# Modelling with Mathematica

by

Hugh Murrell

Submitted in partial fulfilment of
the requirements for the degree
of
**Doctor of Philosophy**
in
**Computer Science**
at
**Natal University.**

November
1994

# Preface

The work described in this thesis was carried out in the Department of Computer Science, Natal University, Durban, under the supervision of Professor Alan Sartori-Angus. The work spanned the period 1990-1994 and has resulted in three single-author publications in the *Mathematica Journal* and a number of multi-author publications in various other journals. The models studied in this thesis are classical and well-known. The *Mathematica* implementations and the resulting animations represent original work by the author and have not been submitted in any form to any other university. Use made of the work of others has been duly acknowledged in the text.

## Acknowledgements

First and foremost, my thanks go to *Professor Alan Sartori-Angus* for supervising this thesis and providing an environment in which I can earn a living **and** study. Thank you Alan.

Thanks are due to *Professor John Swart* and *Professor Fritz Schuddeboom* for many useful discussions concerning mathematical modeling. A special thanks to *John* for his close collaboration on a number of publications.

Thanks are also due to *Silvio Levy* and *Troels Petersen* for editing my articles appearing in *The Mathematica Journal.* They have been responsible for many improvements in style and content.

I would like to thank my current colleagues and previous teachers. In particular my thanks go to my previous supervisor, *Abraham Ungar*, for teaching me how to publish papers, and to my colleague, *Jane Meyerowitz*, for proof reading this work. Others who have influenced my career are *Sunil Maharaj, Hilton Goldstein, Dave Carson, Nik Heideman, Graham Shepherd, Michael O'Rielly, Pat Terry, Mike Lawrie, Jenny and Brian Nevin, Norman Skinner, Costa Zaverdenos* and *Peter Uys.*

I thank my running partners, *Geoffrey, Steven* and *Robert*, for showing an interest in my endeavours even on cold winter mornings; and I thank my sister *Katherine* for many hours of baby-sitting.

I thank my mother, *Barbara Hulley*, for all her financial and emotional support throughout my academic career.

Finally I thank my wife *Sue* and my children *Daniel* and *Benjamin* for all their encouragement and support. A special thanks goes to my wife without whom I would still be driving trains in Pietermaritzburg.

## Dedication

I dedicate this work to the memory of my father, *Bryan Hulley.*

## Abstract

In this thesis a number of mathematical models are investigated with the aid of the modelling package *Mathematica.* Some of the models are of a *mechanical* nature and some of the models are *laboratories* that have been constructed for the purpose of assisting researchers in a particular field.

In the early sections of the thesis mechanical models are investigated. After the equations of motion for the model have been presented, *Mathematica* is employed to generate solutions which are then used to drive *animations* of the model. The frames of the animations are graphical snapshots of the model in motion. *Mathematica* proves to be an ideal tool for this type of modelling since it combines algebraic, numeric and graphics capabilities on one platform.

In the later sections of this thesis, *Mathematica* laboratories are created for investigating models in two different fields. The first laboratory is a collection of routines for performing *Phase-Plane* analysis of planar autonomous systems of ordinary differential equations. A model of a mathematical concept called a *bifurcation* is investigated and an animation of this *mathematical event* is produced.

The second laboratory is intended to help researchers in the *tomography* field. A standard *filtered back-projection algorithm* for reconstructing images from their projections is implemented. In the final section of the thesis an indication of how the tomography laboratory could be used is presented. *Wavelet* theory is used to construct a *new* filter that could be used in filtered back-projection tomography.

# Contents

# 1 Introduction

It is a common cause of complaint that *mathematical modelling* is overlooked in many traditional university mathematics courses. The process of problem formulation, model building, theory application and communication of conclusions to others, especially non-mathematicians, is generally neglected in the undergraduate mathematics curriculum. These abilities should be central to the student's education if he is to survive in the market-place.

The reason for this shortcoming in the traditional syllabus is that, more often than not, extensive computational skills are required not only to solve the equations that arise out of the modelling process but also in the presentation of the conclusions to an unsophisticated public.

The author has experience of modelling in a 4th generation language environment, see for example *[Hughes and Murrell, 1987]* and *[Murrell, 1982]*. The advent of the symbolic system *MuMath*, simplified the algebraic problems encountered in *differential transform theory*, see *[Murrell and Ungar, 1982]* and *[Ungar and Murrell, 1985]*. However, until the appearence of *Mathematica*, no cheap system existed that combined algebraic, numeric and graphics capabilities on a single modelling platform.

In this thesis *Mathematica* is investigated as a modelling tool. Problems of a mechanical nature are posed, solved and animated. Most of the models tackled here have been published by the author in *The Mathematica Journal*, see *[Murrell, 1992, 1993 and 1994]*. *Mathematica* is also an excellent consultation tool to have at hand when collaborating with other researchers, for examples see *[Swart and Murrell, 1991]* and *[Swart, 1994]*. *Mathematica* provides the same relief to a mathematical modeller wrestling with a fortran compiler, a numerical library and a few home grown graphics routines as the advent of spreadsheets provided to an accountant struggling with Cobol programming.

After a short introduction to the *Mathematica* package a classical 18th century problem is investigated. The idea is to show-off *Mathematica's* analytical, numerical and graphics capabilities. Only a small fraction of *Mathematica's* modelling functions will be used in this introductory demonstration but the flavour of the rest of the thesis will be set.

## 1.1 Programming with Mathematica

*Mathematica* was created in 1988 by *Stephan Wolfram* and his associates in *Wolfram Research, Inc.* *Mathematica* is a *functional programming language* with a sophisticated *front-end.* A full description of the language is given in the *Mathematica User's Manual, [Wolfram, 1991]*. Since its creation a plethora of books have appeared describing the *Mathematica* package and its built-in programming language. For a good introduction the reader is referred to *[Blachman, 1992]* and *[Gaylord et al, 1993]*.

*Expressions* are the main type of data in *Mathematica.* Expressions are of the form `h[e1,e2,...]` where `h` is known as the *head* of the expression and the `ei` are the *elements* of the expression. The elements of an expression can be *raw objects* such as *Integer, Real, Rational, Complex, String* or *Symbol* or they may be expressions themselves. The head of an expression may also be an expression.

The front-end allows users to enter expressions which are then evaluated; expressions being produced as output. Some side-effects such as graphics or sound may also be produced. The front-end has an extensive set of abbreviations for commonly used expressions. The reader is referred to the *Mathematica manual* for a full list of abbreviations. Abbreviations are used in both the input and output phases of the front-end. For example on the input

```
In[1]:= Simplify[ 3 / b + a / b ]
```

the following expression is generated:

```
Simplify[Plus[Times[3, Power[b, -1]], Times[a, Power[b, -1]]]]
```

After *simplification* this expression is evaluated as

```
Times[Plus[3, a], Power[b, -1]]
```

and the following text is produced as output.

```
         3 + a
Out[1]=  -----
           b
```

The simplification occurs because `Simplify` has an extensive collection of *transformation rules* associated with it that are used in the *evaluation* process. *Mathematica* performs the same sequence of steps every time an input expression is evaluated.

a) If the expression is a raw object then it is left unchanged.

b) The head of the expression is evaluated.

c) Each element of the expression is evaluated in turn.

d) Any transformation rules that have been defined for the expression are applied.

*Transformation rules* are either built-in or accumulated by the user entering expressions of the form `SetDelayed[h[e1_,e2_,...], g[e1,e2,...]]`. The infix abbreviation for `SetDelayed` is `:=` so to give *Mathematica* the ability to simplify `Log[]` expressions. For example, one could enter the following transformation rule:

```
Log[Times[a_, b___]] := Plus[Log[a], Log[b]]
```

In the above example `a_` stands for any expression while `b___` stands for any sequence of expressions. This method of adding rules to *Mathematica's* data-base will be used throughout this thesis. Transformation rules of this nature are often called *functions* in the *Mathematica* literature, hence the claim that *Mathematica* is a *functional* language.

### 1.1.1 Lists

*Lists* are an important concept in *Mathematica's* programming language. As with all objects in *Mathematica*, lists are also expressions. The internal representation for a list is `List[e1,e2,....]` while the abbreviation for a list is `{e1,e2,...}`. There are a number of very useful built-in functions (or transformation rules) that operate on lists. The following three are used throughout this thesis.

```
Map[h_, {a_, b_, c_, ...}] := {h[a], h[b], h[c], ...}

Apply[h_, {a_, b_, c_, ...}] := h[a, b, c, ...]

Inner[h_, {a1_, a2_, ...}, {b1_, b2_, ...}, g_]  :=
                      h[g[a1, b1], g[a2, b2], ...]
```

`Map` and `Apply` are useful for evaluating functions at data points while `Inner` as its name suggests is a generalized inner product. There are many other list manipulating functions and again the reader is referred to the *Mathematica* manual.

### 1.1.2 Numerics

There are four types of numbers represented in *Mathematica*. They are *Integer, Rational, Real* and *Complex*. Integer and Rational arithmetic is exact whilst the precision for real and complex arithmetic can be set at run time. *Mathematica* has a few built-in functions that implement standard numerical algorithms. Two of these are used throughout this thesis.

`NIntegrate[f, {x, a, b}]` uses a standard adaptive quadrature algorithm to give a numerical approximation to the integral $\int_a^b f(x)dx$.

`NDSolve[eqns, {z1, z2, ...}, {x, xmin, xmax}]` finds a numerical solution for the functions `z1, z2, ...` appearing in the differential equations, `eqns`. These solutions are returned in the form of an `InterpolatingFunction` which is an internal representation of a function that interpolates the numerical solution to the set of differential equations.

Equations are represented in *Mathematica* in the form `lhs == rhs` which is an abbreviation of `Equal[lhs, rhs]`.

Differentials are represented in the form `Derivative[i,j,...][f[x,y,...]]` which stands for $\frac{\partial f}{\partial x^i}\frac{\partial f}{\partial y^j}\cdots$.

*Mathematica* in its standard form is not overendowed with numerical algorithms, however many numerical packages have been written by *Mathematica* users throughout the world and the best of these have been stored in the *Mathematica* archives at `mathsource@wri.com`.

### 1.1.3 Symbolics

*Mathematica* has a rich set of symbolic manipulation routines. Most of the arithmetic operators and standard mathematical functions can work with symbolic expressions as well as with numeric expressions. There are also many functions that are designed specifically for symbolic manipulation purposes. Three of these used throughout this work are:

`Simplify[expr]` performs a sequence of algebraic transformations on `expr` and returns the shortest form it finds. `Simplify` makes use of other basic symbolic manipulation routines such as `Expand`, `Together` and `Factor`.

`Solve[eqn, var]` attempts to solve the equation `eqn` for the variable `var`.

`D[f, x]` gives a symbolic expression for $\frac{\partial f}{\partial x}$.

`Integrate[f, {x, a, b}]` gives a symbolic expression for $\int_a^b f(x)dx$

*Mathematica* also has a large selection of linear algebra functions that can operate on symbolic and numeric expressions. Included are functions such as `Dot`, `Transpose`, `LinearSolve`, `Det`, `Inverse`, `Eigensystem`, and `NullSpace`. Again the reader is referred to the *Mathematica* manual for details.

### 1.1.4 Graphics

In *Mathematica* two dimensional graphics objects are created by generating expressions of the form `Graphics[primitives, options]`. The usual primitives such as `Point[..]`, `Line[..]` and `Text[..]` are available and the reader is referred to the *Mathematica* manual for the full list. Two high-level routines are available for generating plots:

`Plot[f, {x, xmin, xmax}]` generates a plot of $f(x)$.

`ListPlot[ {{x1, y1}, ...} ]` generates an xy plot of a list of data points.

Expressions of the form `Graphics3D[primitives, options]` are used to represent three dimensional graphics. Although the list of primitives and options is extensive not many built-in routines are available for generating three dimensional graphics.

*Mathematica* provides a *DOS* utility called `MSDOSPS.EXE` for rendering *PostScript* graphics on the computer screen. The *PostScript* is generated from 2D and 3D graphics objects by means of the *Mathematica* function `Show[..]`. Hardcopies are also obtained through the *PostScript* language.

Animation plays a large part in this thesis. Animation is accomplished by rendering the required frames through `Show[..]`, writing them to the hard disk and then viewing the animation under *DOS* using another utility called `ANIMATE.EXE`.

*Mathematica* has found applications in many different fields and a number of books espousing the use of *Mathematica* have appeared lately. The interested reader is referred to *[Crandall, 1991]*, *[Gray and Glynn, 1991]*, *[Shaw, 1994]*, *[Vardi, 1991]*, *[Varian, 1993]* and *[Vvedensky, 1992]*. In this introduction the symbolic, numeric and graphics capabilities of *Mathematica* are demonstrated through the medium of a classical problem.

## 1.2 The Brachistochrone Problem

In 1696 *John Bernoulli* proposed the path of quickest passage problem. In a given field of force, what is the quickest route for a particle to travel from one given point to another given point? The Bernoulli brothers and Leibniz solved this problem in 1697 for a particle moving under gravity and proved that the *cycloid* provides the path of quickest descent. The problem gave rise to the so-called *calculus of variations* and it appears in almost every introductory text concerning that topic, see for example *[Goldstein, 1950]*.

In this demonstration *Mathematica* is employed to animate a bead sliding down a wire. The animation will show that a bead sliding down a cycloid reaches its destination before a bead sliding down a straight line.

### 1.2.1 Time of Passage for Sliding Beads

The time of passage problem is as follows: A smooth wire in a vertical plane connects the point $(0, h)$ on the $y$-axis to the point $(k, 0)$ on the $x$-axis. The shape of the wire in the vertical $xy$-plane is described by a parametric curve $(x(s), y(s)), s \geq 0$. The parametric curve is chosen so that $(x(0), y(0)) = (0, h)$ and $(x(s), y(s)) = (k, 0)$ for some $s > 0$. The bead starts from rest at time $t = 0$ and position $(x(0), y(0))$ and proceeds to slide down the smooth wire. In order to animate the sliding bead *Mathematica* must be able to calculate the position of the bead $(x(s_t), y(s_t))$ at any given subsequent time $t > 0$.

Following *[Prescott, 1941]*, the tie-up of position with time is obtained through energy considerations. Conservation of energy gives:

$$\frac{1}{2}m|v(t)|^2 + mg(y(t) - h) = C \tag{1}$$

where $m$ is the mass of the particle, $|v(t)|$ is its speed at time $t$, $g$ is the acceleration due to gravity, $y(t)$ is the height of the particle at time $t$ and $C$ is an arbitrary constant. Now since the particle starts from rest at height $h$ we have $v(0) = 0$ and $y(0) = h$ so that $C = 0$ and equation 1 may be rewritten as

$$(\frac{dx}{dt})^2 + (\frac{dy}{dt})^2 = 2g(h - y(t)) \tag{2}$$

changing the independent variable from time $t$ to path parameter $s$ and rearranging and integrating results in

$$t = \int_0^{s_t} \sqrt{\frac{(\frac{dx}{ds})^2 + (\frac{dy}{ds})^2}{2g(h - y(s))}} ds \tag{3}$$

Given any time $t$ and any particular wire shape $(x(s), y(s))$ the integral in equation 3 must be evaluated and then equation 3 must be solved for $s_t$. The complexity of this task depends on the wire shape. We show how *Mathematica* can deal with the algebra when the wire shape is a straight line or a cycloid.

### 1.2.2 Generating the Cycloid

Following *[Wells, 1967]*, a cycloid is generated by rolling a disk of radius $R$ on the underside of a horizontal line of height $h$. The disk starts with its point of contact at $(0, h)$ and a pen is attached to the disk at its point of contact. As the disk rolls without slipping along the underside of the line $y = h$, the pen traces out a curve with *cusps* at the points $\{(2\pi Rj, h)\}_{j\in Z}$. The parametric equations for such a cycloid are:

$$\begin{aligned} x(s) &= R(s - \sin s) \\ y(s) &= h - R(1 - \cos s) \end{aligned} \tag{4}$$

It is evident that the cycloid will pass through $(0, h)$. In order for the cycloid to pass through $(k, 0)$ the radius $R$ must be chosen such that for some value of the parameter $s$

$$\begin{aligned} k &= R(s - \sin s) \\ 0 &= h - R(1 - \cos s) \end{aligned} \tag{5}$$

Eliminating $R$ from equations 5 results in an equation for the parameter $s$ that has no closed form solution:

$$s = \sin s + \frac{k}{h}(1 - \cos s) \tag{6}$$

However, given any real values for $h$ and $k$ this equation can be solved for $s$ using a numerical fixed-point iteration. In particular, observe that if $h = 2$ and $k = \pi$ then the fixed point solution of equation 6 is $s = \pi$ and substituting back into equation 5 results in a radius of $R = 1$ for the cycloid generating disk. From here on, to avoid the numerical complications of solving equation 6, it is assumed that the bead must travel from $(0, 2)$ to $(\pi, 0)$. *Mathematica* can now be employed to view the cycloid that the bead must slide on:

```
In[1]:= params = {h->2, k->Pi, R->1, g->10};

In[2]:= cycloid = ParametricPlot[
          Evaluate[{R(s - Sin[s]),h - R(1-Cos[s])} /. params],
          {s,0,Pi}, DisplayFunction->Identity];

In[3]:= disk = ParametricPlot[
          Evaluate[{k/2 + R Cos[theta],h/2 + R Sin[theta]} /. params],
          {theta,0,2 Pi}, DisplayFunction->Identity];

In[4]:= rollbar = ParametricPlot[Evaluate[{x,h} /. params],
          {x,0,Pi}, DisplayFunction->Identity];

In[5]:= Show[cycloid, disk, rollbar, DisplayFunction->$DisplayFunction]
```

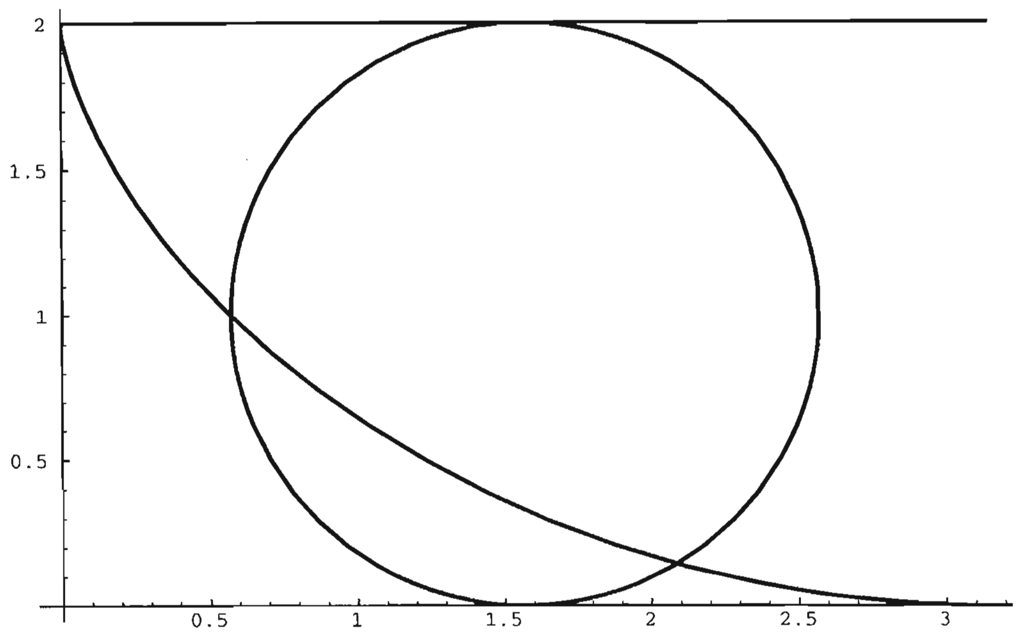

Figure 1: A rolling disk generates a cycloid.

Note that the `DisplayFunction` is switched off until all three parts of the diagram have been generated. The diagram generated by these commands is presented in figure 1.

### 1.2.3 Animating the Sliding Bead

To view the bead sliding down the cycloid equation 3 must be solved for $s_t$. To accomplish this task *Mathematica* is used to generate a *rule* that expresses $s_t$ in terms of $t$ given any parametric curve passing through $(0, h)$ and $(k, 0)$.

```
In[6]:= stRule[xs_,ys_,s_] := First[Solve[ t == Integrate[ Sqrt[
    ( D[xs,s]^2 + D[ys,s]^2 ) / (2 g (h-ys)) ], {s,0,st} ] , st ]];
```

For the cycloid example `stRule[...]` can be used to generate a rule for calculating $s_t$ for any given $t$:

```
In[7]:= cycloidRule = stRule[R(s-Sin[s]),  h-R(1-Cos[s]), s]

                          R
                g Sqrt[-] t
                          g
Out[7]= {st -> -----------}
                        R
```

The total time taken for the bead to slide from $(0, 2)$ to $(\pi, 0)$ is calculated by determining what value of $t$ causes $s_t$ to evaluate to $\pi$.

```
In[8]:= totalCycloidTime = t /.
          First[Solve[Pi == st /. cycloidRule, t]] /. params // N

Out[8]= 0.993459
```

All the information now exists for generating a list of positions for the bead that correspond to evenly spaced time values. In the example that follows 11 positions for the bead on the cycloid are generated. The positions start at $(0, 2)$ and end at $(\pi, 0)$ and are spaced evenly in time.

```
In[9]:= cycloidBeadPositions[n_] := Table[
          {R(st-Sin[st]),  h-R(1-Cos[st])} /. cycloidRule /. params,
          {t,0,totalCycloidTime,totalCycloidTime/(n-1)}] // N

In[10]:= cycloidBeadPositions[11]

Out[10]= {{0, 2.}, {0.00514227, 1.95106}, {0.0405333, 1.80902},
          {0.133461, 1.58779}, {0.305581, 1.30902}, {0.570796, 1.},
          {0.933899, 0.690983}, {1.3901, 0.412215}, {1.92549, 0.190983},
          {2.51842, 0.0489435}, {3.14159, 0.}}
```

In generating the `cycloidBeadPositions` replacement rules have been used:

`/. cycloidRule` replaces every `st` in the parametric equations with the appropriate function of `t` and `/. params` replaces all parameters with their numeric values. The final `// N` converts all numerics to real numbers.

In figure 2 the bead positions are superimposed on the cycloid giving a flash photograph of the sliding bead. The figure is generated using the command:

```
In[11]:= cycloidBeads = ListPlot[cycloidBeadPositions[11],
           Prolog->AbsolutePointSize[9],
           PlotRange->{{0, Pi}, {0, 2}}, DisplayFunction->Identity];

In[12]:= Show[cycloidBeads, cycloid, DisplayFunction->$DisplayFunction]
```

To show that the cycloid is *faster* than a straight line for getting a bead from $(0, 2)$ to $(\pi, 0)$ a flash photograph may be generated for two beads competing with each other along the different routes. The result is shown in figure 3.

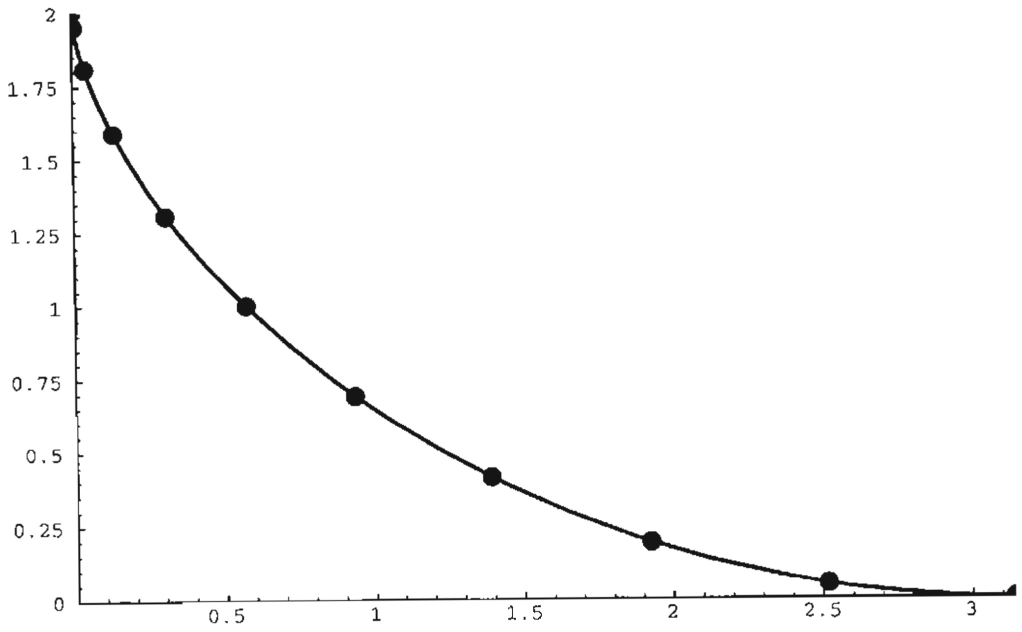

Figure 2: A flash photograph of a bead sliding down a cycloid.

```
In[13]:= straightLineRule = stRule[s,  h(1-s/k), s];

                        2
                 g h k t
Out[13]= {st -> -----------}
                   2    2
                2 (h  + k )

In[14]:= straightLineBeadPositions[n_] := Table[
          {st,  h(1-st/k)} /. straightLineRule /. params,
          {t,0,totalCycloidTime,totalCycloidTime/(n-1)}] // N

In[15]:= straightLineBeads = ListPlot[straightLineBeadPositions[11],
           Prolog->AbsolutePointSize[9],
           PlotRange -> {{0,Pi},{0,2}}, DisplayFunction->Identity];

In[16]:= straightLine = ParametricPlot[
           Evaluate[{s, h(1-s/k)} /. params],
           {s,0,Pi}, DisplayFunction->Identity];

In[17]:= Show[cycloidBeads, straightLineBeads, cycloid, straightLine,
           DisplayFunction->$DisplayFunction]
```

To obtain a real time animation of the sliding beads a single frame is produced for every pair of bead positions. In the following code many bead positions are generated, 51 in fact,

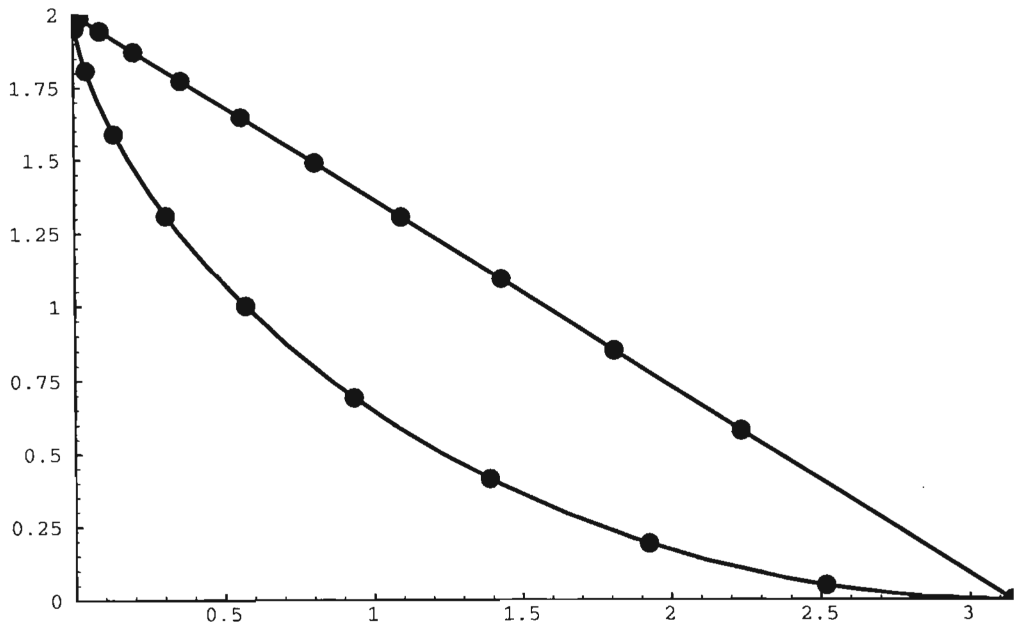

Figure 3: A flash photograph of two competing beads.

to give the illusion of continuous movement.

```
In[18]:= beadAnimationFrameList = Map[ Show[ cycloid, straightLine,
           Graphics[AbsolutePointSize[9], Point[#[[1]]], Point[#[[2]]] ]]&,
           Transpose[ {cycloidBeadPositions[51],
           straightLineBeadPositions[51]} ] ]

In[19]:= DisplayAnimation["BEADS",beadAnimationFrameList]
```

`DisplayAnimation` is an internal *Mathematica* function that takes a list of graphics frames and produces a *DOS* animation file as output. The animation may be viewed by exiting *Mathematica* and using the *DOS* command: `ANIMATE BEADS`. The utility `ANIMATE.EXE` and the animation file `BEADS` are stored in machine readable form on a stiffy disk attached to this thesis. Complete instructions for viewing the animation are given in the appendix.

## 1.3 Conclusion

A short introduction to the modelling capabilities of *Mathematica* has been given and the tone for the rest of the thesis has been set. In the next section animation and flash photograph concepts are employed in the analysis of a mathematical golf swing.

# 2 Animation of a Mathematical Golf Swing

In this section a mathematical model for the swing of a golf club is presented. The model is implemented in *Mathematica* and the final result is an animation of the swing. The animation will demonstrate that a golfer can execute a perfect swing *without* using power from his wrists. This fact is not well documented in golfing literature, which is strange since the use of the wrist muscles in the execution of a golf swing can cause all sorts of problems.

## 2.1 The Mathematical Model

The *double pendulum* provides a simple model of the golf swing *[Williams, 1967]*. The shaft of the club is an outer link of length $a$, while the arms of the golfer form the inner link of length $b$.

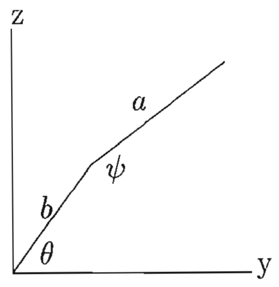

Figure 4: The double pendulum.

The *downswing* is analyzed in two parts. In the first part of the downswing the wrists are locked, $\psi = \psi_0$, while the arms are accelerated from rest with constant angular acceleration $\ddot{\theta} = \alpha$. In the second part of the downswing the arms rotate with constant angular velocity $\dot{\theta} = \omega$, while the wrists are allowed to open freely as if they were a perfect hinge.

The angle at which the first phase ends and the second phase starts will be denoted $\theta_c$. This angle should be selected so that the arms and the club line up at impact with the ball, i.e. $\psi = \pi$ when $\theta = \frac{3\pi}{2}$.

To obtain the equations of motion, we will consider the golf club as a unit mass attached to a light rigid rod of length $a$. Neglecting gravity, the only forces on the club are those exerted by the hands. From the geometry of figure 4 the position of the clubhead for any value of $\theta$ and $\psi$ is given by:

$$\begin{aligned} y &= b\cos\theta + a\cos(\theta+\psi-\pi) \\ z &= b\sin\theta + a\sin(\theta+\psi-\pi) \end{aligned} \tag{7}$$

If $F_y$ and $F_z$ are the forces exerted by the hands on the club in the $y$ and $z$ directions then Newton tells us that:

$$\begin{aligned} F_y = \ddot{y} &= -b\cos\theta\dot{\theta}^2 - b\sin\theta\ddot{\theta} - a\cos(\theta+\psi-\pi)(\dot{\theta}+\dot{\psi})^2 \\ &\quad -a\sin(\theta+\psi-\pi)(\ddot{\theta}+\ddot{\psi}) \\ F_z = \ddot{z} &= -b\sin\theta\dot{\theta}^2 + b\cos\theta\ddot{\theta} - a\sin(\theta+\psi-\pi)(\dot{\theta}+\dot{\psi})^2 \\ &\quad +a\cos(\theta+\psi-\pi)(\ddot{\theta}+\ddot{\psi}) \end{aligned} \tag{8}$$

It is useful to resolve these forces in directions parallel and vertical to the club. Let $F_s$ be the pull along the shaft away from the clubhead and let $F_p$ be the push of the hands perpendicular to the club in a direction of increasing $\theta$. Then $(-F_s, F_p)$ is obtained from $(F_y, F_z)$ by a rotation of $\theta+\psi-\pi$:

$$\begin{aligned} F_s &= -F_y\cos(\theta+\psi-\pi) - F_z\sin(\theta+\psi-\pi) \\ F_p &= -F_y\sin(\theta+\psi-\pi) + F_z\cos(\theta+\psi-\pi) \end{aligned} \tag{9}$$

Substituting equations 8 into equations 9 and simplifying yields:

$$\begin{aligned} F_s &= -b\dot{\theta}^2\cos\psi + b\ddot{\theta}\sin\psi + a(\dot{\theta}+\dot{\psi})^2 \\ F_p &= -b\dot{\theta}^2\sin\psi - b\ddot{\theta}\cos\psi + a(\ddot{\theta}+\ddot{\psi}) \end{aligned} \tag{10}$$

Indeed the following *Mathematica* session confirms these results:

```
In[1]:= y[t]:= b Cos[Theta[t]] + a Cos[Theta[t] + Psi[t] - Pi]

In[2]:= z[t]:= b Sin[Theta[t]] + a Sin[Theta[t] + Psi[t] - Pi]

In[3]:= Fs[t]:= - D[y[t], {t, 2}] Cos[Theta[t] + Psi[t] - Pi] -
                  D[z[t], {t, 2}] Sin[Theta[t] + Psi[t] - Pi]

In[4]:= Fp[t]:= - D[y[t], {t, 2}] Sin[Theta[t] + Psi[t] - Pi] +
                  D[z[t], {t, 2}] Cos[Theta[t] + Psi[t] - Pi]

In[5]:= Simplify[Expand[Fs[t]]]
```

```
                  2                                        2
Out[5]= a Psi'[t]  + 2 a Psi'[t] Theta'[t] + a Theta'[t]  -
                                  2
        b Cos[Psi[t]] Theta'[t]  + b Sin[Psi[t]] Theta''[t]


In[6]:= Simplify[Expand[Fp[t]]]

                                    2
Out[6]= -(b Sin[Psi[t]] Theta'[t] ) + a Psi''[t] + a Theta''[t] -

          b Cos[Psi[t]] Theta''[t]
```

In the first phase of the downswing, $\theta$ increases from $\theta_0$ to $\theta_c$ with constant angular acceleration $\ddot{\theta} = \alpha$, and $\psi$ remains fixed at $\psi = \psi_0$. So during this phase:

$$\begin{array}{rcl} \dot{\theta} & = & \alpha t \\ \theta & = & \frac{\alpha t^2}{2} + \theta_0 \\ F_s & = & -b\alpha^2 t^2 \cos\psi_0 + b\alpha \sin\psi_0 + a\alpha^2 t^2 \\ F_p & = & -b\alpha^2 t^2 \sin\psi_0 - b\alpha\cos\psi_0 + a\alpha \end{array} \tag{11}$$

The first phase ends when $\theta = \theta_c$, that is, at time $t_c = \sqrt{(\theta_c - \theta_0)\frac{2}{\alpha}}$. In the second phase of the downswing, $\theta$ increases from $\theta_c$ to $\frac{3\pi}{2}$ with constant angular speed $\dot{\theta} = \alpha t_c = \omega$. In this phase the wrists are considered to be smooth hinges and so cannot exert forces perpendicular to the club shaft. Therefore $F_p$ is zero and we have:

$$\begin{array}{rcl} \dot{\theta} & = & \omega \\ \theta & = & \theta_c + \omega(t - t_c) \\ F_s & = & -b\omega^2 \cos\psi + a(\omega + \dot{\psi})^2 \\ \ddot{\psi} & = & \frac{b}{a}\omega^2 \sin\psi \end{array} \tag{12}$$

Equations 11 fully describe the first phase of the downswing. A little more work is required to find $\psi$ and $\dot{\psi}$ as functions of $t$ in the second phase. Recalling that $\ddot{\psi} = \dot{\psi}\frac{d\dot{\psi}}{d\psi}$ and using the initial condition $\dot{\psi} = 0$ when $\psi = \psi_0$, the last equation in 12 can be integrated to get:

$$\dot{\psi} = \sqrt{\frac{2b\omega^2}{a}(\cos\psi_0 - \cos\psi)} \tag{13}$$

This equation cannot be integrated further in terms of elementary functions so a numerical procedure is used to calculate $\psi(t)$ for $t > t_c$. Let $t_{n+1} = t_n + \Delta t$ and expand $\psi(t_{n+1})$ in a Taylor polynomial, making use of equation 13 and the last equation in 12 to get:

$$\begin{aligned}\psi(t_{n+1}) &= \psi(t_n + \Delta t) \\ &= \psi(t_n) + \Delta t\dot{\psi}(t_n) + \tfrac{\Delta t^2}{2}\ddot{\psi}(t_n) + O(\Delta t^3) \\ &= \psi(t_n) + \Delta t\sqrt{\tfrac{2b\omega^2}{a}(\cos\psi_0 - \cos\psi(t_n))} + \tfrac{\Delta t^2}{2}\tfrac{b}{a}\omega^2 \sin\psi(t_n) \\ &\quad +O(\Delta t^3)\end{aligned} \tag{14}$$

A sequence of values of $\psi(t)$ for $t > t_c$ can now be generated.

The function `SwingAngles` computes a list of successive configurations $\theta, \psi$, given values for the club length $a$, the arm length $b$, the initial angles $\theta_0$ and $\psi_0$, the parameter $\theta_c$, the initial angular acceleration $\alpha$, and a timestep $dt$.

```
In[1]:= SwingAngles[a_, b_, theta0_, psi0_, thetac_, alpha_, dt_]:=
          Module[{psi, tc, w, tf, kc, ddt, cos0 = Cos[psi0]},
            tc = Sqrt[2(thetac - theta0)/alpha];
            w  = alpha tc;
            tf = (1.6 Pi - thetac)/w + tc;
            kc = Floor[tc/dt];
            ddt = (kc + 1) dt - tc;
            psi = N[psi0 + ddt^2 b w^2 Sin[psi0]/(2a)];
            { Table[{alpha (k dt)^2 /2 + theta0, psi0} // N,
                {k, 0, kc}], {{thetac + w ddt, psi}} // N,
              Table[{thetac + w (k dt - tc) // N,
                     psi = N[psi + dt Sqrt[2 b w^2 (cos0 - Cos[psi])/a] +
                             dt^2 b w^2 Sin[psi]/(2a) ]},
                    {k, kc + 2, Floor[tf/dt] + 1} ] } // Flatten[#, 1]& ]
```

The first phase consists of `kc = Floor[tc/dt]` timesteps, during which `psi` is constant. At the next timestep, `psi` is computed from equation 14 with $\Delta t$ set to `((kc + 1)dt - tc)`. During the rest of the second phase, `psi` is computed iteratively with $\Delta t$ set to `dt`.

```
In[8]:= angles = SwingAngles[0.85, 0.65, 100 Degree,
                             65 Degree, 140 Degree, 60, 0.02]

Out[8]= {{1.74533, 1.13446}, {1.75733, 1.13446},
         {1.79333, 1.13446}, {1.85333, 1.13446},
         {1.93733, 1.13446}, {2.04533, 1.13446},
         {2.17733, 1.13446}, {2.33333, 1.13446},
{2.51166, 1.13608}, {2.69472, 1.15635},
{2.87778, 1.20004}, {3.06084, 1.26758},
{3.2439, 1.35954}, {3.42695, 1.47653},
{3.61001, 1.61903}, {3.79307, 1.78717},
{3.97613, 1.98042}, {4.15919, 2.19728},
{4.34225, 2.43505}, {4.5253, 2.68963},
{4.70836, 2.95554}, {4.89142, 3.22629},
{5.07448, 3.49488}}
```

## 2.2 The Animation

To draw the frames of the animation, we separate the picture into moving and static components. The function `StaticParts` generates graphics primitives for the static component, which depends only on $a$ and $b$:

```
In[9]:= StaticParts[a_, b_] :=
          Module[{r = b/6, c = Sqrt[a(a + 2b)], head, eyes, legs},
            head = Polygon[Table[
              r {0, Sin[i Pi/20], 1 + Cos[i Pi/20]}, {i, 0, 40}]];
            eyes = {PointSize[0.015],
              Point[{0, -r/2, r}], Point[{0, r/2, r}]};
            legs = {Thickness[0.02], Line[
              {{0, -b/3, -c}, {0, 0, -b}, {0, b/3, -c}}] };
            {legs, head, eyes}]
```

The function `MovingParts` draws the elements of each frame that depend on the configuration $\theta, \psi$:

```
In[10]:= MovingParts[a_, b_, {theta_, psi_}] :=
           Module[ {sn = b/(a + b), cs = Sqrt[a(a + 2b)]/(a + b),
                    project, hands, rightshoulder, leftshoulder,
                    clubhead, ball},
             project[{y_, z_}] := {-z sn, y, z cs};
             hands = project[b {Cos[theta], Sin[theta]}];
             rightshoulder = project[
               (b/3) {Cos[theta/2 + Pi/4], Sin[theta/2 + Pi/4]}];
             leftshoulder = project[
               (b/3) {Cos[theta/2 + 5Pi/4], Sin[theta/2 + 5Pi/4]}];
             clubhead = project[
               {b Cos[theta] + a Cos[theta + psi - Pi],
                b Sin[theta] + a Sin[theta + psi - Pi] }];
             ball = If [theta < N[3Pi/2],
      {PointSize[0.02], Point[project[{0, - a - b}]]}, {}];
    {Line[{hands, rightshoulder, leftshoulder,
           hands, clubhead}],
      PointSize[0.025], Point[clubhead], ball,
      EdgeForm[Thickness[0.01]],
      Polygon[{rightshoulder, {0, 0, -b}, leftshoulder}]} ]
```

The swing takes place in a plane in 3-space that is inclined at an angle $sin^{-1}(b/(a+b))$ from the vertical. An auxiliary function `project`, is used to transform the $(y, z)$ coordinates of a point in this plane into the coordinates of the point in 3-space.

The function `DrawFrame` combines the static and moving elements in one frame of the animation sequence and displays them with appropriate graphics options:

```
In[11]:= DrawFrame[a_, b_, {theta_, psi_}] :=
           Show[Graphics3D[
             {StaticParts[a, b], MovingParts[a, b, {theta, psi}]}],
             AspectRatio -> Automatic,
             ViewPoint -> {a + b, 0, a},
             Boxed -> False,
             PlotRange -> (a + b){{-1, 1}, {-1, 1}, {-1, 1}}]]
```

For example, in the swing that was calculated earlier, the change from the first phase to the second phase of the downswing occurs at the 9th frame. The following commands generate figure 5 which shows the position of the golfer at the end of the first phase:

```
In[12]:= angles[[9]]

Out[12]= {2.51166, 1.13608}

In[13]:= DrawFrame[0.85, 0.65, %]
```

To produce the complete animation, we simply `Map` the function `DrawFrame` onto the list generated by `SwingAngles`:

```
In[14]:= AnimateGolfer[a_, b_, theta0_, psi0_, thetac_, alpha_, dt_]:=
          Map[DrawFrame[a, b, #]&,
            SwingAngles[a, b, theta0, psi0, thetac, alpha, dt] ]
```

The *Mathematica* function `DisplayAnimation` can be used to write the *PostScript* code for the frames to a file for animation purposes. For example the command:

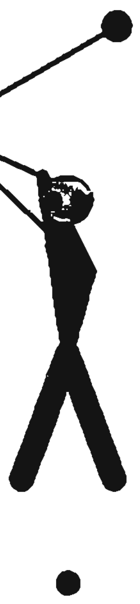

Figure 5: The position at the end of the first phase

```
In[15]:= DisplayAnimation["GOLFER",AnimateGolfer[0.85, 0.65,
           100 Degree, 65 Degree, 140 Degree, 60, 0.02]]
```

produces a DOS animation file which can be viewed via the utility ANIMATE.EXE. The animation file SWING is stored in machine readable form on a stiffy disk attached to this thesis. Complete instructions for viewing the animation are given in the appendix.

## 2.3 Multiflash Photography

The parameters for the swing were obtained by taking measurements from a sequence of flash photographs of Bobby Jones's swing published in *[Williams, 1969]*. The function FlashPhoto combines the animation frames into a single image. Comparison of figure 6 with a published flashphoto of your favorite professional shows that an acceptable downswing is produced.

```
In[16]:= FlashPhoto[a_, b_, theta0_, psi0_, thetac_, alpha_, dt_]:=
          Show[Graphics3D[{StaticParts[a, b],
```

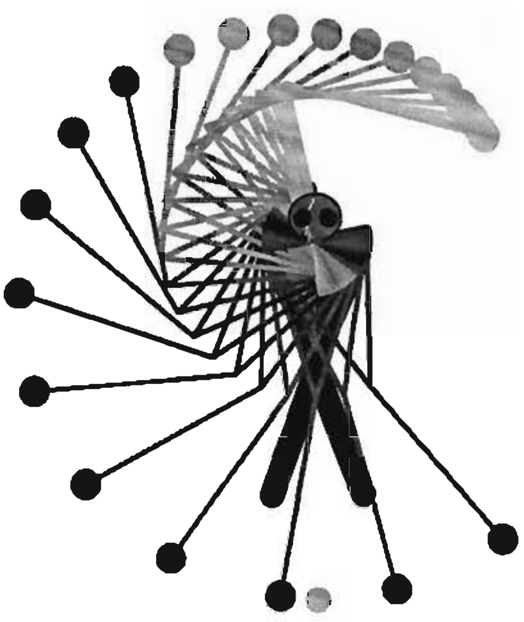

Figure 6: A Multiflash photo of the perfect golf swing

```
        Map[MovingParts[a, b, #]&,
          SwingAngles[a, b, theta0, psi0, thetac, alpha, dt]]},
        AspectRatio -> Automatic, ViewPoint -> {a + b, 0, a},
        Boxed -> False,
        PlotRange -> (a + b){{-1, 1}, {-1, 1}, {-1, 1}}] ]

In[17]:= FlashPhoto[0.85, 0.65, 100 Degree, 65 Degree, 140 Degree, 60, 0.02]
```

## 2.4 Analysis of Forces

Golfing enthusiasts often refer to the *hitting-area* of the downswing. The radial tension in the arms is not felt by the golfer while the force tangential to the path of the hands has to be *manufactured.* Thus a picture of the so-called hitting-area is a polar plot of the component of the forces tangential to the path of the hands. In the first stage of the downswing $F_p$ balances the couple caused by the locking of the wrists. Thus in the first stage only $F_s$ contributes to *hitting* and $F_T = F_s \sin\psi$. In the second stage the wrists are free and both components contribute, $F_T = F_s \sin\psi - F_p \cos\psi$. To show the *hitting area* $F_s$ and $F_p$ are taken from equations 11 and 12 and $F_T$ is then used to *patch* the function SwingAngles and construct a function TangentialForces which computes a list of successive tangential force magnitudes for each $\theta$ in the downswing.

```
In[18]:= TangentialForces[a_, b_, theta0_, psi0_, thetac_, alpha_, dt_]:=
           Module[{psi, tc, w, tf, kc, ddt,
                   cos0 = Cos[psi0], sin0 = Sin[psi0]},
             tc = Sqrt[2(thetac - theta0)/alpha];
             w  = alpha tc;
             tf = (1.6 Pi - thetac)/w + tc;
             kc = Floor[tc/dt];
             ddt = (kc + 1) dt - tc;
             psi = N[psi0 + ddt^2 b w^2 Sin[psi0]/(2a)];
             Map[ Drop[#,{2}]& ,
               { Table[{alpha (k dt)^2 /2 + theta0, psi0,
  (-b alpha^2 (k dt)^2 cos0 + b alpha sin0 + a alpha^2 (k dt)^2) sin0 } // N,
                   {k, 0, kc}], {{thetac + w ddt, psi,
  (-b w^2 Cos[psi] + a(w + Sqrt[2 b w^2 (cos0 - Cos[psi])/a])^2) Sin[psi]
               }} // N,
               Table[{thetac + w (k dt - tc) // N,
                      psi = N[psi + dt Sqrt[2 b w^2 (cos0 - Cos[psi])/a] +
                                dt^2 b w^2 Sin[psi]/(2a) ],
  (-b w^2 Cos[psi] + a(w + Sqrt[2 b w^2 (cos0 - Cos[psi])/a])^2) Sin[psi]
                     } // N,
               {k, kc + 2, Floor[tf/dt] + 1} ] } // Flatten[#, 1]& ] ]
```

Note that each $\psi$ in the downswing is used to calculate the tangential force and then they are discarded.

```
In[19]:= forces = TangentialForces[0.85, 0.65, 100 Degree,
                                   65 Degree, 140 Degree, 60, 0.02]

Out[20]= {{1.74533, 39.00000}, {1.75733, 39.59204},
          {1.79333, 41.36814}, {1.85333, 44.32832},
          {1.93733, 48.47258}, {2.04533, 53.80090},
          {2.17733, 60.31330}, {2.33333, 68.00976},
          {2.51166, 50.03669}, {2.69472, 69.86024},
          {2.87778, 94.41542}, {3.06084, 124.53256},
          {3.24390, 160.60072}, {3.42695, 201.90445},
          {3.61001, 245.68231}, {3.79307, 286.08505},
          {3.97613, 313.59129}, {4.15919, 315.87832},
          {4.34225, 281.15672}, {4.52530, 203.83304},
          {4.70836, 90.00741}, {4.89142, -41.43580},
          {5.07448, -164.68057}}
```

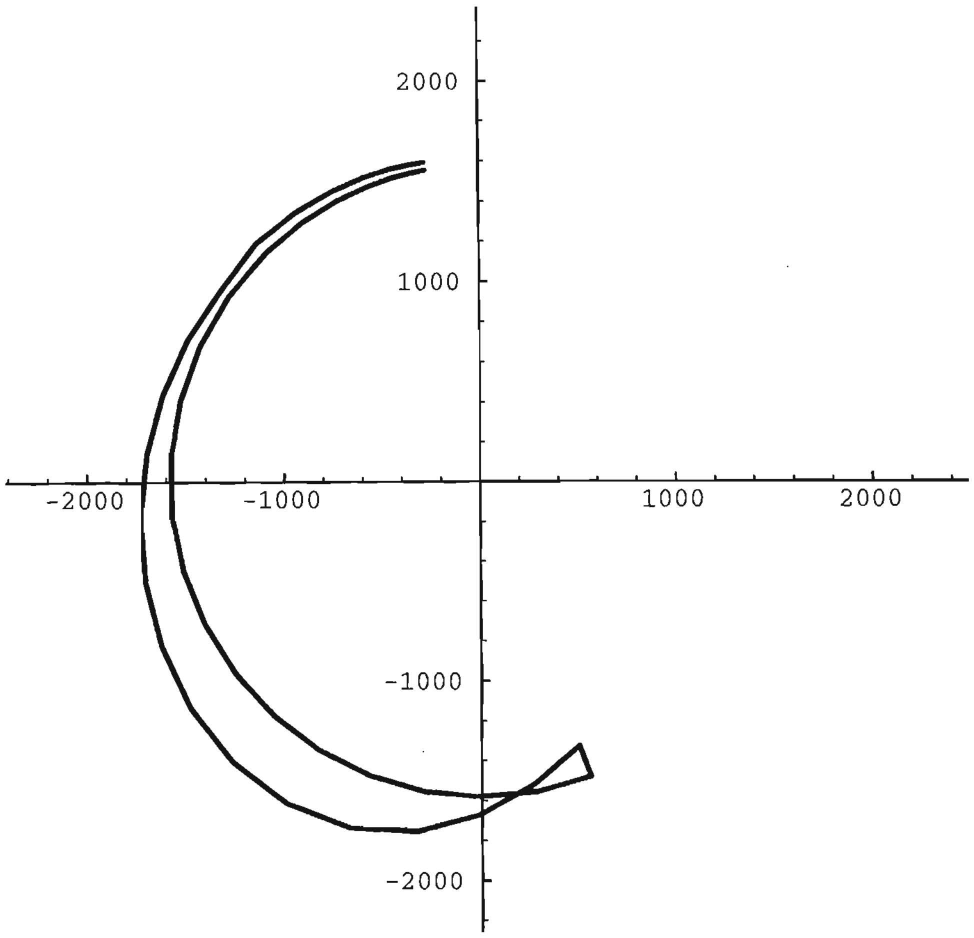

Figure 7: The hitting area of the downswing.

To display the hitting area, we choose a *reference radius*, $R$ corresponding to $F_T = 0.$ and plot the polar coordinates $(\theta, R + F_T)$ together with an arc of radius$R$. A point outside the arc represents an accelerating (positive) force, while a point inside the arc is a retarding force.

```
In[21]:= PolarForcePlot[forces_List]:=
           Module[{ref = 5 Max[Transpose[forces][[2]]]},
             ListPlot[{ Map[ {(ref+#[[2]]) Cos[ #[[1]] ] ,
                                (ref+#[[2]]) Sin[ #[[1]] ]}&, forces],
                        Map[ { ref Cos[ #[[1]] ] , ref Sin[ #[[1]] ] }&,
                          Reverse[forces] ] } // Flatten[#, 1]&,
                        PlotRange -> 2 ref { {-1,1},{-1,1} },
                        AspectRatio -> Automatic, PlotJoined -> True ] ]

In[22]:= PolarForcePlot[forces]
```

Figure 7 shows the hitting area of the downswing where the *fat* portion of the polar plot indicates that hitting is taking place. Note that at contact with the ball all forces are radial and *no hitting* occurs.

## 2.5 Conclusion

It has been demonstrated that an acceptable golf swing can be executed without using power from the wrists. Once the wrists start to unlock, all the force exerted by the hands is in the direction of the club shaft. Most of the hitting in the golf swing should be carried out after the hands have passed the horizontal position. In the next section *Mathematica* is employed in the analysis of a more demanding three dimensional mechanical problem.

# 3 Animation of Rotating Rigid Bodies

In this section *Mathematica* is used to animate the motion of a rigid body under no external forces. As an example, a box model of a rotating tennis racquet is presented.

In the study of the motion of a rigid body under no forces most text books, for example *[Gray and Gray, 1911]*, will show that the tip of the angular velocity vector moves on a fixed plane that is normal to the constant angular momentum vector. As it moves it traces out a curve called the *herpolhode* of the motion. This concept can be hard to visualize but in this section *Mathematica* is used to provide a solution and animation of the relevant equations of motion.

The rigid bodies discussed here will be simple boxes or combinations of boxes but the interested reader should be able to adapt the *Mathematica* code and animate the motion of many other rigid bodies. As a practical example the motion of a tennis racquet rotating under no forces is investigated. A nice example to demonstrate since while rotating through $2\pi$ radians about an axis in the same plane as the head but perpendicular to the shaft, the racquet flips through $\pi$ radians about the shaft axis.

An outline of the classical derivation and solution of the equations of motion for the problem is given first and then a *Mathematica* implementation provides the animation of the strange behaviour of the tennis racquet.

## 3.1 The Equations of Motion

The fundamental ideas in the derivation of the equations of motion of a rigid body moving under no forces are two: the principal axes theorem and the conservation of angular momentum and kinetic energy. These concepts can be found in any reputable mechanics textbook, *[Synge and Griffith, 1959]*. In what follows, boldface characters denote vectors and a dot above a letter indicates differentiation with respect to time.

The principal axes theorem states that for any rigid body it is possible to choose an orthogonal coordinate system, fixed with respect to the body, and such that the center of mass is at the origin and the angular momentum vector, **h**, is given by:

$$\mathbf{h} = A\omega_1\mathbf{x} + B\omega_2\mathbf{y} + C\omega_3\mathbf{z} \tag{15}$$

where **x**, **y** and **z** are unit vectors along the coordinate axes; $A, B$ and $C$ are constants called the *moments of inertia* in the three coordinate directions; and $\omega_1, \omega_2$ and $\omega_3$ are the components of the angular velocity vector $\omega$ with respect to this coordinate system. The coordinate axes in this coordinate system are called the *principal axes* of the rigid body.

The angular velocity vector $\omega$ is defined by the matrix equation:

$$\begin{pmatrix} \dot{\mathbf{x}} \\ \dot{\mathbf{y}} \\ \dot{\mathbf{z}} \end{pmatrix} = \begin{pmatrix} \omega \times \mathbf{x} \\ \omega \times \mathbf{y} \\ \omega \times \mathbf{z} \end{pmatrix} = \begin{pmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \\ \mathbf{z} \end{pmatrix} \tag{16}$$

The matrix in equation 16, obtained by simply expanding the cross products, is called the *infinitesimal rotation matrix* associated with $\omega$.

If a body has enough symmetries (for example, if it is symmetric with respect to reflections in two perpendicular planes) the principal axes are very easy to determine since they are preserved by the symmetries. Only symmetric bodies are considered here but the derivation applies to any rigid body.

Since there are no external forces the center of mass is fixed in space and the angular momentum $\mathbf{h}$ and the kinetic energy $T = \frac{1}{2}(A\omega_1^2 + B\omega_2^2 + C\omega_3^2)$ are constant. Now $\omega \cdot \mathbf{h} = 2T$, so the tip of the angular velocity vector $\omega$ moves on a fixed plane normal to $\mathbf{h}$ and this so-called *invariable plane* lies at a distance of $\frac{2T}{\|\mathbf{h}\|}$ from the center of mass. The path traced out by the tip of the angular momentum vector in the invariable plane is called the *herpolhode*.

## 3.2 Calculating Angular Velocity

The principal axes $\mathbf{xyz}$, being attached to the body, rotate in space. In order to describe the body's movement the position of the principal axes and the angular velocity vector must be expressed as a function of time in terms of a fixed coordinate system $\mathbf{XYZ}$. Assume that the angular velocity $\omega = \omega_1\mathbf{x} + \omega_2\mathbf{y} + \omega_3\mathbf{z}$ is known at time $t = 0$; this corresponds to giving the rigid body a certain initial angular velocity and then letting it go. To find out what happens to the body after $t = 0$, *[Synge and Griffith, 1959, 14.108-14.121]* is followed.

First differentiate equation 15 with respect to time $t$, remembering that $\mathbf{h}$ is constant, while the other variables depend on $t$, then substitute the values of $\dot{\mathbf{x}}, \dot{\mathbf{y}}$ and $\dot{\mathbf{z}}$ given by equation 16 to get *Euler's equations*:

$$\begin{aligned} A\dot{\omega}_1 - (B - C)\omega_2\omega_3 &= 0, \\ B\dot{\omega}_2 - (C - A)\omega_3\omega_1 &= 0, \\ C\dot{\omega}_3 - (A - B)\omega_1\omega_2 &= 0. \end{aligned} \tag{17}$$

Now assume, by permuting the principal axes if necessary, that $A \geq B \geq C$. Solving the equations $\mathbf{h} = A\omega_1\mathbf{x} + B\omega_2\mathbf{y} + C\omega_3\mathbf{z}$ and $T = \frac{1}{2}(A\omega_1^2 + B\omega_2^2 + C\omega_3^2)$ for $\omega_1$ and $\omega_3$ in terms of $\omega_2$ and substituting the results into the second Euler equation gives:

$$\dot{\omega}_2^2 = \frac{(A-C)^2}{B^2}\left(\frac{h^2-2CT}{A(A-C)} - \frac{B(B-C)}{A(A-C)}\omega_2^2\right)\left(\frac{2AT-h^2}{C(A-C)} - \frac{B(A-B)}{C(A-C)}\omega_2^2\right), \tag{18}$$

where $h = ||\mathbf{h}||$. This equation can be reduced to a standard elliptic differential equation by a change of variables. First set

$$\beta = \sqrt{\frac{2AT-h^2}{B(A-B)}}, \quad k = \sqrt{\frac{B-C}{A-B}\frac{2AT-h^2}{h^2-2CT}}, \quad p = \sqrt{\frac{(h^2-2CT)(A-B)}{ABC}}, \tag{19}$$

and then perform the change of variables $\xi = \frac{\omega_2}{\beta}$ and $\tau = pt$ in equation 18, obtaining

$$\left(\frac{d\xi}{d\tau}\right)^2 = (1-\xi^2)(1-k^2\xi^2) \tag{20}$$

The solution to this equation is Jacobi's elliptic function $\xi = sn_k(\tau)$. All three components of angular velocity can now be expressed in terms of Jacobi elliptic functions

$$\begin{aligned} \omega_1 &= \alpha dn_k(p(t-t_c)), \\ \omega_2 &= \beta sn_k(p(t-t_c)), \\ \omega_3 &= \gamma cn_k(p(t-t_c)), \end{aligned} \tag{21}$$

where

$$\alpha = \sqrt{\frac{h^2-2CT}{A(A-C)}}, \quad \gamma = -\sqrt{\frac{2AT-h^2}{C(A-C)}} \tag{22}$$

and $t_c$ is a constant of integration given by

$$t_c = \frac{-1}{p} sn_k^{-1}\left(\frac{\omega_2(0)}{\beta}\right). \tag{23}$$

Note that $\gamma$ is chosen to be negative since substitution into Euler's equations yields $\alpha\beta\gamma < 0$. Also note that the use of elliptic functions requires $k < 1$, which amounts to $h^2 > 2BT$. If this condition does not hold, then the reduction from equation 18 to equation 20 must be performed again after swapping the terms in the big parentheses in equation 18. This will result in a different set of expressions for $\alpha, \beta, \gamma, p$ and $k$.

## 3.3 Calculating Position

Explicit formulas for the angular velocity vector as a function of time have been presented but formulas for the principal axes **xyz** as a function of time must still be found. This problem is approached by finding as a function of time the spacial rotation that maps the stationary axes **XYZ** onto the principal axes **xyz**. This spacial rotation is three dimensional and is conveniently parametrized by the *Euler angles* $\phi, \theta$ and $\psi$, as follows: Start from the fixed frame **XYZ** and rotate it through an angle $\phi$ around the Z-axis. Next rotate through $\theta$ around the rotated Y-axis and finally rotate through $\psi$ around the rotated Z-axis. The result is set equal to **xyz** and this equation defines $\phi, \theta$ and $\psi$. The compound transformation is given by a series of matrix multiplications

$$\begin{pmatrix} \mathrm{x} \\ \mathrm{y} \\ \mathrm{z} \end{pmatrix} = \rho_Z(\psi)\rho_Y(\theta)\rho_Z(\phi) \begin{pmatrix} \mathbf{X} \\ \mathbf{Y} \\ \mathbf{Z} \end{pmatrix} \tag{24}$$

where

$$\rho_Z(\phi) = \begin{pmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{25}$$

is the matrix that describes the first rotation through $\phi$ around the Z-axis and so on. Setting $R_{\psi\theta\phi} = \rho_Z(\psi)\rho_Y(\theta)\rho_Z(\phi)$ one could write down this matrix explicitly but it turns out to be cumbersome and unenlightening. The Euler angles are related to the angular velocity via the angular momentum which is constant and therefore has known components, $h_1, h_2$ and $h_3$ in the frame **XYZ**.

$$\begin{aligned} \begin{pmatrix} h_1 & h_2 & h_3 \end{pmatrix} \begin{pmatrix} \mathbf{X} \\ \mathbf{Y} \\ \mathbf{Z} \end{pmatrix} = \mathbf{h} &= \begin{pmatrix} A\omega_1 & B\omega_2 & C\omega_3 \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \mathbf{y} \\ \mathbf{z} \end{pmatrix} \\ &= \begin{pmatrix} A\omega_1 & B\omega_2 & C\omega_3 \end{pmatrix} R_{\psi\theta\phi} \begin{pmatrix} \mathbf{X} \\ \mathbf{Y} \\ \mathbf{Z} \end{pmatrix} \end{aligned} \tag{26}$$

Comparing the first and last quantities

$$\begin{pmatrix} A\omega_1 & B\omega_2 & C\omega_3 \end{pmatrix} = \begin{pmatrix} h_1 & h_2 & h_3 \end{pmatrix} R_{\psi\theta\phi}^{-1} \tag{27}$$

or transposing and taking into account that the transpose and the inverse of a rotation matrix are the same

$$\begin{pmatrix} A\omega_1 \\ B\omega_2 \\ C\omega_3 \end{pmatrix} = R_{\psi\theta\phi} \begin{pmatrix} h_1 \\ h_2 \\ h_3 \end{pmatrix} \tag{28}$$

Although this is a system of three equations in three unknowns (the Euler angles), it is underdetermined: if $R_{\psi\theta\phi}$ is a solution, so is $R_{\psi\theta\phi}R'$, where $R'$ is a rotation fixing the vector $(h_1, h_2, h_3)$. Thus the most we can get from equation 28 is two of the Euler angles. This is particularly simple if $h_1 = h_2 = 0$, that is, if the frame **XYZ** is chosen so that **h** lies along the $Z$-axis. In this case equation 28 reduces to

$$\begin{aligned} A\omega_1 &= -h \sin\theta \cos\psi \\ B\omega_2 &= h \sin\theta \sin\psi \\ C\omega_3 &= h \cos\theta \end{aligned} \tag{29}$$

where $h = \|\mathbf{h}\| = h_3$ and this results in

$$\theta = \cos^{-1}\left(\frac{C\omega_3}{h}\right), \psi = \tan^{-1}\left(-\frac{B\omega_2}{A\omega_1}\right) \tag{30}$$

To find $\phi$, note that

$$\frac{d}{dt}R_{\psi\theta\phi}\begin{pmatrix} \mathbf{X} \\ \mathbf{Y} \\ \mathbf{Z} \end{pmatrix} = \begin{pmatrix} \dot{\mathbf{x}} \\ \dot{\mathbf{y}} \\ \dot{\mathbf{z}} \end{pmatrix} = \begin{pmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{pmatrix} R_{\psi\theta\phi}\begin{pmatrix} \mathbf{X} \\ \mathbf{Y} \\ \mathbf{Z} \end{pmatrix} \tag{31}$$

The first equality follows by differentiating equation 24 and the second from equation 16. Combining the two equalities results in:

$$\begin{pmatrix} 0 & -\omega_3 & \omega_2 \\ \omega_3 & 0 & -\omega_1 \\ -\omega_2 & \omega_1 & 0 \end{pmatrix} = \frac{d}{dt}\left(\rho_Z(\psi)\rho_Y(\theta)\rho_Z(\phi)\right)\left(\rho_Z(\psi)\rho_Y(\theta)\rho_Z(\phi)\right)^{-1} \tag{32}$$

which can be expanded (preferably in *Mathematica*, not by hand!) to give

$$\begin{aligned} \omega_1 &= \sin\psi\,\dot{\theta} - \sin\theta\cos\psi\,\dot{\phi}, \\ \omega_2 &= \cos\psi\,\dot{\theta} + \sin\theta\sin\psi\,\dot{\phi}, \\ \omega_3 &= \cos\theta\dot{\phi} + \dot{\psi}. \end{aligned} \tag{33}$$

From the first two equations,

$$\dot{\phi} = \frac{\omega_2\sin\psi - \omega_1\cos\psi}{\sin\theta}, \tag{34}$$

and $\phi$ can be obtained by numerical integration.

## 3.4 Implementation

Animation of the motion of a rigid body under no forces is accomplished by implementing the equations above in *Mathematica*. Given some graphics representation of the body at rest, together with its principal moments of inertia ($A$, $B$ and $C$) and initial angular velocity, the Euler angles are computed for evenly spaced values of time $t$. The graphics representation is then rotated from its rest position to the appropriate position at time $t$, to generate an animation frame. To highlight some of the classical theorems in the rigid body literature, objects of interest such as $\mathbf{h}$ and $\omega$ can be attached to the graphical representation of the rigid body.

The function FreeFall animates an object rotating in space under no external forces. An object with graphics representation shape and moments of inertia {mx,my,mz} is rotated in space with initial angular velocities {wx,wy,wz}. The animation goes from time start to time end in n steps

```
In[1]:= FreeFall[ shape_, {mx_,my_,mz_}, {wx_, wy_, wz_},
                                         {start_, end_ , n_} ] :=
    Block[ {i, gr, del, time,(* temporary variables *)
            eps, (* a small real number *)
            tc, (* constant of integration *)
            W1, W2, W3, (* angular velocities as functions of time *)
            w1, w2, w3, (* initial angular velocities *)
            Theta, Phi, Psi, (* Euler angles as functions of time *)
            theta, phi, psi, (* temporary Euler angles *)
            h, T, (* constants of the motion *)
            a, b, c, (* moments of inertia *)
            t0, t1, (* time interval of animation *)
            herpolhode, (* a list of angular velocity co-ords *)
            afile, tfile (* temporary files *)},

      (* switch off the display routine to save time *)
      savedisplayfunction = $DisplayFunction;
      $DisplayFunction = #&;

      (* open the frame file directory file *)
      afile=OpenTemporary[];

      (* initialize the herpolhode to an empty list *)
      herpolhode = {};

      (* make sure everything coming in is numeric *)
       a=N[mx];  b=N[my];  c=N[mz];
      w1=N[wx]; w2=N[wy]; w3=N[wz];
      t0=N[start]; t1=N[end];

      (* check for assumption on magnitudes of moments of inertia *)
      If[a>b && b>=c,Null,Block[{},
           Print["a = ",a,"  b = ",b,"  c = ",c];
           Print["Please permute the xyz frame until a>b>=c"];
           Return[Null]]];

      (* constants of the motion, angular momentum and kinetic energy *)
      h=(a^2 w1^2 + b^2 w2^2 + c^2 w3^2)^(1/2);
      T=(  a w1^2 +   b w2^2 +   c w3^2)/2;
      Print["Angular Momentum = ",h];
      Print["Kinetic Energy   = ",T];
      (* calculate alpha, beta, gamma, k * depending on sign h^2 -2bT *)
  If[ h^2 > 2 b T ,
   Block[{}, Print["Case h^2 > 2 b T"];
         alpha = ( (h^2 - 2 c T) / (a^2 - a c) )^(1/2);
          beta = ( (2 a T - h^2) / (b a - b^2) )^(1/2);
         gamma = ( (2 a T - h^2) / (c a - c^2) )^(1/2);
             p = ( (h^2 - 2 c T) (a - b) / (a b c) )^(1/2);
             k = ( ((b - c)(2 a T - h^2))/
                   ((a - b) (h^2 - 2 c T)) )^(1/2); ],
   Block[{}, Print["Case h^2 < 2 b T"];
         alpha = ( (h^2 - 2 c T) / (a^2 - a c) )^(1/2);
          beta = ( (h^2 - 2 c T) / (b^2 - b c) )^(1/2);
         gamma = ( (2 a T - h^2) / (c a - c^2) )^(1/2);
             p = ( (2 a T - h^2) (b - c) / (a b c) )^(1/2);
             k = ( ((a - b)(h^2 - 2 c T))/
                   ((b - c) (2 a T - h^2)) )^(1/2); ]
    ];

       (* calculate constant of integration *)
  tc = - N[Re[InverseJacobiSN[w2/beta,k] / p]];
  Print["constant of integration = ",tc];

       (* define angular velocities as functions of time *)
  Print["Setting up angular velocity functions"];
  eps=0.000000001;
  If [ Abs[k] > eps ,
     Block[ {} ,
        Print["k greater than eps, using Elliptic functions"];
(*      W1[t_] := N[Re[alpha JacobiDN[p(t-tc),N[k]]]];   *)
        W1[t_] := N[ alpha Sqrt[ 1 - (k^2) (JacobiSN[p(t-tc),N[k]])^2 ] ];
        W2[t_] := N[Re[ beta JacobiSN[p(t-tc),N[k]]]];
        W3[t_] := N[Re[gamma JacobiCN[p(t-tc),N[k]]]]; ],

     Block[ {} ,
        Print["k less than eps, using Trigonometric functions"];
        W1[t_] := N[Re[alpha]];
        W2[t_] := N[Re[ beta Sin[p(t-tc)]]];
        W3[t_] := N[Re[gamma Cos[p(t-tc)]]]; ] ];

  Print["Setting up solutions for Euler angles"];
  Theta[t_] :=    N[ArcCos[ c W3[t] / h ]];
    Psi[t_] :=    N[Re[ArcTan[ - ( b W2[t] ) / ( a W1[t] ) ]]];
    Phi[t_,st_] := Re[NIntegrate[ (W2[s] Sin[Psi[s]] -
       W1[s] Cos[Psi[s]]) / Sin[Theta[s]] , {s,N[st],N[t]} ]];

  del = N[(t1-t0)/n];
```

```
time = N[t0];
theta = Theta[t0];
phi = Phi[t0,tc] + N[Pi];
psi = Psi[t0] + N[Pi];

Print["generating animation frames"];
For[ i = 1 , i <= n+1 , i++ ,
   Block[ {} ,
         (* tack on the angular momentum and velocity vectors *)
         av = (0.99)*{W1[time],W2[time],W3[time]};
         am = (2T / h^2)*{a W1[time] , b W2[time] , c W3[time]};
         gr = Graphics3DJoin[ shape , Graphics3D[
                     { Line[ { {0,0,0} , am } ] ,
                       Line[ { {0,0,0} , av } ] } ] ];
         (* rotate the structure using Euler's angles *)
         Print["time = ",time,"   theta = ",theta,
               "  phi = ",phi,"  psi = ",psi];
         gr = RotateShape[gr,RotationMatrix3D[theta, phi, psi]];
         (* update the herpolhode with the
            tip of the angular velocity vector*)
         herpolhode = Append[herpolhode,Last[Last[Last[Last[gr]]]]];
         (* tack on the herpolhode and the
            invariable plane to the rotated structure *)
         gr = Graphics3DJoin[ gr , Graphics3D[
                  { Line[herpolhode],Polygon[(2T/h)*
                    {{1,1,1},{1,-1,1},{-1,-1,1},{-1,1,1}} ] }]];
         (* render the frame and save the result in a temporary file *)
         gr = Show[gr, RenderAll -> False ,
                       PlotRange -> (2T/h)*{ {-1,1},{-1,1},{-1,1} } ,
                           Boxed -> False ,
                       ViewPoint -> {4,3,-2}];
         tfile = OpenTemporary[];
         Display[tfile,gr];
         (* update the frame index file *)
         WriteString[afile,tfile,"\n"];
         Close[tfile];
         (* if there are more frames, calculate new Euler angles *)
         If[ i <= n , Block[ {},
                             time = time + del;
                             theta = Theta[time];
                             phi = phi + Phi[time,time-del];
                             psi = Psi[time] + N[Pi];
                           ] ]; ] ];
(* close the frame directory and reinstate the display routine *)
Close[afile];
$DisplayFunction = savedisplayfunction;
  ]
```

The implementation of FreeFall is straightforward, and only a few comments are made here.

The code starts by making sure that all arguments are numeric, and that $A > B \geq C$ (the case $A = B = C$ being trivial, since the angular velocity remains constant). The code then computes the constants $\alpha$, $\beta$, $\gamma$, $p$ and $k$ of the previous section, using different formulas for the cases $h^2 > 2bT$ and $h^2 < 2bT$, as explained above.

The constant of integration $t_c$ is computed using the function InverseJacobiSN. Sometimes this function returns a complex number with a very small imaginary part, rather than the desired real number; so the built-in function Re is applied to the result. This trick is used elsewhere in the code. In addition, if $k$ is very close to zero, it is preferable to approximate elliptic functions by trigonometric functions.

When computing $\phi$ the integration is only computed from the last frame, rather than from $t_c$ for each frame; hence the argument st.

To perform the rotation of the model through the previously calculated Euler angles, the following functions are employed.

```
In[2]:= RotationMatrix3D[theta_,phi_,psi_] := N[Transpose[
         { { - Sin[phi] Sin[psi] + Cos[theta] Cos[phi] Cos[psi] ,
               Cos[phi] Sin[psi] + Cos[theta] Sin[phi] Cos[psi] ,
             - Sin[theta] Cos[psi] },
           { - Sin[phi] Cos[psi] - Cos[theta] Cos[phi] Sin[psi] ,
               Cos[phi] Cos[psi] - Cos[theta] Sin[phi] Sin[psi] ,
               Sin[theta] Sin[psi] },
           {   Sin[theta] Cos[phi] ,
               Sin[theta] Sin[phi] ,
               Cos[theta] } } ]]

In[3]:= RotateShape[ shape_, rotmat_ ] := Block[ {}, Return[ shape /.
         { poly:Polygon[_] :> Map[(rotmat . #)&, poly, {2}],
           line:Line[_]    :> Map[(rotmat . #)&, line, {2}],
           point:Point[_]  :> Map[(rotmat . #)&, point,{1}] } ]]
```

RotationMatrix3D[theta,phi,psi] returns the orthogonal matrix required for performing an Euler rotation through Euler angles theta, phi and psi.

RotateShape[shape,rotmat] multiplies all vectors in in the graphics object shape by the rotation matrix rotmat.

## 3.5 Throwing a Tennis Racquet

If a tennis racquet is thrown into the air with angular velocity in the plane of the racquet but perpendicular to its shaft, the racquet seems to flip through half a revolution about its shaft as it rotates through one revolution about its initial angular velocity axis. This action is illustrated in a sequence of frames that can be viewed on a computer screen by installing the appropriate file from the disk accompanying this thesis. In addition, with a little practice it is easy to demonstrate the flipping action in real life. The action of the tennis racquet has been studied by several researchers, see *[Nevin and Jackson, 1977]* and *[Ashbough et al, 1991]*.

To model this motion with the program described in the previous section, a simple *box model* for the tennis racquet is created. If `Box[a,b,c]` represents a box centered at the origin of a rectangular coordinate system $xyz$, with dimensions $-a < x < a$, $-b < y < b$ and $-c < z < c$:

```
In[4]:= Box[a_,b_,c_] := Graphics3D[
         { Polygon[{ {a,b,c},{-a,b,c},{-a,-b,c},{a,-b,c} }],
           Polygon[{ {-a,b,c},{-a,b,-c},{-a,-b,-c},{-a,-b,c} }],
           Polygon[{ {a,b,-c},{-a,b,-c},{-a,-b,-c},{a,-b,-c} }],
           Polygon[{ {a,b,c},{a,b,-c},{a,-b,-c},{a,-b,c} }],
           Polygon[{ {a,b,c},{-a,b,c},{-a,b,-c},{a,b,-c} }],
           Polygon[{ {a,-b,c},{-a,-b,c},{-a,-b,-c},{a,-b,-c} }] } ]
```

then by shifting boxes around a `Racquet[a,b,c]` that combines a stick of length `c` with a square, flat box of side `a` and thickness `b` can be created:

```
In[5]:= TranslateShape[shape_, vec_List] := Block[{tvec = N[vec]},
         shape /. { poly:Polygon[_] :> Map[(tvec + #)&, poly, {2}],
                    line:Line[_]    :> Map[(tvec + #)&, line, {2}],
                    point:Point[_]  :> Map[(tvec + #)&, point,{1}] }
         ] /; Length[vec] == 3

In[6]:= Graphics3DJoin[gr1_,gr2_] :=
         Graphics3D[Join[First[gr1], First[gr2]]]

In[7]:= Racquet[a_,b_,c_] := Graphics3DJoin[
         TranslateShape[Box[b,a,a],{0,0,-a}] ,
         TranslateShape[Box[b,b,c],{0,0,c}] ]
```

As an example the following command will produce the rotated tennis racquet shown in figure 8.

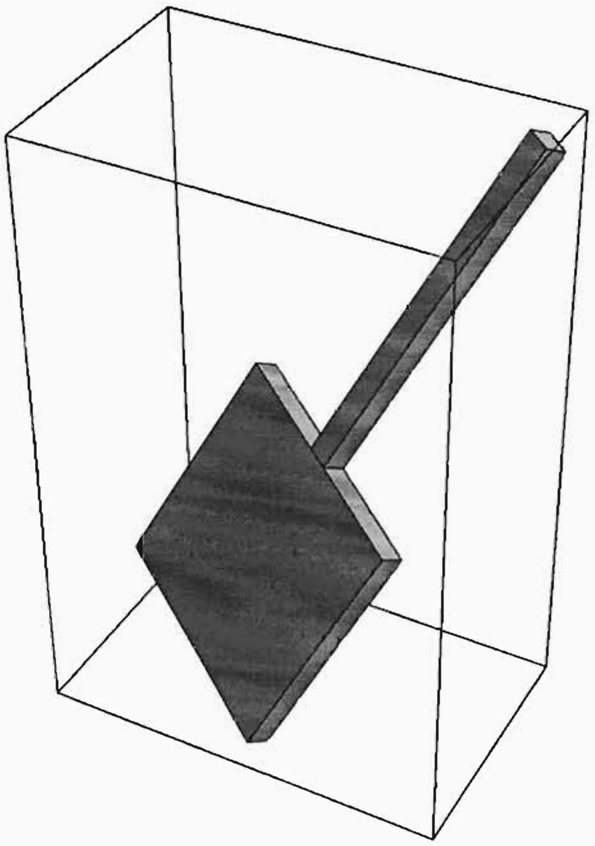

Figure 8: A rotated tennis racquet.

```
In[8]:= Show[RotateShape[Racquet[1,0.1,1.5],RotationMatrix3D[Pi/6,Pi/6,Pi/6]]]
```

## 3.6 Moments of Inertia for a Tennis Racquet

Finding the principal moments of inertia of the racquet is more complicated. To simplify matters, assume the masses of the two boxes making up the racquet are such that the center of mass of the racquet occurs at the join of the shaft to the head. In this case, one can use the parallel axis theorem *[Synge and Griffith, 1959, 7.110]* to show that if the function `BoxMoments[m,a,b,c]` returns the three principal moments of inertia of a box of mass `m` and dimensions `2a` by `2b` by `2c`, then the function `RacquetMoments[m,a,b,c]` will return the three principal moments of inertia of an object of mass `m` and shape `Racquet[a,b,c]`.

```
In[9]:= BoxMoments[m_,a_,b_,c_] := (m/3){b^2 + c^2 , c^2 + a^2 , a^2 + b^2}
```

```
In[10]:= RacquetMoments[m_,a_,b_,c_] := Block[
           {LeftMass, RightMass, LeftMoment, RightMoment},
            LeftMass = m c / (a + b);
            RightMass = m a / (a+b);
            LeftMoment = BoxMoments[LeftMass,b,a,a];
            RightMoment = BoxMoments[RightMass,b,b,c];
            Return[{ LeftMoment[[1]] +  LeftMass a^2 +
                     RightMoment[[1]] + RightMass c^2 ,
                     LeftMoment[[2]] +  LeftMass a^2 +
                     RightMoment[[2]] + RightMass c^2 ,
                     LeftMoment[[3]] + RightMoment[[3]] }]]
```

Now all the pieces are in place to generate an animation of a tennis racquet rotating in space under no external forces. The animation on the accompanying disk was generated with the command

```
In[11]:= FreeFall[Racquet[1,0.1,1.5], RacquetMoments[1,1,0.1,1.5],
                           {0,3,0.1}, {0,4.8,24}]
```

Note that `FreeFall` produces animation frames and writes them to animation files as they are generated without using the *Mathematica* command `DisplayAnimation`. The reason for this is that the author's *Mathematica* system does not have enough memory for holding the entire *Mathematica* system and 25 racquet frames in memory all at one time.

The *Multiflash* concept does not work well with true three dimensional motion. The frames interfere with one another and the result can be unintelligable. However, by chosing a few frames from the animation sequence one can generate an impression of the racquet's flipping motion. In figure 9 five frames from the animation are superimposed on one another.

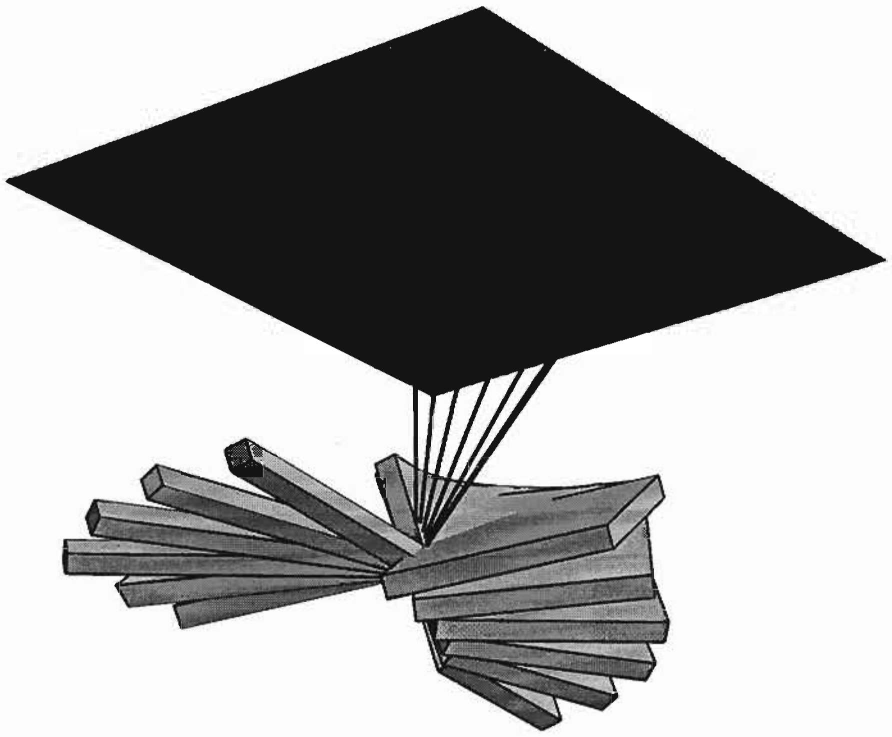

Figure 9: Five frames from the animation sequence.

## 3.7 Conclusion

A complex mechanics problem has been modelled with the aid of *Mathematica.* The result is a realistic animation of the process being modelled. The realisim of the animation assures the modeler that the model is a good one. In the next section *Mathematica* is employed to set up a laboratory for performing *phase plane analysis* of systems of ordinary differential equations.

# 4 A Phase-Plane-Plot Laboratory

This section presents a routine for animating phase-plane plots that vary as a parameter in the governing differential equations varies.

Initially, a procedure for producing a phase-plane plot from a set of planar autonomous differential equations, is investigated. Following that a parameter is allowed to vary in the set of equations and the variation in the phase plots is animated as the parameter changes.

After presenting the animation routine some specific bifurcation problems will be discussed. A standard text-book system is animated first and then a more exotic phase-plane bifurcation that occurs in a population dynamics model is tackled. *Mathematica* is used to find the bifurcation point.

## 4.1 Generating a phase plot

First a recipe is given for producing a phase plot of a planar, autonomous system of ordinary differential equations:

$$\begin{aligned} \dot{x} &= f(x,y) \\ \dot{y} &= g(x,y) \end{aligned} \tag{35}$$

The dependent variables are $x$ and $y$ and the independent variable is assumed to be time and will be denoted by the letter $t$.

The recipe has been known for some time, *[Kaplan, 1958]* and in this exposition *[Sacks, 1991]* is followed. First equations 35 and a bounding box for the phase variables $x$ and $y$ must be supplied. A list of *stationary points* of the system is then determined by solving the equations

$$\begin{aligned} f(x_s,y_s) &= 0 \\ g(x_s,y_s) &= 0 \end{aligned} \tag{36}$$

for $(x_s, y_s)$. Stationary points are fixed points of the system in that any solution that starts at one of these points at time $t = 0$ stays at that point for all time. However solutions that start *near* a stationary point can behave in a variety of different ways depending on the *classification* of the stationary point. This classification is determined by linearising the system in the neighbourhood of the stationary point. Expanding the right hand side of equations 35 in a Taylor series about the stationary point $(x_s, y_s)$, results in:

$$\begin{aligned} \dot{x} &= f(x_s, y_s) + (x - x_s)\frac{\partial f}{\partial x}(x_s, y_s) + (y - y_s)\frac{\partial f}{\partial y}(x_s, y_s) + \cdots \\ \dot{y} &= g(x_s, y_s) + (x - x_s)\frac{\partial g}{\partial x}(x_s, y_s) + (y - y_s)\frac{\partial g}{\partial y}(x_s, y_s) + \cdots \end{aligned} \tag{37}$$

Dropping the higher order terms, making use of equations 36 and using a shorthand notation for the partial derivatives, the linearised system in vector notation is:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \end{bmatrix} = \begin{bmatrix} f_x & f_y \\ g_x & g_y \end{bmatrix} \begin{bmatrix} x - x_s \\ y - y_s \end{bmatrix} \tag{38}$$

The matrix appearing in equation 38 is the *Jacobian* of the system and it is evaluated at the stationary point, $(x_s, y_s)$. A particular solution to equations 38 is given by expressions of the form:

$$\begin{aligned} x(t) &= \alpha e^{\lambda t} \\ y(t) &= \beta e^{\lambda t} \end{aligned} \tag{39}$$

On substituting back into equation 38 and performing a little algebra the following situation is obtained:

$$\begin{bmatrix} f_x - \lambda & f_y \\ g_x & g_y - \lambda \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} e^{\lambda t} = \begin{bmatrix} f_x & f_y \\ g_x & g_y \end{bmatrix} \begin{bmatrix} x_s \\ y_s \end{bmatrix}$$

Now since the right hand side is independent of $t$, the only way that this identity can hold for all $t$ is if

$$\begin{bmatrix} f_x - \lambda & f_y \\ g_x & g_y - \lambda \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

showing that $\begin{bmatrix} \alpha \\ \beta \end{bmatrix}$ and $\lambda$ must be an eigenvector, eigenvalue pair for the Jacobian. For the planar system under discussion there are, in general, two possible eigenvalues with two corresponding independent eigenvectors and a general solution to the linearised system is a linear combination of these particular solutions:

$$\begin{aligned} x(t) &= c_1\alpha_1 e^{\lambda_1 t} + c_2\alpha_2 e^{\lambda_2 t} \\ y(t) &= c_1\beta_1 e^{\lambda_1 t} + c_2\beta_2 e^{\lambda_2 t} \end{aligned} \tag{40}$$

This general solution to the linearised problem can be classified according to the signs of the real and imaginary parts of the eigenvalues, $\lambda_1$ and $\lambda_2$. Broadly speaking, if the real parts of the eigenvalues are negative then the stationary point is an *attractor* of any solution trajectory, $(x(t), y(t))$, that passes close to the stationary point. If the real parts of the eigenvalues are positive then the stationary point is a *repellor.* The classification procedure is well documented and usually forms part of any under-graduate syllabus; further details can be found in *[Kaplan, 1958]* and *[Sacks, 1991].* For animation purposes stationary points are classified into five major cases and one minor case:

| *Classification* | *Real parts* | *Imaginary parts* |
|---|---|---|
| attracting node | negative | zero |
| attracting spiral | negative | non-zero |
| repelling node | positive | zero |
| repelling spiral | positive | non-zero |
| saddle | opposite sign | zero |
| center | zero | |

The *borderline center* with zero real part is not really required since we will only be concerned with animating a situation where one major case changes to another major case passing momentarily through the borderline case. *Mathematica* is perfectly suited for the classification problem. If `Solve` is used to generate a list of stationary points then the Jacobian can be evaluated at the stationary points and the classification itself becomes a series of `If` statements.

```
In[1]:= cpType[jac_] := Module[{e1, e2},
          {e1, e2} = Eigenvalues[jac] // N;
          If[ Re[e1] == 0 && Re[e2] == 0, " center",
            StringJoin[
              If[ Re[e1] < 0 && Re[e2] < 0, "  stable", "  unstable"],
              If[ Im[e1] != 0 && Im[e2] != 0, " spiral",
                  If[ Re[e1] Re[e2] > 0, " node", " saddle"]] ]]]
```

```
In[2]:= ClassifyCP[rhs_, vars_] := Module[{jac = Outer[D, rhs, vars]},
          Map[Print[vars /. #, cpType[jac /. #]]&,
            Solve[rhs == {0, 0}, vars] ]; ]
```

To use ClassifyCP the functions appearing on the right hand side of equation 35 are supplied in a list as the parameter **rhs** while the dependent variables are supplied in a list as the variable **vars**. For example to classify the system $\{\dot{x} = y; \dot{y} = -x - x^2 - y\}$ just enter the command:

```
In[3]:= ClassifyCP[{y, - x - x^2 - y}, {x, y}]

Out[3]= {-1, 0}  unstable saddle
         {0, 0}  stable spiral
```

To obtain a phase plot a set of trajectories is calculated for each stationary point. The stationary points and the trajectories are then plotted clipped to the bounding box of the $xy$ phase-plane. *Mathematica's* numerical integration routine, NDSolve, is used to calculate trajectories. Given a system of differential equations, an initial point in the phase-plane and a time-span over which to integrate, the *Mathematica* code for generating a trajectory is as follows:

```
In[4]:= trajectory[{xdot_, ydot_}, {x_, y_}, {{x0_, y0_}, tm_} ] :=
          Module[ {sol, nSteps = 200, t, tmin, tmax},
            sol = First[ NDSolve[
              {x'[t] == (xdot /. {x -> x[t], y -> y[t]}),
               y'[t] == (ydot /. {x -> x[t], y -> y[t]}),
                x[0] == x0,
                y[0] == y0}, {x, y}, {t, tm}]];
              {tmin, tmax} = sol[[1,2,1]];
            Line[ Map[({x[#], y[#]} /. sol)&,
              Range[tmin, tmax - 0.001, (tmax - tmin)/nSteps]]]]
```

The system of differential equations together with the initial condition and the time span is passed to the Mathematica routine, NDSolve, which returns interpolating functions for the $x$ and $y$ components of the trajectory. The interpolating functions are evaluated at a number of points to produce a graphical representation of the trajectory.

The set of trajectories chosen for sketching depends on the classification of the stationary-point.

For a *node* four trajectories are sketched that start at $t = 0$ at points on an *oval* that surrounds the stationary point. The $x$ and $y$ extents of this oval are defined by a vector,

$\epsilon$, which is chosen to be some small fraction of the size of the bounding box. The four starting points on the $\epsilon$ oval are selected using the eigenvectors of the Jacobian matrix. The coordinates of the trajectory are calculated by using the starting points as initial conditions for equations 35 and numerically integrating from time $t = 0$ to some time $t = tmax$. If the node is a *repelling node* then the trajectory is calculated by letting time run positively, $tmax > 0$, while if the node is an *attracting node* then to see where the trajectory emanated from time must run negatively, $tmax < 0$.

For a *spiral* only two trajectories are sketched. The trajectories both start at the same point on the $\epsilon$ oval and time is allowed to run in both directions.

For a *saddle* a total of twelve trajectories need to be generated. The first four start at points on the $\epsilon$ oval pointed to by the eigenvectors of the Jacobian matrix and time runs positive or negative according to whether the corresponding eigenvalues are positive or negative. The other eight trajectories start at four new points on the $\epsilon$ oval which are generated by taking all possible arithmetic combinations of the two eigenvectors. Eight trajectories are generated by letting time run in both positive and negative directions from these four starting points.

If the stationary-point is the borderline *center* then four trajectories are chosen at increasing distances along the $\epsilon$ vector.

In order to implement these ideas in *Mathematica* use is made of the following function which scales a vector so that it points to a position on the $\epsilon$ oval:

```
In[5]:= scale[vec_,eps_] := vec / Sqrt[Apply[Plus, (vec/eps)^2]]
```

Now given a Jacobian matrix and a small vector $\epsilon$ the required pattern of trajectory starting points is returned by the function `initialPoints`. Note that these initial points are simply *offsets* from the stationary point in question.

```
In[6]:= initialPoints[jac_, eps_] :=
          Module[{esys, lam1, lam2, v1, v2, t1, t2},
            esys = Eigensystem[jac] // N;
            {lam1, lam2} = First[esys];
            If[ Re[lam1] == 0 && Re[lam2] == 0,
              { { eps, 1}, { 2 eps, 1}, { 4 eps, 1}, { 8 eps, 1} },
              ( {v1, v2} = Map[scale[Re[#], eps]&, Last[esys]];
                {t1, t2} = Map[Sign[Re[#]]&, {lam1, lam2}];
                If[ Im[lam1] != 0 && Im[lam2] != 0,
                    {{v1, 10 t1}, {v1, -t2}},
                    If[ Re[lam1] Re[lam2] > 0,
                        {{v1, t1}, {-v1, t1}, {v2, t1}, {-v2, t1}},
                        { {v1, t1}, {-v1, t1}, {v2, t2}, {-v2, t2},
                              {scale[v1 + v2, eps], t1},
                              {scale[v1 + v2, eps], t2},
                              {scale[v1 - v2, eps], t1},
                              {scale[v1 - v2, eps], t2},
                              {scale[- v1 + v2, eps], t1},
                              {scale[- v1 + v2, eps], t2},
                              {scale[- v1 - v2, eps], t1},
                              {scale[- v1 - v2, eps], t2} } ] ] ) ] ]
```

To produce a *sketch* of the phase-plane near the stationary point `initialPoints` is used to classify the stationary point and chose a set of initial points which must then be passed to the integrating routine. This set of trajectories is called a *cluster* and can be generated by mapping `trajectory` onto `initialPoints` as follows:

```
In[7]:= clusterPlot[{xdot_, ydot_}, {x_, xmin_, xmax_},
          {y_, ymin_, ymax_}, jac_, tmax_, eps_, {xc_, yc_}] :=
            { RGBColor[(xc - xmin)/(xmax - xmin), 0.5,
              (yc - ymin)/(ymax - ymin)],
              Map[ trajectory[{xdot, ydot}, {x, y},
                {{xc, yc} + First[#], tmax Last[#]}]&,
                initialPoints[jac /. {x -> xc, y -> yc}, eps] ] }
```

Finally a full phase-plane sketch consists of a number of cluster sketches so the function `clusterPlot` must be mapped to all the stationary points in the bounding box. Before the final phase plot is displayed some `Dashing` is introduced so as to indicate the direction in which the trajectories are progressing.

```
In[8]:= PhasePlot[{xdot_, ydot_}, {x_, xmin_, xmax_},
                {y_, ymin_, ymax_}, tmax_, options___] :=
          Module[ {jac, eps, cp},
            jac = Outer[D, {xdot, ydot}, {x, y}];
            eps = {xmax - xmin, ymax - ymin} / 30.;
            cp = Select[{x, y} /. Solve[{xdot == 0, ydot == 0}, {x,y}],
                  Inner[Greater, #, {xmin, ymin}, And] &&
                    Inner[Less, #, {xmax, ymax}, And] &];
            Show[ Graphics[
              {Dashing[{0.005,0.01,0.010,0.01,0.015,0.01,0.020,0.01}],
                Map[clusterPlot[{xdot, ydot}, {x, xmin, xmax},
                      {y, ymin, ymax}, jac, tmax, eps, #]&, cp]},
              PlotRange -> {{xmin, xmax}, {ymin, ymax}},
              Axes -> True, Frame -> True, options] ] ]
```

As an first example, PhasePlot is used to sketch a phase-plane solution to Lienard's equations, *[Sacks, 1991].*

$$\dot{x} = y, \qquad \dot{y} = -x - x^2 - y \tag{41}$$

This system has a stable spiral at $(0,0)$ and a saddle point at $(-1,0)$. These features which are shown in figure 10 are easily generated using the command:

```
In[9]:= PhasePlot[ {y,-x-x^2-y}, {x,-2,2}, {y,-2,2}, 6,
        PlotLabel -> "Lienard's system", FrameLabel -> {"x","y"} ]}
```

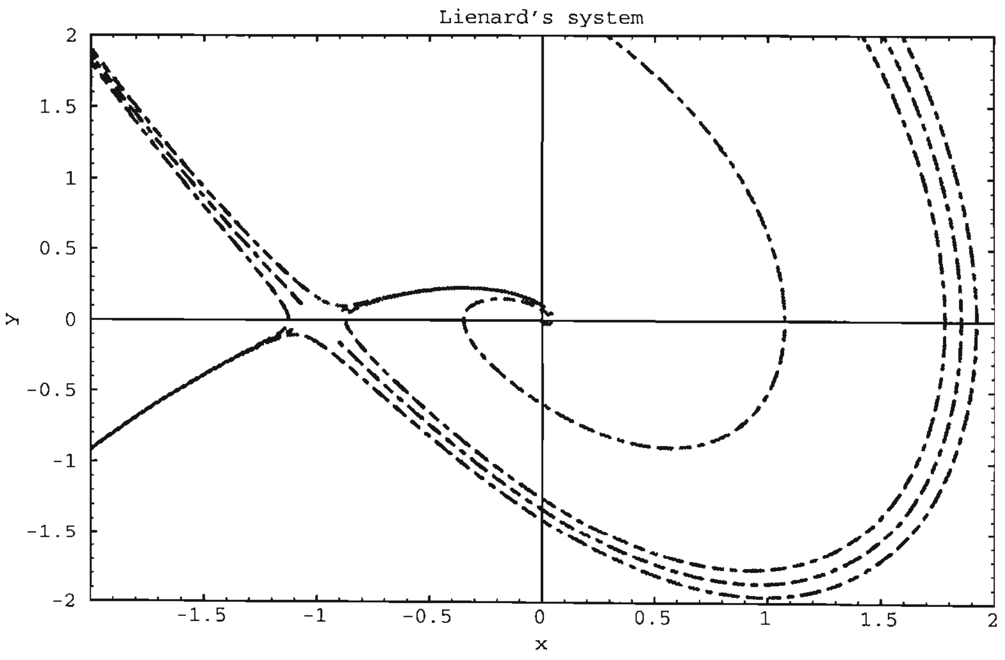

Figure 10: Lienard's system

Although not guaranteed, `PhasePlot` will often bring to light any limit cycles inherent in the system. Consider a standard van der Pol system, *[Kaplan, 1958]*:

$$\dot{x} = y, \qquad \dot{y} = \frac{1}{2}(1 - x^2)y - x \tag{42}$$

This system has an unstable spiral at the origin which is surrounded by a stable limit cycle as shown in figure 11. The *Mathematica* command for producing this phase-plot is:

```
In[10]:= phasePlot[{y, 0.5 (1 - x^2) y - x}, {x,-3,3}, {y,-3,3}, 20,
         PlotLabel -> "van der Pol's system", FrameLabel -> {"x","y"} ]
```

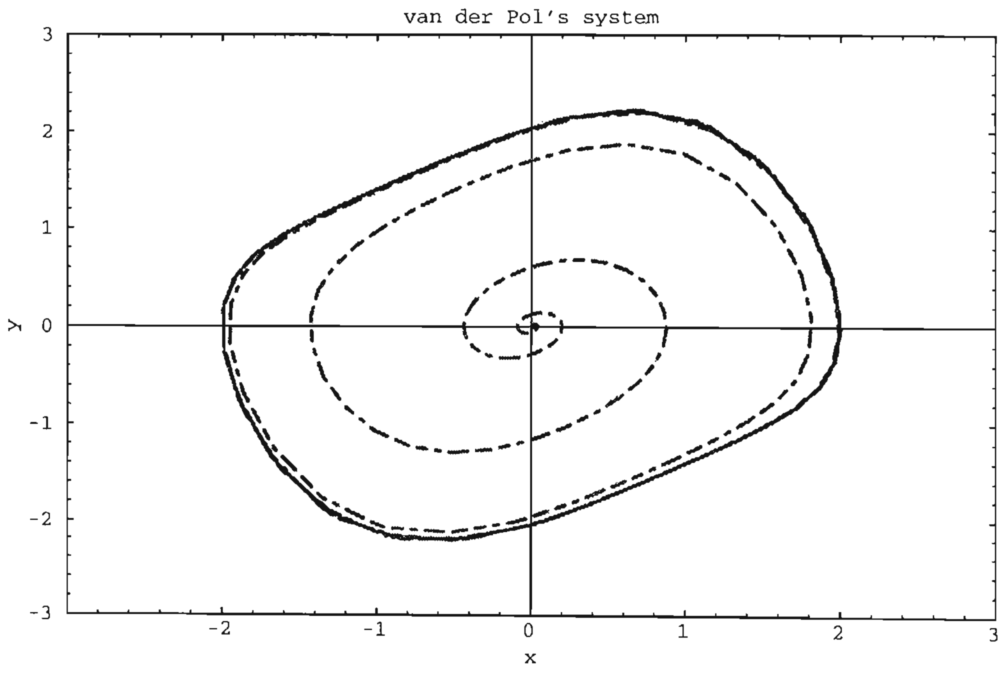

Figure 11: Van der Pol's system

## 4.2 Animating phase plots

In this section a single parameter, $k$, is introduced into the governing differential equations.

$$\begin{aligned} \dot{x} &= f(x,y,k) \\ \dot{y} &= g(x,y,k) \end{aligned} \tag{43}$$

As the parameter varies over a fixed real interval the nature of the phase plot can vary. The stationary points may move in the phase plane and the classification of the stationary points may change. *Mathematica* can be used to animate this change by using `phasePlot` to generate each frame of the animation as the parameter varies:

```
In[11]:= PhasePlotAnimation[eqns_, xrange_, yrange_, t_,
              {k_, kmin_, kmax_}, noFrames_, options___] :=
           Map[PhasePlot[ eqns /. k -> #, xrange, yrange, t,
                   PlotLabel -> StringJoin[
                     ToString[k], " = ", ToString[N[#,3]]], options ]&,
                Range[kmin, kmax, (kmax - kmin)/(noFrames - 1)] ]
```

`phasePlotAnimation` returns a list of `Graphics` objects. The actual animation is viewed by passing this list to the standard Mathematica routine, `DisplayAnimation`.

As an example a parameter is introduced into van der Pol's equations:

$$\dot{x} = y, \qquad \dot{y} = k(1 - x^2)y - x \tag{44}$$

and the frames shown in figure 12 can be produced by varying the parameter, $k$, over the interval $(0.1, 9.0)$.

```
In[12]:= PhasePlotAnimation[ { y, k(1 - x^2)y - x}, {x,y},
           {{-5,5}, {-15,15}}, 50, k, {0.1,9.0}, 28,
           Ticks -> None, FrameTicks -> None,
           PlotRegion -> {{0,0.3},{0,0.3}}  ]
```

When the frames are used in an animation sequence a *deformation* of the limit cycle with increasing $k$ is observed. The frames show a change of classification for the stationary point at $(0, 0)$ as the parameter passes through the value $k = 2$. The classification changes from a repelling spiral to a repelling node. This is called a *bifurcation* of the system. This bifurcation is not very interesting as the limit cycle exists before and after the bifurcation. In the next section a more interesting bifurcation will be animated.

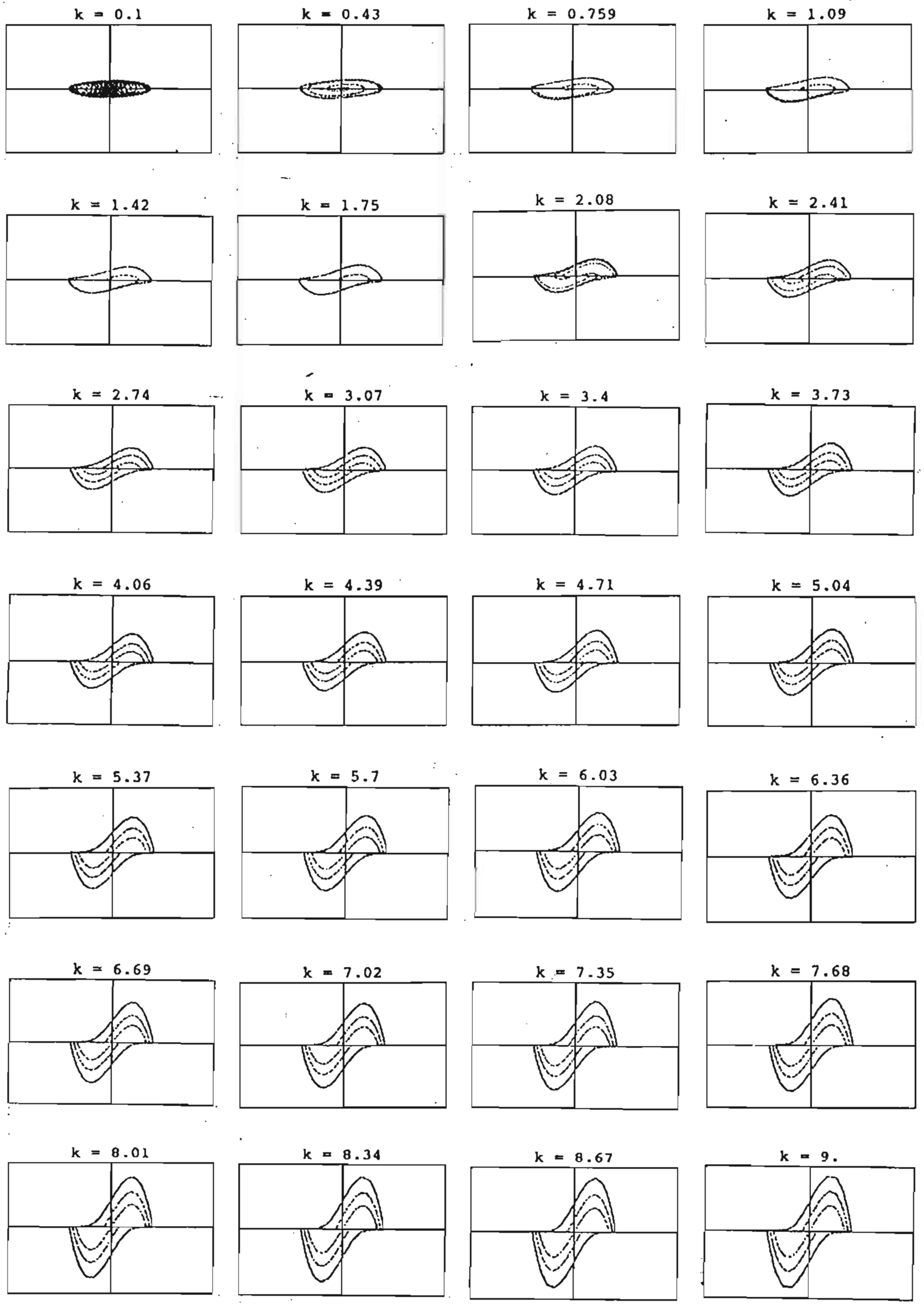

Figure 12: Animation frames for Van der Pol's system

## 4.3 Bifurcation analysis

A more interesting bifurcation occurs when the classification changes from attracting spiral to repelling spiral and visa-versa. In these cases a limit cycle can appear or disappear as the parameter passes through the bifurcation point. This phenomenon is called *Hopf bifurcation*, see *[Marsden and McCracken, 1976]* for a full description.

In a recent paper, *[Swart, 1994]*, the well known autonomous planar model of *[Caughley, 1976]* was used to study limit cycle behaviour in an elephant-tree ecology of the *Tuli Block* game park in northern Botswana. The governing equations are

$$\begin{aligned} \dot{x} &= x(a - bx - \frac{cy}{x+m}) \\ \dot{y} &= y(-k + \frac{hx}{y+n}) \end{aligned} \tag{45}$$

where $x$ is the density of the trees, $y$ is the density of the elephants, $a$ is the natural rate of increase of the trees, $b$ is the degree to which addition of a further unit of tree density depresses the rate of increase of trees, $c$ is the rate of elimination of trees per elephant, $m$ is a threshold density of trees, $k$ is the rate of decrease of elephants in the absence of trees, $h$ is the rate at which elephant decrease is ameliorated at a given ratio of trees to elephants and $n$ is the threshold density of elephants.

Based on observed data from the Tuli Block and other parks in Southern Africa estimates for all the parameters, apart from $k$, were derived in *[Swart, 1994]*, and *[Swart and Duffy, 1987]*. These parameter estimates are:

```
In[13]:= params = { a -> 0.04,
                    b -> 0.000004,
                    c -> 240,
                    h -> 0.000002,
                    m -> 1000,
                    n -> 0.1 }
```

The system governed by equations 45, with parameters indicated above, can exhibit a limit cycle behavior. For example adding `k -> 0.01` to the list of parameters and issuing the *Mathematica* commands:

```
In[14]:= xdot = x(a - b x - c y / (x + m)); ydot = y(- k + h x / (y + n));

In[15]:= phasePlot[ {xdot,ydot} /. params /. k -> 0.01,
                    {x,-500,12000}, {y,-0.05,1.0}, 500,
                    PlotLabel -> "Competing Species",
                    FrameLabel -> {"Trees","Elephants"} ]
```

results in the phase-plot shown in figure 13.

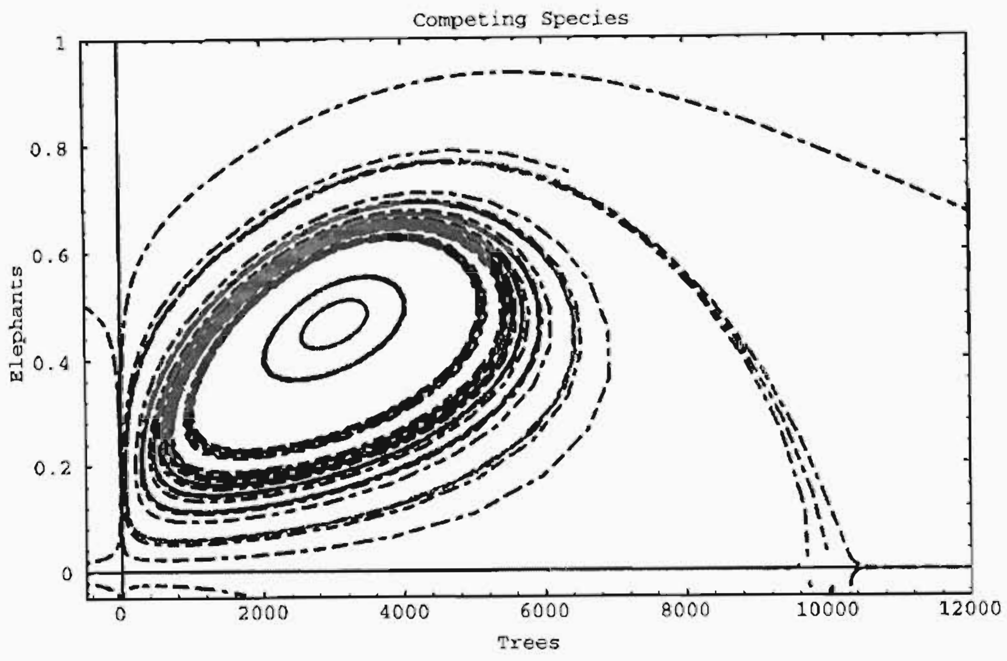

Figure 13: An elephant/tree competing species system (k=0.01)

In *[Swart, 1994]* it is shown that the remaining parameter, $k$ has a Hopf bifurcation point, $k_0$, one side of which, $k > k_0$, the elephant-tree system tends to a positive stable equilibrium point and the otherside of which, $k < k_0$, the elephant-tree system approaches a stable limit cycle.

Large limit cycles are undesirable in game parks. The reason for this is that when the system follows a large limit cycle there will be a period in which the trees have been devastated by the elephants and the elephants are in the process of dying out. This devastation can cause tourism to drop dramatically.

The problem is solved by controlled culling by the game park managers. The question is how much culling is required? In this section *Mathematica* is used to find the bifurcation point $k_0$ and an animation of the phase plots on both sides of $k_0$ is produced. This will allow game park managers to select a desirable value of $k$ that will suit their tourism requirements.

The first task is to find the stationary point of interest in equations 14:

```
In[16]:= xdot = x( a - b x - c y / (x + m));

In[17]:= ydot = y( - k + h x / (y + n));

In[18]:= sPs = Solve[ {Numerator[Together[xdot]] == 0 ,
                       Numerator[Together[ydot]] == 0}, {x,y} ];
```

This results in 6 possible stationary points, the fifth one of which is in the positive quadrant.

```
In[19]:= sP = sPs[[5]]

Out[19]= { x -> (a/b - (c h)/(b k) - m +
            (c^2 h^2 - 2 a c h k + a^2 k^2 + 2 b c h k m +
             2 a b k^2 m + b^2 k^2 m^2 + 4 b c k^2 n)^(1/2)/(b k))/2,
           y -> (-((c h^2)/(b k^2)) + (a h)/(b k) - (h m)/k - 2 n +
            (h (c^2 h^2 - 2 a c h k + a^2 k^2 + 2 b c h k m +
             2 a b k^2 m + b^2 k^2 m^2 + 4 b c k^2 n)^(1/2))/(b k^2))/2 }
```

The bifurcation point occurs when the eigenvalues of the Jacobian are purely imaginary. It is easy to show that the real part of the eigenvalues is given by half the trace of the Jacobian. So first the Jacobian is calculated in the usual manner:

```
In[20]:= jac = { {D[xdot,x], D[xdot,y]}, {D[ydot,x], D[ydot,y]} }

Out[20]= {{a - b x - (c y)/(m + x) + x (-b + (c y)/(m + x)^2),
           -((c x)/(m + x))},
          {(h y)/(n + y),
           -k - (h x y)/(n + y)^2 + (h x)/(n + y)}}
```

Then the value of **k** that makes the trace of the Jacobian zero must be calculated. So the numerator of the trace of the Jacobian is calculated and this results in a large expression in **k**.

```
In[21]:= tr = Numerator[Together[ (jac[[1,1]]+jac[[2,2]]) /. sP /. params ]]
```

```
Out[21]= - 6.09322314049587 10^-9 + 1.650247933884298 10^-6 k -
  0.0001844892561983471 k^2 + 0.00893355371900827 k^3 -
  0.07860422405876961 k^4 - 5.568411386593205 k^5 + 24.42607897153352 k^6 +
  0.00001269421487603306 (2.304 10^-7 - 0.00003456 k + 0.00232 k^2)^(1/2) -
  0.002485950413223141 k (2.304 10^-7 - 0.00003456 k + 0.00232 k^2)^(1/2) +
  0.1696969696969698 k^2 (2.304 10^-7 - 0.00003456 k + 0.00232 k^2)^(1/2) -
  2.475665748393024 k^3  (2.304 10^-7 - 0.00003456 k + 0.00232 k^2)^(1/2) -
  112.0293847566575 k^4  (2.304 10^-7 - 0.00003456 k + 0.00232 k^2)^(1/2) +
  505.0505050505051 k^5  (2.304 10^-7 - 0.00003456 k + 0.00232 k^2)^(1/2)
```

Now, with some luck, *Mathematica* will be able to find roots for this expression:

```
In[22]:= bfs = Solve[ tr == 0,k]

Out[22]= { {k -> -0.0843747900405854},
           {k -> -0.05037192275632139},
           {k -> -0.02000000000005608},
           {k -> -0.00002917829285930187},
           {k -> -0.00002749310579020291 - 0.00002949113462372171 I},
           {k -> -0.00002749310579020291 + 0.00002949113462372171 I},
           {k -> 0.00001840773511470339 - 0.0000420908681419152 I},
           {k -> 0.00001840773511470339 + 0.0000420908681419152 I},
           {k -> 0.00004734903426153325},
           {k -> 0.01068432534785173},
           {k -> 0.04782303038308367},
           {k -> 0.2362393570659732 }  }
```

For physical reasons the bifurcation point must be real and positive. The last four are candidates that fit these requirements. To select the required bifurcation point for k, the stationary point is evaluated using each candidate. It turns out that only the 10th candidate produces a suitable stationary point.

```
In[23]:= sP /. params /. bfs[[10]]

Out[23]= { x -> 3036.98191405709, y -> 0.4684929680033993 }

In[24]:= k0 = k /. bfs[[10]]

Out[24]= 0.01068432534785173
```

Finally the limit cycle animation is produced by letting $k$ vary about $k_0$ by 10% in either direction. The required parameters for *PhasePlotAnimation* are:

```
In[25]:= kInt = {k, 0.5 k0, 1.5 k0};

In[26]:= PhasePlotAnimation[ {xdot,ydot} /. params,
           {x,-500,12000}, {y,-0.05,1.0}}, 500, kInt, 28,
           Ticks -> None, FrameTicks -> None,
           PlotRegion -> {{0,0.3},{0,0.3}} ]
```

The animation frames are shown in figure 14. It is evident from the animation frames that the limit cycle is most prominent in frame `k=0.0104`, while in frame `k=0.0108`, the limit cycle has vanished indicating that a *Hopf bifurcation* has taken place. The animations produced in this section are stored on the stiffy disk attached to the thesis. The instructions for viewing the animations are given in the appendix.

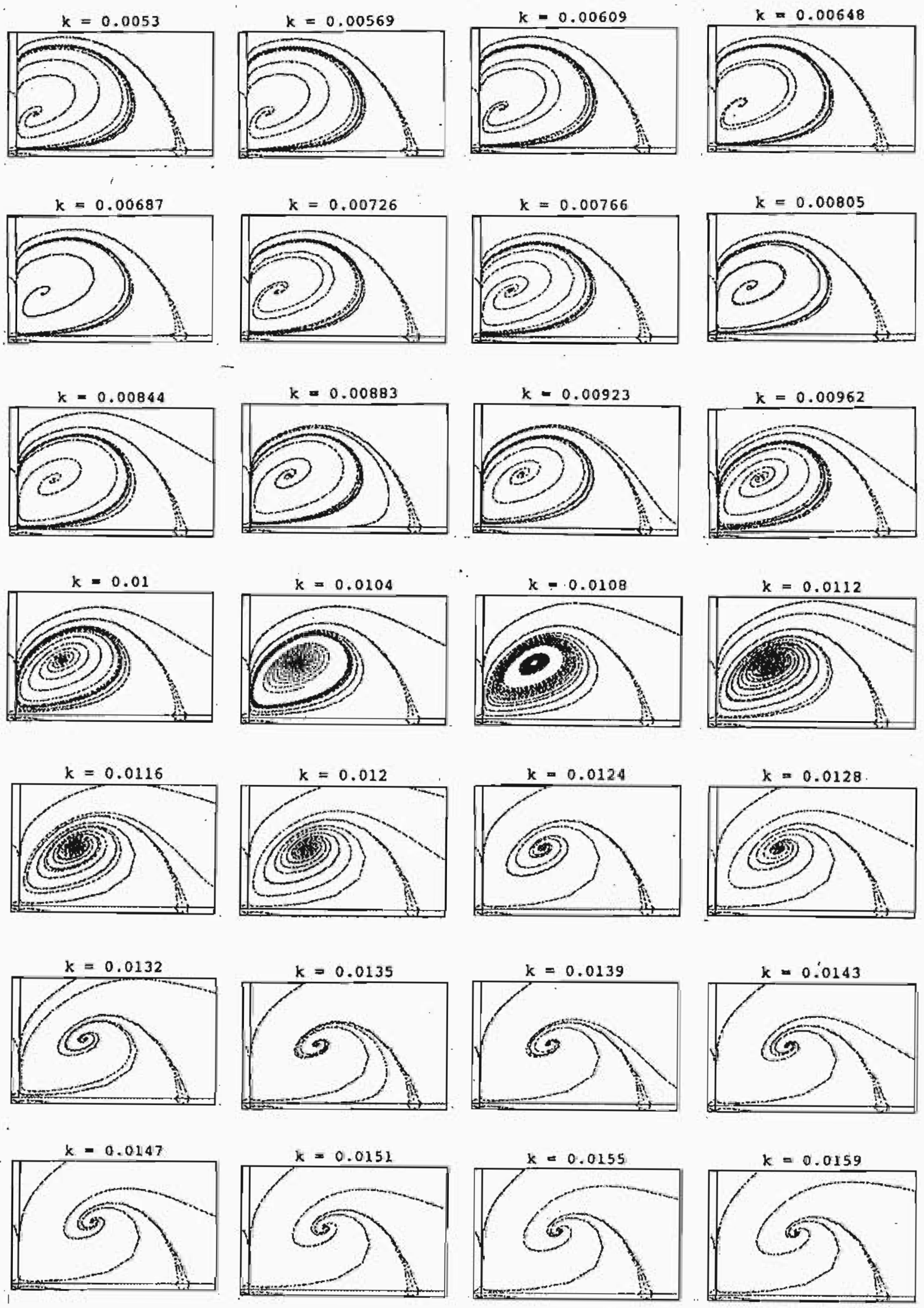

Figure 14: Animation frames for the Elephant/Tree system

## 4.4 Conclusion

There have been many attempts at programs that produce phase-plane plots of one parameter planar systems. The author has had experience using a program called PHASER, see *[Kocak, 1986]*. However, with PHASER, initial conditions for trajectories have to be found by trial and error or hand calculation on the part of the experimentor. In other words PHASER is just an integration machine with no built-in analytical skills.

The state of the art in intelligent analysis of planar systems is probably a program called POINCARE described in *[Sacks, 1991]*. Sacks claims that POINCARE combines theoretical knowledge about differential equations with numeric, symbolic, geometric and probabilistic reasoning.

In this section animation was introduced as a powerful visualization tool for one parameter planar systems. Although the analysis performed does not compete with existing state of the art programs, it has been demonstrated that *Mathematica* could provide a platform on which to develop such a rival. In the next section *Mathematica* is used to build a tomography laboratory.

# 5 A Tomography Laboratory

## 5.1 The Tomography Problem

The tomography problem is to reconstruct a two dimensional image $f(x,y)$ from a set of projections of the image at various angles in the interval $[0,2\pi]$. A *projection* $p_\theta(t)$ is a function of one variable generated by calculating line integrals of $f(x,y)$ along parallel lines. The geometry is given in figure 15.

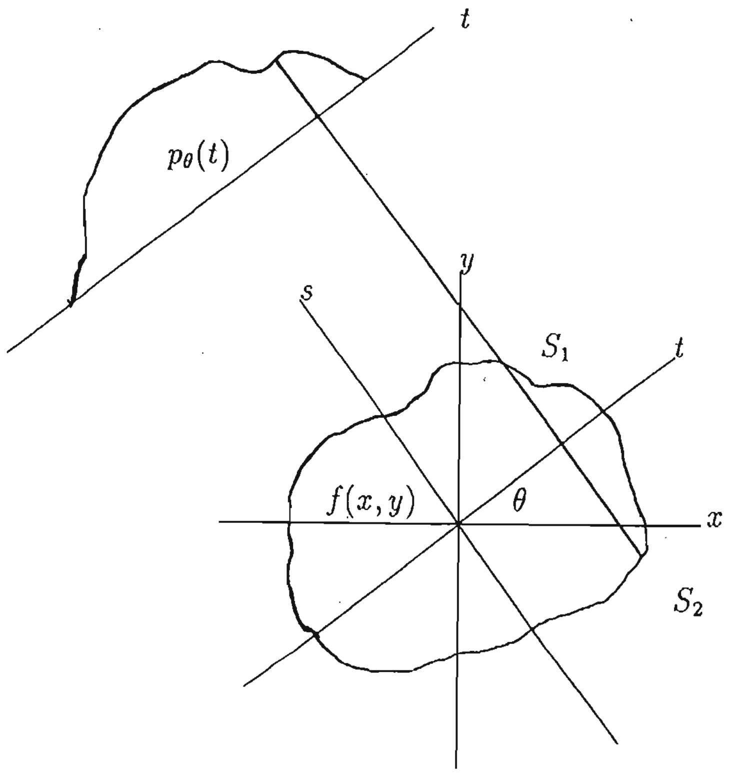

Figure 15: Projection Geometry.

Consideration of figure 15 yields the following mathematical description of a projection:

$$p_\theta(t) = \int_{-\infty}^{\infty} f(x(s,t), y(s,t))ds = \int_{-\infty}^{\infty} f(t\cos\theta - s\sin\theta, s\cos\theta + t\sin\theta)ds \qquad (46)$$

Assuming that the image, $f(x,y)$, lies inside the unit circle a *tomography algorithm* takes a set of projection data, $\{p_{\frac{i\pi}{n-1}}(\frac{2j-(n-1)}{n-1})\}_{i,j=0..n-1}$, as input and produces a reconstructed image, $\hat{f}(x,y)$, as output. As the resolution, $n$, increases $\hat{f}$ should approach $f$. If continuous data is available the tomography algorithm should reconstruct $f$ exactly.

A common method for testing tomography algorithms is to construct artificial images by overlaying ellipses of different brightnesses. The reason for using ellipses is that, as will be shown shortly, the projection through an ellipse can be calculated exactly. Thus a test image with exact projection data can be constructed by a simple algorithm and expensive scanning equipment is not required.

## 5.2 A Projection through an Ellipse

Given an ellipse with major axis of length $2A$ along the $x$ axis and minor axis of length $2B$ along the $y$ axis and center at the origin:

$$\frac{x^2}{A^2}+\frac{y^2}{B^2}=1, \tag{47}$$

consider an image $f(x,y)$, of brightness $\lambda$, if $(x,y)$ is inside the ellipse, and brightness 0, if $(x,y)$ is outside the ellipse. Consideration of figure 15 yields the following formula for a projection through such a centred ellipse:

$$p^c_\theta(t)=\int_{S_2}^{S_1}\lambda ds=\lambda(S_1-S_2) \tag{48}$$

where $S_1$ and $S_2$ are the $s$ components of the intersections of the $t$'th projection with the edges of the ellipse. Calculating the $(x,y)$ coordinates of the points $(t,S_1)$ and $(t,S_2)$ and substituting them into equation 47 for the ellipse and solving for $S_1-S_2$ yields:

$$p^c_\theta(t)=\frac{2AB}{a^2(\theta)}\sqrt{a^2(\theta)-t^2} \tag{49}$$

where

$$a^2(\theta)=A^2\cos^2\theta+B^2\sin^2\theta \tag{50}$$

and thus the projection through an ellipse can be calculated exactly for any projection angle $\theta$. Now consider an ellipse that is rotated about the origin through an angle $\alpha$ and the translated to a point $(x_1, y_1)$ on the $xy$ plane. A projection $p_\theta^{rt}(t)$ through such a rotated and translated ellipse is given by

$$p_\theta^{rt}(t) = p_{\theta-\alpha}^{c}(t-\delta) \tag{51}$$

where $\delta = x_1 \cos\theta + y_1 \sin\theta$. Using this result it is possible to construct a variety of images by superimposing ellipses of various shades of grey and calculate exactly the projection through such an image at any projection angle $\theta$. The reason for this is that the projection operation is linear and to obtain a projection of superimposed ellipses one superimposes projections.

## 5.3 A test Phantom and its Shadow

Shepp and Logan in *[Shepp and Logan, 1974]* were the first to use these ideas for testing tomography algorithms. The data for their test image, or *phantom* as they called it, is published in *[Kak and Slaney, 1988]* and is used in the *Mathematica* tomography laboratory to generate a test image by superimposing ellipses. Each ellipse in the phantom is specified by giving the *center coordinates* of the ellipse, the length of the *semi-major* axis, the length of the *semi-minor* axis, the *orientation* of the ellipse and a *grey-scale* value for the brightness of the ellipse.

```
In[1]:= res=128;

In[2]:= ellipse[{c_,a_,b_,r_,g_}] := Module[ {px, py},
          px=N[{Cos[r Degree],Sin[r Degree]}];
          py=N[{-Sin[r Degree],Cos[r Degree]}];
          Table[
            If[ N[(((#-c).px)^2/a^2 + ((#-c).py)^2/b^2)]&[{x,y}] < 1,g,0],
              {y,-1,1,2/(res-1)}, {x,-1,1,2/(res-1)}]]

In[3]:= SheppLoganData = { { {0,0},          0.92,  0.69,    90,  2  },
                           { {0,-0.0184},    0.874, 0.6624,  90, -0.9},
                           { {0.22,0},       0.31,  0.11,    72, -0.1},
                           { {-0.22,0},      0.41,  0.16,   108, -0.1},
                           { {0,0.35},       0.25,  0.21,    90,  0.3},
                           { {0,0.1},        0.046, 0.046,    0,  0.3},
                           { {0,-0.1},       0.046, 0.046,    0,  0.3},
                           { {-0.08,-0.605},0.046, 0.023,    0,  0.3},
                           { {0.06,-0.605}, 0.046, 0.023,   90,  0.3},
                              { {0,-0.605},    0.023,  0.023,    0,  0.3} }

In[4]:= SheppLoganImage = Apply[Plus,Map[ellipse, SheppLoganData]]
```

The function `ellipse[{c,a,b,r,g}]` generates an image with pixel value `g` inside the ellipse and pixel value `0` outside. The Shepp-Logan phantom is generated by mapping `ellipse` onto the Shepp-Logan data and superimposing the resulting images. *Mathematica* supplies a function for viewing images called `ListDensityPlot` which can be adapted to view the Shepp-Logan phantom shown in figure 16.

```
In[5]:= ShowImage[image_] := ListDensityPlot[image,Mesh->False]

In[6]:= ShowImage[SheppLoganImage]
```

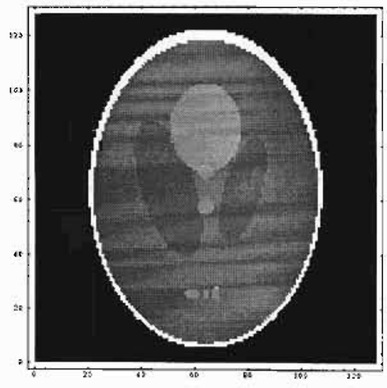

Figure 16: Shepp-Logan phantom, generated by superimposing ellipses.

To obtain a projection through an ellipse centred at the origin with major axis in the $x$ direction, equation 49 must be implemented in *Mathematica*.

```
In[7]:= project[{a_,b_,g_},th_,t_] := Module[ {asq},
          asq = N[a^2 Cos[th]^2 + b^2 Sin[th]^2];
          If[N[t^2] > asq,0.0,N[2 g a b Sqrt[asq-t^2] / asq]]]
```

A *shadow* of a re-orientated translated ellipse can now be generated by implementing equation 51 in *Mathematica* and projecting the ellipse at discrete angles $\{i\frac{\pi}{res-1}\}_{i=0..res-1}$. Each projection is sampled $res$ times and the projections are stored as rows in the shadow image. Since the projection operation is linear a projection of the Shepp-Logan phantom is generated by superimposing shadows cast by each ellipse in the Shepp-Logan data set.

```
In[8]:= ellipseShadow[{c_,a_,b_,r_,g_}] := Module[ {shift},
         shift = c[[1]] Cos[th] + c[[2]] Sin[th];
         Table[ project[{a,b,g},th - r Degree, t - shift ] ,
           {th,0,Pi,Pi/(res-1)},{t,-1,1,2/(res-1)}]]

In[9]:= SheppLoganShadow = Apply[Plus,Map[ellipseShadow, SheppLoganData]]

In[10]:= ShowImage[SheppLoganShadow]
```

The resulting Shepp-Logan shadow image is shown in figure 17.

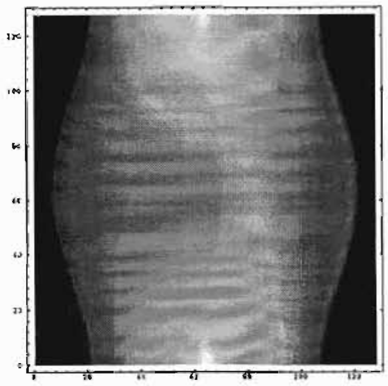

Figure 17: Shepp-Logan shadow, generated by superimposing ellipse shadows.

The task at hand is to reconstruct the Shepp-Logan phantom from the phantom's shadow.

## 5.4 The Projection-Slice Theorem

Most successful reconstruction algorithms are based on the fact that the Fourier transform $P_\theta(\omega)$, of a projection $p_\theta(t)$ through an image $f(x,y)$, is equal to the two dimensional Fourier transform of the image $F(u,v)$, evaluated along the polar line $(\omega\cos\theta, \omega\sin\theta)$.

This fact is known as the *Projection-Slice theorem*, and can be found in any tomography text, see for example *[Kak and Slaney, 1988]*. To verify the theorem start with the two dimensional forward Fourier transform:

$$F(u,v) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)e^{-j2\pi(ux+vy)}dxdy \tag{52}$$

and then express the Fourier transform of a projection in a similar manner. First the Fourier transform of a projection is given by:

$$P_\theta(\omega) = \int_{-\infty}^{\infty} p_\theta(t) e^{-j2\pi\omega t} dt \tag{53}$$

then using the definition of a projection from equation 46 one obtains:

$$P_\theta(\omega) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(t\cos\theta - s\sin\theta, s\cos\theta + t\sin\theta) e^{-j2\pi\omega t} ds dt. \tag{54}$$

Now make the substitution $x = t\cos\theta - s\sin\theta, y = t\sin\theta + s\cos\theta$ which has unit Jacobian to get

$$P_\theta(\omega) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y) e^{-j2\pi(x\omega\cos\theta + y\omega\sin\theta)} dx dy. \tag{55}$$

Comparing equation 55 with equation 52 yields the projection-slice theorem:

$$P_\theta(\omega) = F(\omega\cos\theta, \omega\sin\theta). \tag{56}$$

The projection-slice theorem in this form is seldom used for reconstruction algorithms since when discrete data is available $F(u,v)$ can only be calculated on a polar grid, along the slices, and a 2D interpolation onto a rectangular grid is required before the Fourier inversion of $F(u,v)$ can be attempted. Next, the well-known *filtered-back-projection* algorithm is examined and it is this algorithm that is implemented in the *Tomography Laboratory.*

## 5.5 The Filtered-Back-Projection algorithm

The *filtered-back-projection* algorithm requires only $1D$ interpolation and is the most commonly used reconstruction algorithm in commercial tomography machines. A good description of the algorithm can be found in *[Kak and Slaney, 1988]* but for completeness the derivation of the algorithm is repeated here since a firm grasp of the principles involved is essential to the proper understanding of the *tomography laboratory.* This time start with the two dimensional *inverse* Fourier transform:

$$f(x,y) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} F(u,v)e^{j2\pi(ux+vy)}dudv \tag{57}$$

then switch to polar co-ordinates by means of the transformation $u = \omega\cos\theta, v = \omega\sin\theta$ which has Jacobian, $\omega$.

$$f(x,y) = \int_{0}^{2\pi}\int_{0}^{\infty} \omega F(\omega\cos\theta, \omega\sin\theta)e^{j2\pi\omega(x\cos\theta+y\sin\theta)}d\omega d\theta \tag{58}$$

Now instead of integrating the *half-ray* around a *circle* the same result can be achieved by integrating a *full-ray* around a *semi-circle*. A little manipulation of integration limits then yields:

$$f(x,y) = \int_{0}^{\pi}\int_{-\infty}^{\infty} ||\omega||F(\omega\cos\theta, \omega\sin\theta)e^{j2\pi\omega(x\cos\theta+y\sin\theta)}d\omega d\theta \tag{59}$$

Now use the *projection-slice* theorem, equation 56, to get the *filtered-back-projection* algorithm:

$$f(x,y) = \int_{0}^{\pi}\int_{-\infty}^{\infty} ||\omega||P_\theta(\omega)e^{j2\pi\omega(x\cos\theta+y\sin\theta)}d\omega d\theta \tag{60}$$

To see that equation 60 is indeed a *filtered-back-projection*, implement it in two stages:

1 Filter the projections $p_\theta(t)$ using the so-called *ramp* filter $||\omega||$ to generate filtered projections $p_\theta^\omega(t)$.

$$p_\theta^\omega(t) = \int_{-\infty}^{\infty} ||\omega||P_\theta(\omega)e^{j2\pi\omega t}d\omega \tag{61}$$

2 The filtered projections are then *smeared* back accross the image to build up $f(x,y)$ according to:

$$f(x,y) = \int_{0}^{\pi} p_\theta^\omega(x\cos\theta + y\sin\theta)d\theta \tag{62}$$

The filter in stage 1 is usually done by means of standard Fourier filtering techniques while the back-projection in stage 2 is implemented using a numerical integration technique. To evaluate the integrand in stage 2 one dimensional interpolation is required. In the next section *Mathematica* will be employed to perform the filtering operation in stage 1.

## 5.6 Discretizing the Ramp Filter

Stage 1 of the filtered-back-projection algorithm is a continuous ramp filter. This filtering process must be discretized in order to construct an implementable filtering algorithm. The discretization of the ramp filter is attained via the Fourier sampling theorem. Assume that the projections are *band-limited* with *band-width* $W$, ($P_\theta(\omega) = 0$ whenever $\|\omega\| > W$), where $W$ depends on the sampling size $\tau$. If $\tau < \frac{1}{2W}$ then the sampling theorem reads:

$$p_\theta(t) = \sum_{k=-\infty}^{\infty} p_\theta(k\tau)\frac{\sin 2\pi W(t-k\tau)}{2\pi W(t-k\tau)} \tag{63}$$

Using this representation for a projection the ramp filter in equation 61 can be rewritten as:

$$p_\theta^\omega(t) = \sum_{k=-\infty}^{\infty} p_\theta(k\tau)\int_{-W}^{W}\|\omega\|\left[\int_{-\infty}^{\infty}\frac{\sin 2\pi W(t'-k\tau)}{2\pi W(t'-k\tau)}e^{-j2\pi\omega t'}dt'\right]e^{j2\pi\omega t}d\omega \tag{64}$$

The inner integral in the above equation is just the Fourier transform of a *sinc* function which is well known and documented in most tables, see for example *[Gradshteyn and Ryzhik, 1980]*. In our case the inner integral reduces to $\tau e^{-j2\pi\omega k\tau}$ and the filtered projections can now be expressed as:

$$p_\theta^\omega(t) = \tau\sum_{k=-\infty}^{\infty} p_\theta(k\tau)\int_{-W}^{W}\|\omega\|\cos(2\pi\omega(k\tau-t))d\omega \tag{65}$$

since only the *even* part survives integration over the interval $[-W, W]$. Continuing with the discretization process, it is desirable to generate sample points of the filtered projections $p_\theta^\omega(t)$. Thus we set $t = n\tau$ in the previous equation to get:

$$\begin{aligned} p_\theta^\omega(n\tau) &= \tau\sum_{k=-\infty}^{\infty} p_\theta(k\tau)\int_{-W}^{W}\|\omega\|\cos(2\pi\omega\tau(k-n))d\omega \\ &= \tau p_\theta(n\tau)W^2 + 2\tau\sum_{k=-\infty,k\neq n}^{\infty} p_\theta(k\tau)\int_0^W \omega\cos(2\pi\omega\tau(k-n))d\omega \\ &= \frac{1}{\tau}\left[\frac{1}{4}p_\theta(n\tau) - \sum_{k=-\infty,k\neq n,(k-n)odd}^{\infty}\frac{p_\theta(k\tau)}{\pi^2(k-n)^2}\right] \end{aligned} \tag{66}$$

From this result it is evident that apart from the constant $\frac{1}{\tau}$, the filtered projection data, $\{p_\theta^\omega(n)\}_{n=-\infty}^{\infty}$, can be obtained by convolving the projection data, $\{p_\theta(k)\}_{k=-\infty}^{\infty}$, with the *impulse response*, $\{h(n)\}_{n=-\infty}^{\infty}$, where $h(n)$ is given by:

$$h(n) = \begin{cases} \frac{1}{4} & zero(n), \\ 0 & even(n), \\ \frac{-1}{n^2\pi^2} & odd(n) \end{cases} \tag{67}$$

## 5.7 Implementation

To perform the convolution required by equation 66 in *Mathematica* create a finite-length filter centred at `res/2`.

```
In[11]:= h[n_] := If[n==0, N[1/4], If[OddQ[n], -N[1/(n^2 Pi^2)], 0]]

In[12]:= RampFilter = Table[h[i-(res/2)],{i,1,res}]
```

A pictorial representation of the *ramp filter* can be obtained by using `ListPlot` to display some central values of the filter. The result is shown in figure 18.

```
In[13]:= ShowFilter[v_] := ListPlot[Table[{i-(Length[v]/2),v[[i]]},
           {i, Length[v]/2-10, Length[v]/2+10}],
           PlotRange->All, PlotJoined->True];

In[14]:= ShowFilter[RampFilter]
```

The convolution of a projection vector with this ramp filter is performed with *Mathematica's* fast Fourier transform by zero-padding the projection vector and multiplying in the frequency domain.

```
In[15]:= Ramp = N[ Sqrt[2 res] Fourier[
           RotateRight[ Join[ RampFilter, Table[0,{res}] ], -(res/2-1) ]]];

In[16]:= FastRampFilter[v_,ramp_] := (res/2) Take[ Chop[ InverseFourier[
           Fourier[ Join[v, Table[0,{res}]]] * ramp ]], res];
```

The first stage of the reconstruction process is the attained by filtering all the projections in the shadow image:

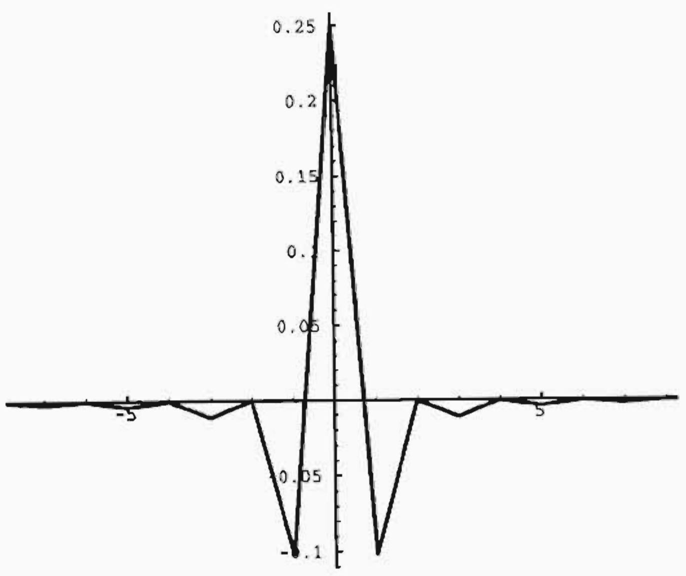

Figure 18: The ramp filter.

```
In[17]:= SheppLoganFilteredShadow = Map[FastRampFilter,SheppLoganShadow]

In[18]:= ShowImage[SheppLoganFilteredShadow]
```

The result of the first stage is shown in figure 19.

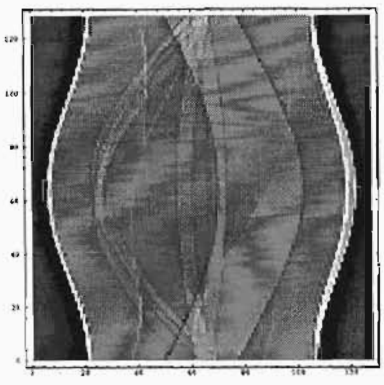

Figure 19: Shepp-Logan filtered shadow, generated via the ramp filter.

The second stage of the reconstruction is performed according to equation 62 by summing contributions from each filtered projection to build up the grey-scale value of the $(x, y)$ pixel in question. One dimensional interpolation is required to calculate an approximation to $p_\theta^\omega$ at the point $x\cos\theta + y\sin\theta$.

```
In[19]:= InterpolatedFilteredShadow =
          Map[Interpolation, SheppLoganFilteredShadow]

In[20]:= backProject[fs_,x_,y_] := If[ N[x^2+y^2] > 1, 0,
          N[Pi/res] Sum[ fs[[i]][ N[1 + (1 + x Cos[(i-1) Pi/(res-1)] +
            y Sin[(i-1) Pi/(res-1)]) ((res-1)/2)] ], {i,1,res} ] ]
```

The final reconstructed image is produced by reconstructing every pixel in the original image:

```
In[21]:= SheppLoganReconstruction = Table[
          backProject[InterpolatedFilteredShadow,N[x],N[y]],
            {y,-1,1,2/(res-1)},{x,-1,1,2/(res-1)}]

In[22]:= ShowImage[SheppLoganReconstruction]
```

The resulting reconstruction is shown in figure 20.

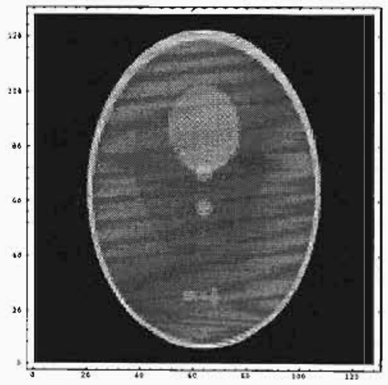

Figure 20: Shepp-Logan reconstruction, generated by back projecting filtered shadows.

## 5.8 Conclusion

This completes the construction of the tomography laboratory. In the next section *wavelet* theory is introduced and a *new* ramp filter for tomography reconstruction is developed.

# 6 Ramp Filters in a Wavelet setting

## 6.1 A new tool for Signal Processing

During the last ten years there has been an explosion of papers dealing with *wavelet* theory. The topic originated with ideas from engineering concerning *subband coding* and *quadrature mirror filters* and really got going with the discovery by Ingrid Daubechies, *[Daubechies, 1988]*, of an orthogonal basis for $L^2(R)$ whose elements, called *wavelets*, have finite support. A complete description of wavelet theory can be found in Ingrid Daubechies's book, *Ten Lectures on Wavelets*, *[Daubechies, 1992]*.

Wavelets have found application in many areas of signal processing. Whenever there is solution to a signal processing problem that involves *Fourier* methods, there should be a role for wavelets to play. In this section a new ramp filter is constructed using ideas from wavelet theory. The technique follows the method of Beylkin, Coifman and Rokhlin for representing operators in bases of compactly supported wavelets, see *[Beylkin, 1992]*. For a complete *functional analysis* treatment of how the subject grew out of the *Calderon-Zygmund* program to its present-day state, read the book *[Meyer, 1992]*. The new ramp filter will be tested for tomography purposes using the *Mathematica* tomography laboratory of the previous section.

## 6.2 The Scaling Function and its associated Wavelet

To construct a wavelet basis for $L^2(R)$ start by looking for a *scaling function* $\phi$ which is a solution to the *scaling equation*:

$$\phi(x) = \sqrt{2}\sum_{k=0}^{L-1} h_k \phi(2x-k) \tag{68}$$

In *[Daubechies, 1988]* it is shown that if the coefficients $h_k$ in equation 68 are chosen to satisfy certain conditions then the translates, $\{\phi(2^j x - k)\}_{k=-\infty}^{\infty}$ form an orthogonal set of spanning functions and a *Riesz basis* for a space $V_j$ such that $V_j \subset V_{j+1}$ and such that $\bigcup_{j=-\infty}^{\infty} V_j$ is dense in $L^2(r)$. The result only holds if $L$ is even and the conditions that the coefficients must satisfy are:

$$\sum_{k=0}^{L-1} h_k h_{k+2s} = \delta(s) \quad , \quad s = 0, 1 \ldots \frac{L}{2} - 1$$

$$\sum_{k=0}^{L-1} k^s(-1)^k h_{L-k} = 0 \quad , \quad s = 0, 1, \ldots \frac{L}{2} - 1 \tag{69}$$

The first set of conditions in equations 69 ensure that $\phi$ is orthogonal to its translates while the second set of conditions ensure that the *wavelet function* $\psi$ defined by:

$$\psi(x) = \sqrt{2}\sum_{k=0}^{L-1}(-1)^k h_{L-k}\phi(2x - k) \tag{70}$$

has translates $\{\psi(2^j x - k)\}_{k=-\infty}^{\infty}$ spanning a space $W_j$ which is the orthogonal complement of $V_j$ in $V_{j+1}$. Equations 69 form a non-linear system from which to determine the $L$ coefficients $h_k$.

## 6.3 Calculating the coefficients of the Scaling Function

For $L = 4$ and $L = 6$ the scaling function coefficients can be computed quite easily using *Mathematica*. For $L = 4$ the required implementation of equations 69 is as follows:

```
In[1]:= n = 2;

In[2]:= L = 2 n;

In[3]:= H = Table[Unique[h],{L}];

In[4]:= G = Reverse[H] Table[(-1)^i,{i,0,L-1}];

In[5]:= rotateRightAndZeroFill[vec_,k_] :=
          RotateRight[vec,k] Join[ Table[0,{k}] , Table[1,{Length[vec]-k}] ];

In[6]:= CondPhi = Table[
          H . rotateRightAndZeroFill[H,2k] == If[k==0,1,0] , {k,0,n-1}];

In[7]:= CondPsi = Join[
          { Apply[Plus,G] == 0 } ,
          Table[ G . Table[i^k,{i,0,L-1}] == 0 , {k,1,n-1} ] ];

In[8]:= h4 = N[ H /. Solve[Join[CondPhi, CondPsi], H][[1]] ]

Out[8]= {0.48296, 0.83651, 0.22414, -0.12940}
```

These coefficients compare favourably with those published in *[Daubechies, 1992]*. Unfortunately this method cannot be used to compute scaling coefficients when $L > 8$. *Mathematica's* non-linear solver cannot cope with the resulting system of equations in a reasonable time span. Daubechies resorted to numerical techniques to obtain scaling coefficients for large $L$. We will require coefficients for $L = 12$ to derive a wavelet based ramp filter and for this purpose the coefficients for $L = 12$ have been lifted from *[Daubechies, 1992]*.

```
In[9]:= h12 = { 0.1115407433501095,
                0.4946238903984533,
                0.7511339080210959,
                0.3152503517091982,
               -0.2262646939654400,
               -0.1297668675672625,
                0.0975016055873225,
                0.0275228655303053,
               -0.0315820393174862,
                0.0005538422011614,
                0.0047772575109455,
               -0.0010773010853085 }
```

## 6.4 Viewing a Scaling Function

What does a scaling function look like? Does a scaling function with compact support exist? The first step in the computation of a scaling function is to assume that its support lies within the interval $[0, L]$ and that $\phi(0) = \phi(L) = 0$. and then determine its value at the rest of the integers in the $[0, L]$ interval by examining the scaling equation 68 at these integers. The result of all this is that to obtain $\phi$ at the integers one must solve an eigenvalue system of the form:

$$\begin{bmatrix} h_1 & h_0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ h_3 & h_2 & h_1 & h_0 & 0 & \cdots & 0 & 0 \\ \vdots & & & & & & & \vdots \\ 0 & 0 & \cdots & 0 & h_{L-1} & h_{L-2} & h_{L-3} & h_{L-4} \\ 0 & 0 & \cdots & 0 & 0 & 0 & h_{L-1} & h_{L-2} \end{bmatrix} \begin{bmatrix} \phi(1) \\ \phi(2) \\ \vdots \\ \phi(L-3) \\ \phi(L-2) \end{bmatrix} = \begin{bmatrix} \phi(1) \\ \phi(2) \\ \vdots \\ \phi(L-3) \\ \phi(L-2) \end{bmatrix} \qquad (71)$$

*Mathematica* has a built in function `Eigensystem[..]` that delivers the required eigenvector quite simply:

```
In[10]:= mat[h_] := Module[
          {row=Join[Reverse[h Sqrt[2]],Table[0,{Length[h]-2}]]},
```

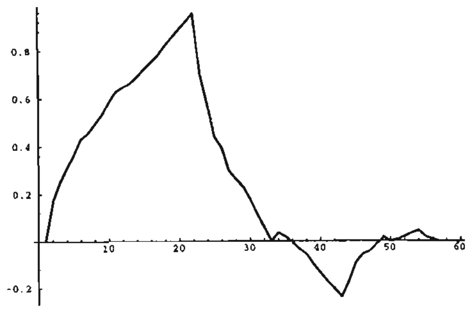
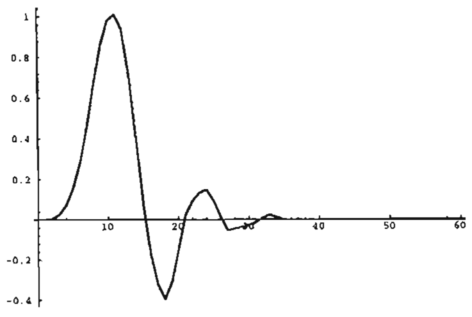

Figure 21: Scaling functions for h4 and h12.

```
          Table[Take[RotateRight[row,2i],-(Length[h]-2)],
          {i,1,Length[h]-2}]];

In[11]:= Phi[x_ /; IntegerQ[x], h_] = If[ x <= 0 || x >= Length[h]-1, 0,
          Module[ {intval=Eigensystem[mat[h]][[2,1]]},
                    intval[[x]] Sign[intval[[1]]] ]];
```

The scaling equation 68 is again used to compute $\phi$ at dyadic points. The recursion terminates when a value of $\phi$ at at an integer is required. Note that due to the recursive nature of the scaling equation intermediate results are saved using *Mathematica's* dynamic programming technique `f[x_] := f[x] = .....`

```
In[12]:= Phi[x_, h_] := Phi[x,h] = If[ x <= 0 || x>= Length[h]-1, 0 ,
          N[Sqrt[2] h . Table[Phi[2x-k,h],{k,0,Length[h]-1}]]];

In[13]:= ShowPhi[h_] := ListPlot[
          Table[Phi[x,h], {x,0,Length[h]-1,(Length[h]-1)/2^6}],
          PlotJoined->True, PlotRange->All];
```

The scaling functions corresponding to the scaling coefficients **h4** and **h12** can now be viewed using the command `ShowPhi[..]`. The results are shown in figure 21. It can be observed that the h12 scaling function is smoother than the **h4** scaling function. This is a general trend. Scaling functions gain $\frac{1}{2}$ a degree of differentiability for every extra 2 scaling coefficients, see *[Strang, 1989]*.

## 6.5 The Ramp Filter as a Convolution

The goal in this section is to construct a tomographic *ramp filter* for a scaling function. The idea is that since any projection data can be approximated by a linear combination of shifted scaling functions any good *ramp filter* for a scaling function will also be a good *ramp filter* for projection data.

To construct a ramp filter for a scaling function it is expedient to use the fact that a ramp filter is equivalent to the Hilbert transform of the first derivative. This representation for the ramp filter can be found in *[Kak and Slaney, 1988]* or in *[Herman et al, 1987]*. Using the notation of the previous section the argument is as follows:

If $p_\theta^\omega(t)$ is a ramp filtered version of $p_\theta(t)$ then

$$\begin{aligned} p_\theta^\omega(t) &= \int_{-\infty}^{\infty} \|\omega\| P_\theta(\omega) e^{j2\pi\omega t} d\omega \\ &= \int_{-\infty}^{\infty} [j2\pi\omega P_\theta(\omega)][-\frac{j\,sign(\omega)}{2\pi}] e^{j2\pi\omega t} d\omega \end{aligned} \quad (72)$$

It is easy to see that the first term in the integrand of the above expression is the Fourier transform of $p_\theta'(t)$. The second term in the integrand is the Fourier Transform of $\frac{1}{2\pi^2 t}$, see *[Kaplan, 1962, pp.282]* for details. Thus using the convolution theorem the above result may be written as:

$$\begin{aligned} p_\theta^\omega(t) &= \frac{1}{2\pi^2 t} * p_\theta'(t) \\ &= \frac{1}{2\pi^2} \int_{-\infty}^{\infty} \frac{p_\theta'(u)}{t-u} du \end{aligned} \quad (73)$$

which confirms that apart from a multiplicative constant, the ramp filter is the *Hilbert transform* of the *derivative*. In *[Beylkin, 1992]* filters for performing differentiation and Hilbert transforms on scaling functions are constructed. Here the construction is repeated using *Mathematica* and the filters are combined according to equation 73 to form a new ramp filter for tomography.

## 6.6 A Filter for Differentiation

The goal here is to approximate the operator $\frac{d}{dx}$, in the space $V_0$ spanned by the scaling function translates $\phi(x-k)$. Given any function $f(x)$, its approximate derivative in $V_0$ is given by $P_0[\frac{d}{dx}[P_0[f(x)]]]$ where $P_0$ is the projection onto $V_0$. Now assuming

$$P_0[f(x)] = \sum_{k \epsilon Z} f_k \phi(x-k) \quad (74)$$

we have

$$\frac{d}{dx}[P_0[f(x)]] = \sum_{k\epsilon Z} f_k\phi'(x-k) \tag{75}$$

and thus

$$P_0[\frac{d}{dx}[P_0[f(x)]]] = \sum_{l\epsilon Z} s_l\phi(x-l) \tag{76}$$

where

$$s_l = \int_{-\infty}^{\infty} \sum_{k\epsilon Z} f_k\phi'(x-k)\phi(x-l)dx. \tag{77}$$

Substituting the last equation into the last but one results in:

$$P_0[\frac{d}{dx}[P_0[f(x)]]] = \sum_{l\epsilon Z}\left[\sum_{k\epsilon Z} f_k r_{l-k}\right]\phi(x-l) \tag{78}$$

where

$$r_l = \int_{-\infty}^{\infty} \phi(x-l)\phi'(x)dx \tag{79}$$

This shows that the coefficients of the derivative of a function are obtained by convolving the coefficients of the function with the sequence $\{r_l\}_{l\epsilon Z}$ given in equation 79. We will call this sequence a *differential filter*. Since a scaling function $\phi$ is fully determined by its scaling coefficients $h_k$ it is possible to obtain expressions for $r_l$ in terms of $h_k$. These expressions are:

$$\begin{aligned} r_l &= 2\left[r_{2l} + \frac{1}{2}\sum_{k=1}^{L/2} a_{2k-1}(r_{2l-2k+1} + r_{2l+2k-1})\right] \\ \sum_l l r_l &= -1 \\ r_l &= -r_{-l} \end{aligned} \tag{80}$$

where $a_n$ are autocorrelation coefficients given by:

$$a_n = 2 \sum_{i=0}^{L-1-n} h_i h_{i+n}, \qquad n = 1, 2, \ldots, L-1. \tag{81}$$

Beylkin was the first to publish this result. The derivation of the first expression in 80 is not difficult, just start with equation 79 and make use of the scaling equation. The second and third expressions in 80 require quite lengthy Fourier arguments. *Beylkin* also shows that $r_l = 0$ whenever $\|l\| > L - 2$. The reader is referred to *[Beylkin, 1992]* for details. Equation 81 is easy to implement in *Mathematica.* Equations 80 require more finesse. Using a suggestion of *Beylkin,* start with $r_1 = -0.5$ and $r_{-1} = 0.5$ so that the second and third parts of equations 80 are satisfied and then iterate using the first part of equations 80 to generate an improved differential filter.

```
In[14]:= AutoCoef[h_] := Table[ 2 Sum[ h[[i+1]] h[[i+1+n]],
         {i,0,Length[h]-1-n}], {n,1,Length[h]-1} ]

In[15]:= DiffCoef[h_,res_] := Module[ {a=AutoCoef[h],
         r=Table[0,{Length[h]-2}],nr,Zeros},
         nr=r; nr[[1]]=-0.5;
         rf[i_]:= If[i<0,-rf[-i],If[ (i==0) || (i>Length[h]-2), 0, r[[i]]]];
         While[Chop[nr]!=Chop[r],
           ( r=nr;
             nr=Table[2(rf[2i] +
               0.5 Sum[a[[2k-1]](rf[2i-2k+1]+rf[2i+2k-1]),
               {k,1,Length[h]/2}]), {i,1,Length[r]}])];
         Zeros=Table[0,{res/2-Length[nr]}];
         Join[Drop[Zeros,1],-Reverse[nr],{0},nr,Zeros]]
```

This code is used to generate a *differential filter* of length 128, associated with h12. The coefficient $r_0$ is situated at position 64 in the filter. The numeric values for $r_0..r_{12}$ are tabulated and compare favourably with those published in *[Beylkin, 1992, Table 1].*

```
In[16]:= Take[DiffCoef[h12,128], {64,76}]

Out[16]= { 0, -0.85013666155593, 0.2585529441414689,
           -0.07244058999766052, 0.01454551104199389,
           -0.001588561543475704, 4.296891570985078*10^-6,
            0.00001202657519572415, 4.206912045116679*10^-7,
           -2.899666805706292*10^-9, 6.9686511520195*10^-13, 0, 0}
```

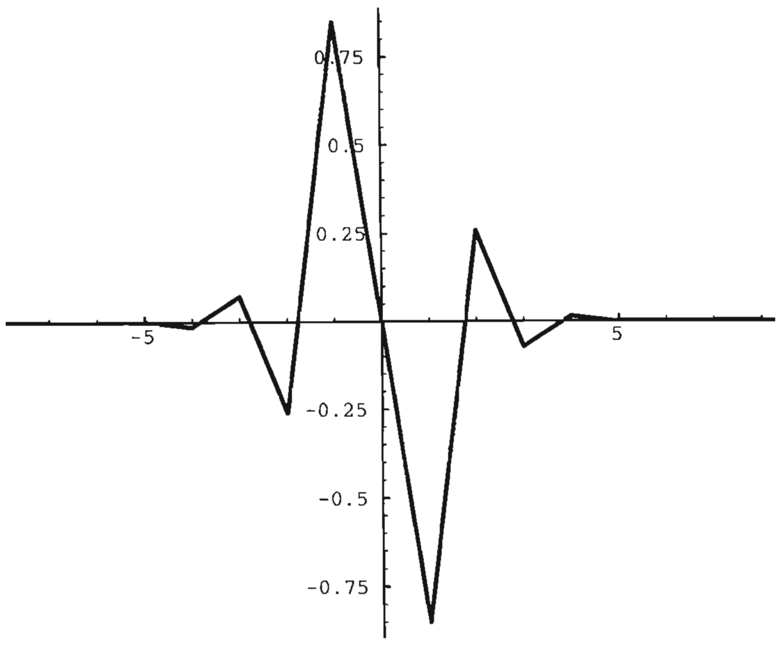

Figure 22: A differential filter from the scaling function h12.

A pictorial representation of the *differential filter* can be obtained by using `ListPlot` to display some values of the filter around $r_0$. The result is shown in figure 22.

```
In[17]:= ShowFilter[v_] := ListPlot[Table[{i-(Length[v]/2),v[[i]]},
           {i, Length[v]/2-10, Length[v]/2+10}],
           PlotRange->All, PlotJoined->True];

In[18]:= ShowFilter[DiffCoef[h12,128]]
```

To generate the new ramp filter we must first use the same techniques to produce a Hilbert transform filter.

## 6.7 A Filter for a Hilbert Transform

To obtain a *Hilbert transform filter* the arguments from the previous section must be repeated with the differential operator $\frac{d}{dx}$ replaced by the Hilbert transform operator $H$. The action of the operator $H$ on any function $f$ is defined by:

$$(H[f])(x) = \frac{1}{\pi}\int_{-\infty}^{\infty} \frac{f(s)}{s-x} ds. \tag{82}$$

Repetition of the previous section's arguments results in the following defining relations for the coefficients of a *Hilbert transform filter*:

$$\begin{aligned} r_l &= r_{2l} + \frac{1}{2}\sum_{k=1}^{L/2} a_{2k-1}(r_{2l-2k+1} + r_{2l+2k-1}) \\ r_l &= -\frac{1}{\pi l} + O(\frac{1}{l^L}) \\ r_l &= -r_{-l} \end{aligned} \tag{83}$$

Note that the factor 2 has disappeared from the first expression and that the second expression now becomes an asymptotic approximation. The $a_k$ are the same autocorrelation coefficients used in the *differential filter*. Again, a derivation for these expressions can be found in *[Beylkin, 1992]*. To compute the *Hilbert transform filter* coefficients start by using the second expression in 83 as an approximation and then iterate using the first and third expressions to compute improved filter coefficients.

```
In[19]:= HilbertCoef[h_,res_] := Module[ {a=AutoCoef[h],
          r=Table[N[-1/(Pi i)],{i,1,Length[h]}], nr, Asymp},
          nr=r; nr[[1]]=-0.5;
          rf[i_]:=If[i<0,-rf[-i],
                   If[ (i==0) , 0,
                     If[i>Length[r], N[-1/(Pi i)], r[[i]] ]]];
          While[Chop[nr]!=Chop[r],
            ( r=nr;
              nr=Table[ (rf[2i] +
                   0.5 Sum[a[[2k-1]](rf[2i-2k+1]+rf[2i+2k-1]),
                     {k,1,Length[h]/2}]), {i,1,Length[r]}])];
          Asymp=Table[N[-1/(i Pi)],{i,Length[nr]+1,res/2}];
          Join[Drop[-Reverse[Asymp],1], -Reverse[nr], {0}, nr, Asymp]]
```

Note that only the first few Hilbert transform filter coeficients are allowed to change value during the iteration. For large $||l||$ the second expression in 83 is used to estimate coeficients. The code is used to generate a *Hilbert transform filter* of length 128, associated with h12. The coefficient $r_0$ is situated at position 64 in the filter. The numeric values for $r_0..r_{12}$ are tabulated and compare favourably with those published in *[Beylkin, 1992, Table 4]*. The central part of the filter is shown in figure 23.

```
In[20]:= Take[HilbertCoef[h12,128], {64,76}]

Out[20]= {0, -0.5883036982236796, -0.07757641365985442,
           -0.1287436950895584, -0.07506362754219426,
```

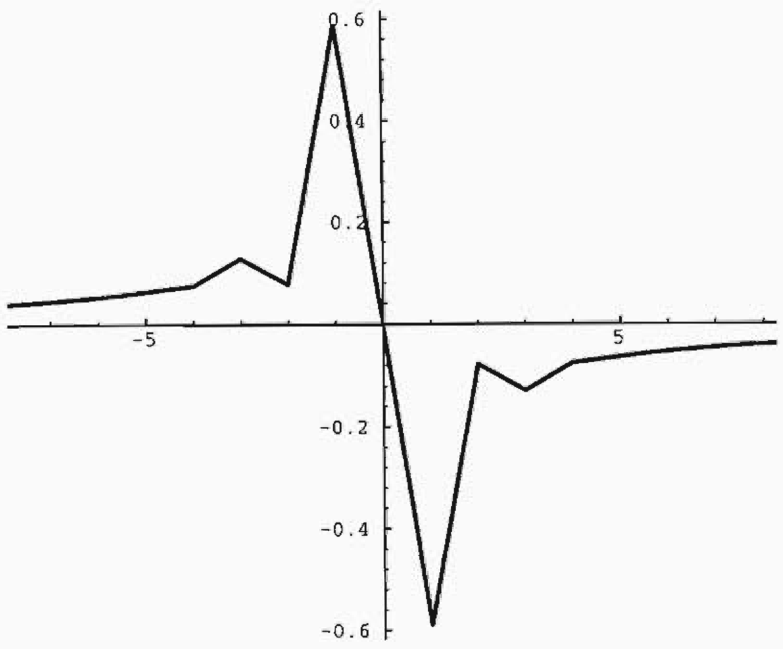

Figure 23: A Hilbert transform filter from the scaling function h12.

```
-0.06416801797769806, -0.0530413661373353,
-0.04547064970868715, -0.03978864124836587,
-0.03536776075990964, -0.03183098767753492,
-0.02893726214706868, -0.02652582378143546}
```

```
In[21]:= ShowFilter[HilbertCoef[h12,128]]
```

## 6.8 Coefficients for a Ramp Filter

Equation 73 shows that a *ramp filter* is equivalent to a differential filter followed by a *Hilbert transform filter*. Thus, since convolution is associative, ramp filter coefficients can be constructed by convolving Hilbert transform filter coefficients with differential filter coefficients. A combination of dot products, circular shifts and zero padding does the trick:

```
In[22]:= Convolve[d_,h_,res_] := Module[ {df,dh,Zeros},
           Zeros = Table[0,{res/2}];
           df = Join[Zeros,d,Zeros];
           hf = Join[Zeros,h,Zeros];
           Drop[ Drop[ RotateRight[
             Table[ RotateRight[df,n] . hf , {n,0,2 res-1}], res-1],
               res/2], -res/2] ];
```

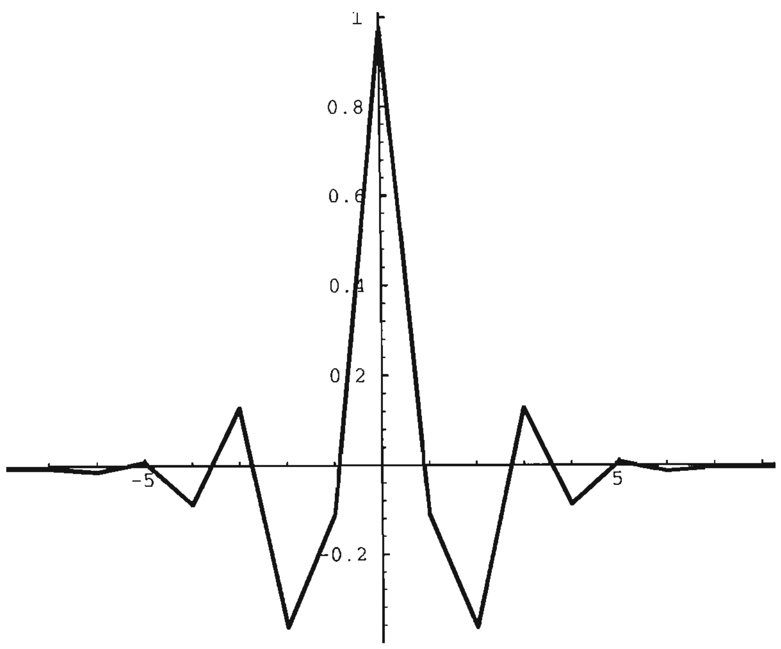

Figure 24: The new Ramp Filter associated with h12, obtained by convolving the Hilbert transform filter with the differential filter.

```
In[23]:= NewRampFilter = Convolve[
           DiffCoef[h12,128], HilbertCoef[h12,128], 128]
```

The new *ramp filter* turns out to be an *even* sequence. This is to be expected as both the differential and the Hilbert filters are *odd.* In keeping with the style so far, the values $r_0..r_{12}$ for the new ramp filter are tabulated here and a pictorial representation of the *ramp filter* is given in figure 24. Note that $r_0$ is no longer zero.

```
In[24]:= Take[NewRampFilter, {64,76}]

Out[24]= {0.976833011377977, -0.1109909447828639, -0.3644561708911,
          0.1281887996668191, -0.0874681026701724, 0.00816052360937506,
         -0.01376979736243151, -0.005579502282393995, -0.005109531721197386,
         -0.003915663821722484, -0.003183687584440462, -0.002630709860396351,
         -0.002210478434002659}

In[25]:= ShowFilter[NewRampFilter]
```

It now remains to test the usefulness of the new *ramp filter* in tomography.

## 6.9 Testing the Ramp Filter

Finally the new *ramp filter* must be evaluated as a reconstruction tool. The tomography laboratory of the previous section is harnessed and `NewRampFilter` is used in the place of `RampFilter`. The result is a new filtered shadow and a new reconstructed Shepp-Logan test image. These images are shown in figures 25 and 26 respectively.

Animation can be employed to compare the two different reconstructions. The original Shepp-Logan test image, the standard reconstruction and the new reconstruction have been combined into a 3-frame animation sequence and stored on the stiffy disk accompanying this thesis. Instructions for viewing the animation can be found in the appendix.

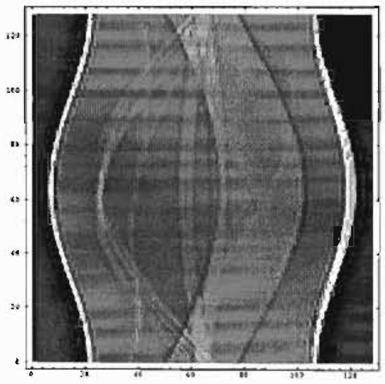

Figure 25: The new Shepp-Logan filtered shadow.

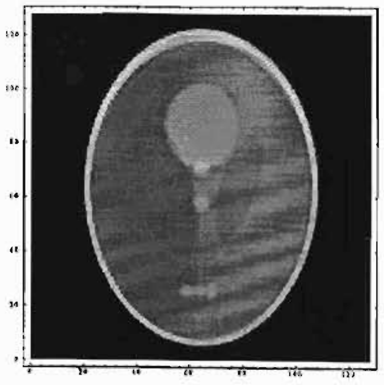

Figure 26: The new Shepp-Logan reconstruction.

## 6.10 Conclusion

*Mathematica* has been employed to investigate the properties of scaling functions. These functions are the so-called *mother wavelets* in the new and expanding field of *wavelet analysis.* The properties of the scaling function have been used to construct a new ramp filter for tomographic reconstruction. The new filter performs well since it has built-in smoothing characteristics.

# 7 Conclusion

This thesis has demonstrated that *Mathematica* provides an excellent modelling environment. Three problems of a mechanical nature have been tackled and two useful laboratories have been created.

Although the problems tackled are not new problems, the *Mathematica* solutions are new as the reference list indicates. The animations of a mathematical golf swing and the tennis racquet flip are new, so to is the *Mathematica* implementation of the phase-plane package. These sections of the thesis have already been published. The author believes that the tomography laboratory will also attract interest from *The Mathematica Journal*. The author has already published in the tomography field, see *[Murrell, 1989]* and *[Murrell and Carson, 1990]*, and is of the opinion that development time for the ideas involved in these publications would have been much shorter if *Mathematica* had been available then.

The potential for modelling with *Mathematica* is unlimited and the author has various future projects in mind. The most inviting of these is to start investigating *Mathematica's* sound capabilities with the goal of building a general purpose *sound analysis laboratory*. The author intends to use such a laboratory to analyse *bird-calls*.

Astronomy provides a rich set of mechanical problems suitable for analysis and animation via *Mathematica*. A general purpose astronomical package has already been developed for *Mathematica*. The author intends to make use of this package to test the hypothesis, *[Hoyle, 1977]*, that Stonehenge was designed with the intention of predicting both lunar and solar eclipses.

The versatility of *Mathematica* has enabled the author to persuade a number of post-graduate students to tackle modelling problems. *Mathematica* is thus an essential tool in both research and teaching environments.

# 8 Appendix

On the accompaning disk there are two ZIP archives and two EXE utilities. The file ANI.ZIP contains all the animations described in this thesis. After *unzipping* the following five animation files should be present:

- BEADS, an animation file for beads sliding down wires.
- SWING, an animation file for the perfect mathematical golf swing.
- RACQUET, an animation file for a racquet rotating under no external forces.
- PHASE, an animation file for the elephant-tree bifurcation event.
- TOMO, the three frame animation file for comparing tomographic reconstructions.

The file SRC.ZIP contains all the *Mathematica* source code described in this thesis. After *unzipping* this file a number of *Mathematica* source files will be present. There is one source file for each section of the thesis.

The two utilities stored on the disk are:

- PKUNZIP.EXE, a program to decompress the .ZIP files.
- ANIMATE.EXE, a program to view the animations.

In order to view an animation, acquire a 386 or 486 machine with at least 8 meg of memory and 10 meg of free disk space and then carry out the following instructions under DOS.

- Create an empty subdirectory on the hard drive and make it the current directory.
- Copy all the files from the stiffy disk to the hard drive subdirectory.
- Decompress the ANI.ZIP using the PKUNZIP.EXE utility.
- View the animation using the ANIMATE.EXE utility.

An example session could be as follows:

```
MD MURRELL
CD MURRELL
COPY B:*.*
PKUNZIP ANI.ZIP
ANIMATE BEADS
ANIMATE SWING
ANIMATE RACQUET
ANIMATE PHASE
ANIMATE TOMO
```

Once an animation is running the user can control the animation by using the keyboard as follows:

- 1..9 sets the animation speed, (1 = slow, 9 = fast).
- P pauses the animation.
- R reverses the animation direction.
- C reverses the direction at the end of a cycle.
- DownArrow steps forwards through the animation.
- UpArrow steps backwards through the animation.
- Q stops the animation.

If any files on the enclosed stiffy disk prove to be unreadable please contact the author at e-mail *murrellh@images.cs.und.ac.za* and a replacement disk will be posted.

## 9 References

Ashbough M.C., Chicone C. and Cushman R. 1991. *The twisting tennis racket* Journal of Dynamics and Differential Equations, 3, pp.67-86.

Beylkin G. 1992. *On the representation of operators in bases of compactly supported wavelets*, SIAM J. Numer. Anal., vol. 6, no. 6, pp. 1716-1740.

Blachman N. 1992. *Mathematica: A Practical Approach*, Prentice Hall.

Caughley G. 1976. *The elephant problem, an alternative hypothesis*, East African Wildlife Journal, vol. 14, pp. 265-283.

Crandall R.E. 1991. *Mathematica for the Sciences*, Addison Wesley.

Daubechies I. 1988. *Orthonormal bases of compactly supported wavelets*, Comm. Pure and Appl. Math., vol. 41, pp. 909-996.

Daubechies I. 1992. *Ten Lectures on Wavelets*, SIAM publications, Philadelphia.

Gaylord R.J., Kamin S.N. and Wellin P.R. 1993. *Introduction to Programming with Mathematica*, Springer Verlag and Telos

Goldstein H. 1950. *Classical Mechanics*, Addison Wesley.

Gradshteyn I.S. and Ryzhik I.M. 1980 *Table of Integrals, Series and Products*, corrected and enlarged edition, translated by Jeffrey A., Academic Press.

Gray A. and Gray J.G. 1911. *Treatise on Dynamics*, MacMillan and Co.

Gray T.W. and Glynn J. 1991. *Exploring Mathematics with Mathematica*, Addison Wesley.

Herman G.T., Tuy H.K., Langenberg K.J. and Sabatier P.C. 1987, *Basic Methods of Tomography and Inverse Problems* edited by Pike E.R., Malvern Physics Series.

Hoyle F. 1977. *On Stonehenge*, Heinemann Educational Books, London.

Hughes D. and Murrell H. 1987. *Non-linear runoff routing, a comparison of solution methods,* Journal of Hydrology, 85, pp.339-347.

Kak A.C. and Slaney M. 1988. *Principles of Computerized Tomographic Imaging*, IEEE Press.

Kaplan W.K. 1958. *Ordinary Differential Equations*, Addison Wesley.

Kaplan W.K. 1962. *Operational Methods for Linear Systems*, Addison Wesley.

Kocak H. 1986. *Differential and Difference Equations through Computer Experiments.* Springer, New York.

Marsden J.E. and McCracken M. 1976. *The Hopf Bifurcation and its Applications*, Springer Verlag.

Meyer Y. 1992. *Wavelets and operators*, Cambridge studies in advanced mathematics, Cambridge University Press.

Murrell H. 1982. *Conductivity Profiles for a Horizontally Uniform Earth*, MSc thesis, Rhodes University.

Murrell H. 1989. *A case for computerized tomography in the undergraduate syllabus,* proceedings of the 15th South African symposium on numerical mathematics, pp.145-158.

Murrell H. 1992. *Animation of rotating rigid bodies,* The Mathematica Journal, Vol. 2, No. 1, pp.61-65.

Murrell H. 1993. *A mathematical golf swing,* The Mathematica Journal, Vol. 3, No. 4, pp.62-65.

Murrell H. 1994. *Planar Phase Plots and Bifurcation Animations*, The Mathematica Journal, Vol 4, No. 3, pp.80-85

Murrell H. and Carson D. 1990. *Image reconstruction via the Hartley transform,* South African Computer Journal, Vol. 1, No. 1, pp.36-42.

Murrell H. and Ungar A. 1982. *From Cagniard's method for solving seismic pulse problems to the method of the differential transform* , Computers and mathematics with applications, Vol. 8, No. 2, pp.103-118.

Nevin J. and Jackson P.J. 1977. *An interesting property of the tennis racquet and dynamically similar rigid bodies*, Inst. of Maths. and Appls.

Prescott J. 1941. *Mechanics of Particles and Rigid Bodies*, Longmans Green and Co.

Sacks E.P. 1991. *Automatic analysis of one-parameter planar ordinary differential equations by intelligent numeric simulation.* Artificial Intelligence, Vol. 48, pp 27-56.

Shaw W.T. and Tigg J. 1994. *Applied Mathematica*, Addison Wesley.

Shepp L.A. and Logan B.F. 1974. *The Fourier reconstruction of a head section*, IEEE Transactions on Nuclear Science, vol. NS-21, pp. 21-43.

Swart J.H. 1994. *Limit cycle behaviour in an elephant-tree ecology*, submitted to, S.A. Journal of Science.

Swart J.H. and Duffy K.J. 1987. *The stability of a predator-prey model applied to the destruction of trees by elephants.* S.A. Journ. Sc., vol. 18, pp.156-158.

Swart J.H. and Murrell H. 1991. *A model of age-dependent population dynamics providing simple criteria for growth or extinction,* Mathematical Biosciences, 103, pp.1-15.

Strang G. 1989. *Wavelets and Dilation Equations: A brief introduction*, SIAM Review, vol. 31, no. 4, pp. 614-627.

Synge J.L. and Griffith B.A. 1959. *Principles of Mechanics*, third edition, New York, McGraw Hill.

Ungar A. and Murrell H. 1985. *The differential transform and its application to an electrostatics image problem,* Computers and mathematics with applications, Vol. 11, No. 6, pp.565-572.

Vardi I. 1991. *Computational Recreations in Mathematica*, Addison Wesley.

Varian H.R. 1993. *Economic and Financial Modeling with Mathematica*, Springer Verlag and Telos.

Vvedensky D. 1992. *Partial Differential Equations with Mathematica*, Addison Wesley.

Wells D.A. 1967. *Lagrangian Dynamics*, Schaum's Outline Series.

Williams D. 1967. *The Dynamics of the Golf Swing.* Quartarly Journal of Mechanics and Applied Mathematics, Vol.XX, pp.247-264.

Williams D. 1969. *The Science of the Golf Swing.* Pelham Books.

Wolfram S. 1991. *Mathematica, A System for doing Mathematics by Computer*, Addison Wesley.