Trojan Asteroids

6895Q

March 2014

Abstract

The three body restricted problem with $e = 0$ is the case of a 'planet' and 'sun' orbiting their barycenter in circular orbits, with an object of negligible mass also orbiting. It can do so in the stable equilibrium Lagrange points L_4 and L_5, $\pi/3$ from the planet-sun axis. A numerical simulations were carried out which demonstrated the stability of L_4 for a planet sun mass ratio less than 0.04. Over a range of planet masses there was found to be a linear relationship with the wander range for a small initial perturbation (as shown in Figure **8**).

1 Introduction

1.1 Two Body Problem

If two solid bodies are in gravitational equilibrium they orbit their common barycenter (centre of mass) in elliptical orbits, with separation, $r(\theta)$:

$$\frac{1}{r(\theta)} \propto 1 + e\cos(\theta)$$

The simplified case of circular orbits, $e = 0$ is considered here, but the offset of each body from the barycenter is preserved.

1.2 Restricted Three Body Problem

In this, a negligible mass is introduced into the above problem, not affecting the two larger masses. This can be in equilibrium at the 'Lagrange points':

The L_4 and L_5 ('triangular') points, situated equidistant from the sun and the planet (as is easily to proven), are the only potentially stable points (if the condition $m < \frac{2}{25+3\sqrt{69}} \approx 1/24.96$ is met[1]). In the circular case considered here, all three objects lie at the vertices of an equilateral triangle.

As points of stable equilibrium, a certain amount of 'wander' about the equilibrium position is possible, dependent on initial displacement/velocity, planet mass and radius. This is the main subject of investigation.

2 Analysis

2.1 Two body

The positions of the bodies m_1 and m_2 with respect to the barycenter are $-m_2/(m_1+m_2)$ and $m_1/(m_1+m_2)$ times $r(\theta)$ respectively. In the circular case here there is the simplification $r(\theta)=a$. The period of this system is:

$$T = 2\pi\sqrt{\frac{a^3}{G\left(M_1+M_2\right)}}$$

where a is the sum of the semi-major axes, the separation distance for the circular case [2].

In this problem, scaled units are used, such that the mass of the more massive object (the 'sun') is 1, the unit of distance is the AU (the Earth's mean distance from the sun) and the unit time is 1 year, giving a gravitational constant of $G = 4\pi^2$. The less massive object is referred to as the 'planet' and has mass 'm'.

Thus the non-elliptic positions of the sun and planet as functions of time can be easily found as:

$$\mathbf{r_s}(t) = -a\frac{m}{1+m}(cos(\omega t)\hat{\mathbf{x}} + sin(\omega t)\hat{\mathbf{y}})$$

$$\mathbf{r_p}(t) = a\frac{1}{1+m}(cos(\omega t)\hat{\mathbf{x}} + sin(\omega t)\hat{\mathbf{y}})$$

where they are chosen to lie in the x-y plane for convenience. $\omega = 2\pi/T$ is the angular frequency.

2.2 Resultant Gravitational field

From the body positions as a function of time, the gravitational field and tensor due to this system at a position $\mathbf{r}$ are defined as:

$$\mathbf{g} = -\frac{G}{|\mathbf{r}-\mathbf{r_s}|^3}(\mathbf{r}-\mathbf{r_s}) - \frac{Gm}{|\mathbf{r}-\mathbf{r_p}|^3}(\mathbf{r}-\mathbf{r_p})$$

$$g'_{ij} = \frac{\partial g_i}{\partial r_j} = -\frac{G}{|\mathbf{r}-\mathbf{r_s}|^5}(|\mathbf{r}-\mathbf{r_s}|^2\delta_{ij} - 3(r_i - r_{si})) - \frac{Gm}{|\mathbf{r}-\mathbf{r_p}|^5}(|\mathbf{r}-\mathbf{r_p}|^2\delta_{ij} - 3(r_i - r_{pi}))$$

2.3 System of equations

The motion of an 'asteroid' - an object with no gravitational mass, not affecting the motions of the other bodies - in this system is governed by the equation:

$$\frac{d^2\mathbf{r}}{dt^2} = \mathbf{g}(\mathbf{r},t)$$

This can be broken down into the 6 first order coupled ODEs:

$$\frac{dr_i}{dt} = v_i$$

$$\frac{dv_i}{dt} = g_i(\mathbf{r}, t)$$

where i ranges over the values x, y, z, which are of the necessary form $dy_i/dt = f_i(y, t)$. The jacobian (j also ranging over x, y, z), $J_{ij} = \partial f_i/\partial y_j$, of the system is:

$$J_{\mathbf{r_i r_j}} = J_{\mathbf{v_i v_j}} = 0$$

$$J_{\mathbf{r_i v_j}} = \delta_{ij}$$

$$J_{\mathbf{v_i r_j}} = g'_{ij}(\mathbf{r})$$

2.4 General adaptive Runge-Kutta methods

A Runge-Kutta method is a stepping algorithm for solving ODEs, determined by its coefficients b_i, b_i^*, c_i, k_i. Each step is given by:

$$y^*_{n+1} = y_n + \sum_{i=1}^{s} b_i^* k_i,$$

where

$$k_1 = hf(t_n, y_n),$$

$$k_2 = hf(t_n + c_2 h, y_n + a_{21} k_1),$$

$$k_3 = hf(t_n + c_3 h, y_n + a_{31} k_1 + a_{32} k_2),$$

$$...$$

There are p such equations for a $O(h^p)$ method. The error in each step is then $O(h^{p+1})$, and can be given by:

$$e_{n+1} = y_{n+1} - y^*_{n+1} = \sum_{i=1}^{s} (b_i - b_i^*) k_i,$$

where the k_i are the same as for the higher-order method. The sizes of the errors can be used to determine the step size used. This method is easily generalised to multiple-dimensional equation systems.

In the simulation, the Runge-Kutta-Fehlberg (RKF45) order 4(5) method was used, with step size control such that the maximum error was constrained to be smaller than 10^{-6}.

2.5 Stationary solution and the corotating frame

In the circular problem, the Lagrangian points L_4 and L_5 are those in the plane of the orbit forming an equilateral triangle with the planet and sun. Bodies orbiting with the same period as the planet are stable at these points.

It is useful in this task to work in the corotating frame, where the Lagrange points are stationary. The vector $\mathbf{R}$ is defined as the position $\mathbf{r}$, rotated such that the planet-sun axis is the x-axis.

3 Implementation

The problem was broken down into several stages solved sequentially (both in the program and in implementation). Each was tested before moving on.

3.1 Data types - '*vec3.h*' and '*constsanddatatypes.h*'

As the problem involved positions in space and was described very well using 3D vectors, the **cav** :: **Vec3** library ('*vec3.h*') was used to provide this functionality. One small change was made to facilitate outputting data sets in an easily plotted format:

```
  void print( std::ostream& os ) const
  {
    os << "(" << m_x[0] << ", " << m_x[1] << ", " << m_x[2] << ")";
  }
```

Was replaced by:

```
void print( std::ostream& os ) const
        {
            os << m_x[0] << ' ' << m_x[1] << ' ' << m_x[2];
        }
```

In the file '*constsanddatatypes.h*' necessary universal constants and two data types (structures) were provided, to store the solar system state (sun and planet position vectors in **solar_state**) and parameters (**system_params**).

3.2 Two Body Problem - '*solar_state.cpp*' and '*gravity.cpp*'

The non-elliptic positions of the sun, $\mathbf{r_s}(t)$, and planet, $\mathbf{r_p}(t)$, as functions of time are calculated in the '*solar_state.cpp*' file using the **calculate_solar_state** function returning a **solar_state** data type. Functions for calculating the omega for a stable circular orbit, the position of the L_4 Lagrange point, $\mathbf{r}_l$, and the corotating vector position $\mathbf{R}$ were also included.

This was the only part of the problem where generality was sacrificed for easier implementation, in the use of $e = 0$ circular orbits.

A full implementation of the gravitational field equations (including the tensor) given in 2.2 is provided, using the **solar_state** data type to make the code as extensible as possible (implementation of $e \neq 0$ thus requiring only changes to '*solar_state.cpp*'). In addition, a testing function, outputting $x, y, g_x(x, y), g_y(x, y)$, is provided.

3.3 Numerical Solution - ‘*funcandjacob.cpp*’ and ‘*simulation.cpp*’

The GSL ODE Library was used for the implementation of an RKF45 method with maximum absolute step size error 10^{-6} (which was obtained by experimentation as not significantly affecting data produced over a very long time scale). A state array y[] was needed and was chosen such that the lower three values contained r and the higher three v, so that the system of equations looked like this:

```
    const cav::Vec3 g = grav_acc(s, p.m, r);

    f[0] = y[3];
    f[1] = y[4];
    f[2] = y[5];

    f[3] = g.x();
    f[4] = g.y();
    f[5] = g.z();
```

where s is the state of the solar system and p.m is m. These functions and the Jacobian (using the GSL matrix library) are implemented in ‘*funcandjacob.cpp*’.

A ‘driver’ function for ease of implementation was used in the **simulate** function:

```
double simulate(double m, double planet_sep, double t0, double t1,
    cav::Vec3 r0, cav::Vec3 v0, int no_outs, bool bound_only, bool
    output)
```

which outputs **no_outs** pieces of data between **t0** and **t1** in the format:

$$t, r_x, r_y, r_z, v_x, v_y, v_z, R_x, R_y, R_z, |\mathbf{R} - \mathbf{r}_l|, \alpha$$

where α is the angle between **R** and the planet-sun axis. The function additionally records the range of α during the simulation and returns this, or, if **bound_only**, ‘-1’ if α drifts out of the range $-\pi/3$ to $2\pi/3$. This was done because in most of the simulations we are only interested in bound orbits, and exiting once an unbound one was found sped the program up significantly (the unbound solutions usually escaping the limits very quickly). Non-exiting simulations generally took less than one second for several hundred ‘years’ of simulation.

3.4 Perturbation - ‘*pert.cpp*’

Several functions are provided which run the above simulation with the same output and return value, but calculate the stable initial state and take various ‘perturbed’ inputs, **r1** and **v1**, such that if these are 0, the asteroid remains in the L_4 point indefinitely:

```
bool simulate_perterbation_from_l4(double planet_m, double
    planet_sep, double final_time, cav::Vec3 r1, cav::Vec3 v1, int
    no_outs, bool output);
```

and

```
double pert_to_angle(double angle, double m, bool out);
```

which starts an additional angle ‘angle’ away from the Lagrangian point.

These functions are only wrappers for the simulation function but provide a more convenient way to access it.

3.5 Header files and ‘*main.cpp*’

For good practice, each ‘.cpp’ file was also given a header file including only the functions used by other ‘.cpp’ files. Each ‘.cpp’ file also included only the necessary header files for itself. In this instance it makes the code more disparate, but is done for maximum extensibility and readability.

The ‘*main.cpp*’ file contains only the main function, which handles only input (and some minor output). It reads from the command line so output can easily be piped to a file or gnuplot.

3.6 Performance

Of the simulations carried out in the problem, as described below, only the range of wander required more than about a second, instead taking about 20. This is because it relies on doing multiple simulations one after another, whereas normally the simulation length is simply linear in the internal time simulated for. As noted above, the possibility of exiting for unbound solutions sped this up significantly by saving time on calculating such solutions.

4 Results and Discussion

4.1 Gravitational Field - option ‘g’

The results of the **test_grav_acc** function (option ‘g’), which loops over positions and generates a gravatational acceleration vector at each point, are shown in Figure **1**. The system is that with $m = 0.3$ and separation 10, after 5 ‘years’ of solar state evolution. The planet and sun positions are also shown.

This is graph is generated from the data using the gnuplot command:

```
plot 'data.txt' u 1:2:3:4 w vectors head filled title
    'Gravatational Field', "<echo ' -0.978925 -2.08977 \n 3.26308
    6.96591'" with points ls 2 title 'Sun and Planet'
```

4.2 Restricted two body problem - option ‘2’

In this option, an asteroid in a stable orbit radius 1 in the y-z plane, with no planet interfering ($m = 0$). The plot of position in Figure **2** shows that the orbit is stable and completely regular - despite the many orbits, the positions at given

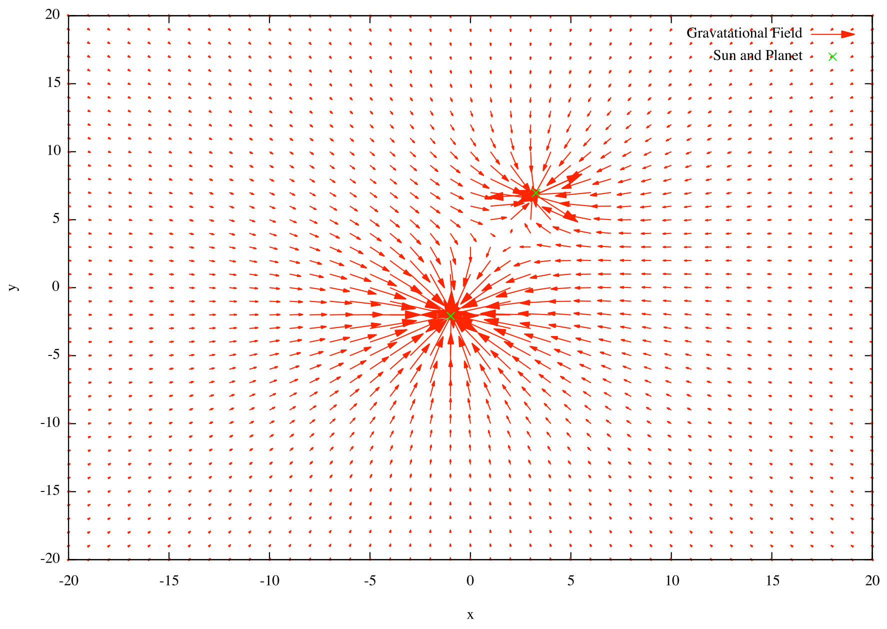

Figure 1: Gravatational field

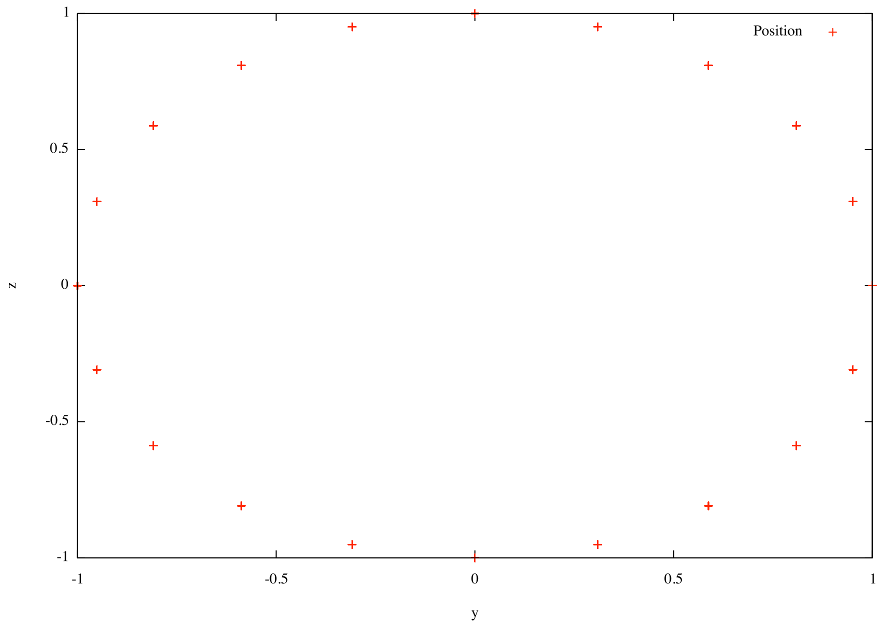

Figure 2: Angle of the unperturbed Jupiter trojan asteroid

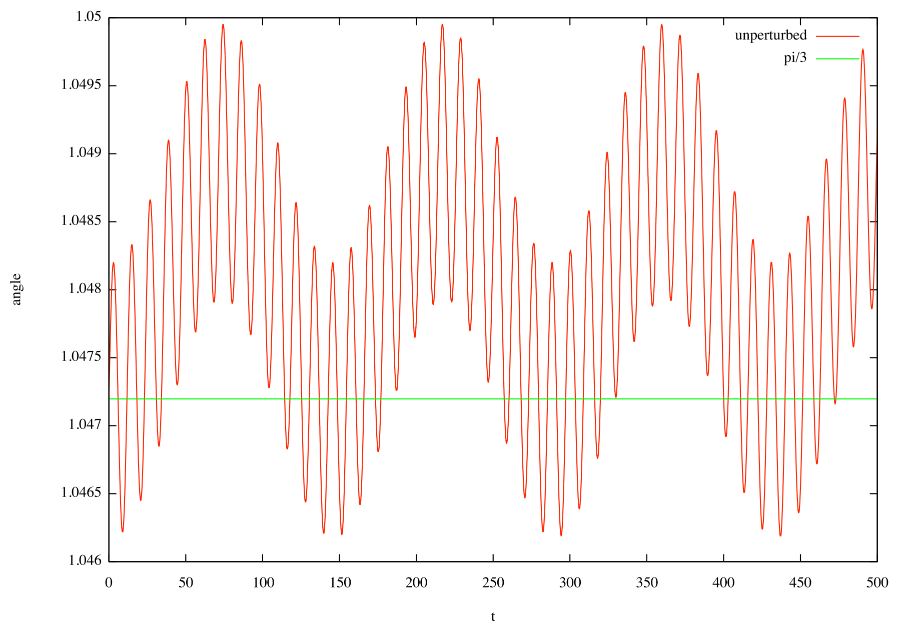

Figure 3: Angle of the unperturbed Jupiter trojan asteroid

times overlap completely (the period is a multiple of the step time for output), which shows that the numerical solution is behaving as intended.

4.3 Unperturbed Jupiter Solution - option ‘j’

Figure **3** shows the angle as a function of time, for the unperturbed solution, with separation 5.2 and mass 0.001 (i.e. Jupiter). It shows that any numerical errors are stable and non-propagating, in this case having a size of less than 0.05 radians, a very small value. It uses columns 1 and 12.

Figure **4** shows the 2:3 columns of the same dataset, demonstrating that the asteroid stays in the same orbit.

4.4 Jupiter z perturbation - option ‘z’

Figures **5** and **6** clearly show that the system is stable for a small perturbation in the z direction (a small velocity perturbation of 0.1 ‘AU per year’ is shown here).

4.5 Perturbed Jupiter Solution - option ‘p’

Here the asteroid is angularly perturbed by 0.15 from the equilibrium state. The position in terms of the corotating frame is shown in figure **7**, showing a closed set of curves indicating oscillations about a stable equilibrium point.

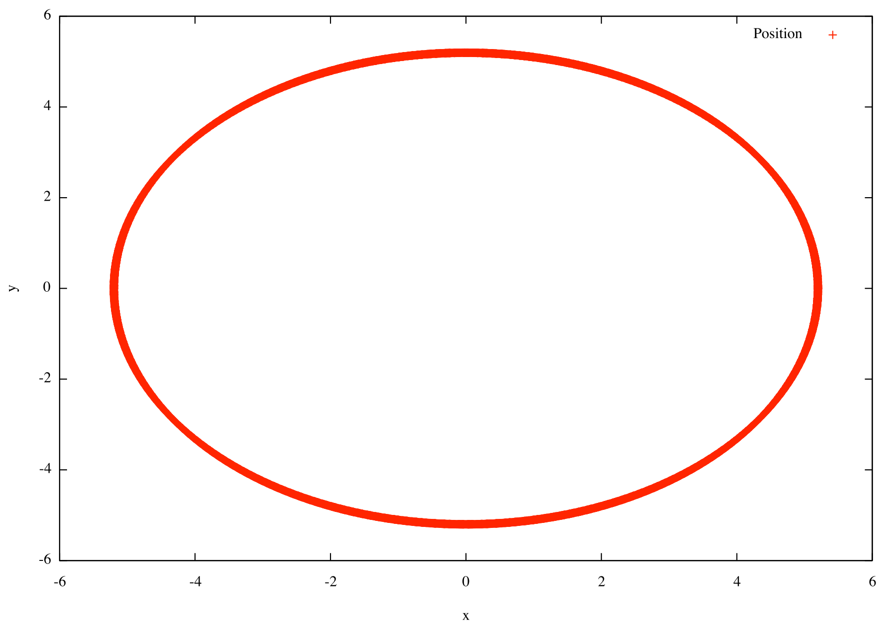

Figure 4: x and y positions of the unperturbed Jupiter trojan asteroid

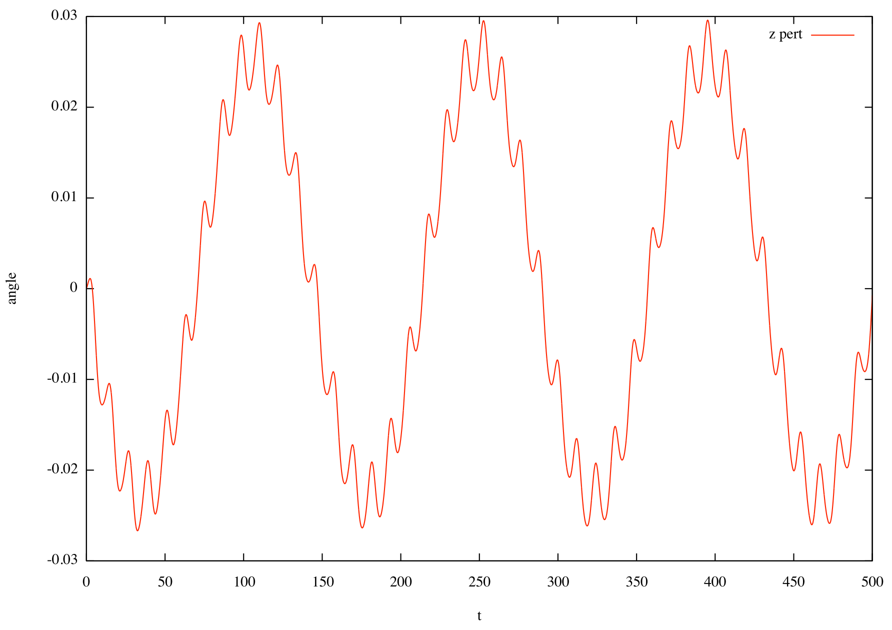

Figure 5: Angle as a function of time for a z perturbation

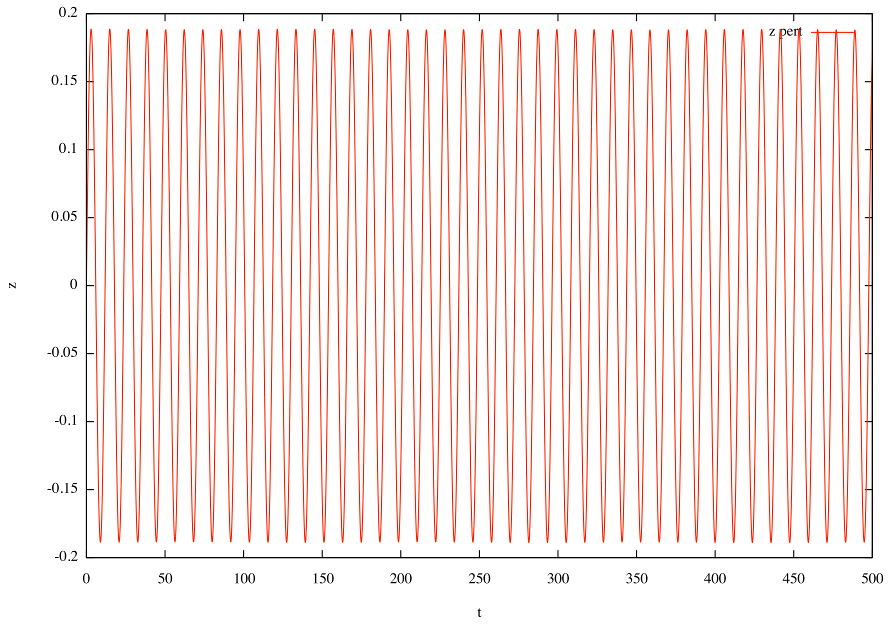

Figure 6: z as a function of time for a z perturbation

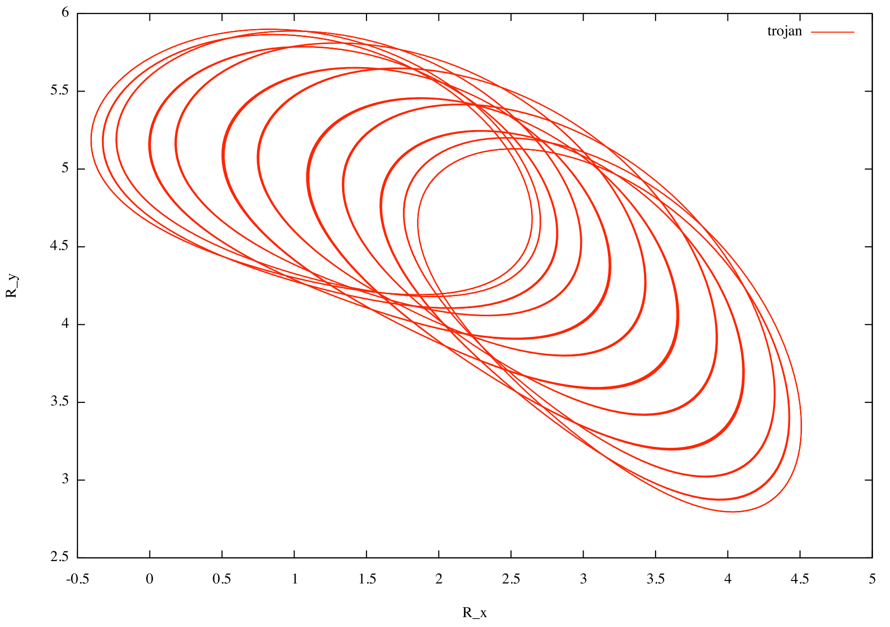

Figure 7: Perturbed trojan asteroid position in corotating frame for jupiter system

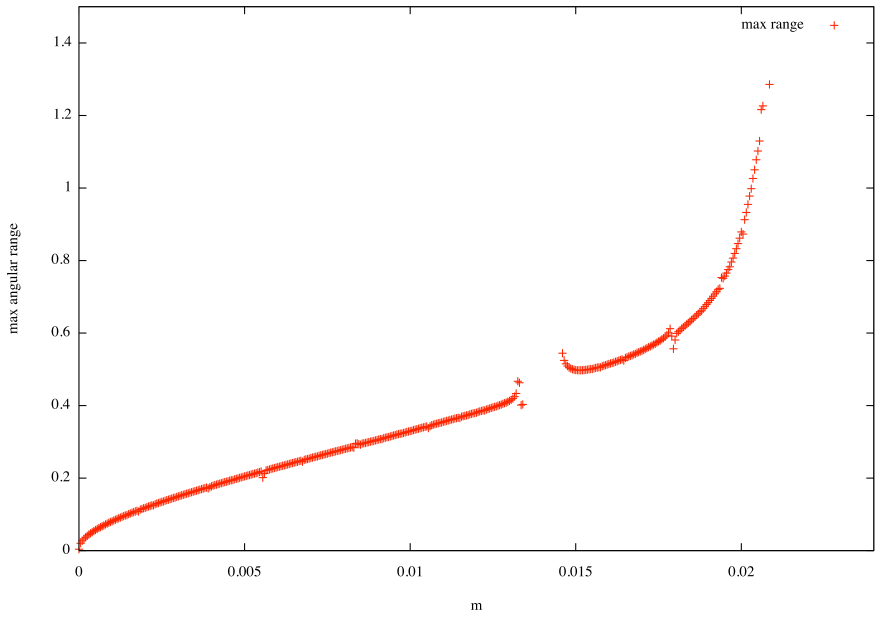

Figure 8: Maximum angular range for a given mass

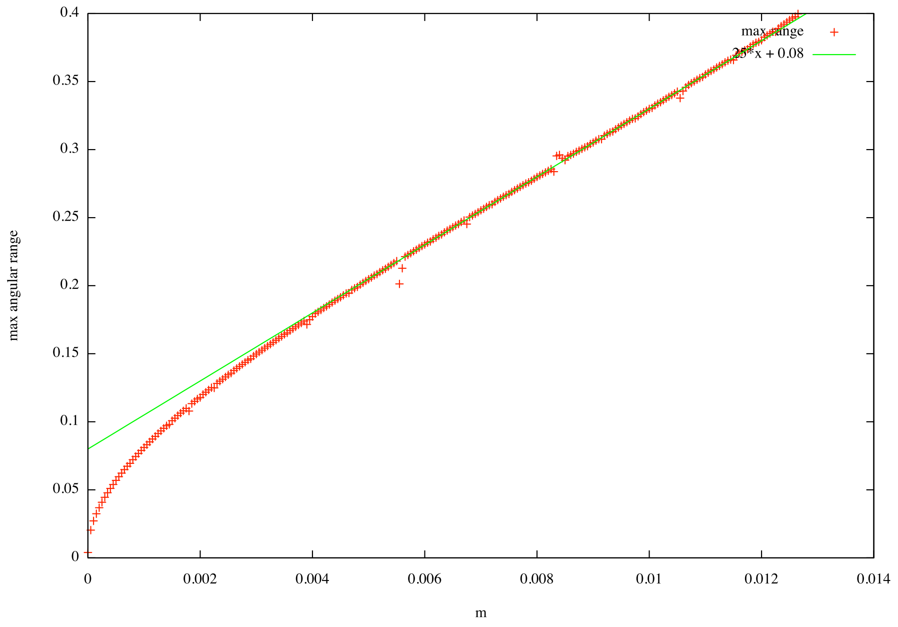

Figure 9: Maximum angular range for a given mass with linear fit

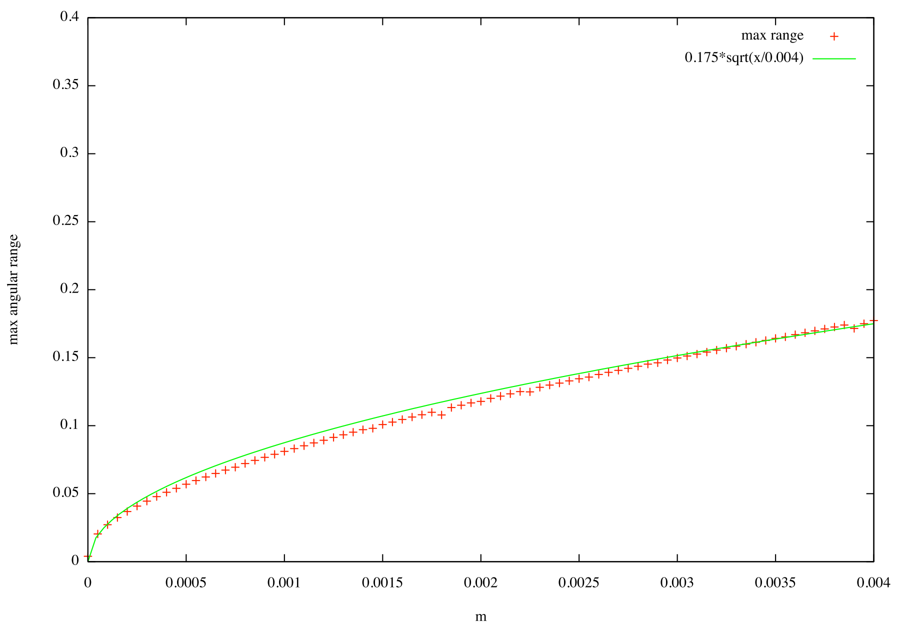

Figure 10: Maximum angular range for a given mass with square root fit

4.6 Range of wander as related to mass - option 'w'

For a system with separation 1, masses are looped over and the mass and the maximum angular range for a given starting angular displacement (2^{-10}) are outputted to produce **8**.

This shows some interesting features. Above $m = 0.04$ no stable patterns occur, as is mentioned in the introduction. Not expected, however, is that for $0.0134 < m < 0.0146$ no stable patterns are possible either. The curve at this point has the appearance of a resonance peak. It is not known whether this is a genuine physical effect or the result of 'resonant' numerical instabilities, though much previous checking had been done of these.

Below these points (and also between them) there is a clear linear relationship between the size of wander and the mass of the planet. Figure **9** shows a close up view with the fitted line $25m + 0.08$ which is an excellent fit for the range $0.04 < m < 0.12$. This can in some way be physically understood: sufficiently small perturbations about a stable equilibrium point must necessarily be linear, and the gravatational potential is directly proportional to the mass.

Near to zero, in the range $0 < m < 0.04$, another fit again is required: $0.175\sqrt{250m}$, shown in figure **10**. Physically, as the mass goes to zero it is obvious that the range must also, but it is not obvious why it does not do so linearly (i.e. why the linear fit above has a nonzero intercept).

5 Conclusions

An extensible and adaptable simulator for the restricted three body problem was produced and various simulations were carried out.

The Lagrangian points have been shown to be stable for small perturbations about the equilibrium, both in the z and within the plane. Many such solutions give non-repeating but stable curves, giving an array of interesting orbital patterns.

The wander range for a given small initial perturbation was investigated and showed that, as expected, above a certain mass orbits were unstable, and as the mass went to zero no such wander occurred. For some range in the middle there was a linear relationship.

Though all of the tests done on the points were using the L_4 point, by symmetry they also hold for the L_5 point. A topic that could be further investigated in future might be how the separation affects the wander in a similar way.

A Code Listing

Compile all using the command:

```
g++ -o trojan.exe gravity.cpp main.cpp pert.cpp simulation.cpp
   solar_state.cpp funcandjacob.cpp `gsl-config --cflags --libs`
```

```
//
//  constsanddatatypes.h
//  trojan_asteroids
//


#ifndef trojan_asteroids_constsanddatatypes_h
#define trojan_asteroids_constsanddatatypes_h

//useful physical constants
const double pi = 3.14159265359;
const double grav_const = 4*pi*pi; //sun set to mass 1
const cav::Vec3 zaxis(0,0,1);


//object for paramaters
typedef struct{
   double m;
   double sep;
} system_params;


//object for solar system state
typedef struct{
   cav::Vec3 rs;  //sun position
   cav::Vec3 rp;  //planet position
} solar_state;
```

```
#endif
```

```
// $Id: vec3.hh,v 1.1 2009/01/25 11:58:14 jsr Exp jsr $
//
//
//
//     printing of vector modified to space seperated with no
   surrounding brackets
//




#ifndef cavlib_vector_hh
#define cavlib_vector_hh

// Define the Vec3 class

// A Vec3 is a vector in three-space. The implementation is all in
// this header file and therefore will be *inlined* for efficiency
   by
// the compiler (if it can...)

#include <cmath>
#include <iostream>

namespace cav {

    class Vec3
    {
    private:

        // The only data needed are three double precision values
            x,y,z:

        double m_x[3];

    public:

        // Construction from x y z values:

        Vec3( double x, double y, double z )
        {
            m_x[0] = x;
            m_x[1] = y;
            m_x[2] = z;
        }

        // Allow empty constructor:

        Vec3( )
        {
            m_x[0] = 0.0;
            m_x[1] = 0.0;
            m_x[2] = 0.0;
        }

        // Destruction: this is not necessary for this class, but we
            provide
        // an empty destructor anyway as an example

        ~Vec3() {}

        // assignment of one vector to another:

        Vec3& operator=( const Vec3& t )
        {
            m_x[0] = t.x();
            m_x[1] = t.y();
            m_x[2] = t.z();
            return *this;
        }

        // copy constructor:

        Vec3( const Vec3& t ) { *this = t; }

        // Data access

        // x() y() z() are const, so cannot be used to modify the
            values:

        double x() const { return m_x[0]; }
        double y() const { return m_x[1]; }
        double z() const { return m_x[2]; }

        // ... but one can change values via [] which returns a
            reference.
        // Note - no bounds checking is done.

        double& operator[]( int i ) { return m_x[i]; }

        // Length of vectors:
```

```
double len2() const
{
    return m_x[0] * m_x[0] + m_x[1] * m_x[1] + m_x[2] *
        m_x[2];
}

double len() const { return sqrt( len2() ); }

// Output:

void print( std::ostream& os ) const
{
    os << m_x[0] << ' ' << m_x[1] << ' ' << m_x[2];
}

// Normalisation, in place:

Vec3& normalise() { return *this /= len(); }

// Unary operators:

Vec3& operator/=( double f )
{
    m_x[0] /= f;
    m_x[1] /= f;
    m_x[2] /= f;
    return *this;
}

Vec3& operator*=( double f )
{
    m_x[0] *= f;
    m_x[1] *= f;
    m_x[2] *= f;
    return *this;
}

Vec3& operator+=( const Vec3& v )
{
    m_x[0] += v.x();
    m_x[1] += v.y();
    m_x[2] += v.z();
    return *this;
}

Vec3& operator-=( const Vec3& v )
```

```
    {
        m_x[0] -= v.x();
        m_x[1] -= v.y();
        m_x[2] -= v.z();
        return *this;
    }

}; // end of Vec3 class

// I/O of Vec3 objects:

inline std::ostream& operator<<( std::ostream& os, const Vec3&
    a )
{
    a.print( os );
    return os;
}

// Inline non-member function utilities for Vec3 objects:

// unary minus negates the vector:

inline Vec3 operator-( const Vec3& v )
{
    return Vec3( -v.x(), -v.y(), -v.z() );
}

// Addition of vectors:

inline Vec3 operator+( const Vec3& v1, const Vec3& v2 )
{
    return Vec3( v1.x() + v2.x(), v1.y() + v2.y(), v1.z() +
        v2.z() );
}

// Subtraction:

inline Vec3 operator-( const Vec3& v1, const Vec3& v2 )
{
    return Vec3( v1.x() - v2.x(), v1.y() - v2.y(), v1.z() -
        v2.z() );
}

// Scaling:

inline Vec3 operator*( const Vec3& v, double f )
{
    return Vec3( f * v.x(), f * v.y(), f * v.z() );
}

inline Vec3 operator*( double f, const Vec3& v )
{
    return Vec3( f * v.x(), f * v.y(), f * v.z() );
}

inline Vec3 operator/( const Vec3& v, double f )
{
    return Vec3( v.x() / f, v.y() / f, v.z() / f );
}

// inner ("dot") product: v1 % v2

inline double operator%( const Vec3& v1, const Vec3& v2 )
{
    return v1.x() * v2.x() + v1.y() * v2.y() + v1.z() * v2.z();
}

// cross product: v1 * v2

inline Vec3 operator*( const Vec3& v1, const Vec3& v2 )
{
    return Vec3( v1.y() * v2.z() - v1.z() * v2.y(),
                 v1.z() * v2.x() - v1.x() * v2.z(),
                 v1.x() * v2.y() - v1.y() * v2.x() );
}

// Return a normalised vector:

inline Vec3 normalise( const Vec3& v )
{
    return v / v.len();
}

// Rotate a vector r *clockwise* about given axis by angle phi:

inline Vec3 rotate( const Vec3& r, const Vec3& rotation_axis,
    double phi )
{
    Vec3 n_hat = normalise( rotation_axis );

    return r * cos( phi ) +
    ( ( (n_hat % r ) * ( 1 - cos(phi) ) ) * n_hat ) -
    ( ( r * n_hat ) * sin(phi) );
}
```

```
} // end namespace cav

#endif
```

```
//
//  solar_state.h
//  trojan_asteroids
//


#ifndef trojan_asteroids_solar_state_h
#define trojan_asteroids_solar_state_h

#include "vec3.h"
#include "constsanddatatypes.h"

solar_state calc_sol_state(const double t, const system_params p);
cav::Vec3 l4_position(double t, system_params p);
double stable_omega(system_params p);
cav::Vec3 corot_pos(double t, system_params p, cav::Vec3 r);
double angle(cav::Vec3 a, cav::Vec3 b, cav::Vec3 c);


#endif
```

```
//
//  solar_state.cpp
//  trojan_asteroids
//  solve the two body problem


#include "solar_state.h"

//calculate angle bac
double angle(cav::Vec3 a, cav::Vec3 b, cav::Vec3 c){
    cav::Vec3 ab = b - a;
    ab.normalise();
    cav::Vec3 ac = c - a;
    ac.normalise();
    return acos(ab%ac);
}
```

```
//orbital periods
double stable_period(system_params p){
    return (2 * pi * pow(p.sep,1.5) * pow(grav_const*(1 +
        p.m),-0.5));
}

double stable_omega(system_params p){
    return 2*pi/stable_period(p);
}


//seperation vector at time t
cav::Vec3 circ_sep_vect(const double t, const system_params p){
    const cav::Vec3 sep0(p.sep,0,0);
    return rotate(sep0,zaxis,stable_omega(p)*t);
}


//calculate and return solar_state for given params and time
solar_state calc_sol_state(const double t, const system_params p){
    solar_state s;
    cav::Vec3 sep_vec = circ_sep_vect(t, p);
    s.rs = sep_vec * (- p.m /(1+p.m));
    s.rp = sep_vec * (1 / (1+p.m));
    return s;
}


cav::Vec3 l4_position(double t, system_params p){
    return rotate(calc_sol_state(t,p).rs + circ_sep_vect(t,p),
        zaxis, pi/3);
}

//convert a r vector to a corotating displacement from l4
cav::Vec3 corot_pos(double t, system_params p, cav::Vec3 r){
    return rotate(r,zaxis,-stable_omega(p)*t);
}
```

```
//
//  gravity.h
//  trojan_asteroids
//

#ifndef trojan_asteroids_gravity_h
#define trojan_asteroids_gravity_h

#include "vec3.h"
```

```
#include "constsanddatatypes.h"
#include "solar_state.h"

cav::Vec3 grav_acc(const solar_state s, const double m, const
    cav::Vec3 r);
double grav_tensor(const cav::Vec3 r, solar_state s, double m, char
    i, char j);
void test_grav_acc();


#endif
```

```
//
//  gravity.cpp
//  trojan_asteroids
//



#include "gravity.h"

#include <iostream>


cav::Vec3 grav_acc(const solar_state s, const double m, const
    cav::Vec3 r){
    cav::Vec3 g(0,0,0);
    cav::Vec3 rfroms = r - s.rs;
    cav::Vec3 rfromp = r - s.rp;
    g = - grav_const * ( rfroms * pow(rfroms.len(),-3) + m * rfromp
        * pow(rfromp.len(),-3) );
    return g;
}


//gravatiational tensor from a single body mass 1 at r=0
double grav_tensor_single(const cav::Vec3 r, const char i, const
    char j){
    double val = 0;

    if (i=='x'){
        val = -3 * r.x();
    } else if (i=='y') {
        val = -3 * r.y();
    } else if (i=='z') {
        val = -3 * r.z();
    } else {
        std::cout << "Error, i is not valid\n";
        return 11111;  //special value, not the best way
    }

    if (i==j) val += r.len2();

    val *= pow(r.len(),-5);

    return val;
}



double grav_tensor(const cav::Vec3 r, solar_state s, double m, char
    i, char j){
    return (-grav_const *(grav_tensor_single((r-s.rs), i, j) +
        m*grav_tensor_single((r-s.rp), i, j)));
}



//test grav_acc
void test_grav_acc(){
    system_params ptest = {0.3,10};
    solar_state s = calc_sol_state(5, ptest);
    cav::Vec3 rtest;

    for (double x=-20; x<=20; x+=1){
        for (double y=-20; y<=20; y+=1){
            rtest = cav::Vec3(x,y,0);

            cav::Vec3 g = grav_acc(s,ptest.m,rtest);
            if(g.len() < 5) std::cout << rtest.x() << ' ' <<
                rtest.y() << ' ' << g.x() << ' ' << g.y() << '\n';
                //avoid really big g near planet and sun
        }
    }

    std::cout << "\nSun in position ";
    s.rs.print(std::cout);
    std::cout << ", planet (mass " << ptest.m << ") in position ";
    s.rp.print(std::cout);
    std::cout << ". Data in format rx,ry,gx,gy.\nTry: \"plot
        'filename' u 1:2:3:4 w vectors head filled, \"<echo ' " <<
        s.rs.x() << ' ' << s.rs.y() << " \\n " << s.rp.x() << ' ' <<
        s.rp.y() << "'\" with points ls 2\"\n";
}
```

```
//
//  funcandjacob.h
//  trojan_asteroids
//


#ifndef trojan_asteroids_func_h
#define trojan_asteroids_func_h


int funcs (double t, const double y[], double f[], void *params);
int jacob (double t, const double y[], double *dfdy, double dfdt[],
    void *params);

#endif
```

```
//
//  funcandjacob.cpp
//  trojan_asteroids
//

#include "funcandjacob.h"

#include "Vec3.h"
#include "solar_state.h"
#include "gravity.h"
#include "constsanddatatypes.h"

#include <gsl/gsl_errno.h>
#include <gsl/gsl_matrix.h>


int funcs (double t, const double y[], double f[], void *params)
{
    system_params p = *(system_params *)params;
    const cav::Vec3 r(y[0],y[1],y[2]);
    const solar_state s = calc_sol_state(t, p);
    const cav::Vec3 g = grav_acc(s, p.m, r);

    f[0] = y[3];
    f[1] = y[4];
    f[2] = y[5];

    f[3] = g.x();
    f[4] = g.y();
    f[5] = g.z();
    return GSL_SUCCESS;
}



int jacob (double t, const double y[], double *dfdy, double dfdt[],
    void *params)
{
    system_params p = *(system_params *)params;
    const cav::Vec3 r(y[0],y[1],y[2]);
    const solar_state s = calc_sol_state(t, p);



    gsl_matrix_view dfdy_mat = gsl_matrix_view_array (dfdy, 6, 6);
        //allow temporary access to save memory

    gsl_matrix * m = &dfdy_mat.matrix;

    gsl_matrix_set_zero(m); //most values are just 0

    gsl_matrix_set (m, 0, 3, 1.0);
    gsl_matrix_set (m, 1, 4, 1.0);
    gsl_matrix_set (m, 2, 5, 1.0);

    gsl_matrix_set (m, 3, 3, grav_tensor(r, s, p.m, 'x', 'x'));
    gsl_matrix_set (m, 3, 4, grav_tensor(r, s, p.m, 'x', 'y'));
    gsl_matrix_set (m, 3, 5, grav_tensor(r, s, p.m, 'x', 'z'));
    gsl_matrix_set (m, 4, 3, grav_tensor(r, s, p.m, 'y', 'x'));
    gsl_matrix_set (m, 4, 4, grav_tensor(r, s, p.m, 'y', 'y'));
    gsl_matrix_set (m, 4, 5, grav_tensor(r, s, p.m, 'y', 'z'));
    gsl_matrix_set (m, 5, 3, grav_tensor(r, s, p.m, 'z', 'x'));
    gsl_matrix_set (m, 5, 4, grav_tensor(r, s, p.m, 'z', 'y'));
    gsl_matrix_set (m, 5, 5, grav_tensor(r, s, p.m, 'z', 'z'));


    dfdt[0] = 0.0;
    dfdt[1] = 0.0;
    dfdt[2] = 0.0;
    dfdt[3] = 0.0;
    dfdt[4] = 0.0;
    dfdt[5] = 0.0;

    return GSL_SUCCESS;
}
```

```
//
//  simulation.h
//  trojan_asteroids
//

#ifndef trojan_asteroids_simulation_h
#define trojan_asteroids_simulation_h
#include "vec3.h"

double simulate(double m, double planet_sep, double t0, double t1,
    cav::Vec3 r0, cav::Vec3 v0, int no_outs, bool bound_only, bool
    output);


#endif
```

```
//
//  simulation.cpp
//  trojan_asteroids
//

#include <iostream>
#include "Vec3.h"
#include "solar_state.h"
#include "funcandjacob.h"
#include "simulation.h"

#include <gsl/gsl_errno.h>
#include <gsl/gsl_odeiv2.h>


//do the simulation. Return -1 if gets too close or too far away
    from planet
double simulate(double m, double planet_sep, double t0, double t1,
    cav::Vec3 r0, cav::Vec3 v0, int no_outs, bool bound_only, bool
    output){
    double t = t0;
    system_params params = {m,planet_sep};
    gsl_odeiv2_system sys = {funcs, jacob, 6, &params};
    gsl_odeiv2_driver * d = gsl_odeiv2_driver_alloc_y_new (&sys,
        gsl_odeiv2_step_rkf45, 1e-7, 1e-9, 0.0);
    double y[6] = { r0.x(), r0.y(), r0.z(), v0.x(), v0.y(), v0.z()
        };

    double maxangle = 0;
    double minangle = pi/3;
    for (int i = 1; i <= no_outs; i++)
    {
        double ti = i * t1 / double(no_outs);
        int status = gsl_odeiv2_driver_apply (d, &t, ti, y);

        if (status != GSL_SUCCESS)
        {
            std::cout << "Error: " << status;
            break;
        }

        //handle outputs
        cav::Vec3 r(y[0],y[1],y[2]);
        cav::Vec3 v(y[3],y[4],y[5]);
        solar_state s = calc_sol_state(t, params);
        cav::Vec3 R = corot_pos(t, params, r);
        double ang = angle({0,0,0},R,{1,0,0});

        if ((ang > 2*pi/3 - 0.05 || ang < 0.05) && bound_only)
            return -1; //return false and stop running if escapes
            from stable position
        if (maxangle < ang) maxangle = ang; //use these to calculate
            angle range
        if (minangle > ang) minangle = ang;

        if(output) std::cout << ti << ' ' << r << ' ' << v << ' ' <<
            R << ' ' << (R-l4_position(t, params)).len() << ' ' <<
            ang-pi/3 << '\n';
    }

    gsl_odeiv2_driver_free (d);
    return (maxangle - minangle);
}
```

```
//
//  pert.h
//  trojan_asteroids
//

#ifndef trojan_asteroids_pert_h
#define trojan_asteroids_pert_h


double simulate_perterbation_from_l4(double planet_m, double
    planet_sep, double final_time, cav::Vec3 r1, cav::Vec3 v1, int
    no_outs, bool output);
double maximum_wander(double m, bool out);
double pert_to_angle(double angle, double m, double sep, bool out,
    double maxt);


#endif
```

```
//
//  pert.cpp
//  trojan_asteroids
//


#include "simulation.h"
#include "Vec3.h"
#include "solar_state.h"




//simulate a steady state solution with small perterbations r1 and
    v1
//such that v = v0 + v1 and r = r0 + r1
double simulate_perterbation_from_l4(double planet_m, double
    planet_sep, double final_time, cav::Vec3 r1, cav::Vec3 v1, int
    no_outs, bool output){
    cav::Vec3 r0 = l4_position(0,{planet_m,planet_sep});
    double modv0 = stable_omega({planet_m,planet_sep})*planet_sep;
    cav::Vec3 v0 = rotate({0,modv0,0},zaxis,pi/3);

    return
        simulate(planet_m,planet_sep,0.0,final_time,r0+r1,v0+v1,no_outs,1,output);
}


double pert_to_angle(double angle, double m, double sep, bool out,
    double maxt){
    double steps = 5000;

    cav::Vec3 r0 = rotate(l4_position(0,{m,sep}),zaxis,angle);
    double modv0 =
        stable_omega({m,sep})*calc_sol_state(0,{m,sep}).rp.len();
    cav::Vec3 v0 = rotate({0,modv0,0},zaxis,pi/3);

    return simulate(m, sep, 0, maxt, r0, v0, steps,1, out);
}
```

```
//
```

```
//  main.cpp
//  trojan_asteroids
//


#include <iostream>
#include "Vec3.h"
#include "solar_state.h"
#include "pert.h"
#include "gravity.h"
#include "simulation.h"




int main (int argc, char* argv[])
{

    if(argc>2) return 1;
    if (argc==1) {
        std::cout << "Program takes inputs direct from command
            line:\n'g': test grav field (outputs some text as well
            as data)\n'2': no planet y-z orbit\n'j': unperturbed
            jupiter\n'z': z perturbation jupiter\n'p': angularly
            perturbed jupiter\n'w': Find range of wander\nExample
            usage: './trojan_asteroids j > ./data.txt'\n";
        return 0;
    }

    switch (*argv[1]) {
        case 'g': test_grav_acc(); break;
        case '2':
            simulate(0,1,0,50,{0,1,0},{0,0,stable_omega({0.0,1})},1000,0,1);
            break;
        case 'j':
            simulate_perterbation_from_l4(0.001,5.2,500,{0,0,0},{0,0,0},10000,1);
            break;
        case 'z':
            simulate_perterbation_from_l4(0.001,5.2,500,{0,0,0},{0,0,0.1},10000,1);break;
        case 'p': pert_to_angle(0.15, 0.001, 5.2, 1, 500); break;

        case 'w': {
            for (double m=0.0; m<0.03; m+=0.00005){
                std::cout << m << ' ' << pert_to_angle(pow(2,-10),
                    m, 1, 0, 80) << '\n';
            }
        }
    }
    return 0;
}
```

References

[1] http://wmap.gsfc.nasa.gov/media/ContentMedia/lagrange.pdf

[2] radley W. Carroll, Dale A. Ostlie. An introduction to modern astrophysics. 2nd edition. Pearson 2007.