Double Pendulum

Freddie Witherden

February 10, 2009

Abstract

We report on the numerical modelling of a double pendulum using C++. The system was found to be very sensitive to both the initial starting conditions and the choice of solver.

1 Introduction

A double pendulum, which consists of one pendulum suspended from another, is a potentially *chaotic system*. This means that for certain parameter ranges a slight change in one of the initial starting conditions can have a dramatic effect on the subsequent motion of the pendulum. As a result the motion of a double pendulum extremely difficult to predict — exhibiting seemingly random or *chaotic* behavior.

The motion of a double pendulum can be modeled using a system of *ordinary differential equations*. However, since these equations have no analytical solution they must instead be approximated numerically. This can be done using a computer with an appropriately written program.

2 Theory

As stated in the introduction a double pendulum is one pendulum suspended from another. This can be seen in Figure 1. Assuming that the system is un-damped and that the connecting rod is massless the following equations of motion can be

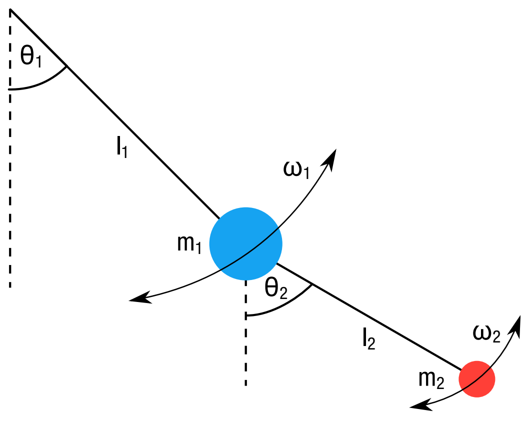


Figure 1: Diagram of a double pendulum.

derived [2, 3]

$$\begin{aligned}
\dot{\theta}_1 &= \omega_1 \\
\dot{\omega}_1 &= \frac{m_2 l_1 \omega_1^2 \sin\Delta\cos\Delta + m_2 g \sin\theta_2 \cos\Delta + m_2 l_2 \omega_2^2 \sin\Delta - M g \sin\theta_1}{M l_1 - m_2 l_1 \cos^2\Delta} \\
\dot{\theta}_2 &= \omega_2 \\
\dot{\omega}_2 &= \frac{-m_2 l_2 \omega_2^2 \sin\Delta\cos\Delta + M(g\sin\theta_1\cos\Delta - l_1\omega_1^2\sin\Delta - g\sin\theta_2)}{M l_2 - m_2 l_2 \cos^2\Delta}
\end{aligned} \tag{1}$$

where $\theta_{1,2}$ is the angle of the bob from the vertical, $\omega_{1,2}$ is the angular momentum of the bob, $l_{1,2}$ is the length of the connecting rod, $m_{1,2}$ is the mass of the bob, $\Delta = \theta_2 - \theta_1$, $M = m_1 + m_2$ and g is acceleration due to gravity.

In order to simulate a double pendulum it is necessary to solve these equations in terms of t. Since there exists no analytical solution it must instead be done numerically — of which there exist several solvers. Considering several methods allows for firstly verification (they should give similar solutions to the same equations) and secondly allows for a comparison to be made between methods. The most simple of these is *Euler's method* which states that given a function $y(t)$ that

$$y(t+\Delta t) = y(t) + \dot{y}(t)\Delta t \tag{2}$$

where Δt is a small time increment and $\dot{y}(t)$ is the derivative of $y(t)$ with respect to time at a time, t. Therefore given the initial starting conditions at $y(0)$ it is possible to calculate $y(t)$. This is easy to generalise in terms of $\theta_{1,2}(t)$ and $\omega_{1,2}(t)$, thus making it possible to model a double pendulum numerically.

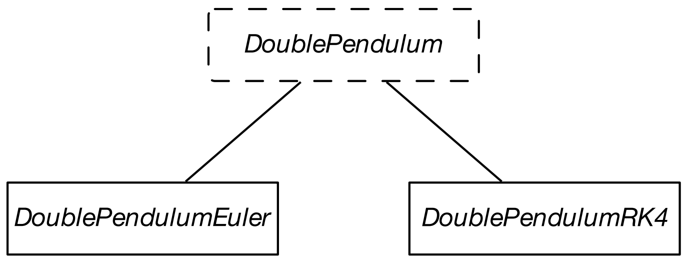


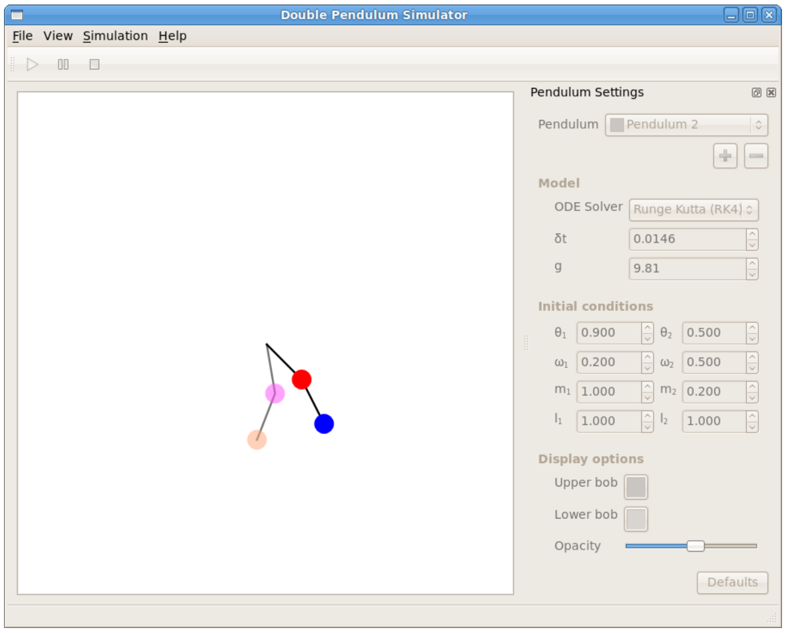
Figure 2: Class inheritance hierarchy for the double pendulum system. `DoublePendulumEuler` makes use of Euler's method for solving the equations of motion while `DoublePendulumRK4` uses a 4th order Runge-Kutta method.

A more advanced method for solving ordinary differential equations is the *Runge-Kutta* method [4]. While Euler's method just computes the derivative once at $y(t)$ Runge-Kutta methods compute the derivative multiple times between $y(t)$ and $y(t+\Delta t)$, using the previous derivative as a starting point. The various derivatives are then combined in a weighted fashion to compute $y(t+\Delta t)$. The most common variant is a *fourth order Runge-Kutta method* which computes the derivative a total of four times for a step Δt. A complete description of the algorithm can be found in [4].

3 Implementation In C++

In order to model the double pendulum system object-orientated C++ was used. An abstract base class, `DoublePendulum`, was used to model the core functionality of a double pendulum system. This contained functions for computing the numerical derivative at a time t and advancing the simulation forward in steps of Δt, while delegating the specific task of solving the equations of motion to subclasses. This can be seen in Figure 2. Doing so both maximised code reuse and made it easy to implement additional solvers.

A graphical fontend was then written around this. By allowing for an arbitrary number of pendulums — each with different solvers/starting conditions — to be added to a *scene* it facilitated a visual comparison. A screen capture of the application can be seen in Figure 3.



Pendulum: Pendulum 2

Model

ODE Solver: Runge Kutta (RK4)

δt: 0.0146

g: 9.81

Initial conditions

θ₁: 0.900 θ₂: 0.500

ω₁: 0.200 ω₂: 0.500

m₁: 1.000 m₂: 0.200

l₁: 1.000 l₂: 1.000

Figure 3: A screen capture of the graphical frontend showing two double pendulums mid-swing.

4 Results, Errors and Discussion

It is possible to investigate the chaotic nature of the system by looking at how a small change to the starting conditions affects the subsequent motion of the pendulum. The effect of $\theta_{1_{init}}$ and $\theta_{2_{init}}$ on the value of θ_2 after 20.0 seconds can be seen in Figure 4. Areas which are consistently shaded are predictable — a small change to $\theta_{1_{init}}$ or $\theta_{2_{init}}$ having little effect on the subsequent motion — while those that are noisy are chaotic.

In order for the results above to be meaningful it is first necessary demonstrate that the algorithms and solvers have been correctly implemented in the program. There are several ways of doing this that do not rely on the assumption that one method is more accurate than another. Firstly, looking at Equation 1 it is clear that

$$\begin{aligned} \lim_{m_2 \to 0} \dot{\theta}_1 &= \dot{\omega}_1 \\ \lim_{m_2 \to 0} \dot{\omega}_1 &= \frac{-Mg\sin\theta_1}{Ml_1} = \frac{-g\sin\theta_1}{l_1} \end{aligned} \tag{3}$$

which are the equations of motion for a single pendulum. For $\theta_1 \ll 1$ it is possible

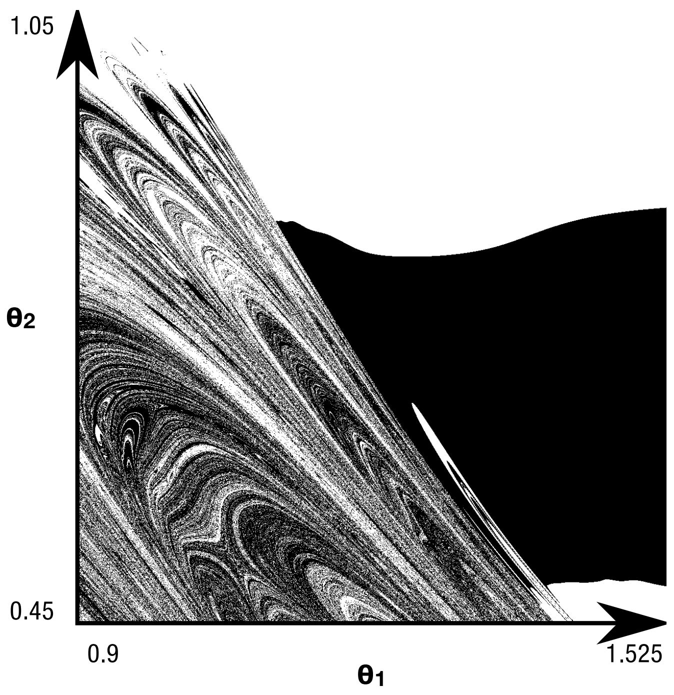


Figure 4: How the value of θ_2 after 20.0 seconds is affected by the initial values of $\theta_{1,2}$. If θ_2 is greater than 0 then a white pixel is plotted while if it is less than zero then black is used. Initial conditions: $\omega_1 = \omega_2 = 0$, $l_1 = 1.0\,\mathrm{m}$, $l_2 = 0.4\,\mathrm{m}$, $m_1 = 1.0\,\mathrm{kg}$ and $m_2 = 0.25\,\mathrm{kg}$.

to solve the equations analytically by using the *small angle approximation* giving *[5]*

$$\theta_1(t) = \theta_{1_{init}} \cos\left(t\sqrt{\frac{l_1}{g}}\right) \tag{4}$$

Hence for a small starting angle θ_1 the motion of the upper-bob should be the same as that predicted by Equation 4. This can be seen in Figure 5. Therefore the program does accurately simulate a single pendulum as m_2 tends to zero.

Another method is to see how the total mechanical energy of the system varies over time. Since gravity is a conservative force and the system is un-damped it should remain constant. A graph of this can be seen in Figure 6. Looking at the graph it is clear that Euler's method is not suitable unless a small Δt is used while the Runge-Kutta method is significantly better at maintaining the total mechanical energy.

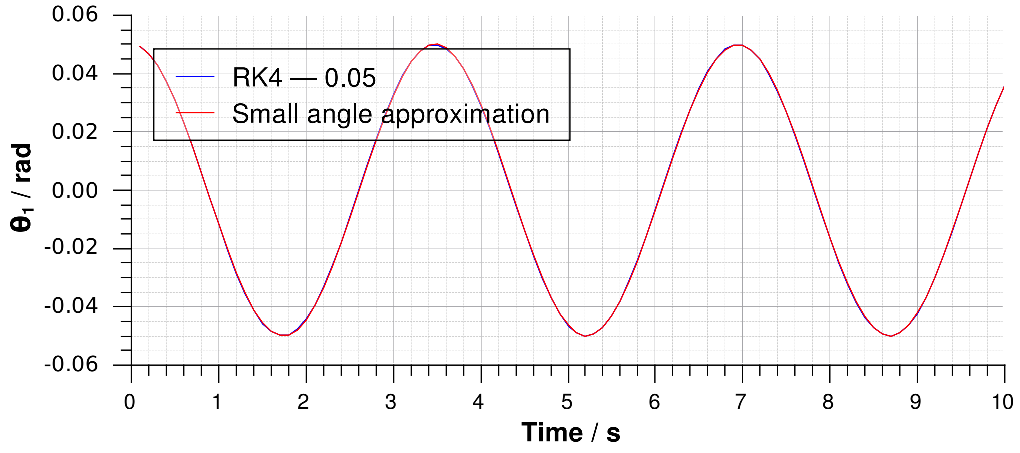


Figure 5: A plot comparing θ against time for a double pendulum with an arbitrary small m_2 and a small-angle approximation. Initial conditions: $\theta_1 = 0.05$, $\omega_1 = 0$, $l_1 = 3.0\,\mathrm{m}$ and $m_1 = 1.0\,\mathrm{kg}$.

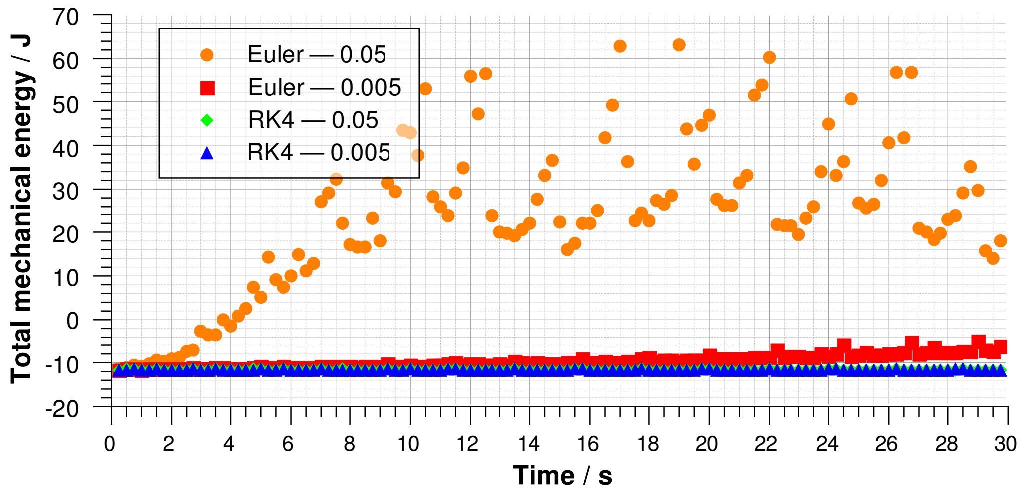


Figure 6: How the total mechanical energy of the system varies over time for various solvers and time steps. Initial conditions: $\theta_1 = \theta_2 = 0.5$, $\omega_1 = \omega_2 = 0$, $l_1 = 1.0\,\mathrm{m}$, $l_2 = 0.4\,\mathrm{m}$, $m_1 = 1.0\,\mathrm{kg}$ and $m_2 = 0.25\,\mathrm{kg}$.

5 Conclusion

The objective was to simulate a double pendulum system using C++ and to investigate both its chaotic behavior and how the simulation is affected by the choice of numerical solver. The system was found to be chaotic in certain parameter ranges, but not universally. The choice of solver and time-step was found to have a big effect on the motion of the system: Euler's method proved to be unsuitable unless a small time-step was used. Fourth-order Runge-Kutta method fared much better — allowing for a much larger time-step to be used than would be possible with Euler's method.

Although with a suitably small time-step the program has the potential to be extremely accurate it is inevitably limited by the machine precision. There is also a significant performance penalty associated with smaller step sizes. One potential solution to this is to investigative *adaptive step-size* Runge-Kutta methods, which use a non-constant step size. By changing the step size dynamically the number of calculations can be reduced, resulting in better performance while obtaining the same accuracy.

References

[1] First Year Computing Laboratory 2008-9, Derek Lee, Imperial College London, 2008.

[2] `http://scienceworld.wolfram.com/physics/DoublePendulum.html`, Double Pendulum, Eric W. Weisstein, last updated 2007.

[3] `http://www.physics.usyd.edu.au/~wheat/dpend_html/`, Double Pendulum, Mike Wheatland, last updated Friday 1st August 2008.

[4] Press W.H. *et al*, 2007, Numerical Recipes, 3e, Cambridge University press, pp907-910.

[5] `http://www.kettering.edu/~drussell/Demos/Pendulum/Pendula.html`, The Simple Pendulum, Daniel A. Russell, Kettering University, accessed Thursday 5th February 2009.

A Program Listing

Since the program itself totals in at over 1,000 lines of C++ code only the core classes, those which simulate the double pendulum, have been attached. The file names correspond those in Figure 2.

doublependulum.h

```cpp
#ifndef DOUBLEPENDULUM_H
#define DOUBLEPENDULUM_H

/**
 * Models the state of a pendulum. This is provided to make it easier to
 * create a DoublePendulum class instance.
 */
struct Pendulum
{
    Pendulum()
        : theta(0.0), omega(0.0), l(0.0), m(0.0)
    {
    }

    Pendulum(double _theta, double _omega, double _l, double _m)
        : theta(_theta), omega(_omega), l(_l), m(_m)
    {
    }

    /**
     * The angle in radians the pendulum makes with the vertical.
     */
    double theta;

    /**
     * The angular momentum of the pendulum.
     */
    double omega;

    /**
     * The length of the connecting rod between the pendulum and the pivot.
     */
    double l;

    /**
     * The mass of the bob on the end of the pendulum.
     */
    double m;
```

```
};

class DoublePendulum
{
public:
    DoublePendulum(const Pendulum& upper, const Pendulum& lower,
                   double dt=0.005, double g=9.81);

    virtual ~DoublePendulum();

    /**
     * Advances the equation in steps of m_dt until newTime is reached.
     *
     * @param newTime   The time to advance the system to. This must be >=
     *                  time().
     */
    void update(double newTime);

    double theta1()
    {
        return m_theta1;
    }

    double omega1()
    {
        return m_omega1;
    }

    double m1()
    {
        return m_m1;
    }

    double l1()
    {
        return m_l1;
    }

    double theta2()
    {
        return m_theta2;
    }

    double omega2()
    {
        return m_omega2;
    }

    double m2()
    {
        return m_m2;
    }

    double l2()
    {
        return m_l2;
    }

    double time()
    {
        return m_time;
    }

    /**
     * Returns a string representation of the solver method used. This can be
     * used to determine which solver is being used by a given instance.
     *
     * @return The name of the solver used by the current DoublePendulum
     *         instance.
     */
    virtual const char *solverMethod() = 0;

protected:
    /**
     * The list of motion ODE which must be solved in order to numerically
     * evaluate the double pendulum system with respect to time.
     */
    enum
    {
        THETA_1,
        OMEGA_1,
        THETA_2,
        OMEGA_2,
        NUM_EQNS
    };

    /**
     * Given theta and omega for the upper- and lower-bobs this method computes
     * the numeric derivatives of each one.
     *
     * The format for yin is: { THETA_1, OMEGA_1, THETA_2, OMEGA_2 }.
     *
     * @param yin   The current values of theta and omega for the system. (As
     *              all other values are constant these do not need to be
     *              passed.)
     * @param dydx  The array to place the computed derivatives in. This has
     *              exactly the same format as yin.
     */
    void derivs(const double *yin, double *dydx);

    /**
     * Called to solve the equations of motion for the system by advancing
     * theta and omega by one step (this->m_dt).
     *
     * Since there are various ways of doing this the exact implementation is
     * left up to sub-classes.
     *
     * @param yin   The current values of theta and omega for the system. The
     *              format is the same as that taken by the derivs method.
     */
    virtual void solveODEs(const double *yin, double *yout) = 0;

    /**
     * Angle of the first pendulum from the vertical (in rad).
     */
    double m_theta1;

    /**
     * Angular acceleration of the first pendulum (dÎž/dt).
     */
    double m_omega1;

    /**
     * Length of the first pendulum (in m).
     */
    const double m_l1;

    /**
     * Mass of the first pendulum (in kg).
     */
    const double m_m1;

    /**
     * Angle of the second pendulum from the vertical (in rad).
     */
    double m_theta2;

    /**
     * Angular acceleration of the second pendulum (dÎž/dt).
     */
    double m_omega2;

    /**
     * Length of the second pendulum (in m).
     */
    const double m_l2;

    /**
     * Mass of the second pendulum (in m).
     */
    const double m_m2;

    /**
     * Step size to take when numerically solving the ODE.
     */
    const double m_dt;

    /**
     * Acceleration due to gravity (usually 9.81 ms^-2).
     */
    const double m_g;

    /**
     * Current time for which omega and theta are evaluated for.
     */
    double m_time;
};

#endif // DOUBLEPENDULUM_H
```

doublependulum.cpp

```
#include "doublependulum.h"

#include <cmath>
#include <cassert>

DoublePendulum::DoublePendulum(const Pendulum& upper, const Pendulum& lower,
                               double dt, double g) :
    m_theta1(upper.theta),
    m_omega1(upper.omega),
    m_l1(upper.l), m_m1(upper.m),
    m_theta2(lower.theta),
    m_omega2(lower.omega),
    m_l2(lower.l), m_m2(lower.m),
    m_dt(dt), m_g(g), m_time(0.0)
{
}

DoublePendulum::~DoublePendulum()
{
}

void DoublePendulum::update(double newTime)
{
    assert(newTime >= m_time);

    do
    {
        const double yin[NUM_EQNS] = { m_theta1, m_omega1, m_theta2, m_omega2 };
        double yout[NUM_EQNS];

        solveODEs(yin, yout);

        m_theta1 = yout[THETA_1];
        m_omega1 = yout[OMEGA_1];
        m_theta2 = yout[THETA_2];
        m_omega2 = yout[OMEGA_2];
    } while ((m_time += m_dt) < newTime);
}

void DoublePendulum::derivs(const double *yin, double *dydx)
{
    // Delta is theta2 - theta1
    const double delta = yin[THETA_2] - yin[THETA_1];

    // ‘Big-M’ is the total mass of the system, m1 + m2;
    const double M = m_m1 + m_m2;

    // Denominator expression for omega1
    double den = M*m_l1 - m_m2*m_l1*cos(delta)*cos(delta);

    // d theta / dt = omega, by definition
    dydx[THETA_1] = yin[OMEGA_1];

    // Compute omega1
    dydx[OMEGA_1] = (m_m2*m_l1*yin[OMEGA_1]*yin[OMEGA_1]*sin(delta)*cos(delta)
                  + m_m2*m_g*sin(yin[THETA_2])*cos(delta)
                  + m_m2*m_l2*yin[OMEGA_2]*yin[OMEGA_2]*sin(delta)
                  - M*m_g*sin(yin[THETA_1])) / den;

    // Again, d theta / dt = omega for theta2 as well
    dydx[THETA_2] = yin[OMEGA_2];

    // Multiply den by the length ratio of the two bobs
    den *= m_l2 / m_l1;

    // Compute omega2
    dydx[OMEGA_2] = (-m_m2*m_l2*yin[OMEGA_2]*yin[OMEGA_2]*sin(delta)*cos(delta)
                  + M*m_g*sin(yin[THETA_1])*cos(delta)
                  - M*m_l1*yin[OMEGA_1]*yin[OMEGA_1]*sin(delta)
                  - M*m_g*sin(yin[THETA_2])) / den;
}
```

doublependulumeuler.h

```
#ifndef DOUBLEPENDULUMEULER_H
#define DOUBLEPENDULUMEULER_H

#include "doublependulum.h"

class DoublePendulumEuler : public DoublePendulum
{
public:
    DoublePendulumEuler(const Pendulum& upper, const Pendulum& lower,
                        double dt=0.05, double g=9.81);

    const char *solverMethod();

    void solveODEs(const double *yin, double *yout);
};

#endif // DOUBLEPENDULUMEULER_H
```

doublependulumeuler.cpp

```
#include "doublependulumeuler.h"

DoublePendulumEuler::DoublePendulumEuler(const Pendulum& upper,
                                         const Pendulum& lower,
                                         double dt, double g):
    DoublePendulum(upper, lower, dt, g)
{
}

const char *DoublePendulumEuler::solverMethod()
{
    return "Euler";
}

void DoublePendulumEuler::solveODEs(const double *yin, double *yout)
{
    double dydx[NUM_EQNS];

    // Calculate the derivatives of the equations at time + dt
    derivs(yin, dydx);

    // Update the values of theta and omega for the two bobs
    yout[THETA_1] = yin[THETA_1] + dydx[THETA_1] * m_dt;
    yout[OMEGA_1] = yin[OMEGA_1] + dydx[OMEGA_1] * m_dt;

    yout[THETA_2] = yin[THETA_2] + dydx[THETA_2] * m_dt;
    yout[OMEGA_2] = yin[OMEGA_2] + dydx[OMEGA_2] * m_dt;
}
```

doublependulumrk4.h

```
#ifndef DOUBLEPENDULUMRK4_H
#define DOUBLEPENDULUMRK4_H

#include "doublependulum.h"

class DoublePendulumRK4 : public DoublePendulum
{
public:
    DoublePendulumRK4(const Pendulum& upper, const Pendulum& lower,
                      double dt=0.005, double g=9.81);

    const char *solverMethod();

    void solveODEs(const double *yin, double *yout);
};

#endif // DOUBLEPENDULUMRK4_H
```

doublependulumrk4.cpp

```
#include "doublependulumrk4.h"

DoublePendulumRK4::DoublePendulumRK4(const Pendulum& upper,
                                     const Pendulum& lower,
                                     double dt, double g) :
    DoublePendulum(upper, lower, dt, g)
{
}

const char *DoublePendulumRK4::solverMethod()
{
    return "Runge Kutta (RK4)";
}

void DoublePendulumRK4::solveODEs(const double *yin, double *yout)
{
    double dydx[NUM_EQNS], dydxt[NUM_EQNS], yt[NUM_EQNS];
    double k1[NUM_EQNS], k2[NUM_EQNS], k3[NUM_EQNS], k4[NUM_EQNS];

    // First step
    derivs(yin, dydx);
    for (int i = 0; i < NUM_EQNS; ++i)
    {
        k1[i] = m_dt * dydx[i];
        yt[i] = yin[i] + 0.5 * k1[i];
    }

    // Second step
    derivs(yt, dydxt);
    for (int i = 0; i < NUM_EQNS; ++i)
    {
        k2[i] = m_dt * dydxt[i];
        yt[i] = yin[i] + 0.5 * k2[i];
    }

    // Third step
    derivs(yt, dydxt);
    for (int i = 0; i < NUM_EQNS; ++i)
    {
        k3[i] = m_dt * dydxt[i];
        yt[i] = yin[i] + k3[i];
    }

    // Fourth step
    derivs(yt, dydxt);
    for (int i = 0; i < NUM_EQNS; ++i)
    {
        k4[i] = m_dt * dydxt[i];
        yout[i] = yin[i] + k1[i] / 6.0 + k2[i] / 3.0 + k3[i] / 3.0 + k4[i] / 6.0;
    }
}
```