Chaotic Magnetic Spinner Toy Dynamics

Jimmy Corno*
School of Physics
Georgia Institute of Technology,
Atlanta, GA 30332-0430, U.S.A
(Dated: December 11, 2003)

A numerical model of the magnetron desk toy is used to study periodic orbits and fixed points.

PACS numbers:
Keywords: periodic orbits, chaos, turbulence, magnetron

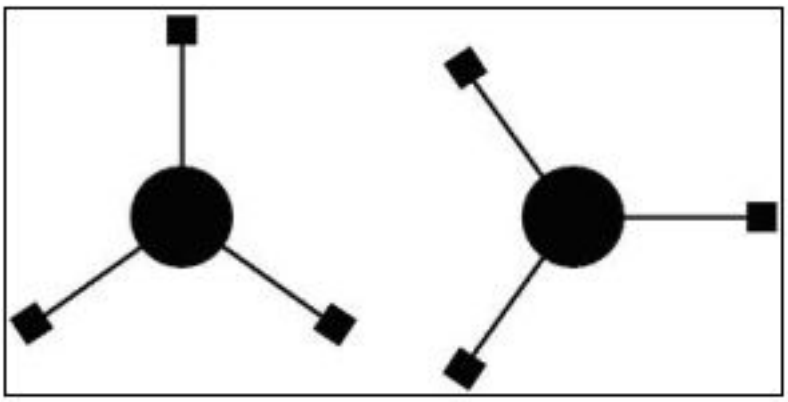
Georgia Tech PHYS 7123:
CHAOS, AND WHAT TO DO ABOUT IT
course project, fall semester 2003

I. INTRODUCTION

My project is an investigation of an interesting but probably unimportant dynamical system. The system appears to be chaotic, though it has a few easy-to-find fixed points and (possibly) stable orbits. It is, unfortunately for me, all original work, as I have been unable to find similar work cited anywhere.
Although I found some interesting results, my confidence in those results is very low. There were some issues with the system that will be discussed in secion IIID.

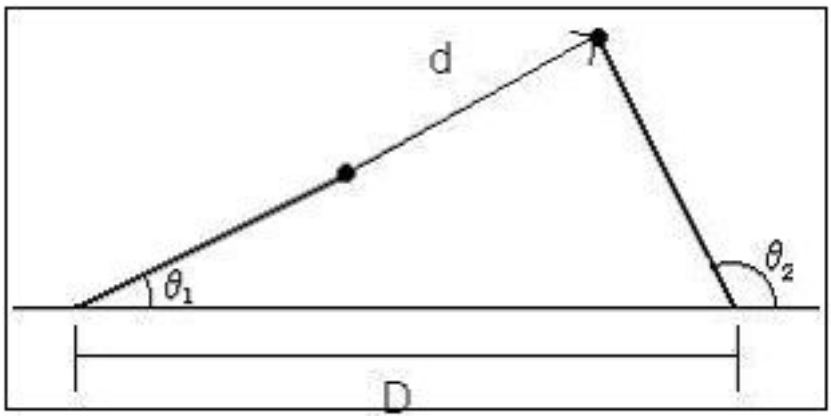
II. THE PROBLEM DEFINED

The magnetron consists of two rotors, each with three arms, lined up on a board. They rotate in the plane without friction. At the end of each arm of each rotor is a bar magnet whose moment is lined up with the arm. As the rotors spin, the bar magnets interact with each other to transfer momentum from one rotor to the other.



The really interesting interactions occur as two magnets pass at the closest point. Changes in the motion of the system can occur very rapidly in this region due to the strong repulsive force of two magnets facing each other.

*Electronic address: James.Corno_at_physics.gatech.edu



A. The Equations

For the purpose of simplicity, the bar magnets are modelled as point magnetic dipoles and friction is ignored. The magnetic field at $\vec{r}$ due to a point magnetic dipole at $\vec{r_0}$ is given by

$$\vec{B}(\vec{r}) = \frac{\mu_0 m}{4\pi}\left[\frac{3\hat{n}(\hat{n}\cdot\hat{m})-\hat{m}}{|\vec{r}-\vec{r_0}|^3}\right] \tag{1}$$

where $\hat{n} = \frac{\vec{r}-\vec{r_0}}{|\vec{r}-\vec{r_0}|}$ and $\vec{r} \neq \vec{r_0}$ (the dipoles can't overlap).
The force on a dipole due to an external magnetic field is given by

$$\vec{F} = (\vec{m}\cdot\vec{\nabla})\vec{B} \tag{2}$$

and the separation of the two dipoles in the diagram is given by

$$\vec{d} = (D + x_2 - x_1)\hat{x} + (y_2 - y_1)\hat{y} \tag{3}$$

so the force on dipole 2 due to dipole 1 is given by

$$F_{2x} = \frac{3\mu_0 m^2}{4\pi}\left[\frac{2x_1x_2d_x + x_2(\vec{d}\cdot\hat{m}_1) + y_1y_2d_x + x_1y_2d_y}{|\vec{d}|^5} - \frac{5(\vec{d}\cdot\vec{m}_1)(x_2d_x^2 + y_2d_xd_y)}{|\vec{d}|^7}\right] \tag{4}$$

and

$$F_{2y} = \frac{3\mu_0 m^2}{4\pi}\left[\frac{2y_1y_2d_y + y_2(\vec{d}\cdot\hat{m}_1) + x_1x_2d_y + y_1x_2d_x}{|\vec{d}|^5} - \frac{5(\vec{d}\cdot\vec{m}_1)(y_2d_y^2 + x_2d_xd_y)}{|\vec{d}|^7}\right] \tag{5}$$

where I have used $x_1 = cos(\theta_1)$. This force must be calulated nine times for each torque calculation (once for each pair of interacting magnets). These forces create torques

$$\tau_{1n} = -x_{1n}(\vec{F}_{1n_{on}21} + \vec{F}_{1n_{on}22} + \vec{F}_{1n_{on}23})_y + y_{1n}(\vec{F}_{1n_{on}21} + \vec{F}_{1n_{on}22} + \vec{F}_{1n_{on}23})_x \tag{6}$$

where $x_{11} = cos(\theta_1)$, $y_{12} = sin(\theta_1 + \frac{2}{3}\pi)$, etc. So, using $\tau = I\ddot{\theta}$, we get

$$\dot{\theta_1} = \omega_1 \tag{7}$$

$$\dot{\omega_1} = \frac{1}{I}(\tau_{11} + \tau_{12} + \tau_{13}). \tag{8}$$

III. IMPLEMENTATION OF MY METHOD

The equations are very long and ugly, but they are not terribly difficult to implement. It helps to calculate the force in cartesion coordinates and switch to polar coordinates for the equations of motion (as should be evident from the way the equations are written). The load is also lightened with the use of some simple trig substitutions for the positions of the magnets (which I did), and small angle approximations could be used for the integration (which I did not do).

A. Numerical implementation

I used a Runge-Kutta-Fehlberg 5th-order algorithm for my integration (Fehlberg because the system was written to allow adaptive step sizes). Nothing fancy, just the four coupled ODE's. Explicit use of energy conservation was not practical, but an energy calculating method was included as an accuracy check. Although the system was written to include an adaptive step method, use of the adaptive step proved to be unreasonably slow.

B. Initialization of the search

Three of the fixed points are easily found by symmetry. Another fixed point can be observed in the toy itself, as it is the rest position of the rotors. The numerical value of its position can be found by including friction in the system (I used an unrealistic drag force proportional to angular velocity). There is also one (apparently) stable periodic orbit that can easily be observed in the toy. That one is discussed in sec:triumph.

C. Desymmetrization

My system has plenty of symmetry, but nothing I was able to utilize. The bar magnets are all identical, so the rotors are symmetrical under rotations of $\frac{2}{3}\pi$. It should therefore be possible to reduce the system to a fundamental domain where $\theta = -\frac{2}{3}\pi \cdots \frac{2}{3}\pi$, but I do not see any benefit in doing that.

D. Numerical troubles

Despite my best efforts at fixing it, the program has some serious problems. First, the unstable fixed points are not fixed; the system drifts away from them every time. But most importantly, energy is not conserved. The problem is particularly evident when the potential energy is comparable to the kinetic energy. In that case, the energy can vary by more than 10%. The problem is definitely not in the energy calculator; that is the simplest part of the program. The step size is also not a problem; the error doesn't change when the step size is changed. That leaves only the implementation of the forces or the integrator, and I know the integrator works. Yet I was unable to find any errors in the equations, even after rederiving them several times. I am completely stumped. My only recourse may be to reformulate the equations with Hamiltonian or Lagrangian mechanics, but it was not possible with my time constraints.

IV. THE HOUR OF TRIUMPH

The program apparently models the system well, with the exception of my energy problem and the lack of a realistic friction force. The behavior of the model mimics the behavior of the toy for the most part. The stable fixed points are there, though they may not be in the correct places. I also believe I have found a very stable periodic orbit (besides those around the stable fixed points). It can be observed in the toy itself with a little manipulation. When the first rotor is spinning with any decent angular velocity and the second rotor is near $\theta = 0$, the second rotor will oscillate slightly while the first rotor spins almost undisturbed. It is very stable motion for small oscillations; I have never seen it deviate, even after thousands of revolutions. I even got a very pretty Poincare map.
My Poincare section here was $\theta_1 = 0$, which was also the starting condition. It is interesting to note that the map is nice even for huge variations in θ_2. At that point it wasn't even oscillating anymore. To look at the raw data, it

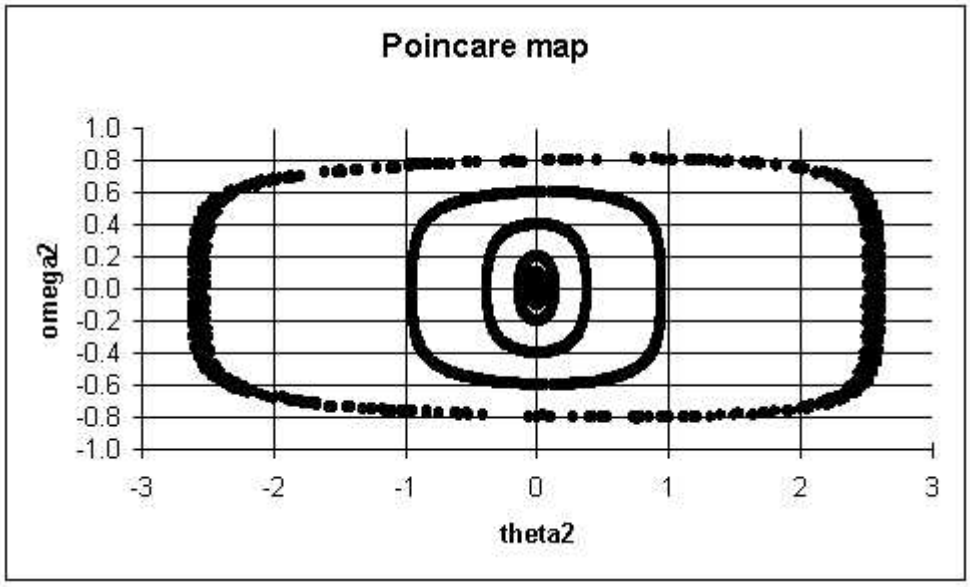


FIG. 1: Poincare map of my stable periodic orbit

appeared that the motion was random. This would be a beautiful result if I had any confidence in the program. Though energy was largely conserved, it was dominated by kinetic energy, so the whole thing could still be way off.

V. DISCUSSION

The ultimate goal of the project was to use a working numerical model of the toy to study the chaotic dynamics. I came nowhere near this goal, of course, as I do not have a working model of the toy. It is all very depressing. My only hope is that with some free time in the next few months I can work it out and find something interesting.

Acknowledgments

I would like to thank Greg Huber for some useful physical insights into the problem.

APPENDIX A: MAGNETRON.JAVA

```
/*Jimmy Corno
 *December 10, 2003
 *Stat Mech 2
 *
 *Runge-Kutta-Fehlberg 4th-5th order integrator for the magnetic spinner toy
 *("magnetron").  The motion is all in terms of the angles of the first magnet
 *on each rotor (theta1 and theta2).
 */

import java.io.*;

class Magnetron {

    static double theta1, theta2, omega1, omega2;   //Angles and angular
                                                    //velocities

    static double x11, x12, x13, x21, x22, x23;     //Cartesian positions of
    static double y11, y12, y13, y21, y22, y23;     //magnets

    static double theta1tmp, theta2tmp, omega1tmp,  //Temporary variables for
                  omega2tmp;                        //integrator use
    static double[] omegaPrime = new double[2];

    static double R, D, m, I, gamma;                //Constants
                                                    //R=rotor arm length
                                                    //D=rotor spacing as a
                                                    //fraction of R
                                                    //m=dipole moment
                                                    //I=rotor moment of inertia
                                                    //gamma=drag coefficient

    static double C = 1.0e-7;                       //mu0/4Pi

    static double d11to21, d11to22, d11to23;        //magnet distances
    static double d12to21, d12to22, d12to23;
    static double d13to21, d13to22, d13to23;

    static double[] F_11on21 = new double[2];       //Interaction forces
    static double[] F_11on22 = new double[2];
    static double[] F_11on23 = new double[2];
    static double[] F_12on21 = new double[2];
    static double[] F_12on22 = new double[2];
    static double[] F_12on23 = new double[2];
    static double[] F_13on21 = new double[2];
    static double[] F_13on22 = new double[2];
    static double[] F_13on23 = new double[2];

    static double[] B_11at21 = new double[2];       //Magnetic fields
    static double[] B_11at22 = new double[2];
    static double[] B_11at23 = new double[2];
    static double[] B_12at21 = new double[2];
    static double[] B_12at22 = new double[2];
    static double[] B_12at23 = new double[2];
    static double[] B_13at21 = new double[2];
    static double[] B_13at22 = new double[2];
    static double[] B_13at23 = new double[2];
```

```
static double h;                                //time step

static double f0, f1, f2, f3, f4, f5;           //derivatives
static double g0, g1, g2, g3, g4, g5;
static double j0, j1, j2, j3, j4, j5;
static double k0, k1, k2, k3, k4, k5;

/*These doubles will be used to avoid the use of all but 4 sines and
 *cosines with every step, since the magnets have a fixed angular
 *separation.*/
static double sin120 = Math.sin(Math.PI*2.0/3.0);
static double cos120 = Math.cos(Math.PI*2.0/3.0);
static double sin240 = -sin120;
static double cos240 = cos120;
static double time, stopTime;

static int n = 0;                           //step count
static int stopN;


public static void main( String[] Args ) {

    h = 0.001;

    theta1 = 0.0;           //Intial conditions
    theta2 = Math.PI;
    omega1 = 0.00001;
    omega2 = 0.0;

    R = 1.0;                //System constants
    D = 2.05;
    m = 200.0;
    I = 0.08;
    gamma = 0.0;

    time = 0.0;
    stopTime = 1.0;
    stopN = 60000;

    System.out.println( "Ei = " + energy(theta1, theta2, omega1, omega2));

    FileOutputStream out;   // declare a file output object
    PrintStream p;          // declare a print stream object

    try{                    // Create a new file output stream
                            // connected to "myfile.txt"
        out = new FileOutputStream("myfile.txt");

        // Connect print stream to the output stream
        p = new PrintStream( out );


        do {
          //  if ( !retry && (n%1000 == 0) ) {
          //      p.println( time + "," + omega2 + ","
          //                          + theta2 );
          //  }
```

```
            advance();

            if( n%100 < h ) {
                System.out.println(time + "\t" + theta2);
            }

        } while ( (time < stopTime) );

    }

    catch (Exception e){
        System.err.println ("Error writing to file");
    }

    System.out.println( "Ef = " + energy(theta1, theta2, omega1, omega2));

    //System.out.println( "n = " + n );
    //System.out.println( "t = " + time );

}

//Advance is the integrator method, uses RK4-5 adaptive step method

public static void advance() {

    theta1tmp = theta1;
    theta2tmp = theta2;
    omega1tmp = omega1;
    omega2tmp = omega2;

    f0 = omega1tmp;
    g0 = omega2tmp;
    omegaPrime = alpha( theta1tmp, theta2tmp, omega1tmp, omega2tmp );
    j0 = omegaPrime[0];
    k0 = omegaPrime[1];

    theta1tmp = theta1 + b10*h*f0;
    theta2tmp = theta2 + b10*h*g0;
    omega1tmp = omega1 + b10*h*j0;
    omega2tmp = omega2 + b10*h*k0;

    f1 = omega1tmp;
    g1 = omega2tmp;
    omegaPrime = alpha( theta1tmp, theta2tmp, omega1tmp, omega2tmp );
    j1 = omegaPrime[0];
    k1 = omegaPrime[1];

    theta1tmp = theta1 + b20*h*f0 + b21*h*f1;
    theta2tmp = theta2 + b20*h*g0 + b21*h*g1;
    omega1tmp = omega1 + b20*h*j0 + b21*h*j1;
    omega2tmp = omega2 + b20*h*k0 + b21*h*k1;

    f2 = omega1tmp;
    g2 = omega2tmp;
    omegaPrime = alpha( theta1tmp, theta2tmp, omega1tmp, omega2tmp );
    j2 = omegaPrime[0];
    k2 = omegaPrime[1];
```

```
    theta1tmp = theta1 + b30*h*f0 + b31*h*f1 + b32*h*f2;
    theta2tmp = theta2 + b30*h*g0 + b31*h*g1 + b32*h*g2;
    omega1tmp = omega1 + b30*h*j0 + b31*h*j1 + b32*h*j2;
    omega2tmp = omega2 + b30*h*k0 + b31*h*k1 + b32*h*k2;

    f3 = omega1tmp;
    g3 = omega2tmp;
    omegaPrime = alpha( theta1tmp, theta2tmp, omega1tmp, omega2tmp );
    j3 = omegaPrime[0];
    k3 = omegaPrime[1];

    theta1tmp = theta1 + b40*h*f0 + b41*h*f1 + b42*h*f2 + b43*h*f3;
    theta2tmp = theta2 + b40*h*g0 + b41*h*g1 + b42*h*g2 + b43*h*g3;
    omega1tmp = omega1 + b40*h*j0 + b41*h*j1 + b42*h*j2 + b43*h*j3;
    omega2tmp = omega2 + b40*h*k0 + b41*h*k1 + b42*h*k2 + b43*h*k3;

    f4 = omega1tmp;
    g4 = omega2tmp;
    omegaPrime = alpha( theta1tmp, theta2tmp, omega1tmp, omega2tmp );
    j4 = omegaPrime[0];
    k4 = omegaPrime[1];

    theta1tmp = theta1 + b50*h*f0 + b51*h*f1 + b52*h*f2 + b53*h*f3
                + b54*h*f4;
    theta2tmp = theta2 + b50*h*g0 + b51*h*g1 + b52*h*g2 + b53*h*g3
                + b54*h*g4;
    omega1tmp = omega1 + b50*h*j0 + b51*h*j1 + b52*h*j2 + b53*h*j3
                + b54*h*j4;
    omega1tmp = omega1 + b50*h*k0 + b51*h*k1 + b52*h*k2 + b53*h*k3
                + b54*h*k4;

    f5 = omega1tmp;
    g5 = omega2tmp;
    omegaPrime = alpha( theta1tmp, theta2tmp, omega1tmp, omega2tmp );
    j5 = omegaPrime[0];
    k5 = omegaPrime[1];


    theta1 = theta1 + h * ( c0*f0 + c2*f2 + c3*f3 + c4*f4 + c5*f5 );
    theta2 = theta2 + h * ( c0*g0 + c2*g2 + c3*g3 + c4*g4 + c5*g5 );
    omega1 = omega1 + h * ( c0*j0 + c2*j2 + c3*j3 + c4*j4 + c5*j5 );
    omega2 = omega2 + h * ( c0*k0 + c2*k2 + c3*k3 + c4*k4 + c5*k5 );

    time = time + h;

    n++;

}

//This double calculates interaction forces

public static double[] F_AonB (double xA, double yA, double xB, double yB) {

    double[] F = new double[2];
    double dx = D + xB - xA;
    double dy = yB - yA;
    double dist = Math.sqrt(dx*dx + dy*dy);
```

```
    F[0] = 3.0*C*m*m*( (2.0*xA*xB*dx + xB*(dx*xA + dy*yA) + yA*yB*dx
            + xA*yB*dy)/Math.pow(dist, 5.0) - 5.0*(dx*xA + dy*yA)
            *(xB*dx*dx + yB*dx*dy)/Math.pow(dist, 7.0));

    F[1] = 3.0*C*m*m*( (2.0*yA*yB*dy + yB*(dx*xA + dy*yA) + xA*xB*dy
            + xB*yA*dx)/Math.pow(dist, 5.0) - 5.0*(dx*xA + dy*yA)
            *(yB*dy*dy + xB*dx*dy)/Math.pow(dist, 7.0));

    return F;

}


//This double calculates angular acceleration.
public static double[] alpha ( double Theta1, double Theta2, double Omega1,
                                    double Omega2) {

    double[] Alpha = new double [2];            //Angular acceleration
                                                //returned to the advance()
                                                //function.

    x11 = Math.cos( Theta1 );
    y11 = Math.sin( Theta1 );
    x21 = Math.cos( Theta2 );
    y21 = Math.sin( Theta2 );

    x12 = x11*cos120 - y11*sin120;
    x13 = x11*cos240 - y11*sin240;
    y12 = y11*cos120 + x11*sin120;
    y13 = y11*cos240 + x11*sin240;

    x22 = x21*cos120 - y21*sin120;
    x23 = x21*cos240 - y21*sin240;
    y22 = y21*cos120 + x21*sin120;
    y23 = y21*cos240 + x21*sin240;

    F_11on21 = F_AonB( x11, y11, x21, y21 );
    F_11on22 = F_AonB( x11, y11, x22, y22 );
    F_11on23 = F_AonB( x11, y11, x23, y23 );
    F_12on21 = F_AonB( x12, y12, x21, y21 );
    F_12on22 = F_AonB( x12, y12, x22, y22 );
    F_12on23 = F_AonB( x12, y12, x23, y23 );
    F_13on21 = F_AonB( x13, y13, x21, y21 );
    F_13on22 = F_AonB( x13, y13, x22, y22 );
    F_13on23 = F_AonB( x13, y13, x23, y23 );

    Alpha[0] = -x11 * ( F_11on21[1] + F_11on22[1] + F_11on23[1] ) +
                y11 * ( F_11on21[0] + F_11on22[0] + F_11on23[0] ) -
                x12 * ( F_12on21[1] + F_12on22[1] + F_12on23[1] ) +
                y12 * ( F_12on21[0] + F_12on22[0] + F_12on23[0] ) -
                x13 * ( F_13on21[1] + F_13on22[1] + F_13on23[1] ) +
                y13 * ( F_13on21[0] + F_13on22[0] + F_13on23[0] );

    Alpha[1] =  x21 * ( F_11on21[1] + F_12on21[1] + F_13on21[1] ) -
                y21 * ( F_11on21[0] + F_12on21[0] + F_13on21[0] ) +
                x22 * ( F_11on22[1] + F_12on22[1] + F_13on22[1] ) -
                y22 * ( F_11on22[0] + F_12on22[0] + F_13on22[0] ) +
```

```
                x23 * ( F_11on23[1] + F_12on23[1] + F_13on23[1] ) -
                y23 * ( F_11on23[0] + F_12on23[0] + F_13on23[0] );

    Alpha[0] = Alpha[0]/I - gamma*Omega1;       //Alpha calculations were
                                                //actually torques.
    Alpha[1] = Alpha[1]/I - gamma*Omega2;

    return Alpha;

}


//Field vectors are used only for energy calculations
public static double[] B_1at2 ( double xA, double yA, double xB, double yB ) {

    double[] B = new double[2];
    double dx = D + xB - xA;
    double dy = yB - yA;
    double dist = Math.sqrt(dx*dx + dy*dy);

    B[0] = C*m*( 3.0 * dx * (dx*xA + dy*yA)/Math.pow(dist, 5.0)
                      - xA/Math.pow(dist, 3.0) );
    B[1] = C*m*( 3.0 * dy * (dx*xA + dy*yA)/Math.pow(dist, 5.0)
                      - yA/Math.pow(dist, 3.0) );

    return B;

}

public static double energy ( double t1, double t2, double o1, double o2 ) {

    double U21, U22, U23, KE;

    x11 = Math.cos( t1 );
    y11 = Math.sin( t1 );
    x21 = Math.cos( t2 );
    y21 = Math.sin( t2 );

    x12 = x11*cos120 - y11*sin120;
    x13 = x11*cos240 - y11*sin240;
    y12 = y11*cos120 + x11*sin120;
    y13 = y11*cos240 + x11*sin240;

    x22 = x21*cos120 - y21*sin120;
    x23 = x21*cos240 - y21*sin240;
    y22 = y21*cos120 + x21*sin120;
    y23 = y21*cos240 + x21*sin240;

    B_11at21 = B_1at2( x11, y11, x21, y21 );
    B_11at22 = B_1at2( x11, y11, x22, y22 );
    B_11at23 = B_1at2( x11, y11, x23, y23 );
    B_12at21 = B_1at2( x12, y12, x21, y21 );
    B_12at22 = B_1at2( x12, y12, x22, y22 );
    B_12at23 = B_1at2( x12, y12, x23, y23 );
    B_13at21 = B_1at2( x13, y13, x21, y21 );
    B_13at22 = B_1at2( x13, y13, x22, y22 );
    B_13at23 = B_1at2( x13, y13, x23, y23 );
```

```
        U21 = -m * x21 * ( B_11at21[0] + B_12at21[0] + B_13at21[0] ) -
               m * y21 * ( B_11at21[1] + B_12at21[1] + B_13at21[1] );

        U22 = -m * x22 * ( B_11at22[0] + B_12at22[0] + B_13at22[0] ) -
               m * y22 * ( B_11at22[1] + B_12at22[1] + B_13at22[1] );

        U23 = -m * x23 * ( B_11at23[0] + B_12at23[0] + B_13at23[0] ) -
               m * y23 * ( B_11at23[1] + B_12at23[1] + B_13at23[1] );

        KE = I * ( o1*o1 + o2*o2 )/2.0;

        return U21 + U22 + U23 + KE;

    }


    //These are the coefficients for the integrator.

    static final double a1 = 0.25;
    static final double a2 = 0.375;
    static final double a3 = 12.0 / 13.0;
    static final double a4 = 1.0;
    static final double a5 = 0.5;

    static final double b10 = 0.25;
    static final double b20 = 3 / 32.0;
    static final double b21 = 9 / 32.0;
    static final double b30 = 1932.0 / 2197.0;
    static final double b31 = -7200.0 / 2197.0;
    static final double b32 = 7296.0 / 2197.0;
    static final double b40 = 439.0 / 216.0;
    static final double b41 = -8;
    static final double b42 = 3680.0 / 513.0;
    static final double b43 = -845.0 / 4104.0;
    static final double b50 = -8.0 / 27.0;
    static final double b51 = 2;
    static final double b52 = -3544.0 / 2565.0;
    static final double b53 = 1859.0 / 4104.0;
    static final double b54 = -11.0 / 40.0;

    static final double c0 = 16.0 / 135.0;
    static final double c2 = 6656.0 / 12825.0;
    static final double c3 = 28561.0 / 56430.0;
    static final double c4 = -0.18;
    static final double c5 = 2.0 / 55.0;

    static final double d0 = 1.0 / 360.0;
    static final double d2 = -128.0 / 4275.0;
    static final double d3 = -2197.0 / 75240.0;
    static final double d4 = 0.02;
    static final double d5 = 2.0 / 55.0;

}
```