2011-04-11 作業解答

1. 用 leapfrog 法計算上週習題的行星運動軌道：計算給定初始條的軌道週期（可用 y 值回到 0 為完成一週期的條件）；計算並比較在不同 τ 時（0.01, 0.001, 0.0001）時 leapfrog 法和上週用的 Euler 法收斂速度的差異。

Leapfrog method is really close to the Euler method in implementation. We just need to take care of alternating the updates of velocities and positions. Also, when the initial velocities are specified at the same time as the positions, we need to extrapolate for their values at half a time step before the initial time.

C++:

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
	double dt, x, y, vx, vy, T;
	cout << "# Input: time_step x0 y0 vx0 vy0 T\n";
	cin >> dt >> x >> y >> vx >> vy >> T;
	cout << "# Got: " << dt << ' ' << x << ' ' << y << ' '
	     << vx << ' ' << vy << ' ' << T << '\n';
	double t = 0;
	// find velocity at t = - dt / 2
	vx -= 0.5 * dt * x / pow(x * x + y * y, 1.5);
	vy -= 0.5 * dt * y / pow(x * x + y * y, 1.5);
	while (t < T) {
		double rt3 = pow(x * x + y * y, 1.5);
		double ax = - x / rt3;
		double ay = - y / rt3;
		vx += dt * ax; // @ t + dt/2
		vy += dt * ay;
		x += dt * vx;  // @ t + dt
		y += dt * vy;
		t += dt;
		cout << t << '\t' << x << '\t' << y << '\n';
	}
	return 0;
}
```

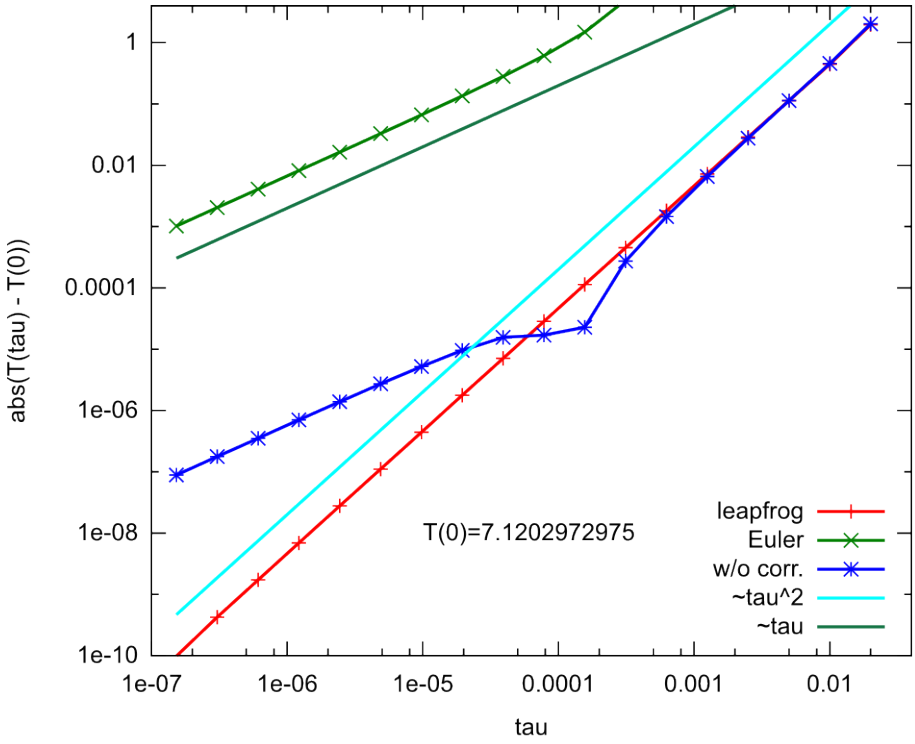
Since we started with y0 = 0 and vy0 > 0, to find out the period of the orbit we can find the time when the y position of the planet crossing the y=0 line from the negative side. (For better accuracy, we interpolate the crossing time using the y values before and after the crossing.) The specialized code is as follows.

C++:

```
#include <iostream>
#include <cmath>
void euler_step(long double dt, long double & x, long double & y,
	long double & vy, long double & vx)
{
	long double rr = powl(x * x + y * y, 1.5);
	long double ax = - x / rr;
	long double ay = - y / rr;
	x += dt * vx;
	y += dt * vy;
	vx += dt * ax;
	vy += dt * ay;
}
void leapfrog_step(long double dt, long double & x, long double & y,
```

```
        long double & vy, long double & vx)
{
        long double rr = powl(x * x + y * y, 1.5);
        long double ax = - x / rr;
        long double ay = - y / rr;
        vx += dt * ax;
        vy += dt * ay;
        x += dt * vx;
        y += dt * vy;
}
long double find_period(long double dt, long double x, long double y,
        long double vx, long double vy, long double T,
        void (* step)(long double, long double &, long double &,
        long double &, long double &))
{
        long double t = 0, py = 0;
        while (t < T) {
                (* step)(dt, x, y, vx, vy);
                t += dt;
                if (py < 0 && y >= 0) return t - dt * y / (y - py);
                py = y;
        }
        return 0;
}
int main()
{
        using namespace std;
        long double tau = 0.02;
        cout.precision(15);
        while (tau > 1e-7) {
                cout << tau;
                if (long double T = find_period(tau, 2, 0, 0.2 + 0.125 * tau,
                        0.2, 16, & leapfrog_step)) cout << '\t' << T;
                if (long double T = find_period(tau, 2, 0, 0.2, 0.2, 16,
                        & leapfrog_step)) cout << '\t' << T;
                if (long double T = find_period(tau, 2, 0, 0.2, 0.2, 16,
                        & euler_step)) cout << '\t' << T;
                cout << endl;
                tau /= 2;
        }
        return 0;
}
```

We compare the results to an estimated exact value of the period T(0) and plot the absolute values of the differences on a double-log plot as follows. Adjusting the estimated T(0), we get nice straight lines for a best value of T(0)≈7.1202972975. We get slope of 1 for the Euler method, which means the error decrease linearly in tau. On the other hand, the leapfrog method has a slope of 2, thus, its error decrease much faster, in second order of tau. Please note that the correction to the initial velocity of the leapfrog method (the "`+0.125*tau`" in the code) is important since it amounts to an error that decrease only linearly in tau. To demontrat this, we also plot the results using leapfrog method without the corrections in the same plot. As can be seen, while initially the error decrease quadratically for large tau, it eventual flips sign and decreases only linearly in tau.



2. 數值積分：以梯形法及 Simpson 法求下列函數在 0~10 區間的積分

$$f(x)=\sin\left[x^2\exp(-x)+\ln(x+8)\right]$$

計算兩方法在 $\Delta x = 1, 0.1, 0.01, 0.001$ 時的積分值，並以此作圖估計兩方法在 $\Delta x \to 0$ 的收斂速度。

When partition the interval into N equal segments, the trapezoidal method amounts to summing up all functional values at the N-1 partition points plus adding in the average of the functional values at two ends. The following code repeat such an integral with finer and finer partitions.

C++:

```
#include <iostream>
#include <cmath>
using namespace std;
double func(double x)
{
	return sin(x * x * exp(- x) + log(x + 8));
}
int main()
{
	cout.precision(16);
	for (size_t n = 10; n < 20000; n *= 2) {
		double xmin = 0;
		double xmax = 10;
		double dx = (xmax - xmin) / n;
		double acc = func(xmin) / 2;
		for (size_t i = 1; i < n; i ++) {
			double x = i * dx;
			acc += func(x);
		}
		acc += func(xmax) / 2;
		acc *= dx;
		cout << dx << '\t' << acc << endl;
```

```
        }
    }
```

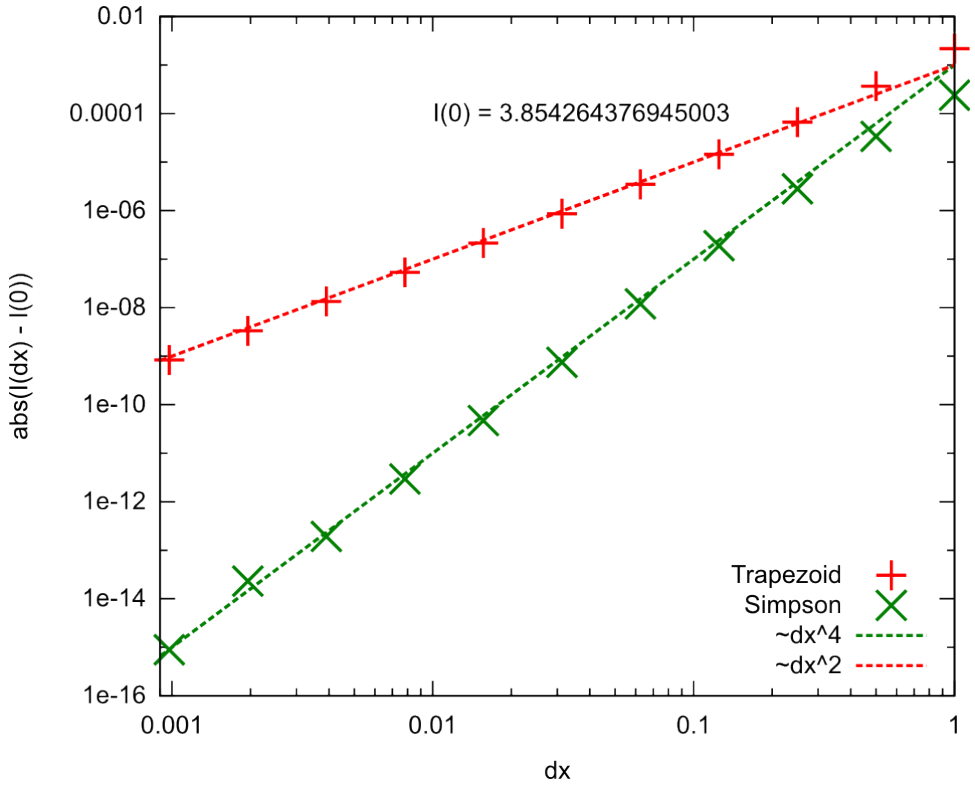
Converting the same code to Simpson's rule requires an extra evaluation of the function at the midpoint of each segment.
C++:

```
    #include <iostream>
    #include <cmath>
    using namespace std;
    double func(double x)
    {
        return sin(x * x * exp(- x) + log(x + 8));
    }

    int main()
    {
        cout.precision(16);
        for (size_t n = 10; n < 20000; n *= 2) {
            double xmin = 0;
            double xmax = 10;
            double dx = (xmax - xmin) / n;
            double pf = func(xmin);
            double acc = 0;
            for (size_t i = 1; i <= n; i ++) {
                double x = i * dx;
                double f = func(x);
                acc += pf + 4 * func(x - dx / 2) + f;
                pf = f;
            }
            acc *= dx / 6;
            cout << dx << '\t' << acc << endl;
        }
    }
```

For the extra functional evaluation at each segment, the Simpson's rule would be twice as slow as the trapezoidal method for the same number of segments. However, it can make up with a much faster convergence that can achieve the same accuracy with a much coarser partition.

Following error analysis is similar to what we done to the orbital period: We first estimate the exact value of the integral from the sequence of the approximate values at different dx. Assuming the errors of these approximations from the estimated exact value decrease with a certain power in dx, the plot of these deviations on double-log scales should give us a straight line. We can judge how good this estimate by how straight the line is on the double-log plot and adjust its value accordingly. The best estimate I got is $I(0) \approx 3.854264376945003$. As can be seen in the following plot, the error of trapezoidal method decreases with dx^2 while the error of Simpson's rule decreases with dx^4.



3. 用 Monte Carlo 法求上述的積分，估計在點數 $n \to \infty$時的收斂速度。（在給定點數時可用不同的 seed 來作取樣，以估計它的標準差）

Graphing the function, we can see it's value fall well within the interval [0,1] between x=0 and x=10. Thus, we can draw random points within this rectangular area to check if they hit below the curve. In order to estimate the accuracy of the Monte Carlo integral I(n) using n random points, we repeat the evaluation with different random seeds. The ensemble generated allows us to estimate the standard deviations of such a method.

C++:

```
#include "ran_nr.hh"
#include <iostream>
#include <cmath>
using namespace std;
double f(double x)
{
	return sin(x * x * exp(- x) + log(x + 8));
}
int main ()
{
	cout.precision(15);
	for (size_t n = 10; n < 200000; n *= 2) {
		double I_tot = 0;
		double I2_tot = 0;
		for (unsigned long seed = 0; seed < 1000; seed ++) {
			RanNR rng(seed);
			size_t cnt = 0;
			for (size_t i = 0; i < n; i ++) {
				double y = rng.uniform();
				double x = 10 * rng.uniform();
				if (y < f(x)) cnt ++;
			}
			double I = cnt * 10.0 / n;
```

```
                I_tot += I;
                I2_tot += I * I;
            }
            double I_ave = I_tot / 1000;
            double I2_ave = I2_tot / 1000;
            double std = sqrt(I2_ave - I_ave * I_ave);
            cout << n << '\t' << I_ave << '\t' << std << endl;
        }
        return 0;
}
```

The best result we got from an estimate using 163840 points is 3.854±0.012. The standard deviation ΔI(n) estimated using 1000 random seeds for each number n of used points is plotted below and shows a scaling of ΔI(n) ~ 1/n^0.5, which is much slower comparing to all the deterministic methods we have seen so far. However, unlike the other methods, the speed of convergence for Monte Carlo integral is independent of the dimensionality of the integral. So, in higher dimensions, it often becomes the only effective method of numerical integral.