

Black Holes and Schwarzschild Orbits

Precession of the Perihelion of Mercury

If two bodies interaction potential energy of two bodies depends only on the magnitude of their relative separation, then angular momentum is conserved. The orbit lies in a plane perpendicular to the total angular momentum vector. The shape of this orbit can change in time while conserving angular momentum. In fact, the orbit will not in general be a closed curve.

The inverse square force law is special. In addition to energy, total momentum, and angular momentum, the Laplace-Runge-Lenz vector

$$\mathbf{A} = \mathbf{p} \times \mathbf{L} - \mu GMm \frac{\mathbf{r}}{r}, \quad (1)$$

where \mathbf{r} , \mathbf{p} and \mathbf{L} are the relative coordinate, relative momentum and angular momentum in the center-of-mass frame, is conserved. The orientation of an elliptical Kepler orbit does not change in time.

The Kepler orbit of the innermost planet Mercury rotates or precesses at the rate of approximately $56''$ per year, or one revolution every 2,300 years. This deviation from a perfect Kepler orbit is caused in part by general relativistic corrections¹.

	arc seconds / century
Precession of the equinoxes	5025.6
Planetary perturbations	531.4
General relativity	42.98 ± 0.04
Quadrupole moment of the Sun	0.0254
Total Theory	5600.0
Observed	5599.7

Table 1: Contributions to the precession of Mercury's perihelion.

Precision Computation of Kepler Orbits

The following program introduces the following improvements that help generate reliable and highly accurate solutions of Newton's equations:

- Use of an extended solution vector with the independent variable as one of the components.
- Template functions in the header in the cpl to make linear algebra programming more convenient.
- Another strategy for determining the orbital period accurately based on making a change of integrating variable.
- Adaptive stepsize control algorithm for Runge-Kutta from Numerical Recipes[1].

_____ Program 1: <http://www.physics.buffalo.edu/phy410-505/topic3/kepler-adapt.cpp> _____

```
#include <cmath>
```

```

#include <cstdlib>
#include <fstream>
#include <iostream>
using namespace std;

#include "linalg.hpp"
using namespace cpl;

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

```

Extending the solution vector with the independent variable

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

```

bool switch_t_with_y = false;    // to interpolate to y = 0

// Derivative vector for Newton's law of gravitation

vector<double> f(vector<double> x) {

    double t = x[0], r_x = x[1], r_y = x[2], v_x = x[3], v_y = x[4];
    double rSquared = r_x*r_x + r_y*r_y;
    double rCubed = rSquared * sqrt(rSquared);

    vector<double> f(5);
    f[0] = 1;
    f[1] = v_x;
    f[2] = v_y;
    f[3] = - GmPlusM * r_x / rCubed;
    f[4] = - GmPlusM * r_y / rCubed;

    if (switch_t_with_y)          // use y as independent variable
        f /= v_y;

    return f;
}

```

Coding linear algebra with template functions

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

```

// Fourth order Runge-Kutta

```

```

void RK4_step(vector<double>& x, double tau) {

    vector<double> k1 = tau * f(x);
    vector<double> k2 = tau * f(x + 0.5 * k1);
    vector<double> k3 = tau * f(x + 0.5 * k2);
    vector<double> k4 = tau * f(x + k3);

    x += (k1 + 2 * k2 + 2 * k3 + k4) / 6.0;
}

```

Adaptive stepsize control for Runge-Kutta

_____ Program 1: <http://www.physics.buffalo.edu/phy410-505/topic3/kepler-adapt.cpp> _____

```

// Adaptive step size control using Runge-Kutta and step doubling
// Algorithm adapted from Numerical Recipes

void adaptiveStep(vector<double>& x, double& tau, double accuracy) {

    const double SAFETY = 0.9, PGROW = -0.2, PSHRINK = -0.25,
                ERRCON = 1.89e-4, TINY = 1.0e-30;
    int n = x.size();
    vector<double> x_half(n), x_full(n), Delta(n);
    vector<double> scale = f(x);
    for (int i = 0; i < n; i++)
        scale[i] = abs(x[i]) + abs(scale[i] * tau) + TINY;
    double err_max;

    while (true) {

        // take two half steps
        double tau_half = tau / 2;
        x_half = x;
        RK4_step(x_half, tau_half);
        RK4_step(x_half, tau_half);

        // take full step
        x_full = x;
        RK4_step(x_full, tau);

        // estimate error
        Delta = x_half - x_full;
        err_max = 0;
        for (int i = 0; i < n; i++)
            err_max = max(err_max, abs(Delta[i]) / scale[i]);
        err_max /= accuracy;
        if (err_max <= 1.0)

```

```

        break;
    double tau_temp = SAFETY * tau * pow(err_max, PSHRINK);
    if (tau >= 0.0)
        tau = max(tau_temp, 0.1 * tau);
    else
        tau = min(tau_temp, 0.1 * tau);
    if (abs(tau) == 0.0) {
        cerr << "adaptiveStep: step size underflow, aborting ..." << endl;
        exit(EXIT_FAILURE);
    }
}

tau *= (err_max > ERRCON ? SAFETY * pow(err_max, PGROW) : 5.0);
x = x_half + Delta / 15.0;
}

```

Determining x -axis crossings accurately by change of variable

_____ Program 1: <http://www.physics.buffalo.edu/phy410-505/topic3/kepler-adapt.cpp> _____

```

// Change independent variable from t to y and step back to y = 0

void interpolate_crossing(vector<double> x, int& crossing) {

    ++crossing;
    switch_t_with_y = true;
    RK4_step(x, -x[2]);
    cout << " crossing " << crossing << "\t t = " << x[0]
        << "\t x = " << x[1] << endl;
    switch_t_with_y = false;
}

int main() {

    cout << " Kepler orbit using adaptive Runge-Kutta with step doubling\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 << " Enter number of periods, step size, and adaptive accuracy: ";
    double periods, dt, accuracy;
    cin >> periods >> dt >> accuracy;
}

```

```

// integrate using fixed step size
vector<double> x0(5);
x0[0] = 0; x0[1] = r_ap; x0[2] = 0; x0[3] = 0; x0[4] = v0;
ofstream data_file("fixed.data");
vector<double> x = x0;
int steps = 0, crossing = 0;
cout << "\n Integrating with fixed step size" << endl;

do {
    for (int i = 0; i < 5; i++)
        data_file << x[i] << '\t';
    data_file << '\n';
    double y = x[2];
    RK4_step(x, dt);
    ++steps;
    if (y * x[2] < 0)
        interpolate_crossing(x, crossing);
} while (x[0] < periods * T);

cout << " number of fixed size steps = " << steps << endl;
cout << " data in file fixed.data" << endl;
data_file.close();

// integrate using adaptive step size
data_file.open("adaptive.data");
x = x0;
steps = crossing = 0;
double dt_max = 0, dt_min = 100;
cout << "\n Integrating with adaptive step size" << endl;

do {
    for (int i = 0; i < 5; i++)
        data_file << x[i] << '\t';
    data_file << '\n';
    double t_save = x[0];
    double y = x[2];
    adaptiveStep(x, dt, accuracy);
    double step_size = x[0] - t_save;
    ++steps;
    if (step_size < dt_min) dt_min = step_size;
    if (step_size > dt_max) dt_max = step_size;
    if (y * x[2] < 0)
        interpolate_crossing(x, crossing);
} while (x[0] < periods * T);

cout << " number of adaptive steps = " << steps << endl;
cout << " step size: min = " << dt_min << " max = " << dt_max << endl;
cout << " data in file adaptive.data" << endl;
data_file.close();
}

```

Homework Problem

Modify the force function to include the effects of general relativity. Assuming that the Sun is a point mass, so mercury get arbitrarily close to its Schwarzschild radius, generate 3 different types of Schwarzschild orbits, choosing the ones you find most interesting from a relativity textbook or http://en.wikipedia.org/wiki/Kepler_problem_in_general_relativity.

References

- [1] W.H. Press, S.A. Teukolsky, W.T. Vetterling and B.P. Flannery, “Numerical Recipes in C” (Cambridge University Press 1992), §16.2 Adaptive Stepsize Control for Runge-Kutta, <http://www.nrbook.com/a/bookcpdf/c16-2.pdf>.