

# Runge-Kutta Methods for ODE Systems

## Ordinary differential equations

An ordinary differential equation (ODE) has *one* independent variable  $x$ , and one or more dependent variables  $y(x), \dots$

Physicists are usually interested in a particular solution of a differential equation given a particular set of initial values of the variables.

Mathematicians prefer to consider the family of solutions of the differential equation as shown in Fig. 1.

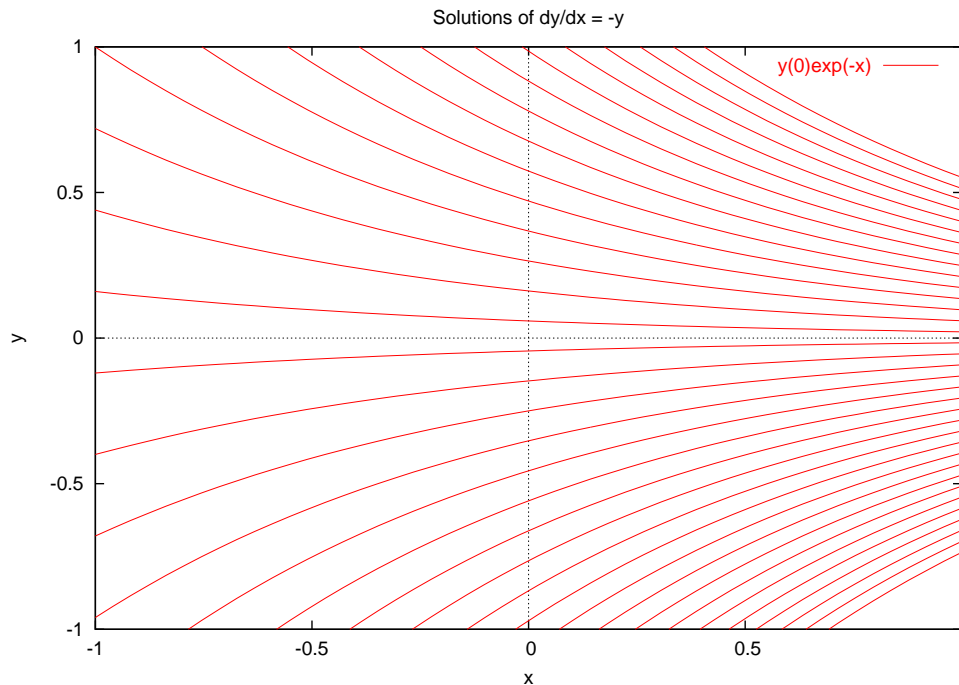


Figure 1: Family of solutions to a first order differential equation.

Example:

$$\frac{dy}{dx} = -y, \tag{1}$$

is a linear first order ODE. It is linear in the solution  $y$  and first order in derivative  $d/dx$ . It describes a *family of solutions* or *curves* in the  $x$ - $y$  plane. There is a unique curve passing through each point in the plane. The slope of the curve at any point is given by the differential equation.

To obtain a unique solution of the ODE a *condition* must be specified to fix the arbitrary *integration constant* which parametrizes the different solutions. For example, an *initial condition* on  $y(x = 0)$  can be chosen.

## Second order ODE's and vector notation

Example: The nonlinear damped and driven pendulum equation

$$\frac{d^2\theta}{dt^2} + q\frac{d\theta}{dt} + \sin\theta = b\cos(\omega_0 t), \quad (2)$$

is a second order nonlinear ODE with  $t$  being the independent variable, which depends on constant parameters  $q$  (damping coefficient),  $b$  (driving amplitude), and  $\omega_0$  (driving angular frequency).

It is always possible to write an ODE of any order as a system of first order ODE's. In this example

$$\frac{d\theta}{dt} = \omega, \quad (3)$$

$$\frac{d\omega}{dt} = -q\omega - \sin\theta + b\cos(\omega_0 t), \quad (4)$$

$$(5)$$

where  $\omega(t)$  is the angular velocity. This system of equations defines a family of curves in  $t$ - $\theta$ - $\omega$  space. To select a particular solution, one can impose initial conditions  $\theta(t=0)$ ,  $\omega(t=0)$ .

The pendulum equations can be written in compact vector notation

$$\frac{d\mathbf{y}}{dx} \equiv \frac{d}{dx} \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \equiv \frac{d}{dt} \begin{pmatrix} \theta \\ \omega \end{pmatrix} = \begin{pmatrix} \omega \\ -q\omega - \sin\theta + b\cos(\omega_0 t) \end{pmatrix} \equiv \begin{pmatrix} f_1(y_1, y_2, x) \\ f_2(y_1, y_2, x) \end{pmatrix} \equiv \mathbf{f}(\mathbf{y}(x), x), \quad (6)$$

where  $\mathbf{y}$  is called the *solution vector*, and  $\mathbf{f}$  is called the *flow vector* or *derivative vector*. To solve the vector differential equation we need to find the unknown solution vector  $\mathbf{y}$  as a function of the independent variable  $x$ , give a flow vector  $\mathbf{f}$  which is a known function of  $\mathbf{y}(x)$  and  $x$ .

## Euler's algorithm in vector notation

This vector notation is also very convenient for expressing many algorithms used to solve ordinary differential equations. For example, the Euler algorithm from Chapter 1 can be expressed simply

$$\mathbf{y}(x + dx) = \mathbf{y}(x) + dx \mathbf{f}(\mathbf{y}(x), x), \quad (7)$$

where  $dx$  is a step size in  $x$ .

## Runge-Kutta methods

Euler's algorithm is not very accurate and also suffers from instabilities. The mathematicians Runge and Kutta[2] discovered a class of algorithms that are based on making multiple Euler steps. These *Runge-Kutta algorithms* have been found to work very well in many applications. They are described in Numerical Recipes[1].

## Second order Runge-Kutta algorithm

The simple Euler algorithm uses the slope (tangent to the curve) of the solution  $y(x)$  at  $x$  to estimate

$$y(x + h) \simeq y(x) + h \left. \frac{dy}{dx} \right|_x = y(x) + hf(y, x) \equiv y(x) + k_1. \quad (8)$$

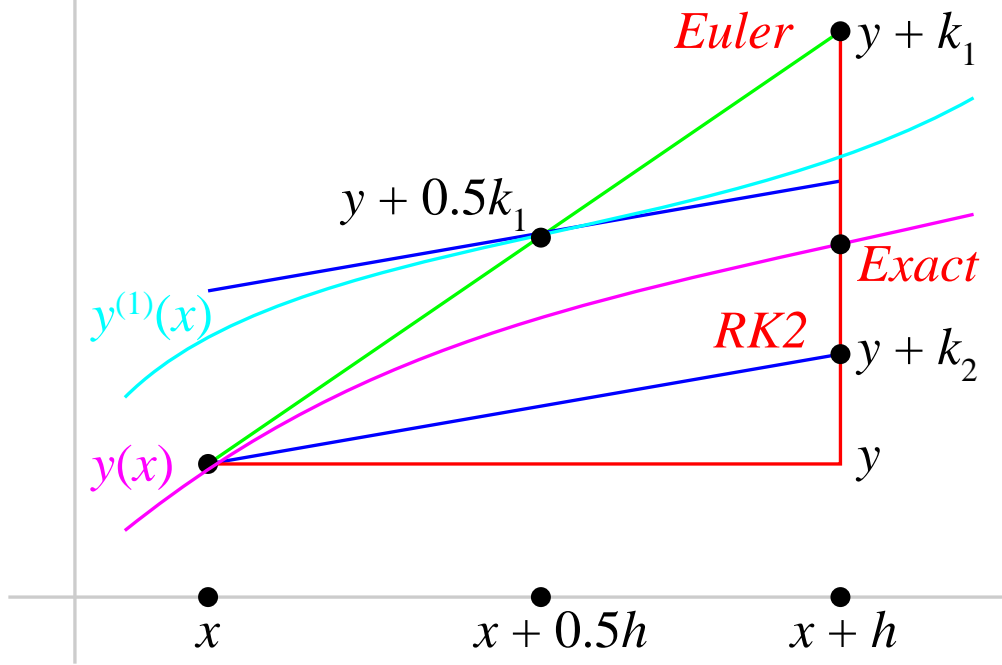


Figure 2: Second order Runge-Kutta algorithm.

The second order Runge-Kutta algorithm takes a *half* Euler step to the point

$$x + 0.5h, \quad y(x) + 0.5k_1. \quad (9)$$

There is a *different* solution curve  $y^{(1)}(x)$  passing through this half-way point. The slope at this point is used to compute

$$k_2 \equiv h \left. \frac{dy^{(1)}}{dx} \right|_{x+0.5h} = hf(y(x) + 0.5k_1, x + 0.5h). \quad (10)$$

The second order Runge-Kutta estimate is then computed

$$y(x + h) \simeq y(x) + k_2. \quad (11)$$

It is easy to show that the error is  $\mathcal{O}(h^3)$ . The left hand side can be expanded in a Taylor series:

$$y(x + h) = y(x) + h \frac{dy}{dx} + \frac{h^2}{2} \frac{d^2y}{dx^2} + \mathcal{O}(h^3) \quad (12)$$

$$= y(x) + hf(y, x) + \frac{h^2}{2} \frac{df(y, x)}{dx} \quad (13)$$

$$= y(x) + hf(y, x) + \frac{h^2}{2} \left[ \frac{\partial f}{\partial y} \frac{dy}{dx} + \frac{\partial f}{\partial x} \right] + \mathcal{O}(h^3) \quad (14)$$

$$= y(x) + hf(y, x) + \frac{h^2}{2} \left[ \frac{\partial f}{\partial y} f + \frac{\partial f}{\partial x} \right] + \mathcal{O}(h^3). \quad (15)$$

$$(16)$$

The right hand side can also be expanded

$$y(x) + k_2 = y(x) + hf \left( y + \frac{1}{2}k_1, x + \frac{h}{2} \right) = y(x) + hf(y, x) + h \left[ \frac{\partial f}{\partial y} \frac{k_1}{2} + \frac{h}{2} \frac{\partial f}{\partial x} \right] + \mathcal{O}(h^3). \quad (17)$$

The two sides agree through terms of  $\mathcal{O}(h^3)$ .

## Fourth order Runge-Kutta algorithm

Runge and Kutta showed that by combining the results of two additional Euler steps, the error can be reduced to  $\mathcal{O}(h^5)$ .

It is also easy to see that these algorithms can be extended to arbitrarily large systems of first order ODE's. The vector form of the fourth order Runge-Kutta algorithm is

$$\mathbf{k}_1 = h \mathbf{f} \left( \mathbf{y}(x), x \right) \quad (18)$$

$$\mathbf{k}_2 = h \mathbf{f} \left( \mathbf{y}(x) + \frac{1}{2} \mathbf{k}_1, x + \frac{h}{2} \right) \quad (19)$$

$$\mathbf{k}_3 = h \mathbf{f} \left( \mathbf{y}(x) + \frac{1}{2} \mathbf{k}_2, x + \frac{h}{2} \right) \quad (20)$$

$$\mathbf{k}_4 = h \mathbf{f} \left( \mathbf{y}(x) + \mathbf{k}_3, x + h \right) \quad (21)$$

$$\mathbf{y}(x+h) = \mathbf{y}(x) + \frac{\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4}{6} + \mathcal{O}(h^5). \quad (22)$$

$$(23)$$

## Kepler Orbit Program using Runge-Kutta

The following program implements the 4th order Runge-Kutta method to generate a Kepler orbit given the desired eccentricity and aphelion distance. Three-point Lagrange interpolation is used to find accurate orbital information at each subsequent aphelion crossing.

\_\_\_\_\_ Program 1: <http://www.physics.buffalo.edu/phy410-505/topic3/kepler-rk4.cpp> \_\_\_\_\_

```
#include <cmath>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <vector>
using namespace std;

const double pi = 4 * atan(1.0);
const double GmPlusM = 4 * pi * pi;

// derivative vector for Newton's law of gravitation

void compute_f(
    const vector<double> &y_vector,
    vector<double> &f_vector
) {

    double r_x = y_vector[0],
           r_y = y_vector[1],
           v_x = y_vector[2],
           v_y = y_vector[3];
```

```

double r_squared = r_x * r_x + r_y * r_y;
double r_cubed = r_squared * sqrt(r_squared);

f_vector[0] = v_x;
f_vector[1] = v_y;
f_vector[2] = - GmPlusM * r_x / r_cubed;
f_vector[3] = - GmPlusM * r_y / r_cubed;
}

// 4th order Runge-Kutta algorithm

void RK4_step(
    const double h,
    vector<double> &y_vector
) {

    int i, n = y_vector.size();
    vector<double> y(n), f(n), k1(n), k2(n), k3(n), k4(n);

    compute_f(y_vector, f);
    for (i = 0; i < n; i++)
        k1[i] = h * f[i];

    for (i=0; i < n; i++)
        y[i] = y_vector[i] + 0.5 * k1[i];
    compute_f(y, f);
    for (i = 0; i < n; i++)
        k2[i] = h * f[i];

    for (i=0; i < n; i++)
        y[i] = y_vector[i] + 0.5 * k2[i];
    compute_f(y, f);
    for (i = 0; i < n; i++)
        k3[i] = h * f[i];

    for (i=0; i < n; i++)
        y[i] = y_vector[i] + k3[i];
    compute_f(y, f);
    for (i = 0; i < n; i++)
        k4[i] = h * f[i];

    for (i = 0; i < n; i++)
        y_vector[i] += (k1[i] + 2 * k2[i] + 2 * k3[i] + k4[i]) / 6.0;
}

// Three point Lagrange interpolation

double interpolate(          // value of parabola at
    const double x0,         // this value of x, given 3 points
    const vector<double> x,  // with these x values and

```

```

    const vector<double> y // these y values
) {

    return (x0 - x[1]) * (x0 - x[2]) / (x[0] - x[1]) / (x[0] - x[2]) * y[0]
        + (x0 - x[2]) * (x0 - x[0]) / (x[1] - x[2]) / (x[1] - x[0]) * y[1]
        + (x0 - x[0]) * (x0 - x[1]) / (x[2] - x[0]) / (x[2] - x[1]) * y[2];
}

int main() {

    cout << " Kepler orbit using the Runge-Kutta method\n"
        << " -----\n"
        << " Enter aphelion distance in AU, and eccentricity: ";
    double r_ap, eccentricity, a, T, v0;
    cin >> r_ap >> eccentricity;
    a = r_ap / (1 + eccentricity);
    T = pow(a, 1.5);
    v0 = sqrt(GmPlusM * (2 / r_ap - 1 / a));
    cout << " Using Kepler's laws to compute\n"
        << " Semimajor axis length = " << a << " A.U.\n"
        << " Period T = " << T << " yr\n"
        << " Speed at aphelion = " << v0 << " A.U./yr\n\n";

    cout << " Enter number of periods and time-step size: ";
    double periods, dt;
    cin >> periods >> dt;
    cout << endl;

    double t = 0;
    vector<double> rv(4);
    rv[0] = r_ap;
    rv[1] = 0;
    rv[2] = 0;
    rv[3] = v0;

    ofstream data_file("kepler-rk4.data");

    int steps = 0, period = 0;
    vector<double> rv_minus2, rv_minus1; // store solution at 2 previous steps
    double t_last = t; // time at last aphelion

    do {

        // save current orbit information in file
        data_file << t << '\t';
        for (int i = 0; i < 4; i++)
            data_file << rv[i] << '\t';
        data_file << '\n';

        // update solution saved at 2 previous steps
        rv_minus2 = rv_minus1;

```

```

rv_minus1 = rv;

// take one Runge-Kutta time step
RK4_step(dt, rv);
t += dt;
++steps;

// process orbit information at each aphelion
if (rv[1] > 0 && rv_minus1[1] < 0) { // aphelion crossing

    ++period;
    cout << " aphelion crossing number " << period << '\n';

    // interpolate to get x value at aphelion
    double x0 = 0;
    vector<double> x(3), y(3);
    x[0] = rv[1], x[1] = rv_minus1[1], x[2] = rv_minus2[1];
    y[0] = rv[0], y[1] = rv_minus1[0], y[2] = rv_minus2[0];
    cout << " aphelion distance = " << interpolate(x0, x, y) << " A.U.\n";

    // interpolate to get time t at aphelion
    y[0] = t, y[1] = t - dt, y[2] = t - 2 * dt;
    double t_ap = interpolate(x0, x, y);
    cout << " current time      = " << t_ap << " years\n"
         << " elapsed time      = " << t_ap - t_last << " years\n"
         << endl;
    t_last = t_ap;
}

} while (t < periods * T);

cout << "\n number of RK4 steps = " << steps << '\n'
     << " data in file kepler-rk4.data" << endl;
data_file.close();
}

```

## Homework Problem

Choose a program whose accuracy you think you can trust and use it to explore changing the inverse square law of gravity to

$$F(r) = \frac{\text{constant}}{r^\beta}, \quad (24)$$

where  $\beta$  is some positive parameter. For example, you can choose initial conditions for  $\beta = 2$  to give a perfectly circular orbit, or an eccentric Kepler ellipse. Then explore how the orbit changes when you start with same initial position and velocity, but change  $\beta$  to 2.1, 2.5, 3.0, ... Your classical mechanics textbook may suggest other interesting values of  $\beta$  to try.

## References

- [1] W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P. Flannery, “Numerical Recipes in C” (Cambridge University Press 1992), §7.1 Runge-Kutta Method, <http://www.nrbook.com/a/bookcpdf/c16-1.pdf>.
- [2] C. Runge, *Math. Ann.* **46**, 167 (1895); M. W. Kutta, *Z. für Math. u. Phys.* **46**, 435 (1901).