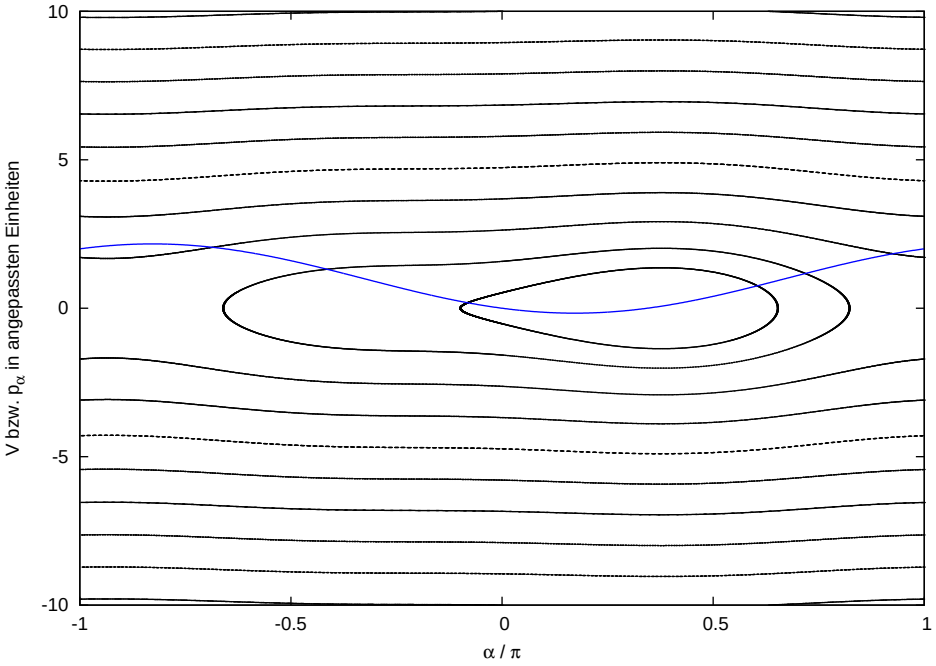

Abbildung 8: Portrait für $\Omega = 2; A = 0.3$.

Portrait für $\Omega = 0.5; A = 3$. Tja, prinzipiell kennen wir dieses Bild; der längere Ausleger macht es nur noch klarer.

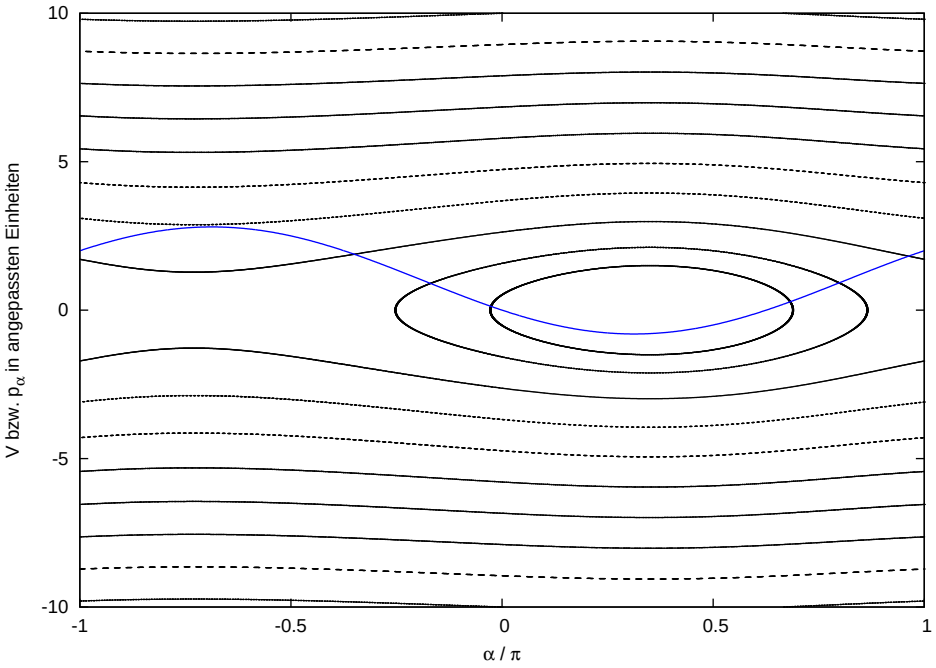

Abbildung 9: Portrait für $\Omega = 0.5; A = 3$.

Portrait für $\Omega = 1; A = 3$**.**
Runde Sache.

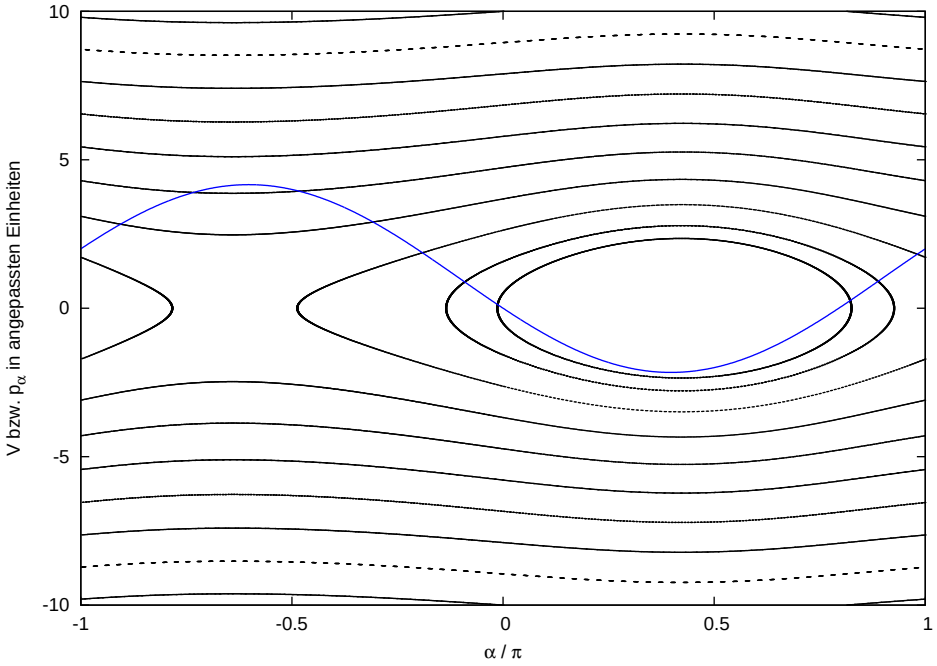

Abbildung 10: Portrait für $\Omega = 1; A = 3$.

Portrait für $\Omega = 2; A = 3$**.**
Noch Fragen?

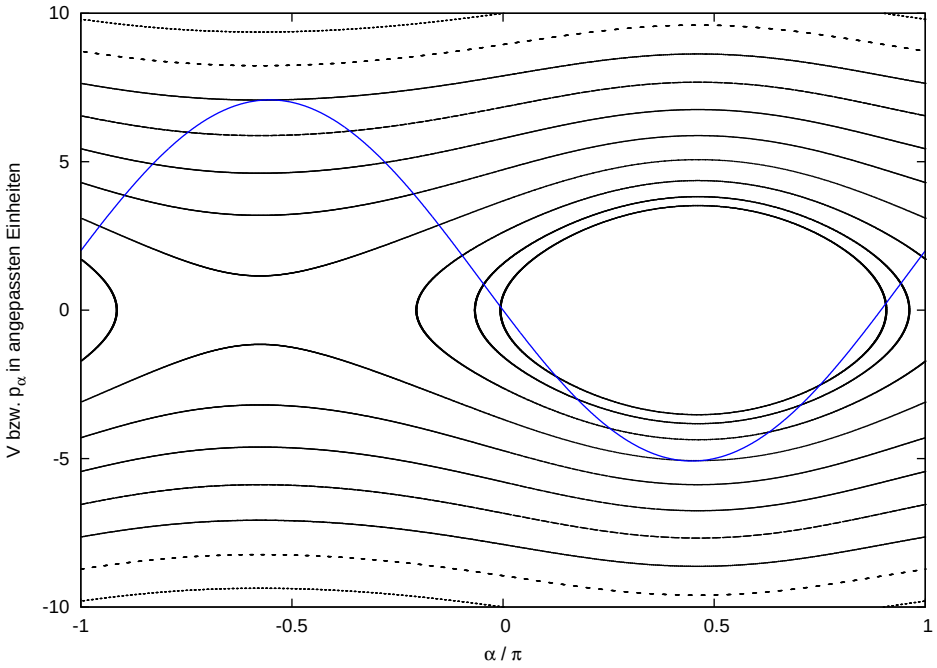

Abbildung 11: Portrait für $\Omega = 2; A = 3$.

4 Kleiner Zusatz: Reibung

So ein streng hamiltonisches System mit brav der Bewegung erhaltener Energie muss ja nicht alles sein, was man betrachtet. Dem aufmerksamen Beobachter mag in `rechner.cpp` die Einbezieung des Parameters `diss` aufgefallen sein, welcher etwas Reibung ins Spiel bringt, wenn er denn nicht gleich Null ist.
So gibt mir

```
cd dissdata; ../rechner 0.1 0.01 6000 2
```

dann etwas andere Bewegungsgleichungen[10]...

$$\begin{aligned}
\dot{\alpha} &= \{\alpha, H\} \\
&= \frac{p_\alpha}{ml^2} \\
\dot{p}_\alpha &= \{p_\alpha, H\} - \mu\dot{\alpha} \\
&= -\omega_0^2 ml^2(-\frac{1}{2}\Omega\cos(\alpha)\sin(\alpha) + \sin(\alpha) - \Omega\cos(\alpha)(A + \frac{1}{2}\sin(\alpha))) - (0.10000000000000000555)\frac{p_\alpha}{ml^2}
\end{aligned}$$

...und auch ein etwas zielgerichteteres Phasenportrait (Abb. 12).

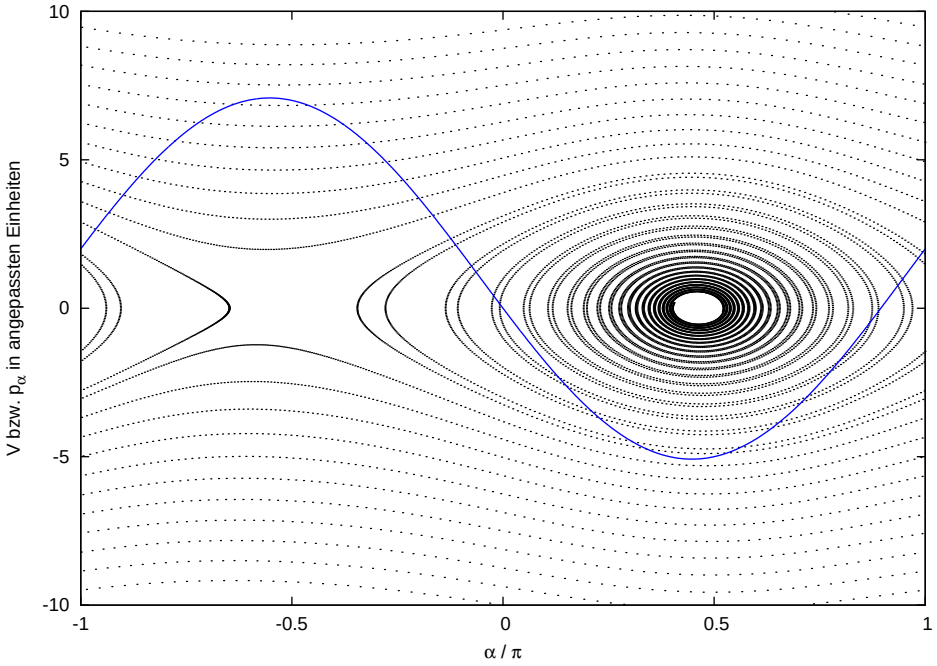

Abbildung 12: Phasenportrait für $\Omega = 2; A = 3$ mit etwas Reibung

Hier sind tatsächlich nur 2 Trajektorien zu sehen – eine von (0,-10) und eine von (0,10) startend –, die sich durch die Reibung beide dem selben Punkt nähern (das Auge ist natürlich nur noch weiß, weil die Trajektorien nur eine bregenzte Zahl Punkte haben).
Ein "reales" Karussell mit anständigem Ausleger und ordentlicher Drehung erreicht also nach gewisser Zeit einen stabilen Winkel, in dem man dann die Fliehkraft genießen kann.

[10] Interessant, wie genau 0.1 wiedergegeben wird!

5 Das Hauptprogramm

Hinter dem Kommando `rechner`, welches die betrachteten Daten und auch Gleichugnen produzierte, verbirgt sich – aufbauend auf den vorgestellten Routinen – dieses Hauptrogramm, niedergeschrieben in `rechner.cpp`:

```
#include "include/common.h"
#include "include/system.h"
#include "include/poisson.h"
#include "include/totex.h"
#include "include/betrachter.h"

ex apunkt;
ex ppunkt;
ex pot;

int usage(char* argv[]);
void vollauto(char* argv[]);
void halbauto(char* argv[]);
void einzel(char* argv[]);

int main(int argc, char* argv[])
{
   pot = poti();
   ex ham = hamilton();

   if( (argc != 5) && (argc != 7) && (argc != 8) ) return usage(argv);

   //Aufstellen der Bewegungsgleichungen ueber die Poissonklammer
   //und etwas Dissipation
   double diss = atof(argv[1]);
   apunkt = poisson(alpha,ham);
   ppunkt = poisson(p,ham) - diss*apunkt;

   if(argc != 8)
   {
      //Grundlegende Feststellung am Anfang
      cout << "Potential: " << pot << endl;
      totex(pot,"potential.tex");
      cout << "Hamilton: " << ham << endl;
      totex(ham,"hamilton.tex");
      //Ausgabe der Gleichungen
      cout << "apunkt = " << apunkt << endl << "ppunkt = " << ppunkt << endl;
      totex(apunkt, "apunkt.tex");
      totex(ppunkt, "ppunkt.tex");
      ofstream parf("rechner.par");
      parf << "diss = " << argv[1] << endl
           << "step = " << argv[2] << endl
```

```
            << "points = " << argv[3] << endl
            << "starts = " << argv[4] << endl;
      parf.close();
   }

   //argc kann nur noch 5,7 oder 8 sein!
   if(argc == 8)  einzel(argv);
   else
   {
      if(argc == 7) halbauto(argv);
      else vollauto(argv);
   }

   return 0;
}

int usage(char* argv[])
{
   cout << "usage: " << argv[0] << " diss step points starts" << endl
        << " for many trajectories for some predefined combinations "
        << "of Omega and A"   << endl  << " or " << endl
        << "       " << argv[0] << " diss step points starts Omega A" << endl
        << " for many trajectories for this combination" << endl
        << " or " << endl
        << "       " << argv[0] << " diss step points Omega A starta"
        << " startp" << endl
        << " for one trajectory printed to stdout" << endl;
   return 1;
}

void vollauto(char* argv[])
{
   double step = atof(argv[2]);
   long points = atol(argv[3]);
   long starts = atol(argv[4]);

   //Betrachtungen fuer die verschiedenen Parameterkombinationen
   for(int i = 0; i <= 4; ++i)
   {
      betrachter(pot, apunkt, ppunkt,0.5*i, 0, step, points, starts);
      betrachter(pot, apunkt, ppunkt,0.5*i, 0.3, step, points, starts);
      betrachter(pot, apunkt, ppunkt,0.5*i, 3, step, points, starts);
   }
}

void halbauto(char* argv[])
{
   double step = atof(argv[2]);
```

```
    long points = atol(argv[3]);
    long starts = atol(argv[4]);
    double Omega = atof(argv[5]);
    double A = atof(argv[6]);

    //Betrachtung fuer die eine Parameterkombination
    betrachter(pot, apunkt, ppunkt, Omega, A, step, points, starts);
}

void einzel(char* argv[])
{
    double step = atof(argv[2]);
    long points = atol(argv[3]);
    double Omega = atof(argv[4]);
    double A = atof(argv[5]);
    double starta = atof(argv[6]);
    double startp = atof(argv[7]);

    betrachter(apunkt, ppunkt, Omega, A, step, points, starta, startp);
}
```

6 Ende

So, dann habe ich gesagt, was ich sagen wollte. Der Rezipient hat nun hoffentlich einen Eindruck davon, wie man denn solch ein analytisch/numerisches Problem direkt in C++ behandeln kann. Erfolgreich, versteht sich.

Nun gilt es, die wirklichen Aufgaben in Angriff zu nehmen – werfen wir die Compiler an!