A Study of the Planar Circular Restricted Three Body Problem and the Vanishing Twist

Joachim Worthington

An essay submitted in partial fulfillment of
the requirements for the degree of
B.Sc. (Honours)

Applied Mathematics
University of Sydney

THE UNIVERSITY OF
SYDNEY

October 2012

CONTENTS

CHAPTER 1

Introduction

Since the 17th century, the N-body problem has held the attention of generations of astronomers and mathematicians. The problem is simple: given a collection of N celestial bodies (be they planets, asteroids, stars, black holes) interacting with each other through gravitational forces, what will their trajectories be? For $N = 2$, the problem has been solved for centuries; for $N \geq 3$, the problem still has no solution in any meaningful sense. As the theory and vocabulary of dynamics have evolved, so too has the analysis of the problem, and indeed the study of the problem has often directly led to the development of new concepts and ideas in dynamics.

In this thesis, we consider the *planar circular restricted three body problem*, a specific case of the N-body problem for $N = 3$. The primary goal is to develop a fast, user-friendly program which can quickly and reliably calculate trajectories from user input. The program will also calculate *Poincaré maps*, which will be used to analyse the system for various parameter values. We then hope to verify the existence of a particular bifurcation called the *twistless bifurcation* for orbits near the Lagrangian points. The twistless bifurcation was found for a general system by Dullin, Meiss and Sterling [**12**], and it is expected that the planar circular restricted three body problem will exhibit the same behaviour.

We begin with a discussion of the history of the problem in Chapter 2, using Barrow-Green [**5**], Valtonen & Karttunen [**10**] and James [**23**] as our primary sources. This background serves a dual purpose, neatly introducing many of the theoretical concepts used to analyse the problem. We discuss several "particular solutions" which illustrate useful ideas and dynamics, and give a summary of the theory of Lagrangian and Hamiltonian mechanics.

In Chapter 3, the solution to the two body problem is presented, and the dynamics for the three body problem are derived. Following Koon, Lo, Marsden & Ross [**22**], we take a Hamiltonian approach to the problem. Other physical considerations such as the *Hill region* and *Lagrangian points* are introduced. Also defined are the *Poincaré map* and *extended phase space*.

Chapter 4 deals with the biggest obstacle in any attempt to integrate trajectories of the N-body problem, regularising collision orbits. Although an elegant split-step integrator can be found for the problem, regularising transforms are still required. The discussion of these transformations follows from Szebehely [**16**], but are here derived in the context of Hamiltonian mechanics. The Levi-Civita, Birkhoff and Thiele-Burrau transformations are discussed. An elegant numerical method for calculating Poincaré maps designed by Henón [**20**] is also presented.

Chapter 5 gives a brief outline of the development and use of the completed program, and a comparison of the observed bifurcations to those predicted by the theory. Also included is a consideration of the Earth-Moon system, predicting the behaviour of a possible Earth-Moon trojan.

CHAPTER 2

Background

The study and theory of the three body problem has developed over the last four centuries concurrent to (and often catalysing) the general theory of dynamical systems. It is therefore natural to explore the history of the problem, not only for context and insight but to introduce key approaches and techniques to be utilised in the project.

2.1. History of the Three Body Problem

The three body problem as we consider it arose very naturally from the work of Newton. In the early 17th century Kepler proposed his laws of planetary motion, describing the orbits of the planets around the sun as ellipses. Newton formalised these ideas in 1687's *Philosophiæ Naturalis Principia Mathematica* [**1**], one of the most important works in the history of science. In particular, the formula for the gravitational force between any two point masses is given as

$$F = G\frac{m_1 m_2}{r^2} \tag{2.1}$$

for two masses m_1 and m_2 separated by a distance of r, and G the *universal gravitational constant*. Some controversy remains whether this law should be attributed to Newton or Hooke, but it is acknowledged that both men made very significant contributions to the development of celestial mechanics.

Having justified the laws proposed by Kepler, Newton turned his attention to systems more complex than a Sun-Planet system. One of his main considerations was the Sun-Earth-Moon system. However, Newton's work in this regard was plagued by difficulties, and he remarked "*...[his] head never ached but with his studies on the moon*". It would not be until after Newton's lifetime that any major progress was made on the three body problem.

In 1747, Alexis Clairaut announced he had successfully constructed a series approximation for the motion of the three masses. After some modification, his approximations accounted for the perigee of the moon (the point at which the moon is nearest to earth), which had been an aim of Newton's. In 1752 Clairaut won the St. Petersburg Academy prize for his work on the problem, and in 1759 the value of his approximations was amply demonstrated when Halley's comet passed Earth within a month of what his equations had predicted, the margin of error he himself had prescribed.

Meanwhile, Leonhard Euler had also turned his attentions to the three body problem. Euler proposed considering the *restricted three body problem*, a simplification of the general problem where one of the bodies is taken to have negligible mass (see 3.2.1). When considered with a circular orbits for the two masses, this is also known as the *Euler three body problem*. Euler also used variation of parameters to study perturbations of the planetary motion.

At the same time as Euler, Joseph Lagrange made significant progress on the general three body problem. Lagrange's major contributions to theory included reducing the problem from a system of differential equations of order 18 to a system of order 7, and describing two types of particular solutions to the general problem (see 2.2.2). Also of major importance, not just to the three body problem but to the general theory of dynamic systems, was his development of *Lagrangian Mechanics* (see 2.3).

Unaware of Lagrange's work, Carl Jacobi reduced the general problem to a sixth order system and the restricted problem to a fourth-order system. A constant of motion was found, known as *Jacobi's integral*, and is the only known conserved quantity of the restricted problem. In 1878, George Hill demonstrated a very useful application of Jacobi's integral, describing the regions of possible motion for the body of negligible mass (see 3.3.2).

Another contributor to the theory of the problem was Charles-Eugéne Delaunay. By taking repeated canonical transformations of the problem, requiring a truly staggering number of calculations, Delaunay completely eliminated the secular terms of the problem. Taking over two decades to complete, Delaunay's methods were published in 1846 but his final results could not be published until 1860 and 1867, when they were published in two large volumes of over nine hundred pages each. A key useful result of the work was the introduction of *Delaunay variables*, a set of canonical action angles which give the equations of motion in Hamiltonian form. Although Delaunay's method was impractical at the time (the expressions involved were extremely complex and converged very slowly), the theory has been highly influential, not only in lunar theory but in fields such as quantum theory as well.

The end of the "classical" period of work on the three body problem was marked by the extremely influential work of Henri Poincaré. In the late nineteenth century, King Oscar II of Sweden established a prize for solving the N-body problem (a more general form of the problem with N rather than 3 masses) on the advice of Gösta Mittag-Leffler, Karl Weierstrass, and Charles Hermite. The statement of the problem was as follows:

Henri Poincaré

> *Given a system of arbitrarily many mass points that attract each other according to Newton's law, under the assumption that no two points ever collide, try to find a representation of the coordinates of each point as a series in a variable that is some known function of time and for all of whose values the series converges uniformly.*

Although Poincaré did not solve the problem as stated, his paper was so progressive and important that he won the prize regardless. However, his initial submission contained a fatal mathematical error, which led to a fallacious stability result for the $N = 3$. After having all copies of the original paper destroyed (at a high personal cost), he published an updated paper which rectified the mistake. In doing so, he raised several influential ideas which would lead to the development of the theory of mathematical chaos, and changed core ideas of the mathematical study of dynamics. He also introduced the concept of a *first return map*, now also known as a *Poincaré map* (see 3.4). For more on Poincaré's work on the three body problem, see Barrow-Green [**5**].

In fact, the problem as it had originally been stated would not be solved until 1912 by Sundman for $N = 3$. The general case would remain unsolved until 1991, when Qiudong Wang published *The Global Solution of N-body Problem* [**8**]. However, in both cases, the series constructed converged so slowly that they were essentially useless in practice. Though his work met the requirements of King Oscar's problem, Wang himself would characterise his result as "*a tricky, simple and useless answer*" while praising the publications that Poincaré did complete (see [**7**]).

After much of the foundations were laid by Poincaré, the twentieth century saw progress on many different fronts. Simplified versions of the restricted three body problem were analysed. For example, the *Copenhagen problem* assumes the two masses to be equal, and were considered by Strömgren and colleagues from 1913 in detail.

Another approach to the problem in last century was the development of statistical analysis of the orbits. Poincaré had demonstrated the system is chaotic, which opens the possibility of systematically analysing orbits based on statistical distributions. In most occurences, the general three body problem results in one mass leaving the two other bodies permanently, and the other two forming a binary system. This is known as an *escape orbit*. Valtonen and Karttunen (see [**10**]) applied statistical methods to analyse the *scattering* of these escape orbits. These results were directly compared to experimental values, which could be carried out thanks to the proliferation of powerful computers at the end of the twentieth century.

An altogether different approach to classify the possible motions was Hénon's analysis of the restricted three body problem. By perturbing the mass of one of the bodies near $\mu = 0$ (an idea explored already by Poincaré), Hénon considered *families* of orbits. This research led to his fascinating books of "generating families" [**2**] [**3**].

2.2. Particular Solutions

Concurrent with the development of the theoretical structure surrounding the problem was the discoveries of several particular solutions in special cases.

2.2.1. Euler's Solution. The earliest and simplest particular solution was discovered by Euler in 1765. In Euler's solution, the three masses are collinear, positioned according to their masses. Then at every future time, the masses remain

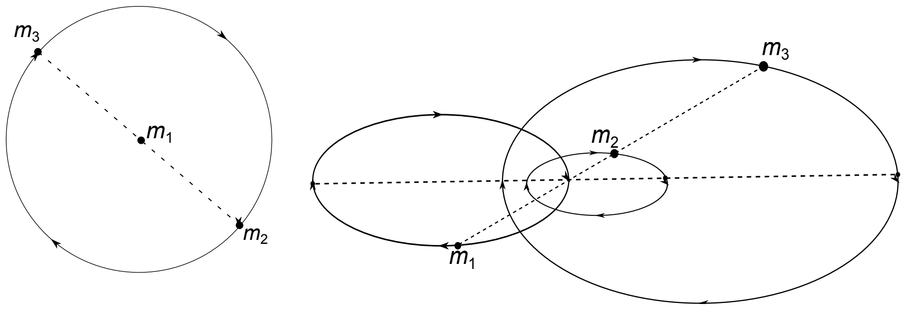


Figure 2.1. The Euler solution: the three bodies remain collinear at all times, in elliptical orbits around the centre of mass. Left: all masses equal. Right: unequal masses.

collinear, and the distances between them remain at the same ratio. In the particular case of the three bodies having equal masses, this corresponds to a situation where two of the bodies rotate in a circle around the third. For unequal masses, each mass will travel in an elliptical orbit around the centre of mass. See Figure 2.1 for an illustration of these behaviours.

Although this solution is interesting, and deeply connected to the theory we will develop, such orbits are unstable. The smallest of perturbations will destroy the symmetry, so such a situation will never be realised in a physical system.

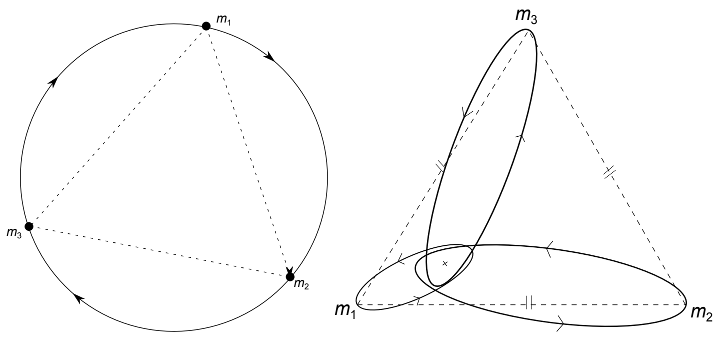


Figure 2.2. The Lagrange solution: the three bodies form an equilateral triangle at all times. Left: three equal masses. Right: unequal masses

2.2.2. Lagrange's Solution. In 1772 Lagrange discovered another particular solution. Lagrange's solution has the three masses occur at the vertices of an equilateral triangle. The masses will then follow elliptical orbits around their centre of mass while remaining in an equilateral triangle formation. When all three masses are equal, this gives a situation where the three bodies trace the same circular orbit fixed distances apart. See Figure 2.2.

Lagrange and Euler's solutions form the only explicit solutions to be discovered in the classical era of investigation. Unlike Euler's solution, Lagrange's can be stable, although only under certain conditions. If one of the masses completely dominates the system, then the orbits become stable. In fact, such orbits explain the existence of *Trojans*. Trojans are satellites (often asteroids) that orbit at specific locations relative to two larger masses. In the restricted case, both Euler and Lagrange's solutions reduce to a description of the Lagrangian points where these trojans are found. These Lagrangian points are discussed in Section 3.3.1.

2.2.3. The Figure-Eight Solution. Long after Euler and Lagrange's solutions, another particular solution was found: the figure-eight solution. First discovered numerically by Cristopher Moore in 1993, its existence was rigorously proved in 2001 by Alain Chenciner and Richard Montgomery [**11**]. Unlike the other particular solutions it only occurs in the case of three equal masses, but remarkably is stable. In the figure eight solution, all three bodies trace the same figure-eight shaped orbit, with the double point at the centre also being the centre of mass. This is illustrated in Figure 2.3.

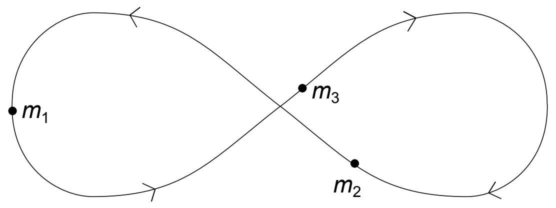


Figure 2.3. The figure-eight solution. The three equal masses chase each other in a figure-eight formation.

Although the solution is not as useful in analysing the problem as Euler or Lagrange's, it is notable for being discovered so recently. Its discovery led to what have been termed *N-body choreographies*, periodic solutions to the N-body problem where all bodies trace out the same path in space. These choreographies were studied by Carlés Simó who found many highly unlikely but extremely interesting orbits. The ongoing research into these kinds of orbits has led to some interesting considerations into the general N-body problem when Einstein's theory of General Relativity is taken into account. For more on these orbits, and some extremely interesting examples, see [**13**].

2.2.4. Hill's Solutions. Of much more practical importance are what are termed "Hill's solutions"[1] to the problem, also known as *tight binaries*. These solutions are very familiar, corresponding to the dynamics of the Sun-Earth-Moon system. In Hill's solution, two of the masses will remain close to each other and form a binary which will orbit the third body which remains further away. This is illustrated in Figure 2.4.

The kinds of orbits described by Hill are the most commonly occuring "nice" orbits, and examples can be found for any sufficiently low energy. These orbits will show up clearly in the model developed in this project.

[1]Despite the name, these do not necessarily occur as explicit solutions like Euler and Lagrange's solutions do.

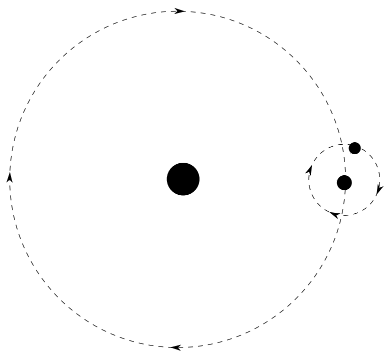


Figure 2.4. An example of a Hill Solution.

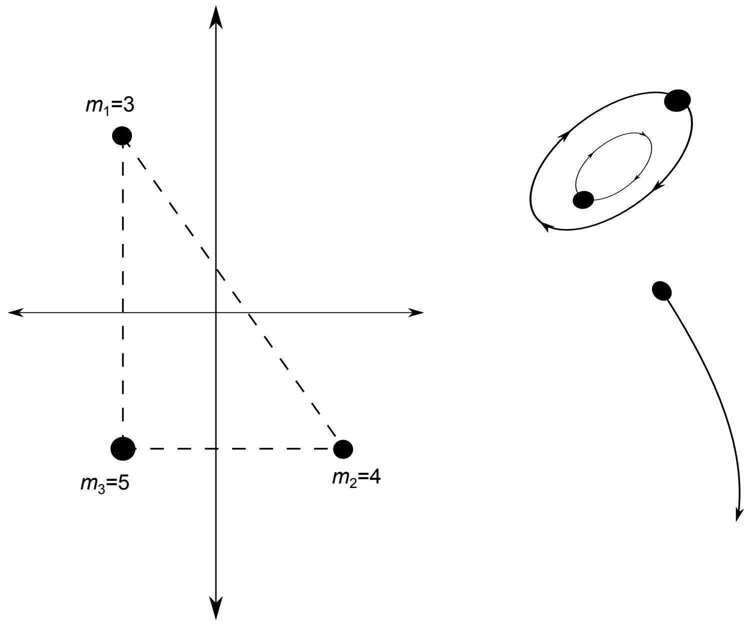


Figure 2.5. Left: The initial condition considered by the Pythagorean problem. The bodies have masses in the ratio 3:4:5 and are at the vertices of the 3-4-5 right angled triangle. Right: an illustration of the usual limiting behaviour of the Pythagorean Problem (and many other initial conditions), an escape orbit.

2.2.5. The Pythagorean Problem. The final set of "particular"[2] orbits we consider is the so-called *Pythagorean problem*. The Pythagorean problem considers an initial configuration of the three bodies with masses of 3, 4 and 5 units places at the corresponding positions on a right angled triangle (see Figure 2.5). It was

[2]Though as we shall see, there is nothing truly "particular" about them.

theorised by Ernst Meissel that such a configuration would lead to periodic orbits. By the 1960s, computers were sufficiently advanced to numerically integrate the problem (as we shall in this paper) and it was demonstrated that the orbits were not periodic, nor particularly unique. However, they did demonstrate a very typical behaviour of the system: one of the three masses drifts away from the others, leaving a stable binary system behind (Figure 2.5). This is common for our restricted problem, and for high energies is the norm rather than an exception. Such orbits are known as *escape orbits*.

2.3. Lagrangian and Hamiltonian Mechanics

Although the three body problem can be analysed from the equations of motion given by classical (or *Newtonian*) mechanics, in this project we will use the Hamiltonian approach. The neccesary theory will be outlined here; for a more thorough treatment, see [**6**].

The theory of *Lagrangian mechanics* was first formulated in 1788 by Lagrange. Unlike in Newtonian mechanics, where a rectangular coordinate system is regularly used, Lagrangian mechanics gives a much more natural way of considering generalized coordinate systems.

For instance, consider the problem of calculating the motion of a pendulum swinging in a two-dimensional plane under the force of gravity. The Lagrangian formulation of this problem can be made in terms of the angle the pendulum sweeps, rather than the x and y coordinates. Thus the problem has one degree of freedom only, in a very natural way. This use of generalised coordinates to incorporate constraints is a major advantage of Lagrangian mechanics.

Having established an appropriate set of coordinates, the Lagrangian itself is a function which is constructed as

$$\mathscr{L} = T - V \tag{2.2}$$

where $\mathscr{L}$ is our Lagrangian function (or simply Lagrangian), T is the *kinetic energy* of the system, and V is the *potential energy* of the system, all written in terms of the generalised coordinates q_i, the corresponding velocities $\dot{q}_i$, and the time t. Once this formula has been derived, the *Principle of Least Action* gives the *Lagrange equations*

$$\frac{\partial \mathscr{L}}{\partial q_i} = \frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\partial \mathscr{L}}{\partial \dot{q}_i}\right) \qquad i = 1, 2, ..., n. \tag{2.3}$$

These are n second-order differential equations for each q_i. In principle, these equations can be solved for a given initial condition to find the subsequent motion of the system.

If the Lagrangian function does not depend on time, the system is *autonomous*. In this case, we have that

$$\frac{\mathrm{d}}{\mathrm{d}t}\left\{\sum_{i=1}^{n} \dot{q}_i \frac{\partial \mathscr{L}}{\partial \dot{q}_i} - \mathscr{L}\right\} = 0. \tag{2.4}$$

Thus the quantity $E = \sum_{i=1}^{n} \dot{q}_i \frac{\partial \mathscr{L}}{\partial \dot{q}_i} - \mathscr{L}$ is constant, and is called the *energy*.

A refinement to these ideas came in 1833, when William Hamilton introduced *Hamiltonian mechanics*. The Hamiltonian formalism follows naturally from the Lagrangian formalism, but has some distinct differences. To switch from the Lagrangian formulation to the Hamiltonian formulation, a *Legendre transform* yields

$$\mathscr{H} = \sum_{i=1}^{n} p_i \dot{q}_i - \mathscr{L}. \tag{2.5}$$

The Hamiltonian $\mathscr{H}$ must be written in terms of the coordinates q_i, the conjugate momenta p_i and time t. If the Lagrangian can be written as

$$\mathscr{L} = \frac{1}{2}\dot{\mathbf{q}}^T T \dot{\mathbf{q}} + \Omega^T \dot{\mathbf{q}} - V(\mathbf{q}) \tag{2.6}$$

then the corresponding Hamiltonian is

$$\mathscr{H} = \frac{1}{2}(\mathbf{p} - \Omega)^T T^- 1(\mathbf{p} - \Omega) + V(\mathbf{q}). \tag{2.7}$$

Having constructed the Hamiltonian, rather than n second-order differential equations, we now have $2n$ first-order differential equations

$$\dot{q}_i = \frac{\partial \mathscr{H}}{\partial p_i} \qquad \dot{p}_i = -\frac{\partial \mathscr{H}}{\partial q_i}. \tag{2.8}$$

These are called *Hamilton's equations*, and again can be solved to give the evolution of the system over time.

A useful property of the Hamiltonian is that the conservation of energy described above is now simplified to $E = \mathscr{H}$. Again, this only applies to time-independent Hamiltonians. We will be exploiting this identity throughout the project.

As noted, much of the power of the Hamiltonian comes from the idea of *generalised coordinates*. We therefore need to construct coordinate and momentum transformations that preserve the Hamiltonian structure (that is, Hamilton's equations will still be valid). Such transformations are called *canonical transformations*, between sets of *canonical coordinates*. These transformations are usually calculated using generating functions. For a more thorough discussion of the theory of generating functions see [**6**]. We include the results and equations for reference. There are four kinds of generating functions, each of which give a relationship between two sets of canonical coordinates $q_1, ..q_n, p_1, ...p_n$ and $Q_1, ...Q_n, P_1, ...P_N$. By taking a function of some combination of old or new coordinates and old or new momenta, a canonical transformation is induced according to the table below[3].

[3]These generating functions can depend on time; for this project, we will not need this detail, so it is omitted.

Function	**Induced Transformation**
$F_1(q_i, Q_i)$	$p_i = \frac{\partial F_1}{\partial q_i} \qquad P_i = -\frac{\partial F_1}{\partial Q_i}$
$F_2(q_i, P_i)$	$p_i = \frac{\partial F_2}{\partial q_i} \qquad Q_i = \frac{\partial F_2}{\partial P_i}$
$F_3(p_i, Q_i)$	$q_i = -\frac{\partial F_3}{\partial p_i} \qquad P_i = -\frac{\partial F_3}{\partial Q_i}$
$F_4(p_i, P_i)$	$q_i = -\frac{\partial F_4}{\partial p_i} \qquad Q_i = \frac{\partial F_4}{\partial P_i}$

CHAPTER 3

Setting Up the Three Body Problem

3.1. The Two Body Problem: exact solution

To begin our analysis, we present a brief treatment of the *two body problem*. The exact solution of the two body problem is a well known result, but foreshadows some concepts we need again later.

Consider two bodies of mass m_1, m_2 and position $\mathbf{r}_1$, $\mathbf{r}_2$ respectively (the same results hold for $\mathbf{r}_1$ and $\mathbf{r}_2$ in $\mathbb{R}^2$ or $\mathbb{R}^3$). The only force acting upon each body is the pull of gravity from the other. Then the Lagrangian describing their motion is

$$\mathscr{L} = \underbrace{\frac{1}{2}m_1|\dot{\mathbf{r}}_1|^2 + \frac{1}{2}m_1|\dot{\mathbf{r}}_2|^2}_{\text{Kinetic Energy}} + \underbrace{\frac{Gm_1m_2}{|\mathbf{r}_1 - \mathbf{r}_2|}}_{\text{Potential Energy}} . \tag{3.1}$$

Let $M = m_1 + m_2$ be the total mass of our system, so the *centre of mass* is defined by

$$\mathbf{r}_G = \frac{m_1\mathbf{r}_1 + m_2\mathbf{r}_2}{M}. \tag{3.2}$$

Then the Lagrangian can be written as

$$\mathscr{L} = \frac{1}{2}M|\dot{\mathbf{r}}_G|^2 + \frac{1}{2}\frac{m_1m_2}{M}|\dot{\mathbf{r}}_2 - \dot{\mathbf{r}}_1|^2 + \frac{Gm_1m_2}{|\mathbf{r}_1 - \mathbf{r}_2|} \tag{3.3}$$

$$= \mathscr{L}_G + \mathscr{L}_R \tag{3.4}$$

where $\mathscr{L}_G$ depends only on the centre of mass, and $\mathscr{L}_R$ does not depend on the centre of mass. As $\mathscr{L}_G$ does not depend on $\mathbf{r_G}$, the centre of gravity moves with a *constant velocity*.

Now consider $\mathbf{r} = \mathbf{r}_1 - \mathbf{r}_2$, equivalent to only considering the motion of $\mathbf{r}_1$ relative to $\mathbf{r}_2$. Then $\mathscr{L}_R$ describes a single particle of mass $\frac{m_1m_2}{M}$ travelling around the gravitational field of a fixed central body of mass M. Its path is therefore a conic section. We only consider elliptical (or *Keplerian*) trajectories[1].

3.2. The Three Body Problem

3.2.1. The Planar Circular Restricted Three Body Problem. We now introduce the third body into the system. However, before continuing a number of simplifications are made.

The first and most prominent simplification is that the mass of the lightest body is negligible. We will refer to this body as a *particle*. This means the motion of the

[1]Trajectories can also be parabolic or hyperbolic, but these are not of concern here.

particle will not affect the orbits of the heavier bodies, which we call the *primary masses* or just primaries. This simplified system is referred to as the *restricted three body problem*.

The next simplification is to fix the orbits of the primaries. As they are only affected by each other and not the particle, the primaries are a two body system, which we solved exactly above. Although we discovered that the two body problem is solved by any conic section, we take a circular orbit for our simplified model. This is the *circular restricted three body problem*. As noted previously, this form of the problem was proposed by Euler.

Our final simplification is to restrict the motion of the particle to the same plane as the orbit of of the primaries. The orbits thus occur on a two-dimensional plane. This is the *planar circular restricted three body problem*, which will be referred to as the PCR3BP in this paper.

3.2.2. Aside: Relevance of the PCR3BP. We should take a moment to justify the use of the PCR3BP as an approximation for the full problem. A number of relevant real-life situations in celestial mechanics are very closely approximated by the PCR3BP, including models of spacecraft trajectories. Some specific applications include modelling comets interacting with Jupiter and the sun. The motion of these comets remain very close to Jupiter's orbital plane (that is, they remain near- planar with Jupiter and the Sun). For a thorough examination of the practical applications of this model, see [**22**]. The assumptions also allow for the modelling of trojans, and can predict their existence and trajectories.

3.2.3. Hamilton's Equations for the PCR3BP. We now derive Hamilton's equations for the PCR3BP. Let the *test particle* (the body of negligible mass) be denoted P. The *primary masses*, or simply *primaries*, are denoted M_1 and M_2. We assume, without loss of generality, that M_1 has a mass greater than or equal to that of M_2. The assumptions of the PCR3BP lead to the following:

- M_1 and M_2 have circular orbits around their centre of mass
- P remains in the same orbital plane as the primaries.
- The motion of the particle does not affect the motion of the primaries

To nondimensionalise the problem, take

- unit of mass the total of the masses of M_1 and M_2
- unit of length the distance between the primaries (which will be constant, as in Section 3.1)
- unit of time such that the period of the orbits of the primaries is 2π

These units force a gravitational constant $G = 1$. The only parameter in the system is the *mass parameter* $\mu \in [0, 1/2]$, such that M_1 has mass $1 - \mu$ and M_2 has mass μ.

Our coordinates have origin at the centre of mass of the two primary masses. This is fairly standard, although some texts place the origin at the larger of the two

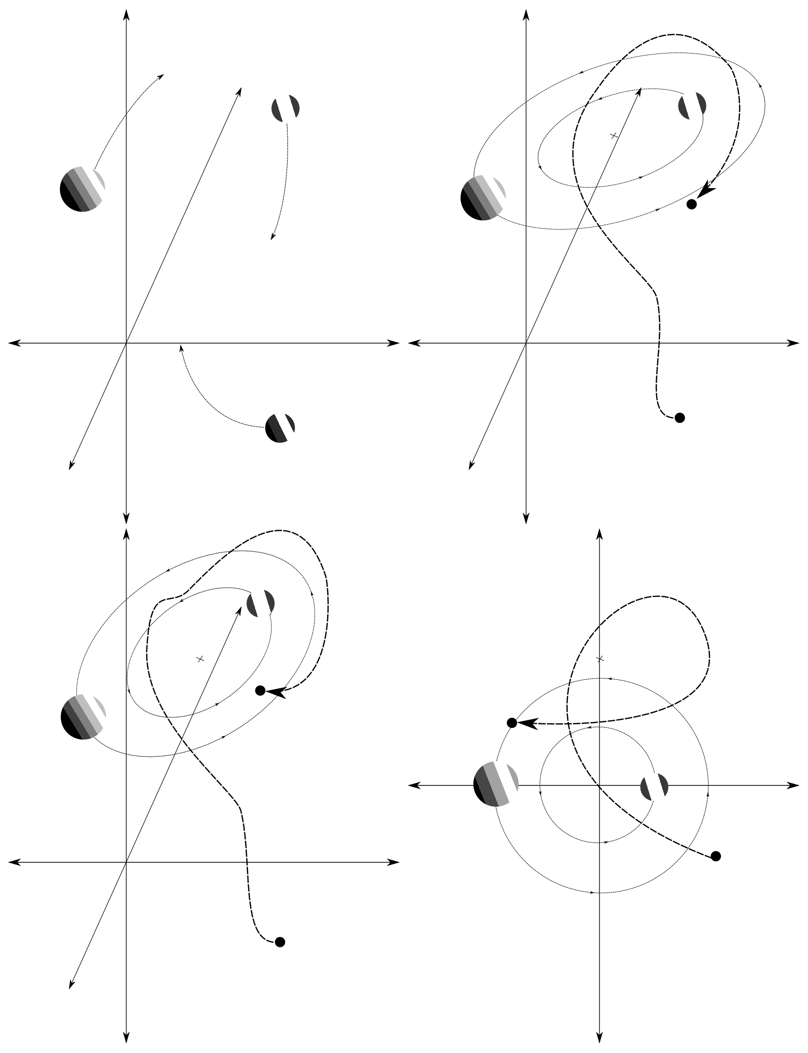


Figure 3.1. The simplifications we take to the three body problem. Top left: the general three body problem. Top right: the restricted three body problem. Bottom left: the circular restricted three body problem. Bottom right: the planar circular restricted three body problem.

bodies[2]. Then the trajectories of the larger bodies are given by

$$(X_1, Y_1) = (-\mu \cos t,\ -\mu \sin t) \quad (X_2, Y_2) = ((1-\mu)\cos t,\ (1-\mu)\sin t) \tag{3.5}$$

as per Section 3.1.

[2]for instance, [**2**].

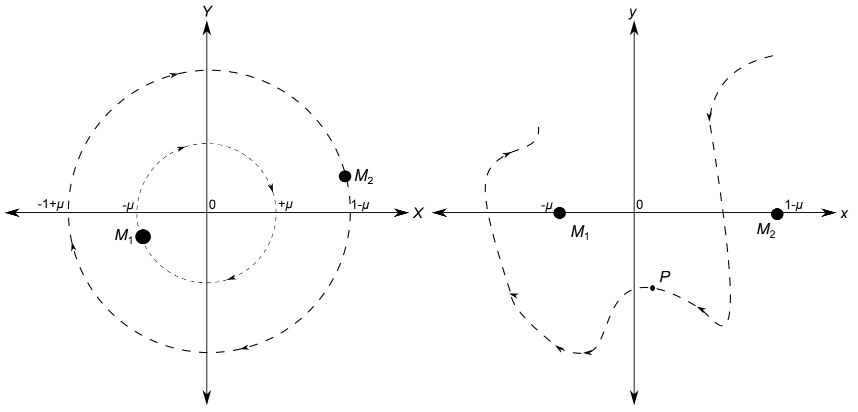


Figure 3.2. At left: fixed coordinates. At right: fixed coordinates for M_1 and M_2 and an illustrative path of the particle P.

We define r_1 and r_2 as the distances of P from M_1 and M_2 respectively:

$$\begin{aligned} r_1^2 &= (X + \mu\cos t)^2 + (Y + \mu\sin t)^2 \\ r_2^2 &= (X - (1-\mu)\cos t)^2 + (Y - (1-\mu)\sin t)^2. \end{aligned} \tag{3.6}$$

Then the Lagrangian describing the motion of P is given by

$$\mathscr{L}_{\text{inertial}}(t, X, Y, \dot{X}, \dot{Y}) = \frac{1}{2}(\dot{X}^2 + \dot{Y}^2) + \frac{1-\mu}{r_1} + \frac{\mu}{r_2} \tag{3.7}$$

This Lagrangian is time dependent. As discussed in Section 2.3, many of the nice properties of a Lagrangian or Hamiltonian system depend on having a time-independent system. Thus a transformation is taken to a *rotating frame of reference* (see Figure 3.2). This coordinate system also gives clearer insight into many behaviours of the three body problem, in particular motion near the Lagrangian points. The transformation is

$$\begin{aligned} X &= x\cos t - y\sin t \\ Y &= x\sin t + y\cos t. \end{aligned} \tag{3.8}$$

The corresponding velocities transform as

$$\begin{aligned} \dot{X} &= \dot{x}\cos t - \dot{y}\sin t - x\sin t - y\cos t \\ \dot{Y} &= \dot{x}\sin t + \dot{y}\cos t + x\cos t - y\sin t. \end{aligned} \tag{3.9}$$

The distances become

$$\begin{aligned} r_1^2 &= (x+\mu)^2 + y^2 \\ r_2^2 &= (x-(1-\mu))^2 + y^2 \end{aligned} \tag{3.10}$$

and the new Lagrangian is

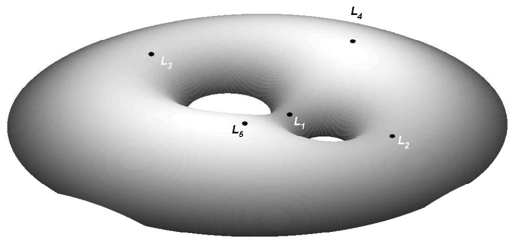


Figure 3.3. The effective potential, U, plotted against x and y for $\mu = 0.3$. Note the singularities around $(-\mu, 0)$ and $(1-\mu, 0)$ marked by "holes" in the potential, and the critical points, which coincide with the Lagrangian points.

$$\mathscr{L}_{\text{rotating}}(x, y, \dot{x}, \dot{y}) = \frac{1}{2}((\dot{x} - y)^2 + (\dot{y} + x)^2) + \frac{1-\mu}{r_1} + \frac{\mu}{r_2}. \tag{3.11}$$

Now, the Hamiltonian formulation is derived from the Lagrangian formulation. The conjugate momenta are

$$p_x = \frac{\partial \mathscr{L}}{\partial \dot{x}} = \dot{x} - y \quad p_y = \frac{\partial \mathscr{L}}{\partial \dot{y}} = \dot{y} + x \tag{3.12}$$

and the Legendre transform defined by

$$\mathscr{H} = \dot{x}p_x + \dot{y}p_y - \mathscr{L} \tag{3.13}$$

gives the corresponding Hamiltonian

$$\begin{aligned} \mathscr{H}(x, y, p_x, p_y) &= \tfrac{1}{2}p_x^2 + p_x y + \tfrac{1}{2}p_y^2 - p_y x - \tfrac{\mu}{r_1} - \tfrac{1-\mu}{r_2} \\ &= \tfrac{1}{2}(p_x + y)^2 + \tfrac{1}{2}(p_y - x)^2 - \tfrac{1}{2}x^2 - \tfrac{1}{2}y^2 - \tfrac{1-\mu}{r_1} - \tfrac{\mu}{r_2}. \end{aligned} \tag{3.14}$$

This value is conserved as the Hamiltonian is independent of time. This conserved value is the *Jacobi integral.*

We define the *effective potential* (illustrated in Figure 3.3) by [3]

$$\begin{aligned} U(x, y) &= -\frac{1}{2}(x^2 + y^2) - \frac{1-\mu}{r_1} - \frac{\mu}{r_2} \\ &= -\frac{1}{2}\{(1-\mu)r_1^2 + \mu r_2^2\} - \frac{1-\mu}{r_1} - \frac{\mu}{r_2} \end{aligned}$$

[3]We have ignored some constant terms. This is allowed with Hamiltonians as Hamilton's equations will remain unchanged, allowing us to add or remove constant terms as we wish.

and the *effective kinetic energy*[4] by

$$T(x,y,p_x,p_y) = \frac{1}{2}\left((x-p_y)^2+(y+p_x)^2\right). \tag{3.15}$$

Then

$$\mathscr{H}(x,y,p_x,p_y) = T(x,y,p_x,p_y)+U(r_1,r_2) \tag{3.16}$$

where U depends only on the location of the particle. (Note that we have written U in terms of r_1 and r_2; this is for convenience in our later calculations.)

We now have Hamilton's equations for the motion of the particle:

$$\begin{aligned}\dot{x} &= \frac{\partial\mathscr{H}}{\partial p_x}\\ &= T_{p_x}\end{aligned} \tag{3.17}$$

$$\begin{aligned}\dot{y} &= \frac{\partial\mathscr{H}}{\partial p_y}\\ &= T_{p_y}\end{aligned} \tag{3.18}$$

$$\begin{aligned}\dot{p_x} &= -\frac{\partial\mathscr{H}}{\partial x}\\ &= -T_x - U_x\\ &= -T_x - U_{r_1}\frac{\partial r_1}{\partial x} - U_{r_2}\frac{\partial r_2}{\partial x}\end{aligned} \tag{3.19}$$

$$\begin{aligned}\dot{p_y} &= \frac{\partial\mathscr{H}}{\partial y}\\ &= -T_y - U_y\\ &= -T_y - U_{r_1}\frac{\partial r_1}{\partial y} - U_{r_2}\frac{\partial r_2}{\partial y}.\end{aligned} \tag{3.20}$$

It is often useful to recontextualise the Hamiltonian in complex space. If $z = x+iy$ and $p_z = p_x - ip_y$ [5] then

$$\begin{aligned}\mathscr{H} =&\frac{1}{2}\left|z+i\bar{p}_z\right| - \frac{1}{2}\{(1-\mu)r_1^2+\mu r_2^2\} - \frac{1-\mu}{r_1} - \frac{\mu}{r_2}\\ =&\frac{1}{2}\left(p_z\bar{p}_z + i(z\bar{p}_z - zp_z)\right) - \frac{1-\mu}{r_1} - \frac{\mu}{r_2}\end{aligned} \tag{3.21}$$

$$r_1 = |z+\mu| \quad r_2 = |z-(1-\mu)|. \tag{3.22}$$

[4]It should be emphasised that this value is *not* the kinetic energy of the particle; this particular splitting is made for mathematical convenience rather than physical reasons.

[5]Strictly speaking, this transformation is not canonical. The Poisson bracket (an object which can be used to test if a given transformation is canonical) is $\{z,p_z\} = \{x+iy, p_x-ip_y\} = \{x,p_x\}+\{y,p_y\} = 2$ rather than 1. If we scale the variables as $z = {}^1/\sqrt{2}(x+iy)$, $p_z = {}^1/\sqrt{2}(p_x-ip_y)$, then the transformation is canonical. In practice, we can safely ignore this point, particularly as the equations will be converted back into separate coordinates at the final step.

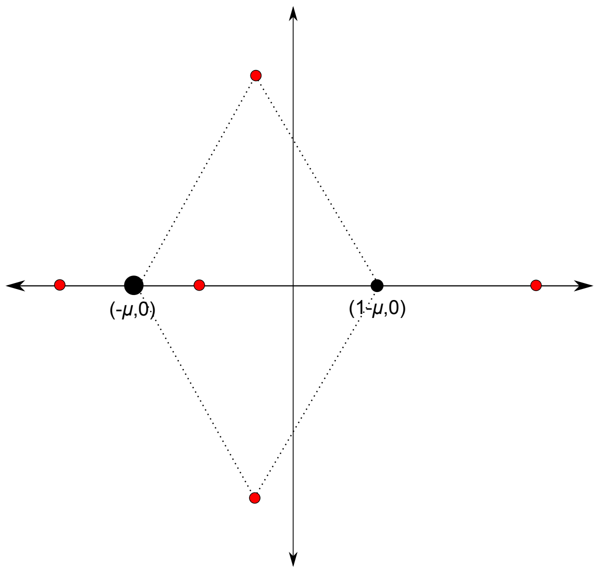


Figure 3.4. The five fixed points for the particle, or *Lagrangians*, marked in red.

3.3. Dynamics of the Three Body Problem

3.3.1. The Lagrangian Points. Important in our analysis are the equilibria of the system, which are referred to as the *Lagrangian points* in this context. There are five of these points, and we use the equations of motion from the Hamiltonian (3.14) to calculate them. By setting $\dot{p}_x = 0, \dot{p}_y = 0$ we get the following equations:

$$0 = (x+\mu)\left((1-\mu)(r_1^{-3}-1)+\mu(r_2^{-3}-1)\right)-\mu(r_2^{-3}-1) \tag{3.23}$$

$$0 = y\left((1-\mu)(r_1^{-3}-1)+\mu(r_2^{-3}-1)\right). \tag{3.24}$$

If $y \neq 0$, it follows that

$$r_1 = r_2 = 1 \tag{3.25}$$

and so there are two Lagrangian points, denoted L_4 and L_5, located at the third vertex of the two equilateral triangles with the two primary masses as vertices. The exact coordinates are $L_4(-\mu+\frac{1}{2}, \frac{\sqrt{3}}{2})$ and $L_5(-\mu+\frac{1}{2}, -\frac{\sqrt{3}}{2})$.

If $y = 0$, then $r_1 = |x+\mu|$ and $r_2 = |x-1+\mu|$ so

$$0 = -x + (x+\mu)(1-\mu)\frac{1}{|x+\mu|^3} + \mu(x+\mu-1)\frac{1}{|x+\mu-1|^3}. \tag{3.26}$$

This has one solution in each of the ranges $(-\infty, -\mu)$, $(-\mu, 1-\mu)$ and $(1-\mu, \infty)$. These solutions can be calculated numerically as they are the roots of polynomials.

It is important to consider the *stability* of these Lagrangian points as well. L_1, L_2 and L_3 are always unstable. L_4 and L_5 are stable only for certain values of the mass ratio; specifically, only when $M_1/M_2 > \frac{25}{2}(1+\sqrt{1-4/625}) \approx 24.96$. For our system, this becomes $\mu < \left(\frac{25}{2}(1+\sqrt{1-4/625})+1\right)^{-1} \approx 0.0385$ (these figures are derived in [**6**]).

As L_4 and L_5 are stable, it is possible for an orbit to perpetuate around these points. Objects which have orbits around L_4 or L_5 are called *trojans*. These trojans will be studied closely in Chapter 5.

3.3.2. Energy and the Hill Region. As our Hamiltonian is independent of time, the Hamiltonian is conserved. We refer to this as the Energy E. Having established this, we can define the *energy manifold*:

$$M(E,\mu) = \{(x,y,p_x,p_y) \mid H(x,y,p_x,p_y) = E\}\,. \tag{3.27}$$

This is a three-dimensional surface in our four-dimensional phase space on which P travels. This is defined for particular values of the parameter μ and a particular fixed energy level E, which will both be manually selected in our numerical computation. This is a general construction for time-independent Hamiltonians.

As the kinetic energy $T \geq 0$ and the effective potential U does not depend on p_x or p_y, we can easily project this manifold onto $x - y$ space:

$$\bar{M}(E,\mu) = \{(x,y) \mid U(x,y) \leq E\}\,. \tag{3.28}$$

This is known as *Hill's region*. Hill's region is easily calculated and provides useful information about the path of the orbits. There are five possible shapes for Hill's region, which are illustrated in Figure 3.5.

Note that the critical points of the effective potential coincide with the Lagrangian points, as shown in Figure 3.3. This will prove useful in our later analysis. In particular, we can calculate the curve of critical values in E - μ space for L_4 and L_5 [6]:

(3.29)

$$\begin{aligned}
E &= U(x,y) && \text{(for critical points)}\\
&= -\frac{1}{2}\{\mu r_1^2 + (1-\mu) r_2^2\} - \frac{\mu}{r_1} - \frac{1-\mu}{r_2} + \mu(1-\mu)\\
&= -\frac{1}{2}\{\mu + 1 - \mu\} - \mu - 1 - \mu + \mu(1-\mu) && (r_1 = r_2 = 1 \text{ at } L_{4,5})\\
&= -\frac{3}{2} + \mu(1-\mu).
\end{aligned}$$

3.4. The Poincaré Map

To help analyse the orbits, we can take a *Poincaré map*. A Poincaré map gives the intersection of an orbit with a particular subspace, called the *Poincaré section*. It is typically used for periodic orbits. The map takes a point in this fixed section, and gives the next intersection between the corresponding orbit and the section.

Definition 3.30 (Poincaré map)**.** *Given a function Φ defining an n-dimensional dynamical system, and S an $n-1$ dimensional surface, a diffemorphism*

$$P : S \to S \tag{3.31}$$

[6]Note that we briefly reintroduce the constants that were dropped previously; this is for consistency with the program to be developed in Chapter 5

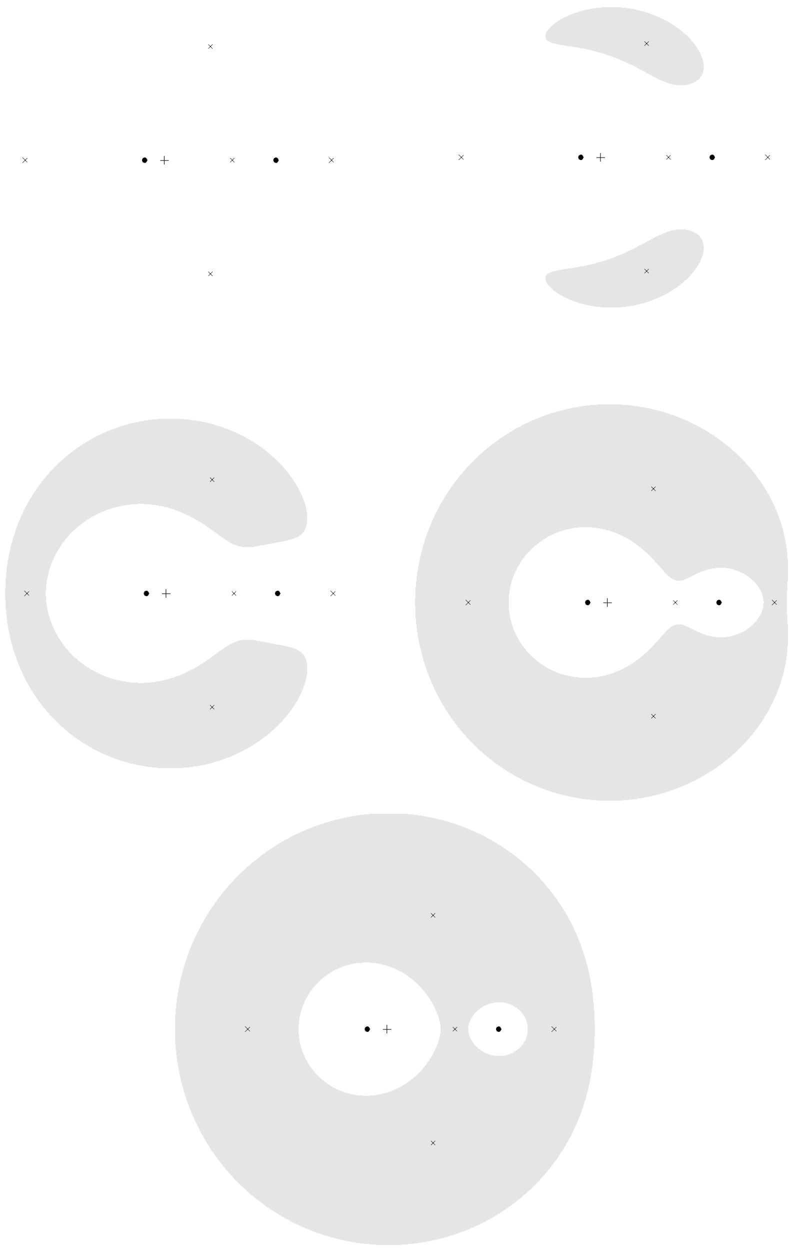


Figure 3.5. The five possible shapes for Hill's region. The five figures show Hill regions for decreasing values of the energy E from top to bottom. The white area is the region of possible motion, or Hill's region, and the grey area is the so-called "forbidden region". Also shown are the Lagrangian points, which coincide with the critical points of Hill's region. Compare these regions to the potential function in Figure 3.3. Also compare the critical points with the Lagrangian points in Figure 3.4.

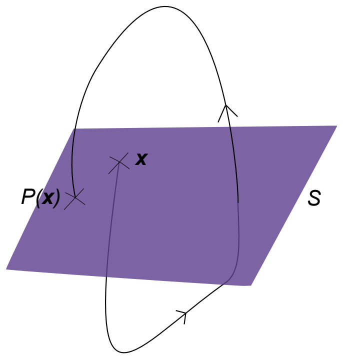


Figure 3.6. A Poincaré Map P. The map takes points on the surface S to the next point where the orbit intersects the surface.

is a ***Poincaré map*** *(or* ***first return map****) if for every point* $\boldsymbol{x} \in S$, $\exists \tau \geq 0$ *such that* $\Phi_\tau(\boldsymbol{x}) \in S$ *and* $\Phi_t(\boldsymbol{x}) \notin S \ \forall\ 0 \leq t < \tau$ *and* $P(\mathbf{x}) = \Phi_\tau(\boldsymbol{x})$. S *is called the* ***Poincaré section***.

For our Poincaré maps, we will also take only the intersecting points that have the same orientation. These ideas are illustrated in Figure 3.6.

Having defined this map, we can now characterise orbits based on the Poincaré map's action on the initial condition. For a *periodic orbit*, there exists some $n \in \mathbb{N}$ such that

$$P^n(\mathbf{x}) = \mathbf{x}. \tag{3.32}$$

We can also define a set

$$R(\mathbf{x}) = \{P^n(\mathbf{x}) | n \in \mathbb{N}\}. \tag{3.33}$$

For a periodic orbit, this set will be finite. As we will see, the nature of this set can be used to classify particular classes of orbits.

For a fixed energy, our system is restricted to the three dimensional energy manifold in equation (3.27). We should thus take a two dimensional section $S \subset M$. In general for a Hamiltonian with n spatial dimensions, we can take an $n-2$ dimensional section.

For this project, we will initially consider the section defined by

$$S_1(E;\mu) = \{(x, y, p_x, p_y) | (x, y, p_x, p_y) \in M(E;\mu); y = 0; p_y > 0\}. \tag{3.34}$$

This section is quite simple. The corresponding map calculates the next intersection of the particle with the x-axis, with a positive y-momentum. We can project the section into $x - p_x$ space, which allows us to visually acquire a lot of information about the dynamics of the system.

When the user makes a selection in the $x - p_x$ space, this uniquely defines an initial condition. The fixed parameters E and μ are used to calculate p_y. Rearranging equation (3.14),

$$\frac{1}{2}p_y^2 - xp_y + \left(-E + \frac{1}{2}p_x^2 + yp_x - \frac{1-\mu}{r_1} - \frac{\mu}{r_2}\right) = 0. \tag{3.35}$$

As E, μ, x and p_x have been chosen by the user, and y has been defined by the section, the quadratic in p_y can be solved to give the complete initial condition, with the choice of root determined by the condition $p_y > 0$.

A second useful section for this problem is

$$\begin{aligned} S_2(E;\mu) = \{(x, y, &p_x, p_y) | (x, y, p_x, p_y) \in M(E;\mu); \\ & y = \sqrt{3}(x+\mu);\;\; p_{perp} \geq 0\}. \end{aligned} \tag{3.36}$$

This section is a line passing through the primary at $(-\mu, 0)$ and L_4. We define p_{perp} as the momentum in the direction perpendicular to this line

$$p_{perp} = \frac{1}{2}(p_x - p_y\sqrt{3}) \tag{3.37}$$

and the momentum parallel to this line as

$$p_{norm} = \frac{1}{2}(p_x\sqrt{3} + p_y). \tag{3.38}$$

We can then project the section down to $x - p_{norm}$ space as we did for the previous section. The reason we study this section and the associated map is that it allows us to see the dynamics of orbits around L_4, which will not show up on S_2. As before equation (3.35) is used to determine the full initial condition (although it must be rewritten in terms of p_{perp} and p_{norm}).

One note here is that for the above sections, $P(\mathbf{x})$ is not necessarily well-defined for all $\mathbf{x} \in S$. In particular for S_2, the corresponding map P_2 is undefined for any particle that will subsequently orbit about the primary at M_2. There is no reason why P should be well defined everywhere for an arbitrarily constructed S.

However, in practice this becomes irrelevant. The maps allow us to see particular orbits we are interested in, and ignore the orbits for which $P(\mathbf{x})$ is undefined. S_1 is used to analyse orbits around either or both of the two primaries, while S_2 is used to analyse behaviour near L_4. If other behaviours need to be considered, new sections can be defined.

3.5. The Extended Phase Space

In the next chapter, time transformations will be applied to our Hamiltonian. For these transformations to be symplectic (that is, preserve the Hamiltonian structure), we introduce the concept of the *extended phase space*. In the extended phase space, we consider the time as "just another coordinate", allowing the construction of time transformations as canonical transformations.

Consider a general time-independent Hamiltonian $\mathscr{H}(q_1, ..., q_n, p_1, ...p_n)^7$. For this system, we have t as not only the integration variable but also as a physical parameter of the system. To generalise our Hamiltonian formulation, we remove this double-duty and let t be treated the same as the other coordinates, $q_{n+1} = t$. Then we need a new integration variable s, parametrising time as $t(s)$. Adjusting this parametrisation amounts to a time transformation.

If we introduce time as a coordinate, we must also introduce its conjugate momentum, p_{n+1}. It can be shown that this conjugate momentum is $p_{n+1} = -E$, the energy of the Hamiltonian. Then

$$\hat{\mathscr{H}}(q_1, ..q_n, t, p_1, ...p_n, E) = \mathscr{H}(q_1, ...q_n, p_1, ...p_n) - E \tag{3.39}$$

is the new Hamiltonian in the extended phase space, with s our new variable of integration. Hamilton's equation for t is simply $\frac{\mathrm{d}t}{\mathrm{d}s} = \frac{\partial \mathscr{H}}{\partial(-E)} = 1$, so t and s are equivalent. Hamilton's equations for the other coordinates are preserved. If we now need to take a time transformation of the form $\tau = f(q_1, ...q_n)t$, it suffices to define

$$\begin{aligned} \bar{\mathscr{H}}(q_1, ..q_n, \tau, p_1, ...p_n, E) &= f(q_1, ...q_n)\hat{\mathscr{H}}(q_1, ..q_n, t, p_1, ...p_n, E) \\ &= f(q_1, ...q_n)\left(\mathscr{H}(q_1, ...q_n, p_1, ...p_n) - E\right). \end{aligned} \tag{3.40}$$

Constructing these new Hamiltonians ensures our time transformations are canonical, and Hamilton's equations will remain valid. For a more thorough treatment of these concepts, see [**15**].

[7]This theory is usually derived for a time-dependent Hamiltonian, but is simplified here for clarity.

CHAPTER 4

Regularisations and other Numerical Considerations

Having derived the necessary equations of motion, the next step is to develop a robust integrator for these equations. This requires a discussion of some key concerns of the problem, most notably *the regularisation of collision orbits*. This chapter will also discuss some candidates for the integrator and other relevant numerical concerns.

4.1. Splitting Method for Hamiltonian

A very powerful method for integrating Hamilton's equations is the use of a *split step integrator*. It works by splitting the Hamiltonian into parts, which can often be integrated exactly.

The idea for a Hamiltonian split step integrator is a specific example of a more general idea. For $y \in \mathbb{R}^n$, $f_i : \mathbb{R}^n \to \mathbb{R}^n$ for $i = 1, 2$, consider the system defined by

$$\dot{y} = f_1(y) + f_2(y). \tag{4.1}$$

Then if ϕ_t is the flow induced by the system above, and ϕ_t^1 and ϕ_t^2 are the flows induced by $\dot{y} = f_1(y)$ and $\dot{y} = f_2(y)$ respectively, we can make the approximation

$$\phi_t \sim \phi^1_{t/2} \circ \phi^2_t \circ \phi^1_{t/2}. \tag{4.2}$$

This is known as a *Strang splitting*, and is a second-order method. More complex integrators of this type can be constructed by composing more $\phi^1_{\alpha t}$ and $\phi^2_{\beta t}$ with appropriate choices of α and β. Further analysis of this process is given in Hairer [**17**].

In particular, in the case of a Hamiltonian, the Hamiltonian often has the natural splitting $\mathscr{H} = T + V$. We can define ϕ_t^T and ϕ_t^V as the flows induced by considering T and V as separate Hamiltonians. Usually with Hamiltonians, T depends only on momentum and V only on position, which makes the corresponding flows very simple. In our case, T depends on position and momentum, but we can still solve for the flow exactly. This idea is often known as a *symplectic integrator*, as ϕ_t^T will conserve the kinetic energy and ϕ_t^V will conserve the potential energy.

Following on from (3.15), the differential equations for the flow induced by T are

$$\begin{aligned} \dot{x} &= T_{p_x} = p_x + y & \dot{y} &= T_{p_y} = p_y - x \\ \dot{p_x} &= -T_x = p_y - x & \dot{p_y} &= -T_y = -p_x - y \end{aligned} \tag{4.3}$$

or, in matrix form,

$$\dot{\mathbf{x}} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ -1 & 0 & 0 & 1 \\ -1 & 0 & 0 & 1 \\ 0 & -1 & -1 & 0 \end{pmatrix} \mathbf{x} = A\mathbf{x}. \tag{4.4}$$

We can explicitly calculate e^{tA}, giving the exact solution

$$\begin{aligned} x(t) =& A_1 \sin 2t + A_2 \cos 2t + C_1 \\ y(t) =& A_1 \cos 2t - A_2 \sin 2t + C_2 \\ p_x(t) =& A_1 \cos 2t - A_2 \sin 2t - C_2 \\ p_y(t) =& -A_1 \sin 2t - A_2 \cos 2t + C_1 \end{aligned}$$

where

$$\begin{aligned} A_1 &= \frac{1}{2}(y(0) + p_x(0)) \quad & A_2 &= \frac{1}{2}(x(0) - p_y(0)) \\ C_1 &= \frac{1}{2}(x(0) - p_y(0)) \quad & C_2 &= \frac{1}{2}(y(0) - p_x(0)). \end{aligned}$$

We can also calculate an explicit flow for the Hamiltonian given by U. The differential equations are

$$\dot{x} = 0 \quad \dot{y} = 0 \quad \dot{p_x} = -U_x \quad \dot{p_y} = -U_y. \tag{4.5}$$

As U depends only on x and y, the flow is (left in terms of partial derivatives for brevity)

$$\begin{aligned} x(t) =& x(0) \qquad & y(t) =& y(0) \\ p_x(t) =& -U_x(x(0), y(0))t + p_x(0) \qquad & p_y(t) =& -U_y(x(0), y(0))t + p_y(0). \end{aligned}$$

Having calculated ϕ_t^T and ϕ_t^U explicitly, the value of the splitting method is clear. We can substitute these into (4.2) to get

$$\phi_t^{\mathscr{H}} \sim \phi_{t/2}^U \circ \phi_t^T \circ \phi_{t/2}^U. \tag{4.6}$$

This integrator is accurate to second order. For our problem, this is more than adequate for many orbits. However, U and $U_{x,y}$ have singularities at $r_1, r_2 = 0$. As the orbits approach these points, the integrator becomes increasingly and fatally unstable. To be able to consistently and accurately integrate around these points, we must deal with these singularities.

4.2. Regularising Collision Orbits

4.2.1. Collision Orbits. Of the utmost importance in any attempt to integrate the three body problem is the consideration of *collision orbits*. These are the orbits which begin or end with $r_1 = 0$ or $r_2 = 0$, where our established equations of motion demonstrate singularities. For an integrator to be useful, it must be stable around these singularities. In particular, when determining trajectory of a probe from one planet to another, the probe will typically begin and end at the surface of the bodies, and so the calculations must be accurate there[1]. Another reason the integrator must be accurate near the primaries is the consideration of *low energy trajectories* which rely on the *slingshot effect* whereby a probe passes very close to planets and moons to achieve a "boost". This is discussed futher in [**22**].

U has simple poles at $r_1, r_2 = 0$. The singularities are non-essential, so we can manage them by taking transformations. This process is called the *regularisation* of collision (and near-collision) orbits. The process is to transform the plane to describe orbits which do collide with the primaries, as well as improve the stability of integrating orbits which come close to the primaries. The next step will be a time transformation, equivalent to taking smaller integration steps as the particle approaches the primaries.

4.2.2. Regularising Transformations. To transform our system of equations to one which we can numerically integrate without issue, we will take two steps: a *regularising transformation* and a *time transformation*.

To make our transformations more legible, the Cartesian plane is transformed into the complex plane, introducing the variable $z = x + iy$. The corresponding momentum transformation is then $p_z = p_x - ip_y$. [2]

Then a general coordinate transformation has the form

$$w = f(z), \quad f : \mathbb{C} \to \mathbb{C}. \tag{4.7}$$

This represents a conformal mapping of the plane, controlling the shape of the orbit. Figure 4.1 illustrates this.

The corresponding canonical momentum transformation can be calculate using the generating function

$$\begin{aligned} F_2(z, p_w) &= wp_w \\ &= f(z)p_w. \end{aligned} \tag{4.8}$$

Then

$$\begin{aligned} p_z &= \frac{\partial F_2}{\partial z} \\ &= f'(z)p_w \end{aligned} \tag{4.9}$$

and so

$$p_w = \frac{p_z}{f'(z)}. \tag{4.10}$$

[1]It should be noted that our derivation assume a point mass for both the primaries and the particle.. However, the difference in scale between the distance between planets and their diameters, as well the relative symmetry of the bodies, means that point masses are a very practical approximation. Relatively few changes need to be made to the model to allow for trajectories which actually begin and/or end on the surface of a planet.

[2]Recall that $p_z = p_x + ip_y$ is not a canonical transformation.

Then an elegant result exists for an appropriate time transformation associated with a given regularisation. As the introduction of p_w introduces a singularity at $f'(z) = 0$, we should take the time transformation defined by

$$\frac{\mathrm{d}t}{\mathrm{d}\tau} = |f'(z)|^2 \tag{4.11}$$

(where t is the "real" time scale and τ is our transformed time scale). The multplicative factor of $|f'(z)|^2$ will cancel with the singularity introduced by $|p_w|^2$. This will transform our equations such that a constant $\mathrm{d}\tau$ timestep will give a stable integrator around the primaries. For a more detailed proof and discussion of this result, as well as further motivation and discussion for the transforms used in this chapter, see Szebehly's *Theory of Orbits* [**16**].

As there are two singularities in our problem, we have a few choices. We can take *local regularisations*, where only one singularity is removed, or a *global regularisation*, where both are removed[3]. Each of these approaches has advantages and disadvantages, and we shall explore both options.

4.3. Local Regularisation

A simple approach is to take a local regularisation, removing only one singularity. We consider the *Levi-Civita transformation*, named for Tullio Levi-Civita who developed the transform when considering the two body problem. The transformation is given by

$$z = w^2 - \mu \tag{4.12}$$

for regularising collision orbits with the primary at $(-\mu, 0)$, illustrated in Figure 4.1. The corresponding transform for regularising collision orbits near the other primary is $z = w^2 + 1 - \mu$. This transformation takes the primary at $(-\mu, 0)$ to the origin, and the primary at $(1 - \mu, 0)$ to ± 1. Although the transformation is not bijective, this has no real impact on our calculations. We can thus choose $w = +\sqrt{z + \mu}$ (taking the principal complex square root) as our inverse.

The theory of *canonical transformations* affords p_w, the conjugate momenta to w that preserves Hamilton's equations. We construct the generating function

$$\begin{aligned} F_2(z, p_w) &= wp_w \\ &= \sqrt{z + \mu}p_w. \end{aligned} \tag{4.13}$$

Then

$$\begin{aligned} p_z &= \frac{\partial F_2}{\partial z} \\ &= \frac{p_w}{2\sqrt{z + \mu}} \\ &= \frac{p_w}{2w}. \end{aligned} \tag{4.14}$$

So the canonical momentum transformation is $p_w = 2wp_z$.

[3]It should be noted that the terminology "local" and "global" here is Birkhoff's, and has no deeper mathematical interpretation.

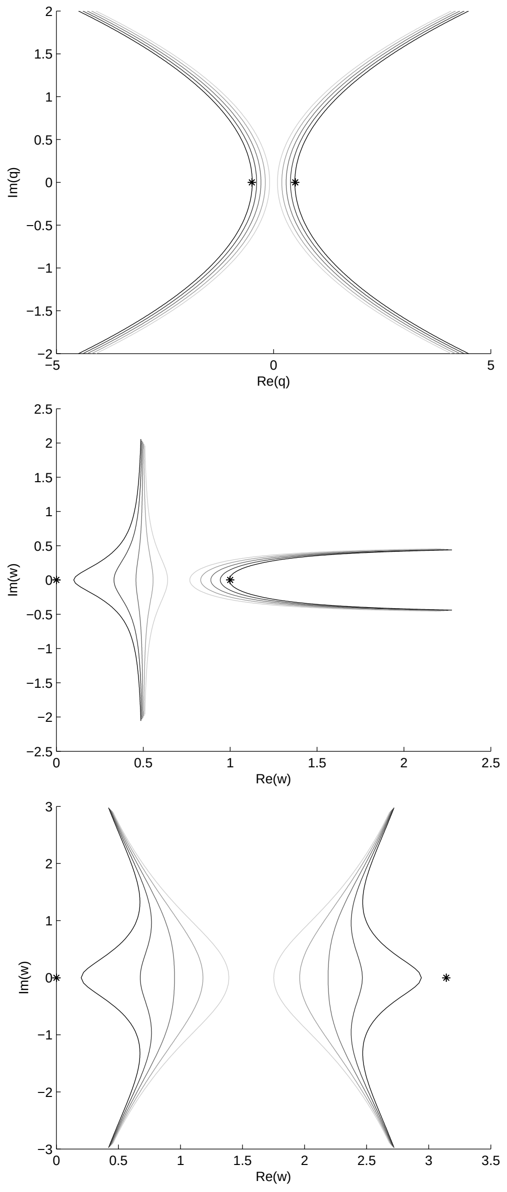


Figure 4.1. An illustration of some regularisations of the plane. In each diagram, the primaries are marked by a $*$. Top: several curves in z-space (before regularisation) for $\mu = {}^1\!/_2$. Middle: the same curves after the Levi-Civita transformation is taken to regularise collisions with $z = -\frac{1}{2}$ only. Bottom: the same curves after taking the Thiele-Burrau transformation. Note that the curves are not actual orbits and are included only to illustrate the regularisation process.

As $r_1 = |z+\mu| = |w^2|$ and $r_2 = |z-1+\mu| = |w^2-1|$, we can easily rewrite the effective potential under this transformation:

$$
\begin{aligned}
U(r_1, r_2) &= -\frac{1}{2}\{(1-\mu)r_1^2 + \mu r_2^2\} - \frac{1-\mu}{r_1} - \frac{\mu}{r_2} \\
&= -\frac{1}{2}\{(1-\mu)|w|^4 + \mu|w^2-1|^2\} - \frac{1-\mu}{|w|^2} - \frac{\mu}{|w^2-1|}.
\end{aligned}
\tag{4.15}
$$

This, along with the same transformation of the kinetic energy T, define a new Hamiltonian $\mathscr{H}_1(w, p_w)$. The details are omitted here for brevity.

Similarly, for regularisation about the primary at $(1-\mu, 0)$, the transformation is

$$w = \sqrt{z - 1 + \mu} \quad p_w = 2wp_z \tag{4.16}$$

and the effective potential becomes

$$U(w) = -\frac{1}{2}\{(1-\mu)|w^2+1|^2 + \mu|w|^4\} - \frac{1-\mu}{|w^2+1|} - \frac{\mu}{|w|^2}. \tag{4.17}$$

4.3.1. Time Transformations. The next step is to take the time transformation. As we have established the structure of the extended phase space in 3.5 and calculated the form of the time transformation, most of the work is already done. We first define our extended phase space Hamiltonian

$$\hat{\mathscr{H}}_1(w, t, p_w, E) = \mathscr{H}_1(w, p_w) - E \tag{4.18}$$

(the energy E is invariant under our transformations) and then take the time transformation given by Equation (4.11)

$$\begin{aligned} \bar{\mathscr{H}}_1(w, t, p_w, E) &= |f'(w)|^2 \hat{\mathscr{H}}_1(w, t, p_w, E) \\ &= 4|w|^2 \hat{\mathscr{H}}_1(w, t, p_w, E). \end{aligned} \tag{4.19}$$

The $|w|^2$ cancels with the singularity in the potential at $w = 0$. Analogously, we can derive a Hamiltonian $\bar{\mathscr{H}}_2(w, t, p_w, E)$ for orbits passing near the other primary.

The integration process is now

Algorithm 4.20 (Local Transformation Integrator)**.** *For some fixed $\theta < \frac{1}{2}$,*

1. *If $r_1 < \theta$, take a (Euler, Runge-Kutta, etc) $\mathrm{d}t$ integration step in $\phi_t^{\bar{\mathscr{H}}_1}$ (the flow induced by $\bar{\mathscr{H}}_1$)*
2. *Else, if $r_2 < \theta$, take a $\mathrm{d}t$ step in $\phi_t^{\bar{\mathscr{H}}_2}$*
3. *Else, take a $\mathrm{d}t$ step in $\phi_t^{\mathscr{H}}$*
4. *Repeat*

Such an integrator works, but is inelegant. The natural question is can we develop a transformation that regularises both singularities at once? In the next section we will see two examples of exactly that: the *Birkhoff transformation* and the *Thiele-Burrau transformation*

4.4. Global Regularisations

4.4.1. The Birkhoff Transformation. The first step in developing our global transformations is to translate the origin so the primaries are at $(-1/2, 0)$ and $(+1/2, 0)$. This lends some symmetry and simplification to our transformation functions. The canonical transformation is given by

$$q = z + \mu - \frac{1}{2}, \qquad p_q = p_z \tag{4.21}$$

($z = x + iy$, as previously). Now $r_1 = |q + \frac{1}{2}|$ and $r_2 = |q - \frac{1}{2}|$.

To motivate Birkhoff's transformations, we look at transformations of the form

$$q = f(w) = \alpha w + \beta w^{-1}. \tag{4.22}$$

The inverse transformation is then

$$w = \frac{q \pm \sqrt{q^2 - 4\alpha\beta}}{2\alpha} \tag{4.23}$$

and if we again take the time transform as defined by (4.11)

$$\frac{\mathrm{d}t}{\mathrm{d}\tau} = |f'(w)|^2 = \frac{|\alpha w^2 - \beta|^2}{|w|^4}. \tag{4.24}$$

Then the singular part of the potential is

$$\frac{1-\mu}{r_1} + \frac{\mu}{r_2} = \frac{1-\mu}{|\alpha w + \beta w^{-1} + \frac{1}{2}|} + \frac{\mu}{|\alpha w + \beta w^{-1} - \frac{1}{2}|} \tag{4.25}$$

$$= \frac{(1-\mu)|w|}{|\alpha w^2 + \beta + \frac{1}{2}w|} + \frac{\mu|w|}{|\alpha w^2 + \beta - \frac{1}{2}w|} \tag{4.26}$$

and applying the time transformation gives

$$|f'(w)|^2 \left(\frac{1-\mu}{r_1} + \frac{\mu}{r_2}\right) = \frac{1}{|w|^3}\left(\frac{(1-\mu)|\alpha w^2 - \beta|^2}{|\alpha w^2 + \beta - \frac{1}{2}w|} + \frac{\mu|\alpha w^2 - \beta|^2}{|\alpha w^2 + \beta + \frac{1}{2}w|}\right). \tag{4.27}$$

Now, our primaries are at $q = \pm\frac{1}{2}$. We wish to eliminate the singularities at these points, and so we need $q = \pm\frac{1}{2}$ to be a factor of the numerators. At $q = \frac{1}{2}$, we have $w = \frac{1}{4\alpha}\left(1 \pm \sqrt{1 - 16\alpha\beta}\right)$. But roots of the numerators in (4.27) are given by $w = \pm\sqrt{\frac{\beta}{\alpha}}$, so

$$\frac{1}{4\alpha}\left(1 \pm \sqrt{1 - 16\alpha\beta}\right) = \pm\sqrt{\frac{\beta}{\alpha}} \tag{4.28}$$

$$16\alpha\beta = 1. \tag{4.29}$$

We can now force $(\frac{1}{2}, 0)$ to be a fixed point of our equation, giving $\frac{1}{2} = \frac{1}{4\alpha}$. Thus $\alpha = \frac{1}{2}$ and $\beta = \frac{1}{8}$. The completed transformation is

$$q = \frac{w}{2} + \frac{1}{8w} = \frac{1}{4}(2\omega + \frac{1}{2\omega}). \tag{4.30}$$

It should be noted from equation (4.27) that a singularity has been introduced in the potential at $|w| = 0$. However this is not an issue, as this corresponds to $|q| \to \infty$. We can thus say our transformation regularises all points in the *finite* plane. The details of the behaviour as $|q| \to \infty$ are not interesting in our problem; this is just our particle floating off into empty space, an escape orbit.

A neat result is the relationship between the distances in the w plane and in the q plane. If

$$\rho_1 = |w + \frac{1}{2}|, \quad \rho_2 = |w - \frac{1}{2}|, \quad \rho = |w| \tag{4.31}$$

and r_1, r_2 are as defined previously, then

$$|f'(w)|^2 = \frac{\rho_1^2 \rho_2^2}{4\rho^4} = \frac{r_1 r_2}{\rho^2} \tag{4.32}$$

$$r_1 = \frac{\rho_1^2}{2\rho} \tag{4.33}$$

$$r_2 = \frac{\rho_2^2}{2\rho} \tag{4.34}$$

so

$$|f'(w)|^2 U = \frac{\rho_1^2 \rho_2^2}{32\rho^6}\left((1-\mu)\rho_1^4 + \mu\rho_2^4\right) + \frac{1}{2\rho^3}\left((1-\mu)\rho_2^2 + \mu\rho_1^2\right). \tag{4.35}$$

Then $\mathscr{H}_{\text{Birkhoff}}$ can be constructed in the extended phase space exactly as it was in the case of the local transformation, and we have a globally regularised Hamiltonian.

4.5. The Thiele-Burrau Transformation

Although the Birkhoff transformation is easier to motivate and construct algebraically, the resulting Hamiltonian is quite messy, and retrieving Hamilton's equations worse again. Therefore, we look at the *Thiele-Burrau transformation*, a connected transformation that yields equations that are easier to manipulate.

Constructed first by Thorvald Thiele for the $\mu = \frac{1}{2}$ case, the transformation was generalised by Carl Burrau for other values of μ (see [**16**] for futher background). It is defined in the q plane by

$$q = \frac{1}{2}\cos w \tag{4.36}$$

(the complex-valued $\cos$ function) or, equivalently

$$q = \frac{1}{4}(e^{iw} + \frac{1}{e^{iw}}). \tag{4.37}$$

The canonical momentum transformation is

$$p_q = \frac{-2p_w}{\sin w}. \tag{4.38}$$

One can compare equations (4.37) and (4.30) to see the close relationship between the Birkhoff and the Thiele-Burrau. In fact, if we let w_B be the coordinates defined by the Birkhoff transformation, and w_T be the coordinates defined by the Thiele Burrau transformation, we can confirm that

$$2w_b = e^{iw_T}. \tag{4.39}$$

We will include the details for the transformation in this case, as we will be using the calculations in our final integrator. Working from equation (3.21),

$$\begin{aligned} \mathscr{H}(z, p_z) =& \frac{1}{2}|z + i\overline{p_z}|^2 - \mu r_1^2 - (1-\mu) r_2^2 - \frac{1-\mu}{r_1} - \frac{\mu}{r_2} \\ =& \frac{1}{2}\left(p_z \overline{p_z} + i(\overline{z p_z} - z p_z)\right) - \frac{1-\mu}{|z+\mu|} - \frac{\mu}{|z-1+\mu|} \end{aligned} \tag{4.40}$$

$$\mathscr{H}(q,p_q) = \frac{1}{2}\left(p_q\overline{p_q} + i(\overline{qp_q} - qp_q + (1/2-\mu)(\overline{p_q}-p_q))\right) - \frac{1-\mu}{|q+1/2|} - \frac{\mu}{|q-1/2|} \tag{4.41}$$

$$\begin{aligned}\mathscr{H}(w,p_w) =& \frac{1}{2}\Bigg(\frac{-2p_w}{\sin w}\overline{\frac{-2p_w}{\sin w}} + i(\overline{\frac{1}{2}\cos w\frac{-2p_w}{\sin w}} \\ &- \frac{1}{2}\cos w\frac{-2p_w}{\sin w} + (1/2-\mu)(\overline{\frac{-2p_w}{\sin w}} - \frac{-2p_w}{\sin w}))\Bigg) \\ &- \frac{1-\mu}{|\frac{1}{2}\cos w + 1/2|} - \frac{\mu}{|\frac{1}{2}\cos w - 1/2|}.\end{aligned} \tag{4.42}$$

In extended phase space, the time transformed Hamiltonian is

$$\begin{aligned}\mathscr{K}(w,t,p_w,E) &= |f'(w)|^2\left(\mathscr{H}(w,p_w) - E\right) \\ &= \frac{1}{4}\sin^2 w\left(\mathscr{H}(w,p_w)-E\right).\end{aligned} \tag{4.43}$$

Using a computer algebra package, we can expand this in terms of u, v, p_u and p_v, where $w = u + iv$ and $p_w = p_u - ip_v$. A useful identity here is

$$\sin^2 w = (\cosh^2 v - \cos^2 u). \tag{4.44}$$

The full Hamiltonian is

$$\begin{aligned}\mathscr{K}(q,t,p_q,E) =& \frac{p_u^2}{2} + \frac{p_v^2}{2} + \frac{1}{2}(\cos u - 2\mu\cos u - \cosh v) \\ &+ \frac{1}{4}\sin u(\cosh v - 2\mu\cosh v + \cos u)p_v \\ &+ \frac{1}{4}\sinh v(\cos u - 2\mu\cos u + \cosh v)p_u \\ &+ \frac{1}{8}(\cos(2u) - \cosh(2v))E\end{aligned} \tag{4.45}$$

which yields the system of Hamilton's equations [4]

$$\dot{u} = p_u + \frac{1}{4}((1-2\mu)\cos u + \cosh v)\sinh v \tag{4.46}$$

$$\dot{v} = p_v + \frac{1}{4}(\cos u + (1-2\mu)\cosh v)\sin u \tag{4.47}$$

$$\begin{aligned}\dot{p_u} =& \frac{(1-2\mu)\sin u}{2} + \frac{p_v\sin^2 u}{4} + \frac{(1-2\mu)\sin u\sinh v p_u}{4} \\ &+ \frac{(\cos u - (1-2\mu)\cosh v)\cos u p_v}{4} + \frac{E\sin(2u)}{4}\end{aligned} \tag{4.48}$$

[4]Care should be taken here to note that $\dot{u}$ etc. now refer to derivatives with respect to τ, our new integration variable, rather than t.

$$\dot{p_v} = \frac{\sinh v}{2} - \frac{\sinh^2 v p_u}{4} + \frac{(\cosh v - (1-2\mu)\cos u)\cosh v p_u}{4} - \frac{(1-2\mu)\sin u \sinh v p_v}{4} + \frac{E\sinh(2v)}{4}. \tag{4.49}$$

For completeness, we should note that the extended variables evolve according to

$$\dot{E} = 0 \tag{4.50}$$

as we'd expect for a system with constant energy, and

$$\begin{aligned} \dot{t} &= \frac{1}{8}(\cos(2u) - \cosh(2v)) \\ &= \frac{1}{4}(\cos^2 u - \cosh^2 v) \end{aligned} \tag{4.51}$$

which corresponds to the time transformation $\frac{\mathrm{d}t}{\mathrm{d}\tau}$ taken in (4.43).

We thus have a complete description of the regularised system. It is this transformation that will be implemented in the integrator we will develop in the next chapter.

4.6. Numerical Scheme

Having derived the equations of motion, we now need to develop an numerical integrator. Unfortunately, a splitting method as per Section 4.1 is not obvious. Fortunately, equations (4.46) to (4.49) do not require a consideration of the step size or distance from the singularities thanks to the regularisation process. Thus, we can "blindly" integrate them using an off-the-shelf method such as Runge-Kutta.

The Runge-Kutta methods are iterative methods used to approximate solutions to systems of ordinary differential equations. We take the fourth-order Runge-Kutta scheme, known as RK4. In general, the scheme is given as:

Algorithm 4.52 (Fourth Order Runge-Kutta scheme)**.** *For a system given by*

$$\dot{\mathbf{x}} = f(t, \mathbf{x}) \quad \mathbf{x}(t_0) = \mathbf{x}_0. \tag{4.53}$$

for some timestep Δt, the approximation to $\mathbf{x}(t_0 + \Delta t)$ is given by

$$\mathbf{x}(t_0 + \Delta t) = \mathbf{x}(t_0) + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4) \tag{4.54}$$

where

$$\begin{aligned} \mathbf{k}_1 &= f(t_0, \mathbf{x}_0)\Delta t \\ \mathbf{k}_2 &= f(t_0 + \frac{1}{2}\Delta t, \mathbf{x}_0 + \frac{1}{2}\mathbf{k}_1)\Delta t \\ \mathbf{k}_3 &= f(t_0 + \frac{1}{2}\Delta t, \mathbf{x}_0 + \frac{1}{2}\mathbf{k}_2)\Delta t \\ \mathbf{k}_4 &= f(t_0 + \Delta t, \mathbf{x}_0 + \mathbf{k}_3)\Delta t. \end{aligned} \tag{4.55}$$

Iterating this scheme approximates propogating the initial condition $\mathbf{x}(t_0)$.

Many other Runge-Kutta schemes exist for higher orders and other choices of coefficients in k_i. We will see in the next chapter that this scheme is more than accurate enough for our purposes.

4.7. Calculating the Poincaré Map

Another difficulty is the numerical calculation of the Poincaré maps constructed in Section 3.4. Although in our implementation we will use a naïve but adequate approach, we present here brief outline of a clever method proposed by Henon [**20**] for calculating these maps that could be implemented in the future.

Algorithm 4.56 (Henon's Method for Calculating Poincare Maps)**.** *First consider a dynamical system*

$$\dot{\mathbf{x}} = f(\mathbf{x}) \quad \mathbf{x} \in \mathbb{R}^n \tag{4.57}$$

and the associated Poincaré section

$$S = \{(x_1, x_2, ...x_n) | (x_n - a) = 0\} \tag{4.58}$$

with Poincaré map P. By dividing the system by $\frac{\mathrm{d}x_n}{\mathrm{d}t}$ and inverting the last equation, we get the new system

$$\begin{aligned} \frac{\mathrm{d}x_1}{\mathrm{d}x_n} &= \frac{f_1}{f_n} \\ \frac{\mathrm{d}x_2}{\mathrm{d}x_n} &= \frac{f_2}{f_n} \\ &\vdots \\ \frac{\mathrm{d}x_{n-1}}{\mathrm{d}x_n} &= \frac{f_{n-1}}{f_n} \\ \frac{\mathrm{d}t}{\mathrm{d}x_n} &= \frac{1}{f_n}. \end{aligned} \tag{4.59}$$

Now the process to calculate the Poincaré map is given as

1. *Take an initial condition $\mathbf{x}_0 \in S$*
2. *Propogate* (4.57)*, calculating $(x_n - a)$ at each step*
3. *Stop when $(x_n - a)$ changes sign*
4. *Take single integration step of size $\Delta x_n = -(x_n - a)$ in system* (4.59)

Then $P(\mathbf{x}_0) = \mathbf{x}$, up to the accuracy of our integration method.

This algorithm must be adjusted for a more general section. Consider the Poincaré section defined by

$$S = \{(x_1, x_2, ...x_n) | D(x_1, x_2, ...x_n) = 0\} \tag{4.60}$$

for some function D. Then we introduce the variable

$$x_{n+1} = D(x_1, x_2, ...x_n) \tag{4.61}$$

and

$$f_{n+1}(\mathbf{x}) = \sum_{i=1}^{n} f_i(\mathbf{x}) \frac{\partial D}{\partial x_i}. \tag{4.62}$$

Then the section is defined by $x_{n+1} = 0$*, and we can follow the procedure described above.*

Although this method is accurate to the same level as our integration method and theoretically does not require much analytical work, it does require the introduction of a lot of new structure in the program to be developed in the next chapter. A much simpler algorithm is used

Algorithm 4.63 (Simple Method for Calculating Poincare Maps)**.** *Given the system in* (4.57) *and the associated section,*

1. *Take an initial condition* $\mathbf{x}_0 \in S$
2. *Propogate* (4.57)*, calculating* D*, the signed distance to the section, at each point*
3. *Stop when* D *changes sign*
4. *Extrapolate* $P(\mathbf{x}_0)$ *from the current point, the previous point and their distances from the section*

Although this method is less accurate and far less defensible, our integrator will take sufficiently small steps that we can calculate these points to a sufficiently high accuracy.

CHAPTER 5

Results

Having established the necessary theory and discussed notable properties of the system, the next step is to implement this theory as a computer program. Although individual orbits and particular behaviours can be derived analytically, a user-friendly program allows us to quickly get a broader (if perhaps less detailed) insight into the dynamics of the system, especially across a large range of parameter values. In particular, the study of the Poincaré map for the system is very intuitive in a visual setting, and allows numerical analysis of the bifurcations of the system without the (extensive) analytical theory.

5.1. Program Design and Methodologies

The integration of Equations (4.46)-(4.49) using the Runge-Kutta method 4.52 is itself simple. A numerical computing environment such as MATLAB or Mathematica has built-in functions that not only integrate the differential equations but display the results in very few lines of code.

However, we require the program to be interactive and user-friendly, and hopefully run quickly. To achieve these goals, the solution is implemented in C++. C++ is a lower-level language than MATLAB or Mathematica, and so requires considerably more detailed code, but runs much faster and allows for a much greater flexibility and level of control. OpenGL is used for the graphics and interactivity, with GLUT (the OpenGL Utility Toolkit) providing simpler access to graphic commands.

The full code is provided in Appendix B. The control structure of the program is summarised in Figure 5.1.

To evaluate the precision of the calculations, we define the *energy error*

$$EE = |E - \mathscr{H}(x, y, p_x, p_y)| \tag{5.1}$$

(that is, the difference between the energy parameter and the currently calculated energy value). Informally, this value is $\mathcal{O}(10^{-9})$ for the periodic orbits we study, and other orbits which do not travel too far from the primaries. This can be adjusted by changing the size of a single integration step. As the trajectories stray from the primaries, this energy error increases. It is thought that the error comes from the mechanics of the Thiele-Burrau transform, which effectively "squashes" the space far from the primaries. The limitations of C++ (in particular, the accuracy of a `double`) lead to inaccuracies as the space becomes more squashed. For this project we do not consider the dynamics of the particle past the influence of the primaries, so this is of little concern. However, we could possibly integrate these trajectories

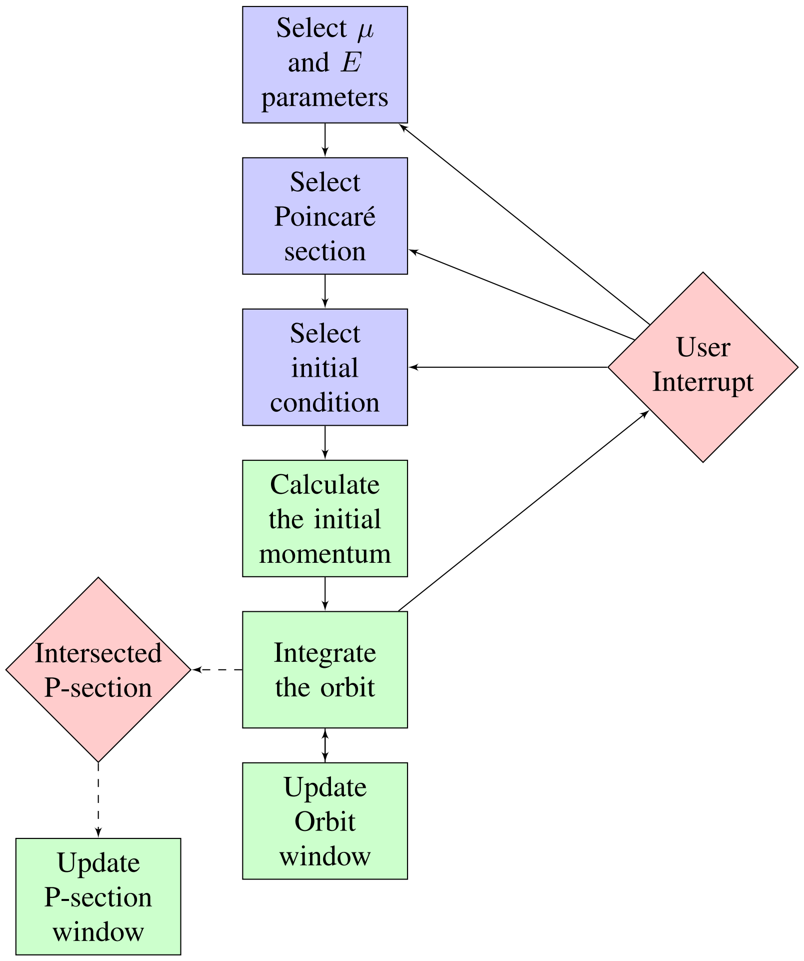


Figure 5.1. Control flow diagram for the program. The user makes selections for the parameters and initial condition using either the point-and-click interface or by direct entry at the command line. Equation (3.35) or similar is used to calculate the full initial condition. The program then integrates the orbits and calculates the corresponding Poincaré map until it is interrupted by the user.

by switching to the split-step integrator described in Section 4.1, which is very accurate (if slower) away from the primaries.

Figure 5.2 shows the User Interface for the program. This has five windows

- The *message window*, used to display relevant data. The current choices of parameters and initial condition are shown, as well as the "Energy Error", defined as the difference between the selected energy value and the current (calculated) energy value. This window can be right clicked to access a menu. The menu allows the user to manually enter an initial condition in the console (not shown), as well as switch Poincaré section type, change the orbit colour, and other options.

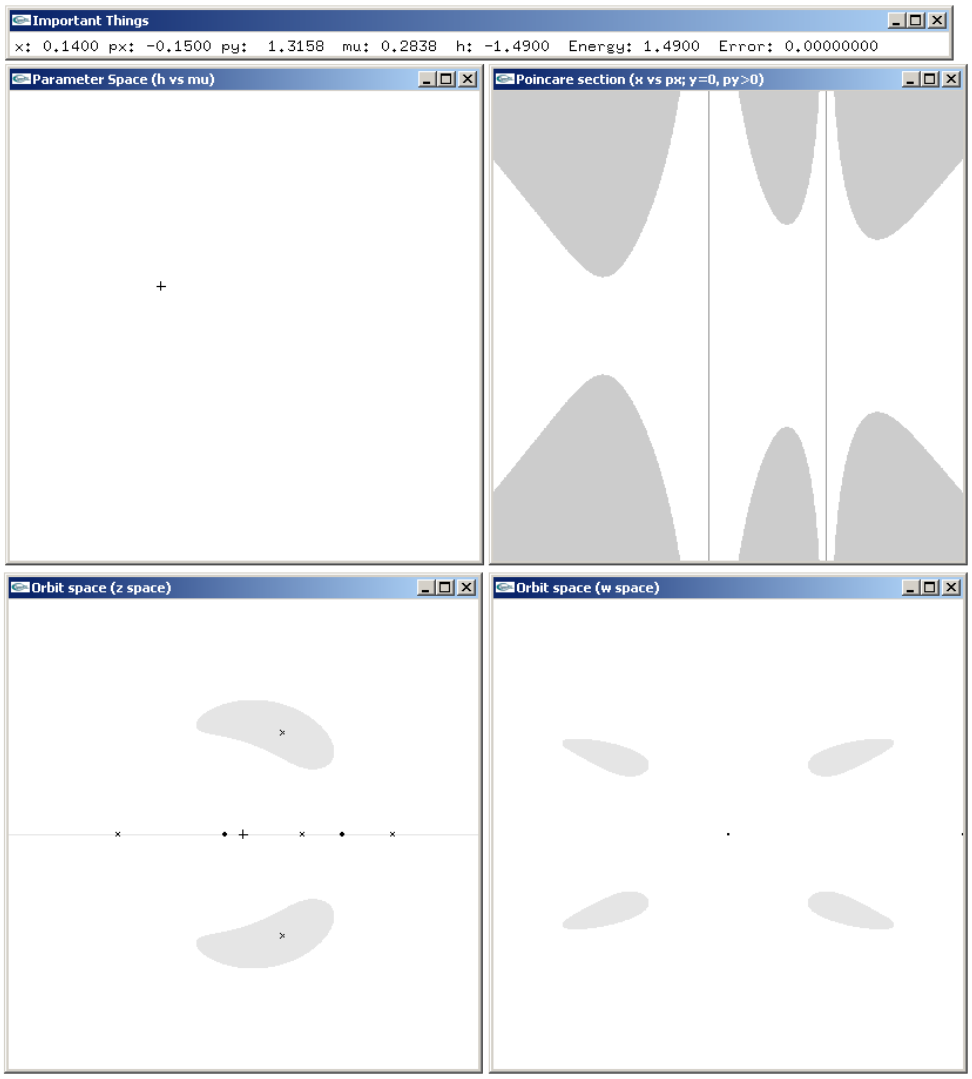
- The *parameter window*. This window is used to select the values of the E and μ parameters, with horizontal position determining μ and vertical position determining E.
- The *Poincaré section window*. Clicking in this window allows the user to select an initial condition for an orbit, which will immediately begin integration. Also, whenever an orbit passes through the section, the point is drawn on the section, visually showing the Poincaré map defined in Section 3.4. The grey regions delineate the Hill region, and the grey lines mark the positions of the primaries.
- The *z-space orbit window*, which shows the motion of the particle in z coordinates. One orbit is shown at a time, and will automatically be displayed as the trajectory is calculated. Also marked are the locations of the primaries, the centre of mass, and the five Lagrangians. The Hill region is delineated with a grey regions. A light grey line shows the currently selection Poincaré section. This window can be hidden to improve performance.
- The *w-space orbit window*, which shows the motion of the particle under the Thiele-Burrau coordinates. This is mostly provided for reference and testing. Again, it can be hidden to improve performance.

5.2. Analysing Orbits

5.2.1. Interpreting the Poincaré Section. The general procedure followed when using the program is to select the parameter, then select a variety of initial conditions in the section to build up an image of the Poincaré map. These diagrams give a visual representation of the qualitative differences between different orbits. Figure 5.3 shows this idea for the $y = 0$ Poincaré section, showing the initial conditions which will lead to Hill orbits (trajectories that orbit one of the primaries). The Hill orbits show up as closed curves (as indeed will any quasiperiodic orbits) while chaotic orbits appropriately are chaotic under the Poincaré map. Figure 5.4 shows a map for the section through L_4. This allows comparisons to be made between the nature of the trajectories. Figure 5.5 shows another way of using the program: by studying the sections, "interesting" orbits can be found. Orbits that follow more complex paths than the circles associated with Hill orbits can be found quite naturally, and "minimised" in the sense that periodic orbits can be found near quasiperiodic orbits.

5.2.2. The Vanishing Twist. As noted in the introduction, a goal of this project is to demonstrate a particular bifurcation in the system called the *vanishing twist*. This follows from the theory developed in *Generic twistless bifurcations* by Dullin, Meiss and Sterling [**12**].

In the paper, Dullin et al. proved that in the neighbourhood of the tripling bifurcation of a fixed point of an area-preserving map, there is what they call a *twistless*


Important Things
Parameter Space (h vs mu)
Poincare section (x vs px; y=0, py>0)
Orbit space (z space)
Orbit space (w space)

```
x: 0.1400 px: -0.1500 py:  1.3158  mu: 0.2838  h: -1.4900  Energy: 1.4900  Error: 0.00000000
```

Figure 5.2. The program's interface. At top: message window. Top left: parameter window. Top right: Poincaré window. Bottom left: orbit window in z-space. Bottom right: orbit window in w-space.

bifurcation. We can access this idea for our system by defining the *rotation number* of our Poincaré map.

Definition 5.2 (Rotation number)**.** *For a map $f : [0, 2\pi) \to [0, 2\pi)$, the rotation number of f is defined as*

$$\Omega(f) = \lim_{n\to\infty} \frac{f^n(x) - x}{n}. \tag{5.3}$$

In the paper, bifurcations occur at points corresponding to special values of this rotation number. In particular, the twistless bifurcation occurs when the derivative

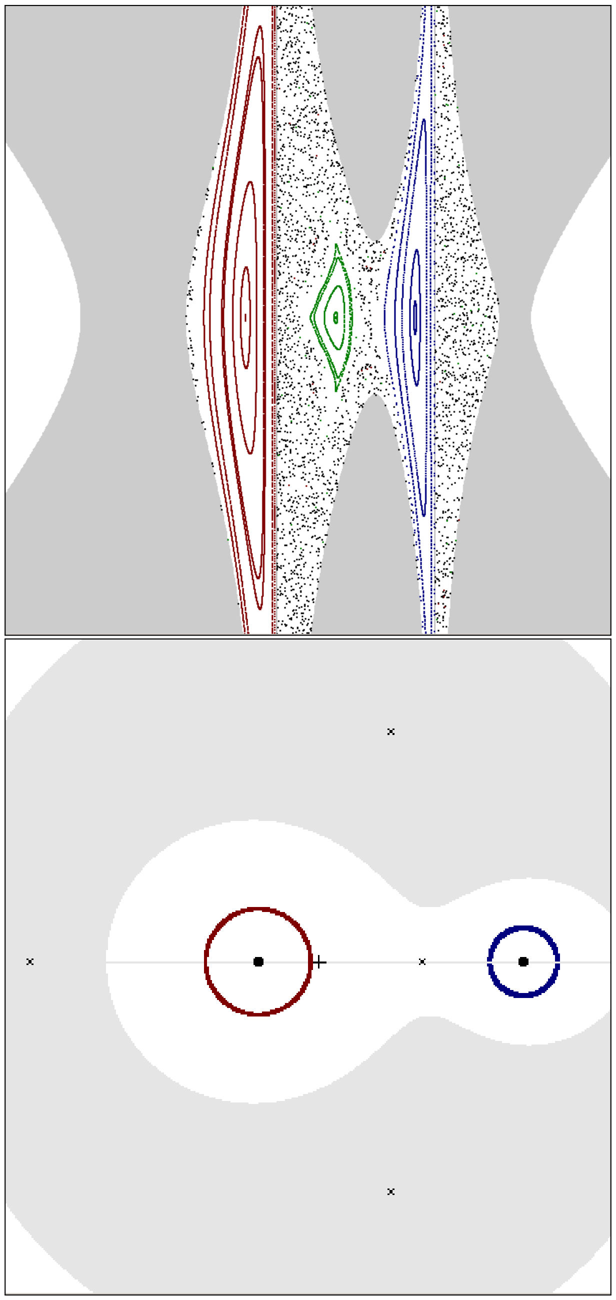


Figure 5.3. Using the Poincaré section to differentiate types of orbits around the primaries (Hill orbits). These figures use the Poincaré section defined by $y = 0$, through both primaries. The top figure shows the Poincaré map. The red section corresponds to clockwise orbits around the left primary. The green section corresponds to anti-clockwise orbits around the left primary. The blue section corresponds to clockwise orbits around the right primary. The black dots are regions where the map is chaotic, and will (after a long computation time) eventually fill out this space. The bottom figure shows particular orbits for the red and blue sections in the z-coordinate space.

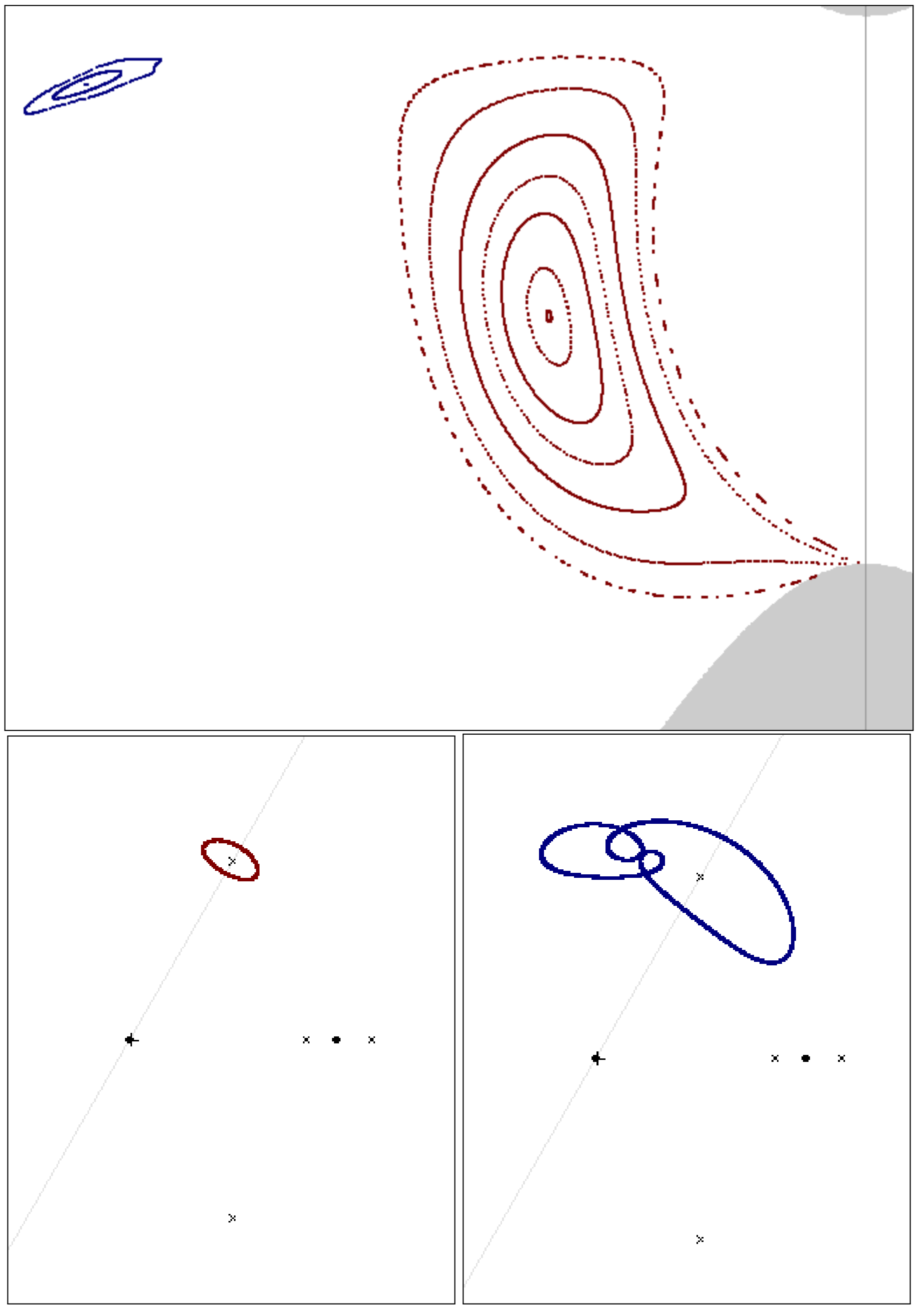


Figure 5.4. Using the Poincaré section to differentiate types of orbits around L_4. As indicated by the grey lines in the lower figures, these figures use the Poincaré section that passes through L_4 and the heavier of the primaries. The top figure shows the Poincaré map for $\mu = 0.0122$, $E = -1.4937$. The bottom figures show two orbits periodic corresponding to the points at the centre of the red and blue sections of the section respectively. The qualitative difference between the two types of orbits can be confirmed visually: in the blue orbit the particle's trajectory has extra "loops".

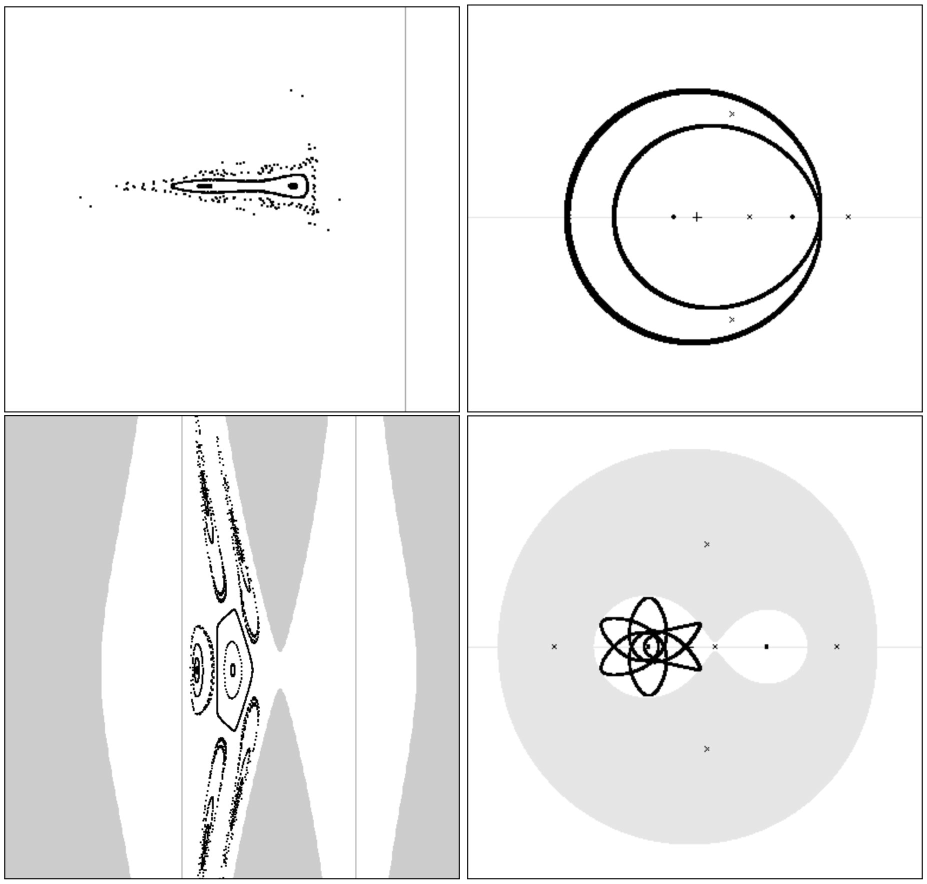


Figure 5.5. At left: Poincaré sections for two choices of parameters. At right: orbits corresponding to regions in these sections. For the top images, $\mu = 0.20$, $h = 0.57$. For the bottom images, $\mu = 0.32$, $h = -1.95$.

of the rotation number as a function of the action vanishes. The structures around this bifurcation are shown in Figure 5.6.

The goal is then to demonstrate these behaviours in the PCR3BP. The equilibrium at L_4 is a suitable canditate for this structure, as the Poincaré map has a fixed point. A fixed energy is considered, close (but not too close) to the critical value of the energy at L_4 (which, as found in Equation (3.29) is $E = -1.5 + \mu(1 - \mu)$).

The Poincaré map (as a function of energy) can, for quasiperiodic orbits, be defined on a curve isomorphic to the circle. Then the rotation number can be defined as in 5.2 as a function of the action. At particular values of this, we observe bifurcations. In particular, rational rotation numbers lead to *resonance* resulting in a bifurcation. A formula for the mass ratios where this occurs is a known result. *Introduction to Hamiltonian Dynamical Systems and the N-Body Problem* [**24**]

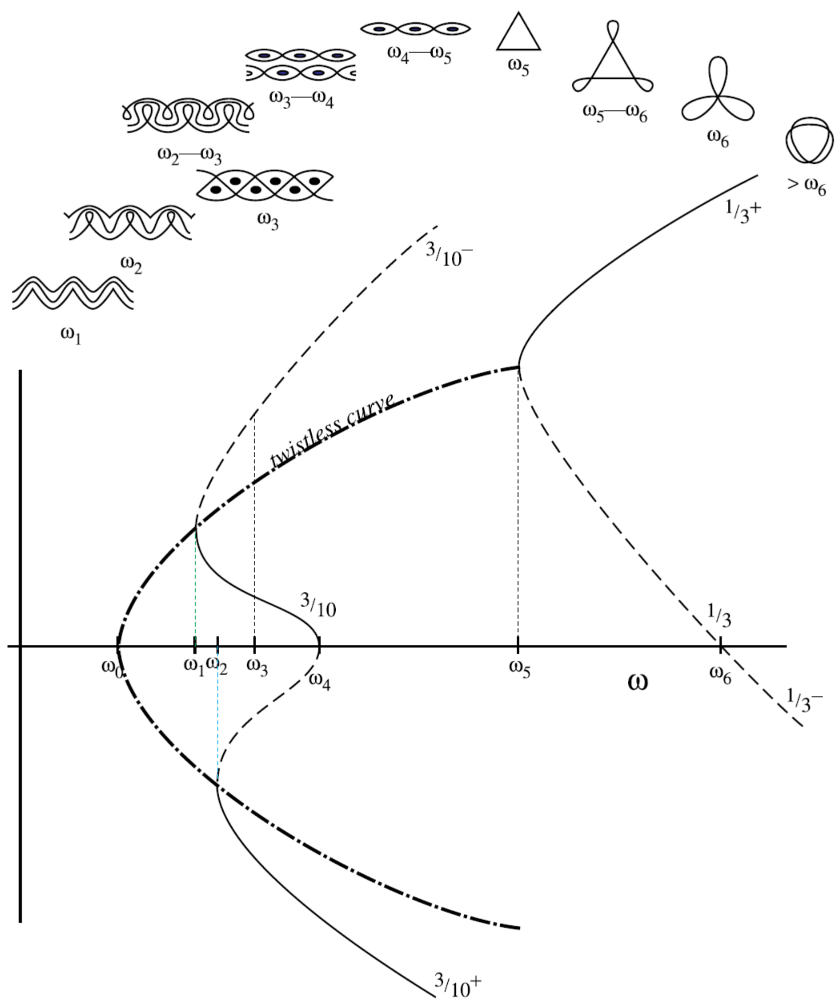


Figure 5.6. The bifurcations near the twistless bifurcation, and representative phase portraits. Picture credit: *Generic twistless bifurcations* [**12**].

gives the formula

$$\mu_r = \frac{1}{2}(1 - \sqrt{1 - \frac{16r^2}{27(r^2+1)^2}}) \tag{5.4}$$

where μ_r is the value of μ where the $\frac{1}{r}$ resonance occurs (where the rotation number is $\frac{1}{3}$). This formula is only valid for $E = -1.5 + \mu(1 - \mu)$ (the critical energy value found in (3.29)), but gives a good estimate for close energy values.

We can then consider the rotation number of the map as a function of the action $\Omega(J)$. Then for periodic orbits bifurcations occur at *resonant* values of the rotation number. In particular,

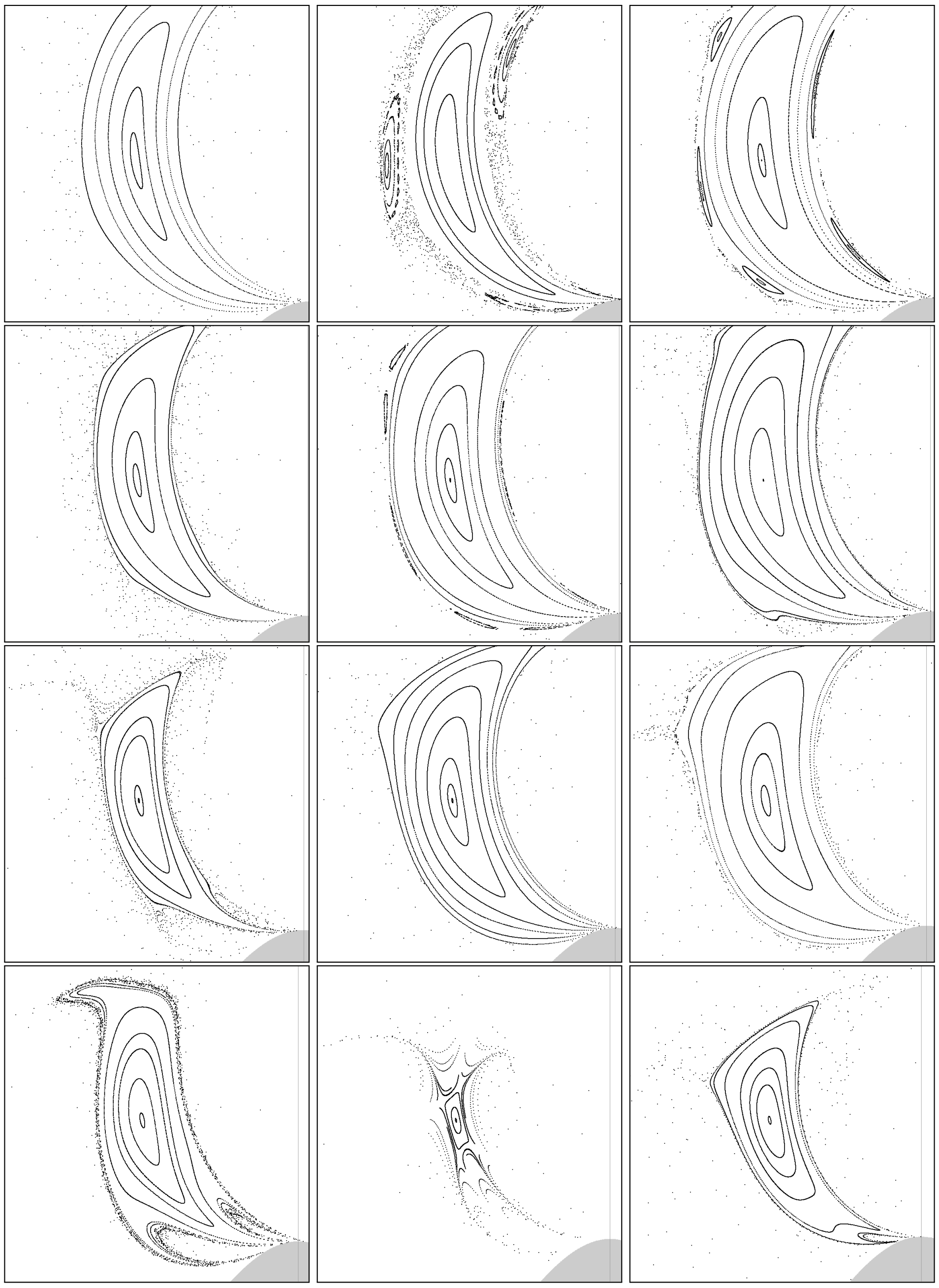


Figure 5.7. Bifurcation analysis for the Poincaré map near L_4. For these plots, $E = -1.49$.
Top row, left to right: $\mu = 0.0025,\ 0.0030,\ 0.0035$.
Second row: $\mu = 0.0040,\ 0.0045,\ 0.0050$.
Third row: $\mu = 0.0055,\ 0.0060,\ 0.0065$.
Bottom row: $\mu = 0.0075,\ 0.0080,\ 0.0085$.
This continued in Figure 5.8. The bifurcations described by Figure 5.6 can be seen. In particular, note the bifurcation at $\mu = 0.008$, and the complex structure at $\mu = 0.003$.

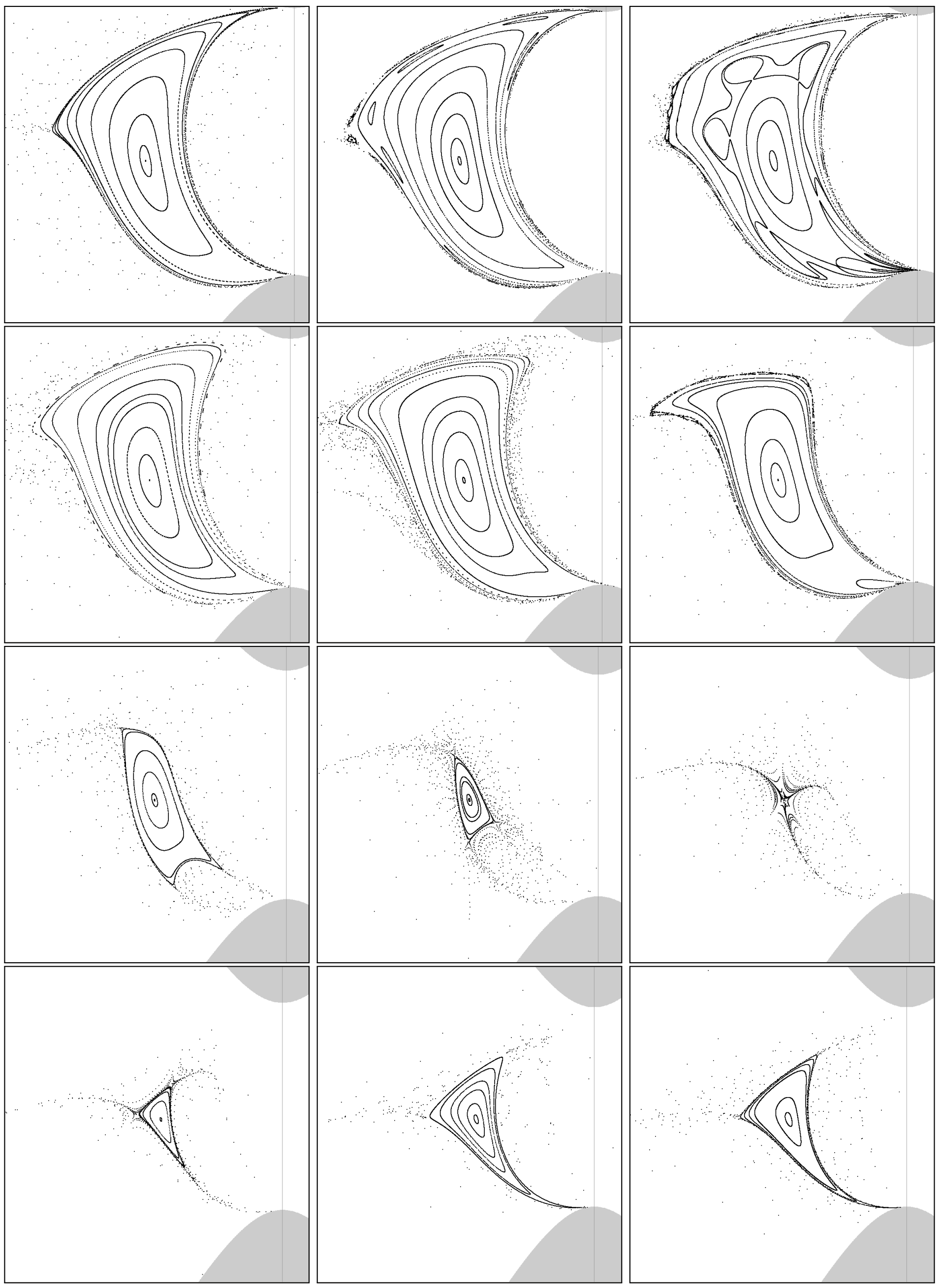


Figure 5.8. Bifurcation analysis for the Poincaré map near L_4. Continued from Figure 5.7
Top row, left to right: $\mu = 0.0090,\ 0.0095,\ 0.0100.$
Second row: $\mu = 0.0105,\ 0.0110,\ 0.0115.$
Third row: $\mu = 0.0120,\ 0.0125,\ 0.0130.$
Bottom row: $\mu = 0.0135,\ 0.0140,\ 0.0145.$
In particular note the bifurcation at $\mu = 0.0130$ corresponding to the $1/3$ bifurcation in Figure 5.6, and the structure at $\mu = 0.0100$ which is detailed in Figure 5.9

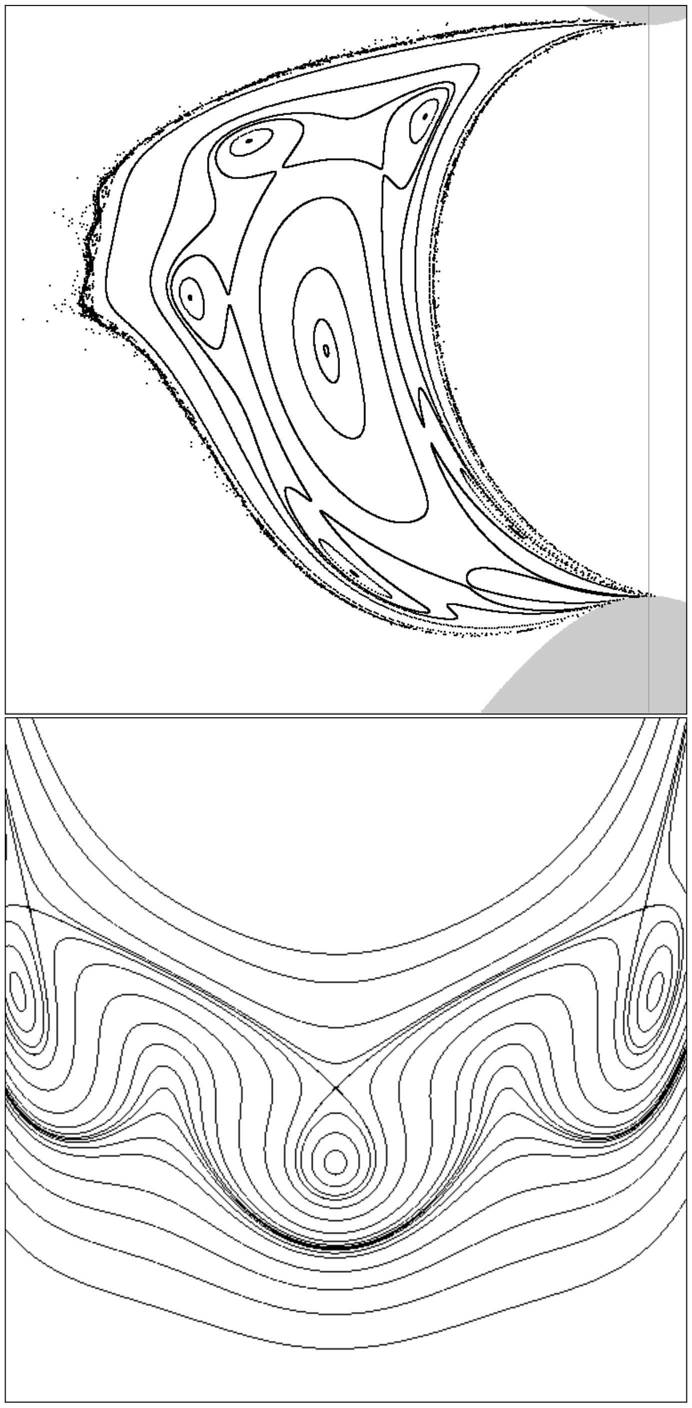
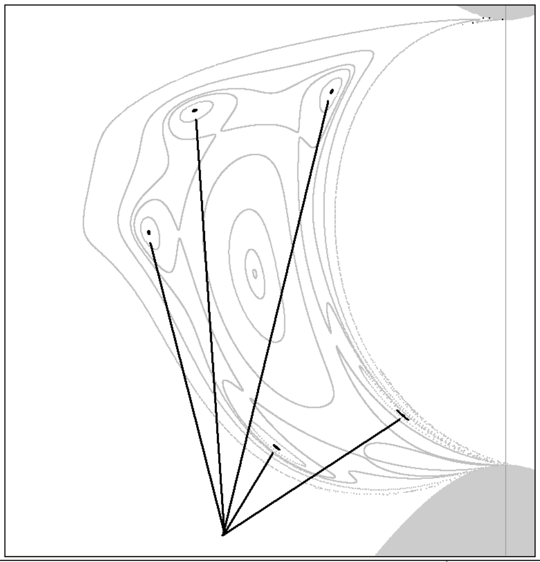


Figure 5.9. Top: the Poincaré map for the Poincaré section through L_4, for the parameter values $\mu = 0.0100$, $E = -1.49$. Bottom: the structure predicted near the vanishing twist bifurcation by [**12**]. Also visible is the additional structure in the top left of the top diagram; this small island chain is a remnant of a resonant rotation number. Picture credit for lower figure: *Generic twistless bifurcations* [**12**].



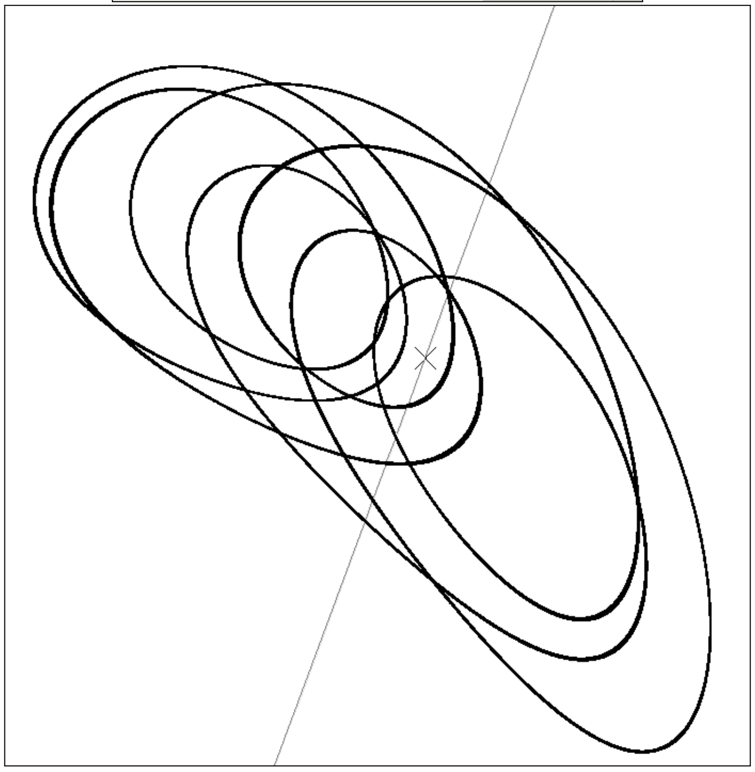


Figure 5.10. Top figure: Figure 5.9, with a particular set highlighted which is invariant under the Poincaré map . Bottom figure: The orbit around L_4 corresponding to this invariant set. Recall that points on the Poincaré section are generated when the orbit crosses the grey line where the section is defined.

- If $\Omega(0)$ is rational the orbit is resonant, and a chain of "islands" form. See $\mu = 0.0035$ and $\mu = 0.0045$, $\mu = 0.0070$ and $\mu = 0.0095$ in Figures 5.7 and 5.8 for examples of this.
- If $\Omega(J)$ is rational with $\Omega(J) = \frac{p}{q}$ for $p, q \in \mathbb{N}$, we get chains of islands with q smaller islands in the big island. $\mu = 0.0030$ in Figure 5.7 corresponds to this with $\Omega(J) = 3/10$. However, there appear to be more than 10 smaller islands, though is most likely a quirk of the way the Poincaré map has been constructed or calculated.
- A circle where $\Omega'(J) = 0$ is called twistless.

Based on the diagrams from the paper by Dullin et al., we can identify the ranges where our system undergoes these bifurcations, by comparing Figures 5.7 and 5.8 and the phase portraits in Figure 5.6. The figures show the evolution of the structure of the Poincaré map for fixed energy and μ being varied between $\mu = 0.0025$ and $\mu = 0.0145$[1].

Of particular note are the bifurcations which occur near $\mu = 0.0080$ and $\mu = 0.0130$, and the structure near the twistless bifurcation at $\mu = 0.0100$. This structure is shown in more detail in Figure 5.9. A striking orbit in this structure is shown in Figure 5.10.

There are some key differences between the PCR3BP and the system considered by Dullin et al, most notably the second parameter. However, we can still observe that the bifurcations match the theoretical predictions very closely.

5.2.3. The Earth-Moon system. One final analysis is to study the structure of the Poincaré map when the energy parameter is altered, rather than the mass parameter. The results of this are shown in Figures 5.11 and 5.12. Of note is the choice of mass parameter for this analysis; we take $\mu = 0.01215$. This is the mass parameter for the Earth-Moon system. The moon's orbit around the Earth has a low eccentricity of about 0.05 (from [**25**]), so the "circular" part of our approximation is reasonable. As of October 2012, no trojans have been discovered in the Earth-Moon system. This does not mean they do not exist; it was only in July 2011 that the first trojans in the Sun-Earth system was found, an asteroid called TK$_7$ (see [**26**]). The plots show possible stable orbits around L_4 for $-1.4855 < E < -1.4975$ (although the possibility of stable orbits for other energy values cannot be discounted).

[1]Recall that orbits around L_4 are only stable when $\mu < 0.0385$, so only a relatively small range of the parameter is being considered.

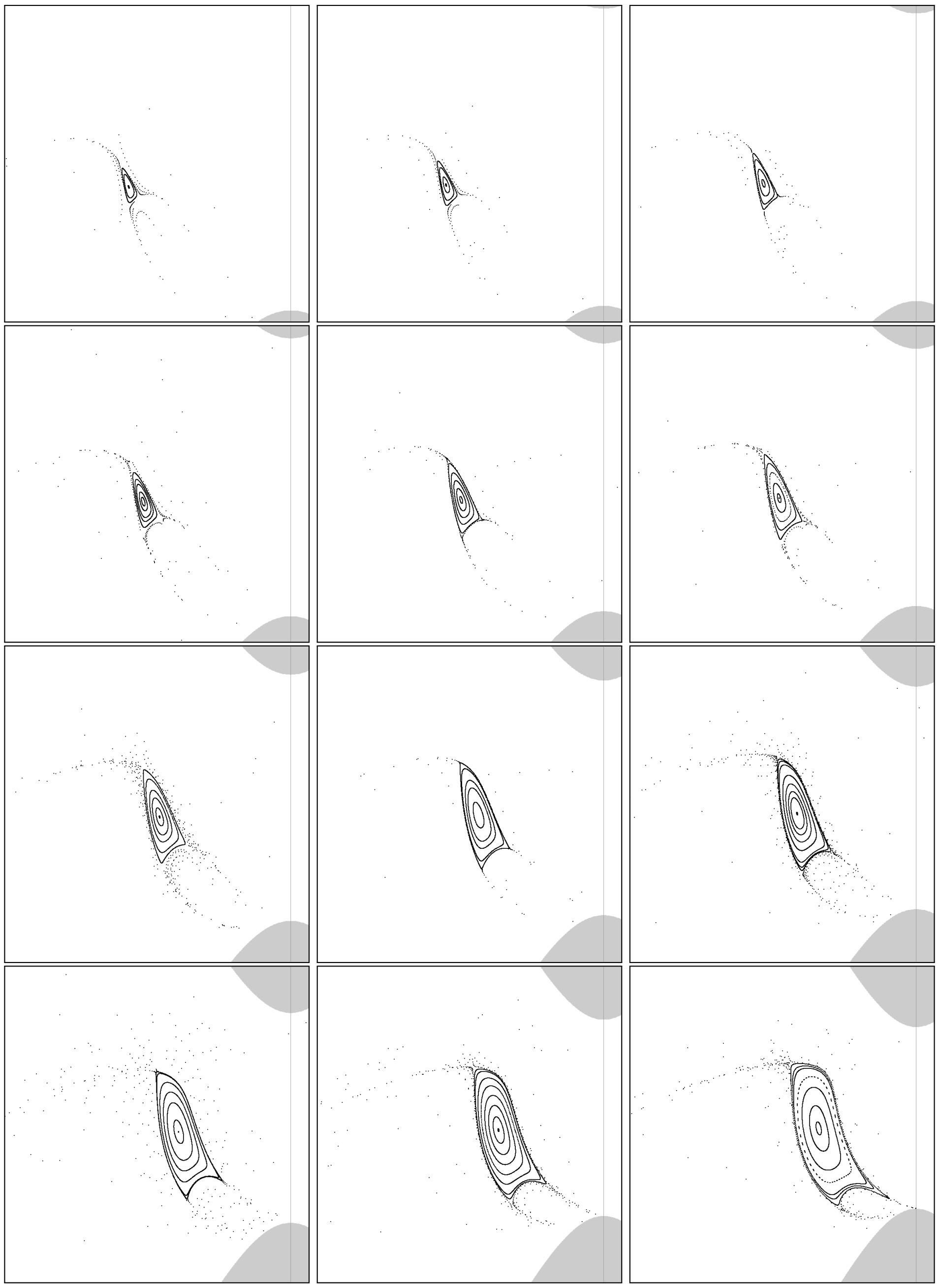


Figure 5.11. Analysing the Poincaré section for the Earth-Moon system near L_4. For the Earth-Moon system, $\mu = 0.01215$. The above images are the result of adjusting the Energy parameter.
Top row, left to right: $E = -1.4855$, -1.4860, -1.4865.
Second row: $E = -1.4870$, -1.4875, -1.4880.
Third row: $E = -1.4885$, -1.4890, -1.4895.
Bottom row: $E = -1.4900$, -1.4905, -1.4910.

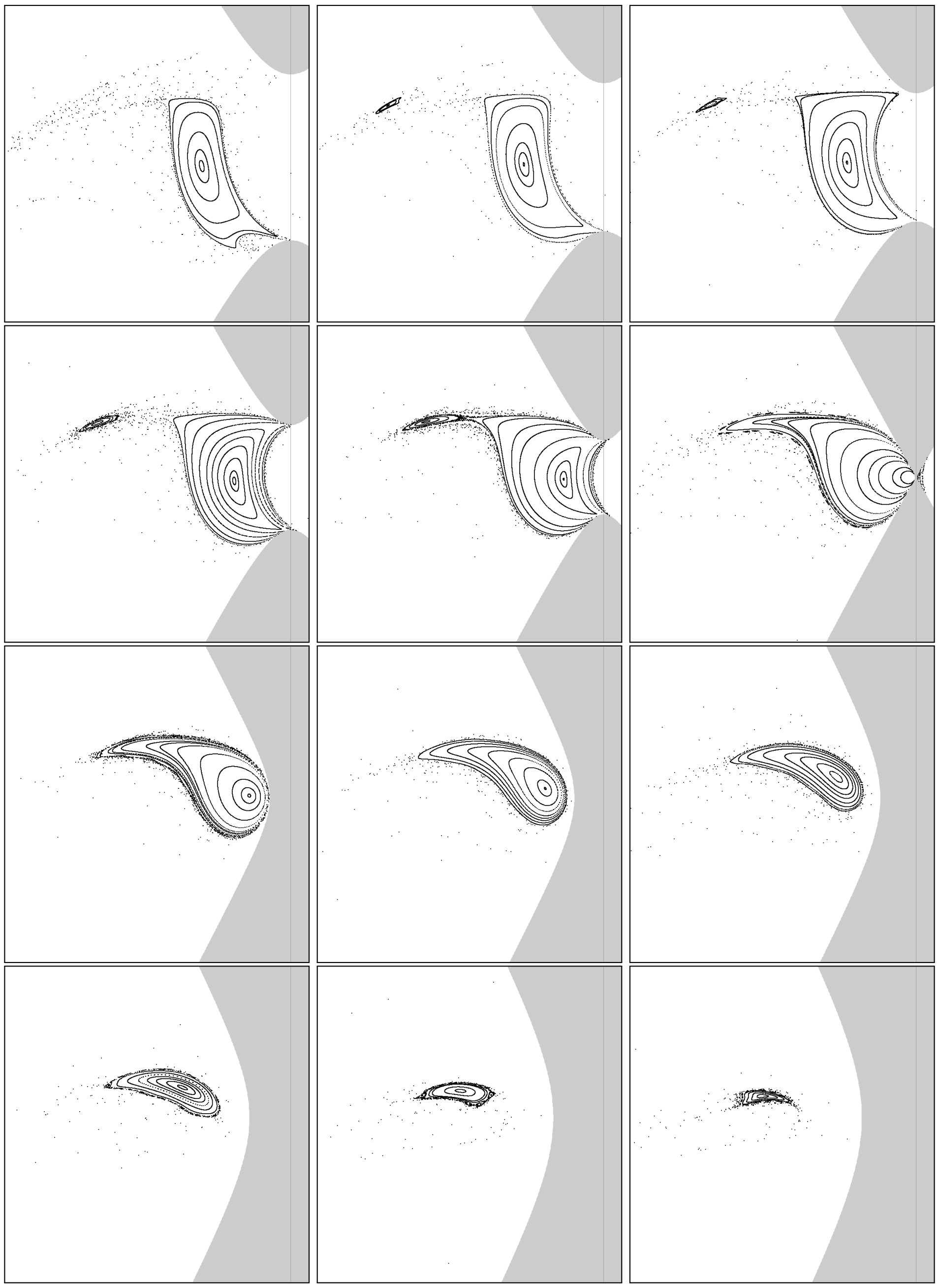


Figure 5.12. The Poincaré section for the Earth-Moon system near L_4.
Continued from Figure 5.11.
Top row, left to right: $E = -1.4915$, -1.4920, -1.4925.
Second row: $E = -1.4930$, -1.4935, -1.4940.
Third row: $E = -1.4945$, -1.4950, -1.4955.
Bottom row: $E = -1.4960$, -1.4970, -1.4975.

CHAPTER 6

Conclusion

Although the PCR3BP is a significant simplification of the general three body problem, it is a rich problem with much unexplored depth. Our analysis has hopefully given the reader an appreciation of the key concerns of the system, and provided a robust derivation of the equations of motion in a Hamiltonian context. The equations derived provide a good basis for any analysis of the PCR3BP, and many of the ideas can be translated to similar problems.

The integrator developed in Chapter 5 provides a powerful tool for a hands-on numerical study of the problem. The code provided could easily be expanded to incorporate other ideas; for instance, the Poincaré map integrator describe in Section 4.7 could be implemented with only minor additions. Much of the code base, including the graphical routines, could be reused in future projects.

The primary mathematical success of the project is the verification of the twistless bifurcation for this system. The motion of objects near the L_4 and L_5 Lagrangian points can be very complex. The discovery of the bifurcations corresponding to those derived for a general system in [**12**] gives a mathematical basis for classifying the types of orbits that can occur around these points. Of particular note is the spectacular motion observed close to the twistless bifurcation, illustrated in Figure 5.10.

We also constructed a similar bifurcation diagram for the Earth-Moon system in Section 5.2.3. Although this does not exhibit the same bifurcation structure as adjusting the mass parameter, it exhibits some very interesting behaviour, which could be later examined analytically. Such work could prove useful in searching for trojans in the Earth-Moon system.

The work of this project raises several possible avenues for future investigation. The more obvious additions include the implementation of Henon's method for calculating Poincaré maps, as mentioned above, and functions to automate the process of building bifurcation plots like Figure 5.7. Much of the structure of the program could be reused in a study of the PR3BP (without restricting the primaries to circular orbits), or perhaps even the R3BP (in three dimensional configuration space). A deeper analytical study could be made of the motion around L_4 and L_5, following from the results in [**24**]. This would be motivated by the numerical results and structures yielded from our program. In particular, the rotation number could be explicitly calculated, which could be used to narrow the estimates for the parameter values at the observed bifurcations.

References

[1] I. Newton, Philosophiæ Naturalis Principia Mathematica [1687]. Retrieved from University of California Digital Library

[2] M. Hénon. Generating Families in the Restricted Three-Body Problem [Springer, 1997]

[3] M. Hénon. Generating Families in the Restricted Three-Body Problem: II Quantitative Study of Bifurcations [Springer, 2001]

[4] A. Katko and B. Hasselblatt. Introduction to the Modern Theory of Dynamical Systems [Cambridge University Press, 1995]

[5] J. Barrow-Green. Poincaré and the Three Body Problem [The American Mathematical Society, 1997]

[6] L. Poladian. Lagrangian and Hamiltonian Dynamics *The University of Sydney School of Mathematics & Statistics* [2011]

[7] Q. Wang. On the Homoclinic Tangles of Henri Poincaré [University of Arizona, 2010]

[8] Q. Wang The Global Solution of N-body Problem. *Celestial Mechanics and Dynamical Astronomy* Vol 50 [1991] p. 73-88

[9] D. Ó'Mathúna. Integrable Systems in Celestial Mechanics *Birkhauser* [2008]

[10] M. Valtonen and H. Karttunen. The Three-Body Problem. *Cambridge University Press* [2006]

[11] R. Montgomery. A New Solution to the Three-Body Problem *Notices of the AMS* Volume 48, Number 5 [2001]

[12] H. R. Dullin, J. D. Meiss and D. Sterling. Generic twistless bifurcations. *Nonlinearity* 13 [2000] p. 203-224

[13] R. Montgomery. N-body choreographies *Scholarpedia* (2010) 5(11):10666, revision #91542

[14] C. Simó. New families of Solutions in N–Body Problems. *Progress in Mathematics, Vol 201* [2001], p. 101–115

[15] L. A. Pars. A Treatise on Analytical Dynamics. *Ox Bow Pr* [1981]

[16] V. Szebehely. Theory of Orbits: The Restricted Problem of Three Bodies *Academic Press* [1967]

[17] E. Hairer, C. Lubich and G. Wanner Geometric Numerical Integration: Structure-Preserving Algorithms for Ordinary Differential Equations. 2nd edition. *Springer* [2006] p. 179-236

[18] E. Hairer Variable time step integration with symplectic methods *Applied Numerical Mathematics* 25 [1997] p. 219-227

[19] R. Broucke. Regularizations of the Plane Restricted Three-Body Problem. *Icarus 4* [1965] pp. 8-18

[20] M. Henón. On the Numerical Computation of Poincaré Maps *Physica D* 5 p. 412-414

[21] W. Tucker, Computing Accurate Poincaré maps [2002] *Physica D* 171 p. 127-137

[22] W. S. Koon, M. W. Lo, J. E. Marsden and S. D. Ross, Dynamical Systems, the Three-Body Problem and Space Mission Design [2008]

[23] J.D. Mireles James. Celestial Mechanics Notes, Set 4: The Circular Restricted Three Body Problem *University of Texas* [2006]

[24] K. R. Meyer, G. R. Hall. Introduction to Hamiltonian Dynamical Systems and the N-Body Problem, *Springer* [1992]

[25] Earth's Moon: Facts & Figures from the NASA Website. Accessed from *http://solarsystem.nasa.gov/planets/profile.cfm?Object=Moon&Display=Facts*

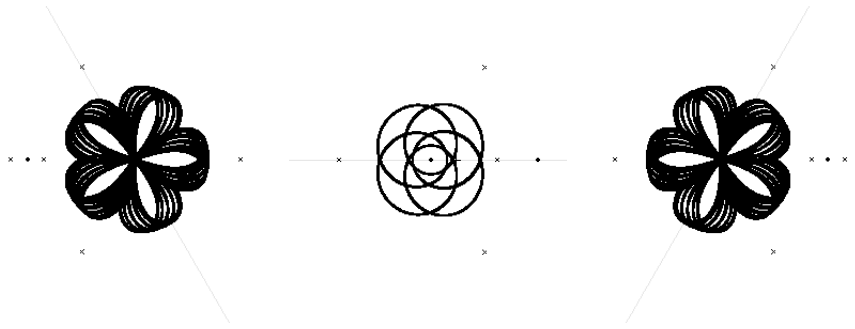
[26] NASA's WISE Mission Finds First Trojan Asteroid Sharing Earth's Orbit from the NASA Website. Accessed from *http://www.nasa.gov/mission_pages/WISE/news/wise20110727.html*

APPENDIX A

Acknowledgements

The author would like to thank Holger Dullin, whose expertise and enthusiasm have made this project not only possible but extremely gratifying.

Thanks must also go to William Rowley, whose advice on C++ was invaluable, and Ting-Ying Chang, whose input has been exceedingly propitious.



APPENDIX B

Code

```
/*
  THREE BODY PROBLEM
  JOACHIM WORTHINGTON
  2012
  jwor6721@uni.sydney.edu.au

  Some functions, in particular those pertaining to unused
      transformations, are unfinished/unused in the final code.
  They are left in for reference and possible future use.

*/


#include <stdio.h>
#include <tchar.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <GL/glut.h>
#include <GL/gl.h>
#include <iostream>
#include <string>
#include <sstream>
#include <complex>

#include "stdafx.h" //To be removed when compiling outside VC++

std::stringstream sstm;
using namespace std;

/* ULTRATIGHT: For finding structure near the vanishing twist
#define MAX_MU 0.014    //Maximum value of mu parameter; minimum
    is 0
#define MAX_ENERGY 1.495  //Really the minimum energy; .energy
    between 0 and -this
#define MIN_ENERGY 1.485  //Again, really the maximum energy;
    consider -this
#define MAX_PX 0.1      //Maximum selection of p_x in the poincare
     window
```

```
#define MIN_PX -0.15      //Minumum selection of p_x in the
    poincare window
#define MAX_X 0.50      //Maximum selection of x in the poincare
    window
#define MIN_X 0.38      //Minumum selection of x in the poincare
    window*/


/* L4: For analysing orbits around L4
#define MAX_MU 0.04     //Maximum value of mu parameter; minimum
    is 0.
#define MAX_ENERGY 1.5    //Really the minimum energy; .energy
    between 0 and -this
#define MIN_ENERGY 1.47   //Again, really the maximum energy;
    consider -this
#define MAX_PX 0.1      //Maximum selection of p_x in the poincare
     window
#define MIN_PX -0.15      //Minumum selection of p_x in the
    poincare window
#define MAX_X 0.50      //Maximum selection of x in the poincare
    window
#define MIN_X 0.35      //Minumum selection of x in the poincare
    window*/


/*FOR PRIMARY ORBITS: for orbits that orbit one or both primaries
    */
#define MAX_MU 0.5      //Maximum value of mu parameter; minimum
    is 0
#define MAX_ENERGY 5.0    //Really the minimum energy; .energy
    between 0 and -this
#define MIN_ENERGY -1.0   //Again, really the maximum energy;
    consider -this
#define MAX_PX 2.0      //Maximum selection of p_x in the poincare
     window
#define MIN_PX -2.0     //Minumum selection of p_x in the poincare
     window
#define MAX_X 2.0     //Maximum selection of x in the poincare
    window
#define MIN_X -2.0      //Minumum selection of x in the poincare
    window

#define SQRT3ON2 0.8660254038
#define SQRT3 1.732050808
#define PI 3.14159265   //Constant relating the circumference of a
     circle to its diameter
#define NEWTON_STEPS 20   //Number of steps to take in calculating
     L1,2,3
```

```
#define TOO_CLOSE 0.1   //For the switching integrator
#define TOO_FAR 5     //If the particle has flown away

#define CROSS_SIZE 0.02   //Size of the Cross Marker
#define PARAM_X_RES 0.01  //Keyboard right increases mu by
#define PARAM_Y_RES 0.01  //Pressing up increases h by this
#define PLANET_RADIUS 0.02  //Size the Larges Bodies are drawn at
#define HILL_DETAIL 1   //Higher settings render the Hill Region
    more carefully, slowly
#define HILL_RES 1000   //The resolution at which to draw the Hill
     Region in the P-window

//Values for the Menu
#define INTEGRATE 0
#define HILL_ON 1
#define RED 2
#define GREEN 3
#define BLUE 4
#define BLACK 5
#define POIN_ON 6
#define ZOOM_IN 7
#define ZOOM_OUT 8
#define ENTER_POIN 9
#define ENTER_PARAM 10
#define ORBIT_ON 11
#define ORBIT_ON_W 12
#define SWITCH_POIN 13
#define EACH_NEW 14

double dt=0.01;       //0.01 is a reliably correct, 0.05 is
    usually okay

//Varibles to store window data
int paramWindow;
int paramWindowHeight;
int paramWindowWidth;
int orbitWindow;
int orbitWindowHeight;
int orbitWindowWidth;
int orbitWindowW;
int orbitWindowWHeight;
int orbitWindowWWidth;
int poinWindow;
int poinWindowHeight;
int poinWindowWidth;
int UIWindow;
int UIWindowHeight;
int UIWindowWidth;
int menu;


double scaleOut=0.5;    //Scale for the orbit window


bool orbitWindowOn=true;  //Hide the z-space Orbit Window to
    achieve better performance
bool orbitWindowOnW=true; //Hide the W-space Orbit Window to
    achieve better performance
bool hillRegionOn=true;   //Turn on Hill Region plotting
bool velocityCurveOn=false;   //Turn on Velocity Curve plotting (
    system intensive)
bool poinRegionOn=true;   //Turn on Selectable Poincare Region
bool sectionTwo=true; //Use the Poincare section through the
    Trojan and the Larger Body
bool sectionThree=false;  //Use the Poincare section through the
    Trojan and the Smaller Body
bool clearOrbits=true;  //Clear each orbit upon picking a new
    Poincare section
bool fastMode=false;  //Turns off double buffering

bool integrateOn=false;   //While true, integrate the orbit
bool newOrbit=true;   //If we press space, then press it again,
    continue last orbit
int intSteps=100;   //Number of steps to integrate before checking
     the idle stuff again
int colour=3;     //Number corresponding to the colour to use 0 is
     red, 1 is blue, 2 is green

double muP=0.0;   //Mass Parameter
double hP=-1;   //Energy Parameter;

double poinX; //Initial condition
double poinPX;  //Initial condition
double poinPY;  //Initial condition (determined by x,p_x,E and mu)
double coords[4]; //Stores (u,v,p_u,p_v)

void renderBitmapString(double x, double y, void *font, char *
    string); //Writes a string to the screen

//GLUT functions for the various windows
void renderSceneParam(void);
void reshapeParam (int w, int h);
void mouseParam(int button, int state, int x, int y);
void sKeyboardParam(int key, int x, int y);
void reshapeOrbit (int w, int h);
void renderSceneOrbit(void);
void redrawOrbit(void);
void mouseOrbit(int button, int state, int x, int y);
void reshapeOrbitW (int w, int h);
void renderSceneOrbitW(void);
void redrawOrbitW(void);
void mouseOrbitW(int button, int state, int x, int y);
void reshapePoin (int w, int h);
void renderScenePoin(void);
void mousePoin(int button, int state, int x, int y);
void sKeyboardPoin(int key, int x, int y);
void reshapeUI (int w, int h);
void renderSceneUI(void);

void idle();  //GLUT idle loop
void createGLUTMenus();
void processMenuEvents(int option);

double H_w(double* c);  //Calculate energy in terms of w, pw in T-
    B space
double Energy(double x, double y, double px, double py);  //Energy
     in terms of z, pz
double K_E(double* c);  //The phase space Hamiltonian (T-B space)
double initPY(double x, double px); //Calculate the initial
    condition
double uHat(double x, double y);  //The effective potential energy
double T(double x, double y, double px, double py);   //Effective
    kinetic energy
bool hillRegion(double x, double y);  //Checks if points are in
    the Hill region
void verifyParameters();  //Checks that mu and h are sensible
double *delT(double x, double y, double px, double py);   //
    Kinetic derivative
void tStep(double*, double step); //Kinetic step in untransformed
    coordinates
double *delUhat(double*); //Effective Potential derivative
void uHatStep(double*, double step);  //Effective Potential step
    in untransformed coordinates
void integrationStep(); //Generic "total integration step"
    function

void zToW_lc1(double*); //Local transformation at left singularity
void wToZ_lc1(double*); //Inverse local transformation left
    singularity

void zToW_tb(double*);  //Thiele Barrau Transform
void wToZ_tb(double*);  //Inverse Theiele Barrau Transform
void uStep_tb(double*, double step);  //Potential step under T-B
void tStep_tb(double*, double step);  //Kinetic step under T-B
void f_tb(double *c, double *result);
  //Derivatives for the Thiele-Barrau transformed Hamiltonian

double *delUhat_lc1(double*); //Effective Potential derivative for
     left singularity local transform
void uHatStep_lc1(double*, double step);  //EP step for left
    singularity local transform
double *delThat_lc1(double*); //Kinetic derivative for left
    singularity local transform
void tHatStep_lc1(double*, double step);  //Kinetic step for left
    singularity local transform

void birkhoff(double*);     //Birkhoff transformation
double birkhoff_inv(double*); //Inverse Birkhoff transformation
double* delUhat_birkhoff(double*);  //Effective Potential
    derivative for Birkhoff transform
void uHatStep_birkhoff(double*, double step); //EP step for
    Birkhoff transform

void runge_kutta_4(double* y0, void (*derivs)(double*,double*),
    double step);
  //Runge-Kutta method for four-dimensional time-independent
      function

double L1(); //Calculate the Lagrangian points
double L2();
double L3();
double Lagrangian_f(double x, int p); //For calculating the
    Lagrangian points
double Lagrangian_fdash(double x, int p);


void renderBitmapString(double x, double y, void *font, char *
    string) {
  //Prints a string on the screen at x,y
  char *c;
  glRasterPos2f(x, y);
  for (c=string; *c != '\0'; c++) {
    glutBitmapCharacter(font, *c);
  }
}

//PARAMETER WINDOW
void renderSceneParam(void) {
  //Render the Parameter window

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glClearColor(1.0, 1.0, 1.0, 1.0);
  //Draw a cross at the Selected Parameter Point
  int pSpaceW=paramWindowWidth;
  double ratioW=MAX_MU/pSpaceW;
  double crossX=(muP/ratioW);
  crossX=2*crossX/paramWindowWidth-1;

  int pSpaceH=paramWindowHeight;
  double ratioH=(MAX_ENERGY-MIN_ENERGY)/pSpaceH;
  double crossY=(-1*(hP+MIN_ENERGY)/ratioH);
  crossY=-1*(2*crossY/paramWindowHeight-1);

  glBegin( GL_LINES );
    glColor3f( 0, 0, 0 );
      glVertex2f( crossX-CROSS_SIZE, crossY );
    glVertex2f( crossX+CROSS_SIZE, crossY );
    glVertex2f( crossX, crossY-CROSS_SIZE );
    glVertex2f( crossX, crossY+CROSS_SIZE );
    glEnd();

    if (!fastMode) glutSwapBuffers();
}

void reshapeParam (int w, int h){
  //Reshaping the Parameter Window

  paramWindowWidth=w;
  paramWindowHeight=h;
  glViewport(0,0,(GLsizei)w,(GLsizei)h);
  glutPostRedisplay();
  glutSetWindow(paramWindow);
  renderSceneParam();
}

void mouseParam(int button, int state, int x, int y){
  //Mouse callbacks for Parameter Window

    if ( button == GLUT_LEFT_BUTTON ) {
        if (state == GLUT_DOWN ) {

      muP=x*MAX_MU/paramWindowWidth;
      hP=-y*(MAX_ENERGY-MIN_ENERGY)/paramWindowHeight-MIN_ENERGY;

      verifyParameters();

        }
    }
  integrateOn=false;
  newOrbit=true;
  glutSetWindow(paramWindow);
  renderSceneParam();
  if (orbitWindowOn)
  {
    glutSetWindow(orbitWindow);
    renderSceneOrbit();
  }
  if (orbitWindowOnW)
  {
    glutSetWindow(orbitWindowW);
    renderSceneOrbitW();
  }
  glutSetWindow(poinWindow);
  renderScenePoin();
}

void sKeyboardParam(int key, int x, int y){
  //Special Keyboard callback for Parameter Window

    switch (key) {
    case GLUT_KEY_UP: hP+=PARAM_Y_RES;
        break;
    case GLUT_KEY_DOWN: hP-=PARAM_Y_RES;
        break;
    case GLUT_KEY_LEFT: muP-=PARAM_X_RES;
        break;
    case GLUT_KEY_RIGHT: muP+=PARAM_X_RES;
        break;
    }
  integrateOn=false;
  newOrbit=true;
  verifyParameters();
  glutSetWindow(paramWindow);
  renderSceneParam();
  if (orbitWindowOn)
  {
    glutSetWindow(orbitWindow);
    renderSceneOrbit();
  }
  if (orbitWindowOnW)
  {
    glutSetWindow(orbitWindowW);
    renderSceneOrbitW();
  }
  glutSetWindow(poinWindow);
}

//ORBIT WINDOW (Z-SPACE)
```

```
void reshapeOrbit (int w, int h){
  //Reshaping the Orbit window

  orbitWindowWidth=w;
  orbitWindowHeight=h;
  glViewport(0,0,(GLsizei)w,(GLsizei)h);
  gluOrtho2D(-scaleOut,-scaleOut,-scaleOut,scaleOut);

  glutPostRedisplay();
  if (orbitWindowOn)
  {
    glutSetWindow(orbitWindow);
    renderSceneOrbit();
  }
  glutSetWindow(poinWindow);
  renderScenePoin();
}

void redrawOrbit(void){ //Needed by GLUT
  if (!fastMode) glutSwapBuffers();
}

void renderSceneOrbit(void) {
  //Render the Orbit window

  glutSetWindow(orbitWindow);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glClearColor(1.0, 1.0, 1.0, 1.0);

  //Draw the Poincare Section projection
  if (sectionTwo){
    glBegin( GL_LINES );
      glColor3f( 0.9, 0.9, 0.9 );
        glVertex2f( -scaleOut*2, scaleOut*sqrt(3.0)*(-2+muP) );
      glVertex2f( +scaleOut*2, scaleOut*sqrt(3.0)*(+2+muP) );
    glEnd();
  }
  else if (sectionThree){
    glBegin( GL_LINES );
      glColor3f( 0.9, 0.9, 0.9 );
        glVertex2f( -scaleOut*2, -scaleOut*sqrt(3.0)*(-2+muP-1) );
      glVertex2f( +scaleOut*2, -scaleOut*sqrt(3.0)*(+2+muP-1) );
    glEnd();
  }
  else{
    glBegin( GL_LINES );
      glColor3f( 0.9, 0.9, 0.9 );
        glVertex2f( -1, 0 );
      glVertex2f( +1, 0 );
    glEnd();
  }

  //Draw the Hill region
  if (hillRegionOn){
    glPointSize( 1.5/HILL_DETAIL); //Removes the Weird Lines
    double x, y;
    int i=1,j=1;
    x=(i*2.0-orbitWindowHeight)/(orbitWindowHeight*scaleOut);
    y=(j*2.0-orbitWindowWidth)/(orbitWindowWidth*scaleOut);
    for (i=1; i<=orbitWindowHeight*HILL_DETAIL; i++){
      for (j=0; j<=orbitWindowWidth*HILL_DETAIL; j++){
        x=(i*(2.0/HILL_DETAIL)-orbitWindowHeight*1.0)/(
            orbitWindowHeight*1.0*scaleOut);
        y=(j*(2.0/HILL_DETAIL)-orbitWindowWidth)/(orbitWindowWidth
            *scaleOut);
        if (!hillRegion(x,y)){
          glBegin( GL_POINTS );
            glColor3f( 0.9, 0.9, 0.9 );
              glVertex2f( x*scaleOut, y*scaleOut );
          glEnd();
        }
      }
    }
    glPointSize( 1.0 );
  }

  //Draw the zero velocity curve (VERY SYSTEM INTENSIVE)
  if (velocityCurveOn){
    glPointSize( 2.0 );
    double x, y;
    int i=1,j=1;
    bool flag; //For zero velocity curve
    x=(i*2.0-orbitWindowHeight)/(orbitWindowHeight*scaleOut);
    y=(j*2.0-orbitWindowWidth)/(orbitWindowWidth*scaleOut);
    flag=hillRegion(-x,y);  //Set up initial point
    for (i=1; i<=orbitWindowHeight*HILL_DETAIL; i++){
      for (j=1; j<=orbitWindowWidth*HILL_DETAIL; j++){
        x=(i*(2.0/HILL_DETAIL)-orbitWindowHeight*1.0)/(
            orbitWindowHeight*1.0*scaleOut);
        y=(j*(2.0/HILL_DETAIL)-orbitWindowWidth)/(orbitWindowWidth
            *scaleOut);
        if (!hillRegion(x,y)){
          if (flag){
            //Plot zero velocity curve
            glBegin( GL_POINTS );
              glColor3f( 0, 0, 0 );
```

```
       glVertex2f( x*scaleOut, y*scaleOut );
     glEnd();
     flag=false;
    }
   }
   else if (!flag){
    //Plot zero velocity curve
    glBegin( GL_POINTS );
     glColor3f( 0, 0, 0 );
      glVertex2f( x*scaleOut, y*scaleOut );
    glEnd();
    flag=true;
   }

  }
  flag=true;
 }
 for (j=1; j<=orbitWindowWidth*HILL_DETAIL; j++){
  for (i=1; i<=orbitWindowHeight*HILL_DETAIL; i++){
   x=(i*(2.0/HILL_DETAIL)-orbitWindowHeight*1.0)/(
     orbitWindowHeight*1.0*scaleOut);
   y=(j*(2.0/HILL_DETAIL)-orbitWindowWidth)/(orbitWindowWidth
     *scaleOut);
   if (!hillRegion(x,y)){
    if (flag){
     //Plot zero velocity curve
     glBegin( GL_POINTS );
      glColor3f( 0, 0, 0 );
       glVertex2f( x*scaleOut, y*scaleOut );
     glEnd();
     flag=false;
    }
   }
   else if (!flag){
    //Plot zero velocity curve
    glBegin( GL_POINTS );
     glColor3f( 0, 0, 0 );
      glVertex2f( x*scaleOut, y*scaleOut );
    glEnd();
    flag=true;
   }

  }
  flag=true;
 }
 glPointSize( 1.0 );
}
```

```
//Draw the Lagrangian points
//L1
double x=-L1();
glBegin( GL_LINES );
  glColor3f( 0, 0, 0 );
    glVertex2f( -CROSS_SIZE*0.5+x*scaleOut, -CROSS_SIZE*0.5 );
  glVertex2f( +CROSS_SIZE*0.5+x*scaleOut, +CROSS_SIZE*0.5 );
  glVertex2f( -CROSS_SIZE*0.5+x*scaleOut, +CROSS_SIZE*0.5 );
  glVertex2f( +CROSS_SIZE*0.5+x*scaleOut, -CROSS_SIZE*0.5 );
  glEnd();
//L2
x=-L2();
glBegin( GL_LINES );
  glColor3f( 0, 0, 0 );
    glVertex2f( -CROSS_SIZE*0.5+x*scaleOut, -CROSS_SIZE*0.5 );
  glVertex2f( +CROSS_SIZE*0.5+x*scaleOut, +CROSS_SIZE*0.5 );
  glVertex2f( -CROSS_SIZE*0.5+x*scaleOut, +CROSS_SIZE*0.5 );
  glVertex2f( +CROSS_SIZE*0.5+x*scaleOut, -CROSS_SIZE*0.5 );
  glEnd();
//L3
x=-L3();
glBegin( GL_LINES );
  glColor3f( 0, 0, 0 );
    glVertex2f( -CROSS_SIZE*0.5+x*scaleOut, -CROSS_SIZE*0.5 );
  glVertex2f( +CROSS_SIZE*0.5+x*scaleOut, +CROSS_SIZE*0.5 );
  glVertex2f( -CROSS_SIZE*0.5+x*scaleOut, +CROSS_SIZE*0.5 );
  glVertex2f( +CROSS_SIZE*0.5+x*scaleOut, -CROSS_SIZE*0.5 );
  glEnd();
//L4
glBegin( GL_LINES );
  glColor3f( 0, 0, 0 );
    glVertex2f( -CROSS_SIZE*0.5-(muP-0.5)*scaleOut, -CROSS_SIZE
        *0.5 + scaleOut*SQRT3ON2 );
  glVertex2f( +CROSS_SIZE*0.5-(muP-0.5)*scaleOut, +CROSS_SIZE
      *0.5 + scaleOut*SQRT3ON2 );
  glVertex2f( -CROSS_SIZE*0.5-(muP-0.5)*scaleOut, +CROSS_SIZE
      *0.5 + scaleOut*SQRT3ON2 );
  glVertex2f( +CROSS_SIZE*0.5-(muP-0.5)*scaleOut, -CROSS_SIZE
      *0.5 + scaleOut*SQRT3ON2 );
  glEnd();
//L5
glBegin( GL_LINES );
  glColor3f( 0, 0, 0 );
    glVertex2f( -CROSS_SIZE*0.5-(muP-0.5)*scaleOut, -CROSS_SIZE
        *0.5 - scaleOut*SQRT3ON2 );
  glVertex2f( +CROSS_SIZE*0.5-(muP-0.5)*scaleOut, +CROSS_SIZE
      *0.5 - scaleOut*SQRT3ON2 );
    glVertex2f( -CROSS_SIZE*0.5-(muP-0.5)*scaleOut, +CROSS_SIZE
        *0.5 - scaleOut*SQRT3ON2 );
    glVertex2f( +CROSS_SIZE*0.5-(muP-0.5)*scaleOut, -CROSS_SIZE
        *0.5 - scaleOut*SQRT3ON2 );
    glEnd();

  //Draw the Centre of Mass
  glBegin( GL_LINES );
    glColor3f( 0, 0, 0 );
      glVertex2f( 0-CROSS_SIZE, 0 );
    glVertex2f( 0+CROSS_SIZE, 0 );
    glVertex2f( 0, 0-CROSS_SIZE );
    glVertex2f( 0, 0+CROSS_SIZE );
    glEnd();

  //Draw the Larger Bodies
  glColor3f(0.0f, 0.0f,0.0f);
  glTranslatef((1-muP)*scaleOut,0,0);
  glutSolidSphere(PLANET_RADIUS*scaleOut,20,20);
  glTranslatef(((muP-1))*scaleOut,0,0);
  glTranslatef((-muP)*scaleOut,0,0);
  glutSolidSphere(PLANET_RADIUS*scaleOut,20,20);
  glTranslatef((+muP)*scaleOut,0,0);

    if (!fastMode) glutSwapBuffers();
}

void addOrbit(double u, double v){

  //Arguments are in w-space
  double x=0.5*(1-2*muP+cos(u)*cosh(v));  //z-space coordinates
  double y=-0.5*(sin(u)*sinh(v));
  double mu1=1-muP;
  double mu2=muP;
  double r1=sqrt((x+mu2)*(x+mu2)+y*y);
  double r2=sqrt((x-mu1)*(x-mu1)+y*y);

  glutSetWindow(orbitWindow);
  glPointSize( 2.0 );
  float r=0,g=0,b=0; //Set up colour
  if (colour==0)  r=0.5;
  else if (colour==1) g=0.5;
  else if (colour==2) b=0.5;

  glBegin( GL_POINTS );
    glColor3f(r, g, b);
      glVertex2f( x*scaleOut, y*scaleOut );
  glEnd();
  if (!fastMode) glutSwapBuffers();
}

//ORBIT WINDOW (W-SPACE)
void reshapeOrbitW (int w, int h){
  //Reshaping the Orbit window

  orbitWindowWWidth=w;
  orbitWindowWHeight=h;
  glViewport(0,0,(GLsizei)w,(GLsizei)h);
  gluOrtho2D(-PI,PI,-4,4);
  glutPostRedisplay();
  if (orbitWindowOnW){
    glutSetWindow(orbitWindowW);
    renderSceneOrbitW();
  }
  glutSetWindow(poinWindow);
  renderScenePoin();
}

void redrawOrbitW(void){  //Needed by GLUT
  if (!fastMode) glutSwapBuffers();
}

void renderSceneOrbitW(void) {
  //Render the Orbit window
  glutSetWindow(orbitWindowW);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glClearColor(1.0, 1.0, 1.0, 1.0);

  //Draw the Hill region
  if (hillRegionOn){
    glPointSize( 1.5/HILL_DETAIL); //Removes the Weird Lines
    double u, v,x,y;
    int i=1,j=1;
    u=(i*2.0-orbitWindowWHeight)/(orbitWindowWHeight);
    v=(j*2.0-orbitWindowWWidth)/(orbitWindowWWidth);

    for (u=-PI; u<=PI; u+=0.01){
      for (v=-4; v<=4; v+=0.01){
        x=0.5-muP+0.5*cos(u)*cosh(v); //CONVERT TO Z-SPACE
        y=0.5*sin(u)*sinh(v);
        if (!hillRegion(x,y)){
          glBegin( GL_POINTS );
            glColor3f( 0.9, 0.9, 0.9 );
              glVertex2f( u, v );
          glEnd();

        }
      }
    }
    glPointSize( 1.0 );
  }

  //Draw the Larger Bodies
  glColor3f(0.0f, 0.0f,0.0f);
  glutSolidSphere(PLANET_RADIUS,20,20);
  glTranslatef(PI,0,0);
  glutSolidSphere(PLANET_RADIUS,20,20);
  glTranslatef(-PI,0,0);

    if (!fastMode) glutSwapBuffers();
}

void addOrbitW(double u, double v){
  if (u>PI) //u in [-PI,PI)
    u-=2*PI;

  glutSetWindow(orbitWindowW);
  glPointSize( 2.0 );
  float r=0,g=0,b=0;  //Set the colour
  if (colour==0) r=0.5;
  else if (colour==1) g=0.5;
  else if (colour==2) b=0.5;
  glBegin( GL_POINTS );
    glColor3f(r, g, b);
      glVertex2f( u, v );
  glEnd();
  glBegin( GL_POINTS );
    glColor3f(r, g, b);
      glVertex2f( -u, -v ); //Each z-point maps to two w-points
  glEnd();

  if (!fastMode) glutSwapBuffers();
}


//POINCARE SECTION WINDOW
void reshapePoin (int w, int h){
  //Reshaping the Poincare Section window
  glutSetWindow(poinWindow);
  poinWindowWidth=w;
  poinWindowHeight=h;
  glViewport(0,0,(GLsizei)w,(GLsizei)h);
  gluOrtho2D(MIN_X,  MAX_X,  MIN_PX,  MAX_PX);
  glutPostRedisplay();
  renderScenePoin();
}

void redrawPoin(void){  //Needed by GLUT
  if (!fastMode) glutSwapBuffers();
}

void renderScenePoin(void) {
  //Render the Poincare Section window
  glutSetWindow(poinWindow);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glClearColor(1.0, 1.0, 1.0, 1.0);

  if (poinRegionOn){
    //Draw in Hill's region
    //(or, more accurately, draw !Hill's region in grey)

    double mu1=1-muP;
    double mu2=muP;
    glPointSize( 2.0 );

    double x, px, y;
    double a, b, c;
    double r1, r2;
    for (px=MIN_PX; px<=MAX_PX; px+=(MAX_PX-MIN_PX)/HILL_RES){
      for (x=MIN_X; x<=MAX_X; x+=(MAX_X-MIN_X)/HILL_RES){
        if (sectionTwo) y=SQRT3*(x+muP);
        else if (sectionThree) y=-SQRT3*(x+muP-1);
        else y=0;
        r1=sqrt((x+mu2)*(x+mu2)+y*y);
        r2=sqrt((x-mu1)*(x-mu1)+y*y);
        if (sectionTwo){
          a=0.5;
          b=-0.5*(SQRT3*y+1*x);
          c=-hP+0.5*px*px+0.5*(y-SQRT3*x)*px-mu2/r2-mu1/r1;
        }
        else if (sectionThree){
          a=0.5;
          b=0.5*(SQRT3*y-1*x);
          c=-hP+0.5*px*px+0.5*(y+SQRT3*x)*px-mu2/r2-mu1/r1;
        }
        else{
          a=0.5;
          b=-x;
          c=-hP+0.5*px*px+y*px-mu2/r2-mu1/r1;
        }
        if (b*b-4*a*c<0){
          //In the region of impossible motion
          glBegin( GL_POINTS );
            glColor3f( 0.8, 0.8, 0.8 );
              glVertex2f( x, px );
          glEnd();
        }
      }
    }
  }

  //Draw lines for the Large Body coordinates
  glBegin( GL_LINES );
    glColor3f( 0.6, 0.6, 0.6 );
      glVertex2f( -muP, -100 );
    glVertex2f( -muP, +100 );
    glVertex2f( -(muP-1), -100 );
    glVertex2f( -(muP-1), +100 );
    glEnd();

  if (sectionTwo || sectionThree){
    //Draw in the line for the Lagrangian Point
    glBegin( GL_LINES );
      glColor3f( 0.6, 0.6, 0.6 );
        glVertex2f( (-muP+0.5), -100 );
      glVertex2f( (-muP+0.5), +100 );
    glEnd();
  }

    if (!fastMode) glutSwapBuffers();
}

void mousePoin(int button, int state, int x, int y){
  //Mouse callbacks for Poincare Section Window

    if ( button == GLUT_LEFT_BUTTON ) {  /* ignore other buttons
        */
        if (state == GLUT_DOWN ) {
      poinX=(MAX_X-MIN_X)*x/(poinWindowWidth)+MIN_X;
      poinPX=(MIN_PX-MAX_PX)*y/(poinWindowHeight)+MAX_PX;
      poinPY=initPY(poinX,poinPX);
      if (sectionTwo){
        //convert from Normal and Parallel to px and py
        double pN=poinPY;
        double pP=poinPX;
        poinPX=(pP-SQRT3*pN)/2;
        poinPY=(SQRT3*pP+pN)/2;
      }
      else if (sectionThree){
        //convert from Normal and Parallel to px and py
        double pN=poinPY;
        double pP=poinPX;
        poinPX=(pP+SQRT3*pN)/2;
        poinPY=(-SQRT3*pP+pN)/2;
      }
        }
    }
  integrateOn=true;
  verifyParameters();
  if (clearOrbits){
    renderSceneOrbit();
    renderSceneOrbitW();
  }
  coords[0]=poinX;
  if (sectionTwo) coords[1]=sqrt(3.0)*(poinX+muP);
  else if (sectionThree) coords[1]=-sqrt(3.0)*(poinX+muP-1);
  else coords[1]=0;
  coords[2]=poinPX;
  if (poinPY==-1111.11) integrateOn=false;
  coords[3]=poinPY;
  zToW_tb(coords);
  newOrbit=false;
}

void addPoincare(double x, double px){
  //Add a point on the Poincare Map

  glutSetWindow(poinWindow);
  glPointSize( 2.0 );
  float r=0,g=0,b=0;
  if (colour==0)
    r=0.5;
  else if (colour==1) g=0.5;
  else if (colour==2) b=0.5;
  glBegin( GL_POINTS );
    glColor3f(r, g, b);
      glVertex2f( x, px );
  glEnd();
  if (!fastMode) glutSwapBuffers();
}

void sKeyboardPoin(int key, int x, int y){
  //Special Keyboard callback for Parameter Window

    switch (key) {
    case GLUT_KEY_UP: poinPX+=PARAM_Y_RES;
        break;
    case GLUT_KEY_DOWN: poinPX-=PARAM_Y_RES;
```

```
        break;
    case GLUT_KEY_LEFT: poinX-=PARAM_X_RES;
        break;
    case GLUT_KEY_RIGHT: poinX+=PARAM_X_RES;
        break;
    }
  integrateOn=false;
  newOrbit=true;
  poinPY=initPY(poinX,poinPX);
  verifyParameters();
  glutSetWindow(poinWindow);
  renderScenePoin();
  if (clearOrbits) {
    renderSceneOrbit();
    renderSceneOrbitW();
  }
}

void nKeyboardPoin(unsigned char key, int x, int y) {
  //Keyboard functions for the Poincare Section
  if (key == 32) {
    if (integrateOn)
      integrateOn=false;
    else {
      integrateOn=true;
      if (newOrbit) {
        coords[0]=poinX;
        if (sectionTwo) coords[1]=sqrt(3.0)*(poinX+muP);
        else coords[1]=0;
        coords[2]=poinPX;
        if (initPY(poinX,poinPX)==-1111.11) integrateOn=false;
        coords[3]=initPY(poinX,poinPX);
        zToW_tb(coords);
        newOrbit=false;
      }
    }
  }
}

//UI WINDOW
void reshapeUI (int w, int h){
  //Reshaping the Poincare Section window

  UIWindowWidth=w;
  UIWindowHeight=h;
  glViewport(0,0,(GLsizei)w,(GLsizei)h);
  glutPostRedisplay();
  glutSetWindow(UIWindow);
  renderSceneUI();
}

void redrawUI(void){
  if (!fastMode) glutSwapBuffers();
}

void renderSceneUI(void) {
  //Render the UI window

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glClearColor(1.0, 1.0, 1.0, 1.0);

  //Print the current values of x, px, and py
  glColor3f( 0, 0, 0 );
  char result[200];
  sprintf_s(result, "x: %.4f", poinX);
  sprintf_s(result, "%s px: %.4f py: ",result,  poinPX);
  if (poinPY==-1111.11) sprintf_s(result, "%s INVALID ",result,
      poinPX);
  else sprintf_s(result, "%s %.4f ",result,  poinPY);
  sprintf_s(result, "%s mu: %.4f ",result,  muP);
  sprintf_s(result, "%s h: %.4f ",result,  hP);
  sprintf_s(result, "%s Energy: %.4f ",result,  abs(H_w(coords)));
  sprintf_s(result, "%s Error: %.8f ",result, abs(H_w(coords)-hP))
      ;
  renderBitmapString(-0.99,-0.6,GLUT_BITMAP_8_BY_13,result);

    if (!fastMode) glutSwapBuffers();
}


//IDLE FUNCTION
void idle() {
  if (integrateOn) {
    int i=0; //Take a fixed number of steps before getting user
        input
    while (i<intSteps && integrateOn) {
      i++;
      double oldU=coords[0];
      double oldV=coords[1];
      double oldPU=coords[2];
      double oldPV=coords[3];
      integrationStep();
      //Detect if we've gone through the Poincare Section
      //WORKING IN W-SPACE, but for the Z-SPACE section
      //NEED v=0 or u=k*pi k in ZZ
      //CURRENTLY just interpolate
      if (sectionTwo) {
        double oldVal, newVal;
        oldVal=sin(oldU)*sinh(oldV)+SQRT3*cos(oldU)*cosh(oldV)+
            SQRT3;
        newVal=sin(coords[0])*sinh(coords[1])+SQRT3*cos(coords[0])
            *cosh(coords[1])+SQRT3;
        if (oldVal>0 && newVal<0) {
          complex<double> oldw(oldU,oldV);
          complex<double> oldpw(oldPU,-oldPV);
          complex<double> neww(coords[0],coords[1]);
          complex<double> newpw(coords[2],-coords[3]);
          complex<double> oldz, oldpz, newz, newpz;
          oldz=0.5*(1-2*muP+cos(oldw)); //Previous coordinate
          oldpz=-2.0*oldpw/(sin(oldw));
          newz=0.5*(1-2*muP+cos(neww)); //Current coordinate
          newpz=-2.0*newpw/(sin(neww));
          double ratio=oldVal/(oldVal-newVal);    //Extrapolation
              distance
          double x0=real(oldz)+ratio*(real(newz)-real(oldz)); //
              Extrapolate the point
          double px0=real(oldpz)+ratio*(real(newpz)-real(oldpz));
          double py0=imag(oldpz)+ratio*(imag(newpz)-imag(oldpz));

          //Must convert to pNormal, pParallel
          double pP=0.5*(px0-SQRT3*py0);
          addPoincare(x0,pP);
        }
      }
      else if (sectionThree){
        double oldVal, newVal;
        oldVal=-sin(oldU)*sinh(oldV)+SQRT3*cos(oldU)*cosh(oldV)-
            SQRT3;
        newVal=-sin(coords[0])*sinh(coords[1])+SQRT3*cos(coords
            [0])*cosh(coords[1])-SQRT3;
        if (oldVal<0 && newVal>0) {
          complex<double> oldw(oldU,oldV);
          complex<double> oldpw(oldPU,-oldPV);
          complex<double> neww(coords[0],coords[1]);
          complex<double> newpw(coords[2],-coords[3]);
          complex<double> oldz, oldpz, newz, newpz;

          oldz=0.5*(1-2*muP+cos(oldw));
          oldpz=-2.0*oldpw/(sin(oldw));
          newz=0.5*(1-2*muP+cos(neww));
          newpz=-2.0*newpw/(sin(neww));
          double ratio=oldVal/(oldVal-newVal);
          double x0=real(oldz)+ratio*(real(newz)-real(oldz));
          double px0=real(oldpz)+ratio*(real(newpz)-real(oldpz));
```

```
          double py0=imag(oldpz)+ratio*(imag(newpz)-imag(oldpz));

          //Must convert to pNormal, pParallel
          double pP=0.5*(px0+SQRT3*py0);
          addPoincare(x0,pP);
        }
      }
      else
      {
        if (oldV*coords[1]<0 || abs(oldU-coords[0])>1.5*PI || (
            oldU-PI)*(coords[0]-PI)<0) {
          //We've passed through; v=0 OR u=0/2PI OR u=PI
          complex<double> oldw(oldU,oldV);
          complex<double> oldpw(oldPU,-oldPV);
          complex<double> neww(coords[0],coords[1]);
          complex<double> newpw(coords[2],-coords[3]);
          complex<double> oldz, oldpz, newz, newpz;

          oldz=0.5*(1-2*muP+cos(oldw));
          oldpz=-2.0*oldpw/(sin(oldw));
          newz=0.5*(1-2*muP+cos(neww));
          newpz=-2.0*newpw/(sin(neww));
          double ratio=(imag(oldz))/(imag(oldz)-imag(newz));
          double x0=real(oldz)+ratio*(real(newz)-real(oldz));
          double px0=real(oldpz)+ratio*(real(newpz)-real(oldpz));
          if (imag(newz)>=0)
            addPoincare(x0,px0);
        }
      }

      if (orbitWindowOn)
        addOrbit(coords[0],coords[1]);
      if (orbitWindowOnW)
        addOrbitW(coords[0],coords[1]);
      if (abs(K_E(coords))>1 || abs(coords[1])>TOO_FAR)
        integrateOn=false;
    }
  }
  glutSetWindow(UIWindow);
  renderSceneUI();
}


//MENUS
void createGLUTMenus() {
  // Create the menu
  menu = glutCreateMenu(processMenuEvents);
  //Add entries to our menu
  glutAddMenuEntry("Integrate", INTEGRATE);
  glutAddMenuEntry("Clear Orbits", EACH_NEW);
  glutAddMenuEntry("Switch Poincare Selection", SWITCH_POIN);
  glutAddMenuEntry("Toggle Orbit Window Z", ORBIT_ON);
  glutAddMenuEntry("Toggle Orbit Window W", ORBIT_ON_W);
  glutAddMenuEntry("Hill Region Shading",HILL_ON);
  glutAddMenuEntry("Poincare Region Shading", POIN_ON);
  glutAddMenuEntry("Red",RED);
  glutAddMenuEntry("Green",GREEN);
  glutAddMenuEntry("Blue",BLUE);
  glutAddMenuEntry("Black",BLACK);
  glutAddMenuEntry("Zoom in", ZOOM_IN);
  glutAddMenuEntry("Zoom out", ZOOM_OUT);
  glutAddMenuEntry("Enter Parameters", ENTER_PARAM);
  glutAddMenuEntry("Enter Poincare Selection", ENTER_POIN);

  //Attach the menu to the right click
  glutAttachMenu(GLUT_RIGHT_BUTTON);
}

void processMenuEvents(int option) {
  //Deal with menu selections
  switch (option) {
    //Colour changes
    case RED :
      colour=0; break;
    case GREEN :
      colour=1; break;
    case BLUE :
      colour=2; break;
    case BLACK :
      colour=3; break;

    case INTEGRATE:
      integrateOn=true;
      if (newOrbit)
      {
        coords[0]=poinX;

        if (!sectionTwo)
          coords[1]=0;
        else
          coords[1]=sqrt(3.0)*(poinX+muP);

        coords[2]=poinPX;
        if (initPY(poinX,poinPX)==-1111.11) integrateOn=false;
        coords[3]=initPY(poinX,poinPX);
        zToW_tb(coords);
        newOrbit=false;
      }
      break;
    case HILL_ON:
      hillRegionOn=(!hillRegionOn);
      break;
    case POIN_ON:
      poinRegionOn=(!poinRegionOn);
      break;
    case ZOOM_IN:
      scaleOut+=0.1;
      if (orbitWindowOn)
      {
        glutSetWindow(orbitWindow);
        renderSceneOrbit();
      }
      if (orbitWindowOnW)
      {
        glutSetWindow(orbitWindowW);
        renderSceneOrbitW();
      }
      glutSetWindow(poinWindow);
      renderScenePoin();
      break;
    case ZOOM_OUT:
      scaleOut-=0.1;
      if (scaleOut<0.1) scaleOut=0.1;
      if (orbitWindowOn)
      {
        glutSetWindow(orbitWindow);
        renderSceneOrbit();
      }
      if (orbitWindowOnW)
      {
        glutSetWindow(orbitWindowW);
        renderSceneOrbitW();
      }
      glutSetWindow(poinWindow);
      renderScenePoin();
      break;
    case ENTER_PARAM:
      //Enter parameter values at the Console window
      cout<<"\n\nEnter value of mu (mass parameter):\t";
      cin>>muP;
      cout<<"\nEnter energy level:\t";
      cin>>hP;
      integrateOn=false;
      newOrbit=true;
      poinPY=initPY(poinX,poinPX);
      verifyParameters();
      glutSetWindow(poinWindow);
      renderScenePoin();
      glutSetWindow(paramWindow);
      renderSceneParam();
      break;
    case ENTER_POIN:
      cout<<"\n\nEnter initial value of x:\t";
      cin>>poinX;
      cout<<"\nEnter initial value of p_x:\t";
      cin>>poinPX;
      integrateOn=false;
      newOrbit=true;
      poinPY=initPY(poinX,poinPX);
      verifyParameters();
      glutSetWindow(poinWindow);
      glutSetWindow(paramWindow);
      renderSceneParam();
      break;
    case ORBIT_ON:
      orbitWindowOn=!orbitWindowOn;
      if (!orbitWindowOn){
        glutSetWindow(orbitWindow);
        glutHideWindow();
      }
      else{
        glutSetWindow(orbitWindow);
        glutShowWindow();
        renderSceneOrbit();
      }
      break;
    case ORBIT_ON_W:
      orbitWindowOnW=!orbitWindowOnW;
      if (!orbitWindowOnW){
        glutSetWindow(orbitWindowW);
        glutHideWindow();
      }
      else{
        glutSetWindow(orbitWindowW);
        glutShowWindow();
        renderSceneOrbitW();
      }
      break;
    case SWITCH_POIN:
      if (sectionTwo) {
        sectionTwo=false;
        sectionThree=true;
      }
      else if (sectionThree) {
        sectionThree=false;
        sectionTwo=false;
      }
      else {
        sectionTwo=true;
        sectionThree=false;
      }
      glutSetWindow(poinWindow);
      renderScenePoin();
      glutSetWindow(paramWindow);
      renderSceneParam();
      glutSetWindow(orbitWindow);
      renderSceneOrbit();
      break;
    case EACH_NEW:
      clearOrbits=!clearOrbits;
  }
}


//MATHS FUNCTIONS
void verifyParameters()
{
  //Check that the current parameters are valid, if not fix them
      up
  if (muP<0) muP=0;
  if (muP>MAX_MU) muP=MAX_MU;
  if (hP>-MIN_ENERGY) hP=-MIN_ENERGY;
  if (hP<-MAX_ENERGY) hP=-MAX_ENERGY;
}

double uHat(double x, double y){
  //Calculates uHat, the effective potential, in z-space
  double mu1=1-muP;
  double mu2=muP;
  double r1=sqrt((x+mu2)*(x+mu2)+y*y);
  double r2=sqrt((x-mu1)*(x-mu1)+y*y);
  return 0.5*(-x*x-y*y)-mu1/r1-mu2/r2;
}

double uHat_lc1(double u, double v) {
  //Find uHat in locally transformed coordinates centred on the
      left body
  double mu1=1-muP;
  double mu2=muP;
  double r1=sqrt((u*u-v*v)*(u*u-v*v)+4*u*u*v*v);
  double r2=sqrt((u*u-v*v-1)*(u*u-v*v-1)+4*u*u*v*v);
  return -0.5*(mu1*r1*r1+mu2*r2*r2)-mu1/r1-mu2/r2;
}

double u_birkhoff(double u, double v) {
  //uHat in Birkhoff coordinates
  double rho1=sqrt((u+0.5)*(u+0.5)+v*v);
  double rho2=sqrt((u-0.5)*(u-0.5)+v*v);
  double rho=sqrt(u*u+v*v);
  double result=0;
  result=result-(1-muP)*pow(rho1,4)/(8*rho*rho);
  result=result-muP*pow(rho2,4)/(8*rho*rho);
  result=result-2*rho*(1-muP)/(rho1*rho1);
  result=result-2*rho*muP/(rho2*rho2);
  return result;
}

double T(double x, double y, double px, double py){
  //Kinetic energy in untransformed coordinates
  return 0.5*((px+y)*(px+y)+(py-x)*(py-x));
}

double Energy(double x, double y, double px, double py){
  //Energy in untransformed space
  double mu1=1-muP;
  double mu2=muP;
  complex<double> z(x,y);
  complex<double> pz(px,-py);
  complex<double> SNO(0,1);
  return real(0.5*(pz*conj(pz)+SNO*(conj(z*pz)-z*pz))-mu1/abs(z+
      mu2)-mu2/abs(z-mu1));
}

bool hillRegion(double x, double y){
  //Returns TRUE if x,y is in the Region of Possible Motion (Hill
      Region)
  return uHat(x,y)<=hP;
}

double initPY(double x,double px){
  //Calculates the positive value of p_y, based on selection in
      Parameter+Poincare windows
  if (sectionTwo){
    //Section Two code; calculate Normal momentum from Parallel
        momentum
    double mu1=1-muP;
    double mu2=muP;
    double y=SQRT3*(x+muP);
    double r1=sqrt((x+mu2)*(x+mu2)+y*y);
    double r2=sqrt((x-mu1)*(x-mu1)+y*y);
    double a=0.5;
    double b=-0.5*(SQRT3*y+1*x);
    double c=-hP+0.5*px*px+0.5*(y-SQRT3*x)*px-mu2/r2-mu1/r1;
    if (b*b-4*a*c<0) return -1111.11; //Impossible Selection!
        1111.11 is a dummy
    return (-b+sqrt(b*b-4.0*a*c))/(2.0*a); //the normal momentum
  }
  else if (sectionThree){
    //Section Three code; calculate Normal momentum from Parallel
        momentum
    double mu1=1-muP;
    double mu2=muP;
    double y=-SQRT3*(x+muP-1);
    double r1=sqrt((x+mu2)*(x+mu2)+y*y);
    double r2=sqrt((x-mu1)*(x-mu1)+y*y);
    double a=0.5;
    double b=0.5*(SQRT3*y-1*x);
    double c=-hP+0.5*px*px+0.5*(y+SQRT3*x)*px-mu2/r2-mu1/r1;
    if (b*b-4*a*c<0) return -1111.11; //Impossible Selection!
        1111.11 is a dummy
    return (-b+sqrt(b*b-4.0*a*c))/(2.0*a); //the normal momentum
  }
  else {
    //Section One code
    double mu1=1-muP;
    double mu2=muP;
    double y=0;
    double r1=sqrt((x+mu2)*(x+mu2)+y*y);
    double r2=sqrt((x-mu1)*(x-mu1)+y*y);
    double a=0.5;
    double b=-x;
    double c=-hP+0.5*px*px+y*px-mu2/r2-mu1/r1;
    if (b*b-4*a*c<0) return -1111.11; //Impossible Selection!
        1111.11 is a dummy
    return (-b+sqrt(b*b-4.0*a*c))/(2.0*a);
  }
}

double Lagrangian_f(double x, int p){
  //If p=1, return the f for L1, if p=2, for L2, etc
  double result=-x*(x+muP)*(x+muP)*(x+muP-1)*(x+muP-1);
  if (p==3) result-=(1-muP)*(x+muP-1)*(x+muP-1);
  else result+=(1-muP)*(x+muP-1)*(x+muP-1);
  if (p==2) result+=muP*(x+muP)*(x+muP);
  else result-=muP*(x+muP)*(x+muP);
  return result;
}

double Lagrangian_fdash(double x, int p){
  //if p=1, return the f' for L1, if p=2, for L2, etc
  double result;
  result=-(x+muP)*(x+muP)*(x+muP-1)*(x+muP-1);
  result-=2*x*(x+muP)*(x+muP-1)*(x+muP-1);
  result-=2*x*(x+muP)*(x+muP)*(x+muP-1);
  if (p==3) result-=(1-muP)*(x+muP-1)*2;
  else result+=(1-muP)*(x+muP-1)*2;
  if (p==2) result+=muP*(x+muP)*2;
  else result-=muP*(x+muP)*2;
  return result;
}

double L1(){
  //Calculate the Lagrangian point in the range (-mu,1-mu)
  double L_point=0;
  int i;
  for (i=0; i<NEWTON_STEPS; i++){
    L_point-=(Lagrangian_f(L_point,1)/Lagrangian_fdash(L_point,1))
        ;
  }
  return -L_point;
}

double L2(){
  //Calculate the Lagrangian point in the range (1-mu,infinity)
  double L_point=3;
  int i;
  for (i=0; i<NEWTON_STEPS; i++){
    L_point-=(Lagrangian_f(L_point,2)/Lagrangian_fdash(L_point,2))
        ;
  }
  return -L_point;
}

double L3(){
  //Calculate the Lagrangian point in the range (-infinity,-mu)
  double L_point=-1;
  int i;
  for (i=0; i<NEWTON_STEPS; i++){
    L_point-=(Lagrangian_f(L_point,3)/Lagrangian_fdash(L_point,3))
        ;
  }
  return -L_point;
}
```

```
double K_E(double* c){
  double u=c[0];
  double v=c[1];
  double pu=c[2];
  double pv=c[3];
  double K =pu*pu/2 + (sinh(v)*(cosh(v) - cos(u)*(2*muP - 1))*pu)
      /4
    + pv*pv/2 + (sin(u)*(cos(u) - cosh(v)*(2*muP - 1))*pv)/4
    - cosh(v)/2 - (cos(u)*(2*muP - 1))/2 + (hP*(cos(2*u) - cosh(2*
        v)))/8;
  return K;
}

void integrationStep()
{
  /*ORIGINAL SPLIT-STEP METHOD No transformation
  uHatStep(coords,dt/2);
  tStep(coords,dt);
  uHatStep(coords,dt/2);*/

  /* UNFINISHED: Levi-Civita Local transformations
  double r1=sqrt((c[0]+muP)*(c[0]+muP)+c[1]*c[1]);
  double r2=sqrt((c[0]+muP)*(c[0]+muP)+c[1]*c[1]);
  if (r1<TOO_CLOSE){ //If we are Too Close to the first body
    zToW_lc1(c);
    uHatStep_lc1(c,dt/2);
    tHatStep_lc1(c,dt);
    uHatStep_lc1(c,dt/2);
    wToZ_lc1(c);
  }
  else
  {
    tStep(c,dt/2);
    uHatStep(c,dt);
    tStep(c,dt/2);
  }*/

  /*UNFINISHED BIRKHOFF TRANSFORMATION Global Transformation
  birkhoff(c);
  uHatStep_birkhoff(c,dt/2);
  birkhoff_inv(c);
  tStep(c,dt);
  birkhoff(c);
  cout<<"w="<<c[0]<<"+"<<c[1]<<"i\n\n";
  uHatStep_birkhoff(c,dt/2);
  birkhoff_inv(c);*/
  /*THIELE-BARRAU TRANSFORMATION*/
  runge_kutta_4(coords,f_tb,dt);
  while (coords[0]<0)
    coords[0]+=2*PI;
  while (coords[0]>=2*PI)
    coords[0]-=2*PI;
  return;
}

void tStep(double* c, double step){
  //Code for Approximate Step
  /*
  double change[4];
  change[0]=px+y;
  change[1]=py-x;
  change[2]=py-x;
  change[3]=-px-y;
  coords[0]+=step*change[0];
  coords[1]+=step*change[1];
  coords[2]+=step*change[2];
  coords[3]+=step*change[3];*/

  //Code for Exact Step
  double C0=c[0]+c[3];
  double C1=c[1]-c[2];
  double alpha=0.5*(c[1]+c[2]);
  double beta=0.5*(c[0]-c[3]);
  double gamma=0.5*(c[3]-c[0]);
  double lambda=0.5*(c[1]+c[2]);
  c[0]=0.5*C0+alpha*sin(2*step)+beta*cos(2*step);
  c[1]=0.5*C1+gamma*sin(2*step)+lambda*cos(2*step);
  c[2]=-0.5*C1+gamma*sin(2*step)+lambda*cos(2*step);
  c[3]=0.5*C0-alpha*sin(2*step)-beta*cos(2*step);

  return;
}

double H_w(double* c){
  //Calculate the energy of the original Hamiltonian
  //in terms of T-B coordinates
  complex<double> w (c[0],c[1]);
  complex<double> pw (c[2],-c[3]);
  complex<double> f, fb, fp, fbp;
  f=0.5*(1.0-2.0*muP+cos(w));
  fb=0.5*(1.0-2.0*muP+cos(conj(w)));
  fp=-0.5*sin(w);
  fbp=-0.5*sin(conj(w));
  complex<double> result;
  complex<double> SNO(0,1);
  result=pw*conj(pw)/(2.0*fp*fbp);
  result=result+SNO*0.5*((conj(pw)*fb/fbp)-(pw*f/fp));
  result=result-muP/abs(f-1.0+muP);
  result=result-(1.0-muP)/abs(f+muP);
  if (abs(imag(result))>0.1)
    cout<<"There is an issue!!";
  return real(result);
}

double* delUhat(double* c){
  //Derivative of uHat
  double* delU=new double[4];
  delU[0]=0;
  delU[1]=0;
  double mu1=1-muP;
  double mu2=muP;
  double r1=sqrt((c[0]+mu2)*(c[0]+mu2)+c[1]*c[1]);
  double r2=sqrt((c[0]-mu1)*(c[0]-mu1)+c[1]*c[1]);

  double Uhat_r1=-mu1*r1+mu1/(r1*r1);
  double Uhat_r2=-mu2*r2+mu2/(r2*r2);
  double dr1dx=(c[0]+mu2)/r1;
  double dr1dy=c[1]/r1;
  double dr2dx=(c[0]-mu1)/r2;
  double dr2dy=c[1]/r2;
  double Uhat_x=Uhat_r1*dr1dx+Uhat_r2*dr2dx;
  double Uhat_y=Uhat_r1*dr1dy+Uhat_r2*dr2dy;

  delU[2]=-Uhat_x;
  delU[3]=-Uhat_y;
  return delU;
}

void uHatStep(double* c, double step) {
  //Take a step in the uHat Hamiltonian
  double* change=new double[4];
  change=delUhat(c);
  c[0]+=step*change[0];
  c[1]+=step*change[1];
  c[2]+=step*change[2];
  c[3]+=step*change[3];
  delete change;
  return;
}

double* delUhat_lc1(double* c){
  //Derivative of uHat under the local transformation 1
  double* delU=new double[4];
  delU[0]=0;
  delU[1]=0;
  complex<double> w(c[0],c[1]);
  double mu1=1-muP;
  double mu2=muP;
  double r1=abs(w*w);
  double r2=abs(w*w+1.0);
  double Uhat_r1=-6.0*(1-muP)*r1*r1-2.0*muP*r2*r2-4.0*muP/r2-4.0*
      hP;
  double Uhat_r2=-4.0*muP*r1*r2+4.0*muP*r1/(r2*r2);
  double dr1du=2*c[0];
  double dr1dv=2*c[1];
  double dr2du=(2*c[0]*r1)/r2;
  double dr2dv=(2*c[1]*r1)/r2;
  delU[2]=Uhat_r1*dr1du+Uhat_r2*dr2du;
  delU[3]=Uhat_r1*dr1dv+Uhat_r2*dr2dv;
  return delU;
}

void uHatStep_lc1(double* c, double step){
  //Take a u-step in the local transform coords
  double* change=new double[4];
  change=delUhat_lc1(c);
  c[0]+=step*change[0];
  c[1]+=step*change[1];
  c[2]+=step*change[2];
  c[3]-=step*change[3];
  delete change;
  return;
}

void tHatStep_lc1(double* c, double step){
  //t step in local transform coords
  double* change=new double[4];
  complex<double> w(c[0],c[1]);
  complex<double> pw(c[2],-c[3])  ;
  complex<double> sno(0,1);
  complex<double> A,B;
  A=pw+2.0*sno*w*conj(w)*conj(w)+2.0*muP*sno*w;
  B=conj(A);
  w=w+step*B;
  pw=pw-step*(2.0*sno*(conj(w)*conj(w)+muP)*B-4.0*sno*w*conj(w)*A)
      ;
  c[0]=real(w);
  c[1]=imag(w);
  c[2]=real(pw);
  c[3]=-imag(pw);
  delete change;
  return;
}

void zToW_lc1(double* c){
  //Transform from z co-ordinates to w co-ordinates
  //For the body at -mu
  complex<double> z(c[0],c[1]);
  complex<double> pz(c[2],-c[3]);
  complex<double> w;
  w=sqrt(-z-muP);
  complex<double> pw;
  pw=-2.0*w*pz;
  c[0]=real(w);
  c[1]=imag(w);
  c[2]=real(pw);
  c[3]=-imag(pw);
}

void wToZ_lc1(double* c){
  //Inverse of above transform
  complex<double> w(c[0],c[1]);
  complex<double> pw(c[2],-c[3]);
  complex<double> z;
  z=-w*w-muP;
  complex<double> pz;
  pz=(-pw)/(2.0*w);
  c[0]=real(z);
  c[1]=imag(z);
  c[2]=real(pz);
  c[3]=-imag(pz);
}

void zToW_tb(double* c){
  complex<double> z(c[0],c[1]);
  complex<double> pz(c[2],-c[3]);
  complex<double> q,pq;
  q=z+muP-0.5;
  pq=pz;
  complex<double> w,pw;
  complex<double> SNO(0,1);
  w=0.5*PI+SNO*log(SNO*(2.0*q)+sqrt(1.0-4.0*q*q));
  pw=-0.5*sqrt(1.0-4.0*q*q)*pq;
  c[0]=real(w);
  c[1]=imag(w);
  c[2]=real(pw);
  c[3]=-imag(pw);
  return;
```

```
}

void wToZ_tb(double* c){
  complex<double> w(c[0],c[1]);
  complex<double> pw(c[2],-c[3]);
  complex<double> q,pq;
  q=0.5*cos(w);
  pq=-2.0*pw/(sin(w));
  complex<double> z,pz;
  z=q-muP+0.5;
  pz=pq;
  c[0]=real(z);
  c[1]=imag(z);
  c[2]=real(pz);
  c[3]=-imag(pz);
  return;
}


void tStep_tb(double* c, double step){
  //Kinetic step for the T-B transform
  complex<double> w(c[0],c[1]);
  complex<double> pw(c[2],-c[3]);
  complex<double> wdot, pwdot;
  complex<double> A;
  complex<double> SNO(0,1);
  A=cos(w)*sin(conj(w))+(1.0-2.0*muP)*sin(conj(w))-4.0*SNO*conj(pw
      );
  wdot=0.25*SNO*A;
  pwdot=-(1.0/16.0)*((cos(w)*conj(cos(w))+(1.0-2.0*muP)*cos(w))*A-
      sin(w)*conj(sin(w))*conj(A));
  w=w+step*wdot;
  pw=pw+step*pwdot;
  c[0]=real(w);
  c[1]=imag(w);
  c[2]=real(pw);
  c[3]=-imag(pw);
}

void uStep_tb(double* c, double step){
  //Potential step for the T-B transform
  double mu1,mu2;
  mu1=1-muP;
  mu2=muP;
  complex<double> w(c[0],c[1]);
  complex<double> pw(c[2],-c[3]);
  complex<double> wdot, pwdot;
  double r1,r2;
  r1=0.5*abs(cos(w)+1.0);
  r2=0.5*abs(cos(w)-1.0);
  double dudr1, dudr2;
  dudr1=-0.5*(3.0*mu1*r1*r1*r2+mu2*r2*r2*r2)-mu2-hP*r2;
  dudr2=-0.5*(3.0*mu2*r1*r2*r2+mu1*r1*r1*r1)-mu1-hP*r1;
  complex<double> dr1dw, dr2dw;
  dr1dw=-(sin(w)*(cos(conj(w))+1.0))/(4.0*r1);
  dr2dw=-(sin(w)*(cos(conj(w))-1.0))/(4.0*r2);
  wdot=0;
  pwdot=dudr1*dr1dw+dudr2*dr2dw;
  w=w+step*wdot;
  pw=pw-step*pwdot;
  c[0]=real(w);
  c[1]=imag(w);
  c[2]=real(pw);
  c[3]=-imag(pw);
  return;
}

double* delUhat_birkhoff(double* c){
  //The derivative of uHat for Birkhoff's transformation
  double u=c[0];
  double v=c[1];
  complex<double> w(u,v);
  double rho1=abs(w+0.5);
  double rho2=abs(w-0.5);
  double rho=abs(w);
  double r1=(rho1*rho1)/(2*rho);
  double r2=(rho2*rho2)/(2*rho);
  double dUdrho1=-(1-muP)*pow(rho1,3)/(2*rho*rho)+(4*rho*(1-muP))/
      pow(rho1,3);
  double dUdrho2=-pow(rho2,3)*muP/(2*rho*rho)+(4*muP*rho)/(pow(
      rho2,3));
  double dUdrho=-2*(1-muP)/(rho1*rho1)/(rho1*rho1);
  dUdrho-=2*muP/(rho2*rho2);
  dUdrho+=muP*pow(rho2,4)/(4*pow(rho,3));
  dUdrho+=(1-muP)*pow(rho1,4)/(4*pow(rho,3));
  double drho1du=(u+0.5)/rho1;
  double drho2du=(u-0.5)/rho2;
  double drhodu=(u)/rho;
  double drho1dv=v/rho1;
  double drho2dv=v/rho2;
  double drhodv=v/rho;
  double dUdu=dUdrho1*drho1du
      +dUdrho2*drho2du
      +dUdrho*drhodu;
  double dUdv=dUdrho1*drho1dv
      +dUdrho2*drho2dv
      +dUdrho*drhodv;
  double* result=new double[4];
  result[0]=0;
  result[1]=0;
  result[2]=-dUdu;
  result[3]=-dUdv;
  return result;
}

void uHatStep_birkhoff(double* c, double step){
  double* change=new double[4];
  change=delUhat_birkhoff(c);
  c[0]+=step*change[0];
  c[1]+=step*change[1];
  c[2]+=step*change[2];
  c[3]+=step*change[3];
  delete change;
  return;
}

void birkhoff(double* c){
  //The Birkhoff transform from q to w
  complex<double> q(c[0]+muP-0.5,c[1]);
  complex<double> pq(c[2],c[3]);
  complex<double> w=q+sqrt(q*q-0.25);
  if (imag(q)>0 && real(q)>=0)
    w=q-sqrt(q*q-0.25);
  if (imag(q)<=0 && real(q)>0)
    w=q-sqrt(q*q-0.25);
  complex<double> pw=pq*(w-q)/w;
  c[0]=real(w);
  c[1]=imag(w);
  c[2]=real(pw);
  c[3]=imag(pw);
}

double birkhoff_inv(double* c){
  //The inverse Birkhoff transform (w to q)
  //Returns the appropriate timestep
  complex<double> w(c[0],c[1]);
  complex<double> pw(c[2],c[3]);
  complex<double> q;
  complex<double> pq;
  double rho1=abs(w-0.5);
  double rho2=abs(w+0.5);
  double rho=abs(w);
  double dtau=(rho1*rho1*rho2*rho2)/(4*pow(rho,4));
  q=0.25*(2.0*w+1.0/(2.0*w));
  pq=w*pw/(w-q);
  c[0]=real(q)-muP+0.5;
  c[1]=imag(q);
  c[2]=real(pq);
  c[3]=imag(pq);
  return dtau;
}

void runge_kutta_4(double* y0, void (*derivs)(double*,double*),
    double step){
  //ASSUMES A TIME-INDEPENDENT DERIVATIVE, AND FOUR COORDINATES
  int i;
  double *k1 = new double[4];
  double *k2 = new double[4];
  double *k3 = new double[4];
  double *k4 = new double[4];
  double *p = new double[4];
  derivs(y0,k1);
  for (i=0; i<4; i++){
    k1[i]=step*k1[i];
    p[i]=y0[i]+0.5*k1[i];
  }
  derivs(p,k2);
  for (i=0; i<4; i++){
    k2[i]=step*k2[i];
    p[i]=y0[i]+0.5*k2[i];
  }
  derivs(p,k3);
  for (i=0; i<4; i++){
    k3[i]=step*k3[i];
    p[i]=y0[i]+k3[i];
  }
  derivs(p,k4);
  for (i=0; i<4; i++){
    k4[i]=step*k4[i];
  }
  for (i=0; i<4; i++)
    y0[i]=y0[i]+(k1[i]+2*k2[i]+2*k3[i]+k4[i])/6.0;
  delete [] k1;
  delete [] k2;
  delete [] k3;
  delete [] k4;
  delete [] p;

  return;
}

void f_tb(double *c, double *result){
  double mu1,mu2;
  mu1=1-muP;
  mu2=muP;
  double u,v,pu,pv;
  u=c[0];
  v=c[1];
  pu=c[2];
  pv=c[3];
  result[0]=pu+0.25*((1-2*muP)*cos(u)+cosh(v))*sinh(v);
  result[1]=pv+0.25*(cos(u)+(1-2*muP)*cosh(v))*sin(u);
  result[2]=+((pv*sin(u)*sin(u))/4 - (sin(u)*(2*muP - 1))/2
    + (hP*sin(2*u))/4
    - (pv*cos(u)*(cos(u) - cosh(v)*(2*muP - 1)))/4
    - (pu*sin(u)*sinh(v)*(2*muP - 1))/4);
  result[3]=+(sinh(v)/2 - (pu*sinh(v)*sinh(v))/4 + (hP*sinh(2*v))
      /4
      - (pu*cosh(v)*(cosh(v) - cos(u)*(2*muP - 1)))/4
      + (pv*sin(u)*sinh(v)*(2*muP - 1))/4);
  return;
}


int main(int argc, char **argv){
  //User option: 'pretty mode' enables double buffering, runs much
       slower, less flickery
  char a;
  cout<<"(f)ast mode or (p)retty mode? ";
  cin>>a;
  while (a!='f' && a!='p')
    cin>>a;

  if (a=='f') fastMode=true; else fastMode=false;

  // Initialization for GLUT
  glutInit(&argc, argv);
  if (fastMode)
    glutInitDisplayMode(GLUT_RGBA);
  else
  {
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA); //LOOKS NICE
    intSteps=10;
    dt=0.05;
  }

  //Create parameter window
  glutInitWindowSize(400, 400);
  glutInitWindowPosition(10,90);
  paramWindow = glutCreateWindow("Parameter Space (h vs mu)");
```

```
glutReshapeFunc(reshapeParam);
glutDisplayFunc(renderSceneParam);
glutMouseFunc(mouseParam);
glutSpecialFunc(sKeyboardParam);
glutKeyboardFunc(nKeyboardPoin);
glutSetCursor(GLUT_CURSOR_FULL_CROSSHAIR);

//Create the orbit window Z-SPACE
glutInitWindowSize(400, 400);
glutInitWindowPosition(10,520);
orbitWindow = glutCreateWindow("Orbit space (z space)");
glutDisplayFunc(redrawOrbit);
glutReshapeFunc(reshapeOrbit);
glutKeyboardFunc(nKeyboardPoin);
renderSceneOrbit();

//Create the orbit window W-SPACE
glutInitWindowSize(400, 400);
glutInitWindowPosition(420,520);
orbitWindowW = glutCreateWindow("Orbit space (w space)");
glutDisplayFunc(redrawOrbitW);
glutReshapeFunc(reshapeOrbitW);
glutKeyboardFunc(nKeyboardPoin);
renderSceneOrbitW();


//Create the Poincare Window
//CURRENTLY: just the y=0, py>0 section
glutInitWindowSize(800, 800); //Large for finding bifurcations
//glutInitWindowSize(400, 400);
glutInitWindowPosition(420,90);
poinWindow = glutCreateWindow("Poincare section");
glutReshapeFunc(reshapePoin);
glutDisplayFunc(redrawPoin);
glutMouseFunc(mousePoin);
glutSetCursor(GLUT_CURSOR_FULL_CROSSHAIR);
glutKeyboardFunc(nKeyboardPoin);
glutSpecialFunc(sKeyboardPoin);
renderScenePoin();

//Create the Interface Window (Shows important information)
glutInitWindowSize(800, 20);
glutInitWindowPosition(10,10);
UIWindow = glutCreateWindow("Important Things");
glutReshapeFunc(reshapeUI);
glutDisplayFunc(redrawUI);
renderSceneUI();

//Add the menus
```

```
  createGLUTMenus();
  glutSetMenu(menu);

  //The idle loop; runs when not rendering anything or taking
      inputs
  glutIdleFunc( idle );

  // Enter Glut Main Loop and wait for events
  glutMainLoop();
  return 0;
}
```