PHYS580 Lab03 Report

Yicheng Feng
PUID: 0030193826

September 11, 2019

Workflow: I use the Linux system, and code C++ in terminals. For visualization, I use the C++ package – ROOT (made by CERN) to make plots. At the end, the reports are written in LaTeX.

The codes for this lab are written as the following files:

- `runge_kutta.h` and `runge_kutta.cxx` for the class `RungeKutta` to solve general ordinary differential equation sets.
- `pendulum.h` and `pendulum.cxx` for the class `Pendulum` to calculate ODE set of the pendulum.
- `lab3.cxx` for the main function to make plots.

To each problem of this lab report, I will attach the relevant parts of the code. If you want to check the validation of my code, you need to download the whole code from the link `https://github.com/YichengFeng/phys580/tree/master/lab3`.

(1) First, take the linear oscillator with no damping and no driving force (i.e., completely harmonic). Investigate the stability and accuracy of three numerical methods: Euler, Euler-Cromer, and Runge-Kutta 2$^{\text{nd}}$ order. Demonstrate your conclusions with a few chosen parameters and plots. Be sure to include the cumulative (global) error analyses for both θ and E (energy) in your report, comparing the different methods to each other and to the theoretical expectations. Do NOT use Matlab built-in functions for these approximations, rather, code the respective algorithms into your programs explicitly.

Physics explanation:

We set the parameters: length $L = 9.8$ m, gravity $g = 9.8$ m s^{-2}, the initial angle $\theta_0 = 0.08$ rad, the initial angular velocity $\omega_0 = 0$, and the time step $\Delta t = 0.02$ s.

We can see from Fig. 1 and Fig. 2 that the Euler approximation is inadequate, because the amplitudes of θ and ω have obivious deviation from the exact value, and the energy is not conserved and increasing. After 4.5 periods, the energy is increased by about 78%, which is a huge error.

However, the Runge-Kutta 2$^{\text{nd}}$ order approximation (Fig. 5 and 6) is much better with the same setting. The θ nad ω's deviation from the exact values are small, after 4.5 periods, the deviation is roughly 0.15% of the amplitude. Although the energy is still not conserved, the increasing rate is small: after 4.5 periods, the deviation is roughly 5.5×10^{-5} of the real energy.

The Euler-Cromer approximation (Fig. 3 and 4) seems also good and adequate. Due to the algorithm of this method, the global error of energy is oscillating around 0 instead of just increasing. The amplitude of the energy deviation is about 1%. The deviation of θ is similar to the energy, oscillating with amplitude 1% of the maximum θ. The deviation of ω is similar to the Euler case, oscillating and increasing. After 4.5 periods, the relative error of ω is about 0.05%.

Plots:

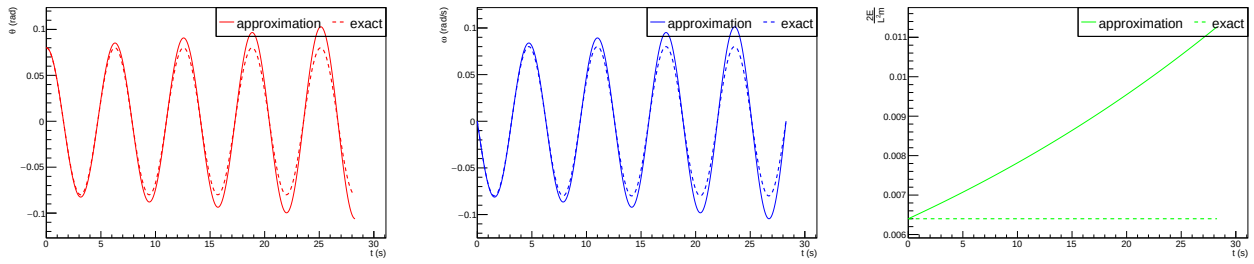

Figure 1: The θ, ω and E depending on time t with Euler approximation. The dashed lines are the exact values.

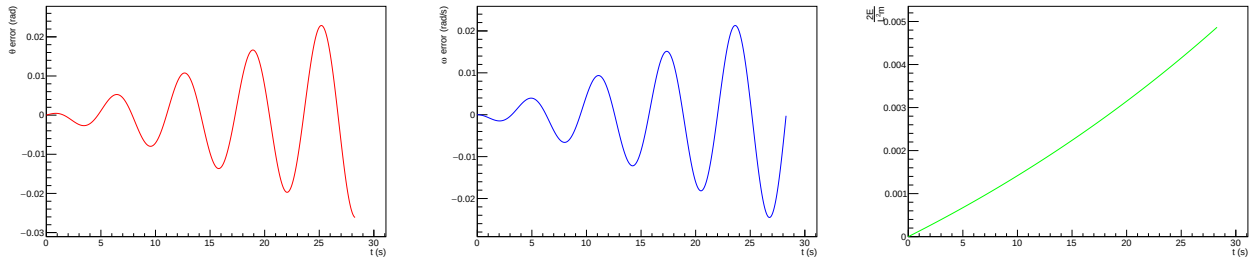

Figure 2: The θ, ω and E's deviation (global error) from the exact values depending on time t with Euler approximation.

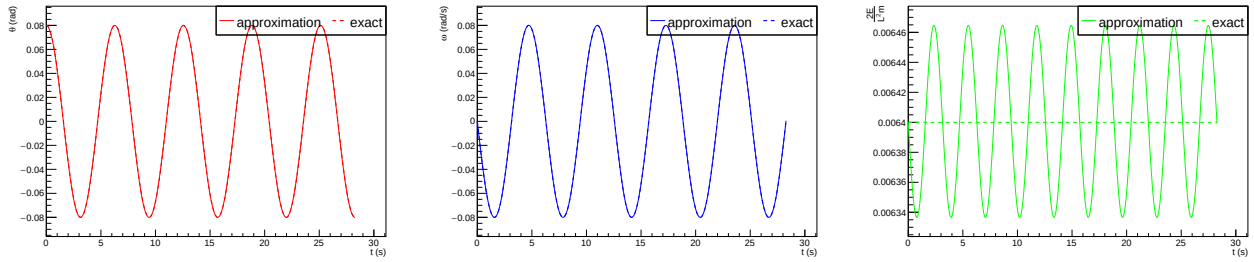

Figure 3: The θ, ω and E depending on time t with Euler-Cromer approximation. The dashed lines are the exact values.

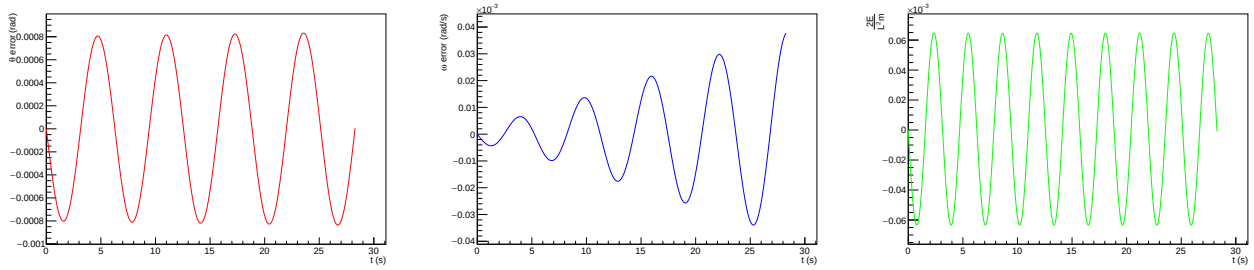

Figure 4: The θ, ω and E's deviation (global error) from the exact values depending on time t with Euler-Cromer approximation.

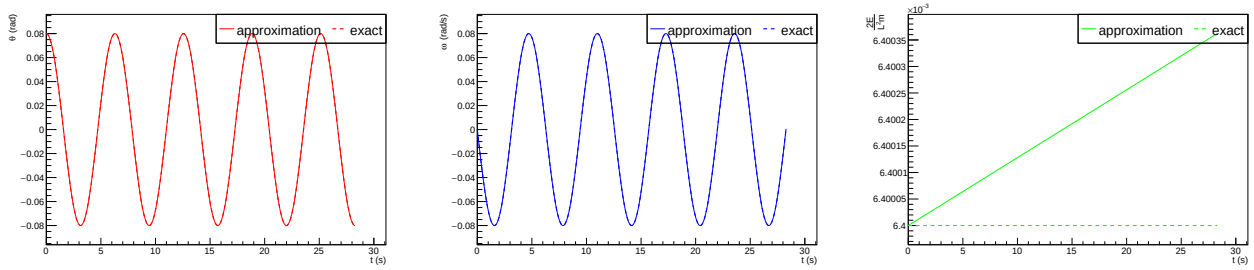

Figure 5: The θ, ω and E depending on time t with 2^{nd} order Runge-Kutta approximation. The dashed lines are the exact values.

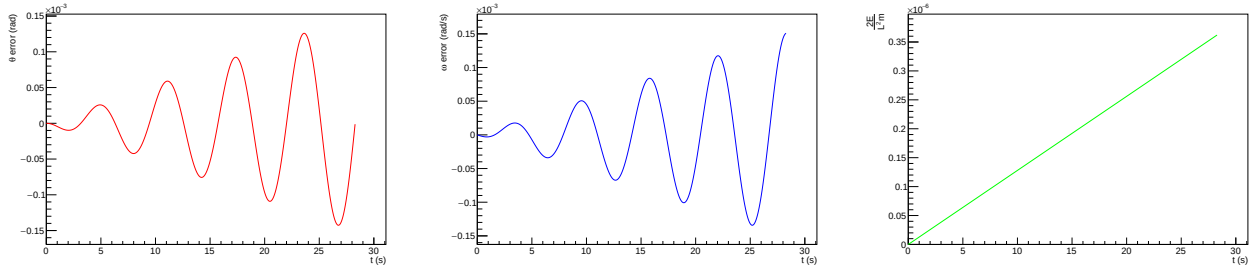

Figure 6: The θ, ω and E's deviation (global error) from the exact values depending on time t with 2^{nd} order Runge-Kutta approximation.

Relevant code:

The Euler approximation:

```
//------------------------------------------------------------------------//

void RungeKutta::cal_rk1() {
        if(!check() || !clear()) {
                cout << "WARNING: check() not passed, no calculation!" << endl;
                return;
        }
        //cout << "check() passed" << endl;

        _n_stps = 0;

        double t = _t_start;
        vector<double> x = _x_start;

        vector<double> F1(_n_eqns);

        vector<double> x1(_n_eqns);

        double t1;

        int nfv = _fv.size(); // nfv = _n_eqns
        while(!_stop(t, x)) {
                // Fill into output
                _t.push_back(t);
                for(int ifv=0; ifv<nfv; ifv++) {
                        _x[ifv].push_back(x[ifv]);
                }
                _n_stps ++;

                // F1
                t1 = t;
                for(int ifv=0; ifv<nfv; ifv++) {
                        x1[ifv] = x[ifv];
                }
                for(int ifv=0; ifv<nfv; ifv++) {
                        F1[ifv] = _fv[ifv](t1, x1);
                }

                t += _dt;
                for(int ifv=0; ifv<nfv; ifv++) {
                        x[ifv] += F1[ifv]*_dt;
                }
        }

        _t.push_back(t);
        for(int ifv=0; ifv<nfv; ifv++) {
                _x[ifv].push_back(x[ifv]);
        }
        _n_stps ++;
}

//------------------------------------------------------------------------//
```

The Euler-Cromer approximation:

```
//------------------------------------------------------------------------//

void RungeKutta::cal_Euler_Cromer() {
```

```
        if(!check() || !clear()) {
                cout << "WARNING: check() not passed, no calculation!" << endl;
                return;
        }
        //cout << "check() passed" << endl;

        _n_stps = 0;

        double t = _t_start;
        vector<double> x = _x_start;

        vector<double> F1(_n_eqns);

        int nfv = _fv.size(); // nfv = _n_eqns
        while(!_stop(t, x)) {
                // Fill into output
                _t.push_back(t);
                for(int ifv=0; ifv<nfv; ifv++) {
                        _x[ifv].push_back(x[ifv]);
                }
                _n_stps ++;

                // F1
                // update x right after each calculation
                //for(int ifv=0; ifv<nfv; ifv++) {
                for(int ifv=nfv-1; ifv>=0; ifv--) {
                        F1[ifv] = _fv[ifv](t, x);
                        x[ifv] += F1[ifv]*_dt;
                }
                t += _dt;
        }

        _t.push_back(t);
        for(int ifv=0; ifv<nfv; ifv++) {
                _x[ifv].push_back(x[ifv]);
        }
        _n_stps ++;
}

//-------------------------------------------------------------------------//
```

The Runge-Kutta 2nd approximation:

```
//-------------------------------------------------------------------------//

void RungeKutta::cal_rk2() {
        if(!check() || !clear()) {
                cout << "WARNING: check() not passed, no calculation!" << endl;
                return;
        }
        //cout << "check() passed" << endl;

        _n_stps = 0;

        double t = _t_start;
        vector<double> x = _x_start;

        vector<double> F1(_n_eqns);
        vector<double> F2(_n_eqns);

        vector<double> x1(_n_eqns);
```

```
        vector<double> x2(_n_eqns);

        double t1, t2;

        int nfv = _fv.size(); // nfv = _n_eqns
        while(!_stop(t, x)) {
                // Fill into output
                _t.push_back(t);
                for(int ifv=0; ifv<nfv; ifv++) {
                        _x[ifv].push_back(x[ifv]);
                }
                _n_stps ++;

                // F1
                t1 = t;
                for(int ifv=0; ifv<nfv; ifv++) {
                        x1[ifv] = x[ifv];
                }
                for(int ifv=0; ifv<nfv; ifv++) {
                        F1[ifv] = _fv[ifv](t1, x1);
                }
                // F2
                t2 = t + 0.5*_dt;
                for(int ifv=0; ifv<nfv; ifv++) {
                        x2[ifv] = x[ifv] + F1[ifv]*0.5*_dt;
                }
                for(int ifv=0; ifv<nfv; ifv++) {
                        F2[ifv] = _fv[ifv](t2, x2);
                }

                t += _dt;
                for(int ifv=0; ifv<nfv; ifv++) {
                        x[ifv] += F2[ifv]*_dt;
                }
        }

        _t.push_back(t);
        for(int ifv=0; ifv<nfv; ifv++) {
                _x[ifv].push_back(x[ifv]);
        }
        _n_stps ++;
}

//-------------------------------------------------------------------------//
```

The input functions for the simplified pendulum ODE set:

```
//-------------------------------------------------------------------------//

// x[0]: theta; x[1]: omega
double Pendulum::f_theta(double t, const vector<double> &x) {
        // dtheta/dt = omega
        return x[1];
}

//-------------------------------------------------------------------------//

double Pendulum::f_omega_simplified(double t, const vector<double> &x) {
        return -_g/_l*x[0] - _q*x[1] + _F*sin(_O*t+_P);
}

```

```
//------------------------------------------------------------------------//

double Pendulum::f_energy_simplified(double t, const vector<double> &x) {
        // 2E/m/l^2
        return _g/_l*x[0]*x[0]+x[1]*x[1];
}

//------------------------------------------------------------------------//
```

The exact solution to the simplified pendulum ODE set:

```
//------------------------------------------------------------------------//

// the exact solution of the simplified ODE
double Pendulum::f_theta_simplified_exact(double t, const vector<double> &x_start) {
        double Omg = sqrt(_g/_l);
        double Amp = sqrt(x_start[0]*x_start[0]+x_start[1]*x_start[1]/Omg/Omg);
        double Phi = atan2(x_start[0], x_start[1]/Omg);
        return Amp*sin(Omg*t+Phi);
}

//------------------------------------------------------------------------//

// the exact solution of the simplified ODE
double Pendulum::f_omega_simplified_exact(double t, const vector<double> &x_start) {
        double Omg = sqrt(_g/_l);
        double Amp = sqrt(x_start[0]*x_start[0]+x_start[1]*x_start[1]/Omg/Omg);
        double Phi = atan2(x_start[0], x_start[1]/Omg);
        return Amp*Omg*cos(Omg*t+Phi);
}

//------------------------------------------------------------------------//
```

(2) Second, still for the linear oscillator, turn on some damping and/or driving force. Using the method and parameters you determined in (1) to be adequate, demonstrate the overdamped, underdamped, and resonant regimes (cf. Section 3.2 of the textbook). Set $l = 9.8$ m to simplify things a bit.

Physics explanation:

We copy the parameters from (1): length $L = 9.8$ m, gravity $g = 9.8$ m s^{-2}, the initial angle $\theta_0 = 0.08$ rad, the initial angular velocity $\omega_0 = 0$, and the time step $\Delta t = 0.02$ s. We use the 2$^{\text{nd}}$ order Runge-Kutta approximation in this problem, since it has the smallest error.

Plots:

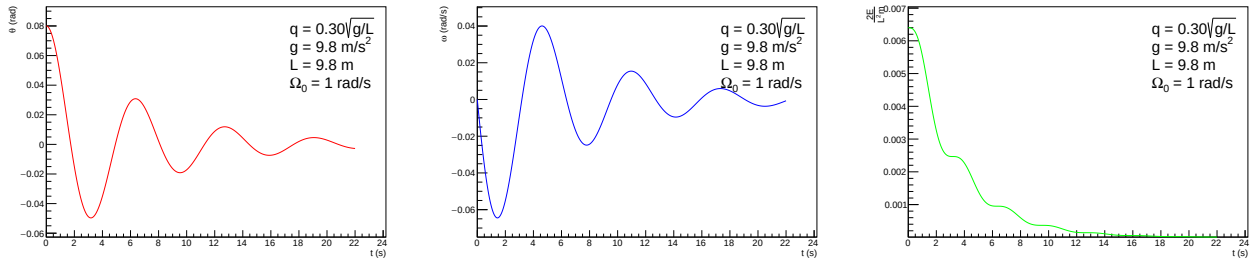

Figure 7: The θ, ω and E depending on time t with RK2 approximation. The underdamped case, $q = 0.3\sqrt{q/l} < 2\sqrt{q/l}$.

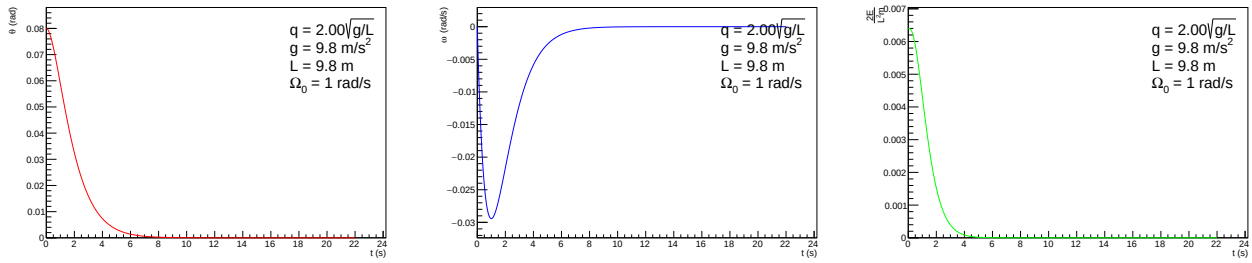

Figure 8: The θ, ω and E depending on time t with RK2 approximation. The critical case, $q = 2\sqrt{q/l}$.

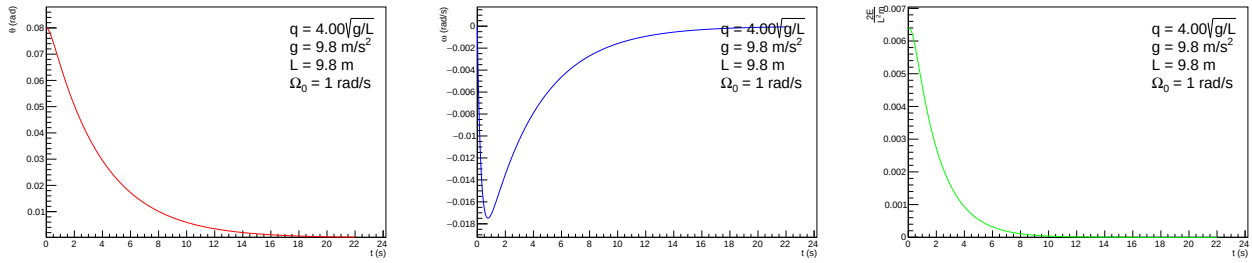

Figure 9: The θ, ω and E depending on time t with RK2 approximation. The overdamped case, $q = 4\sqrt{q/l} > 2\sqrt{q/l}$.

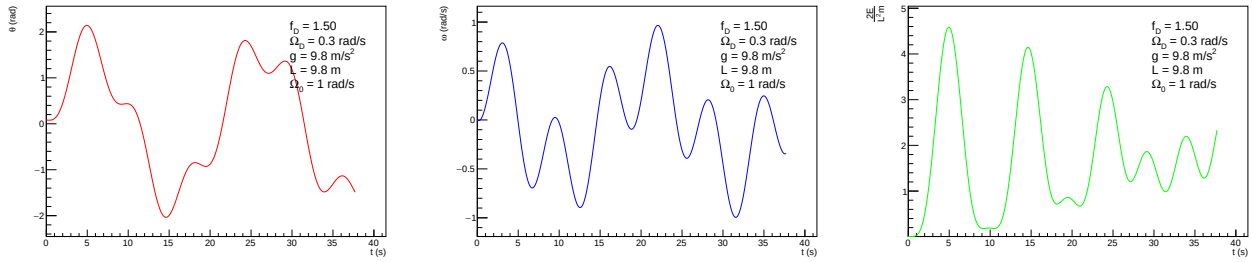

Figure 10: The θ, ω and E depending on time t with RK2 approximation. The drive force $f_D = 1.5$, and drive frequency $\Omega_D = 0.3 < \sqrt{g/l}$.

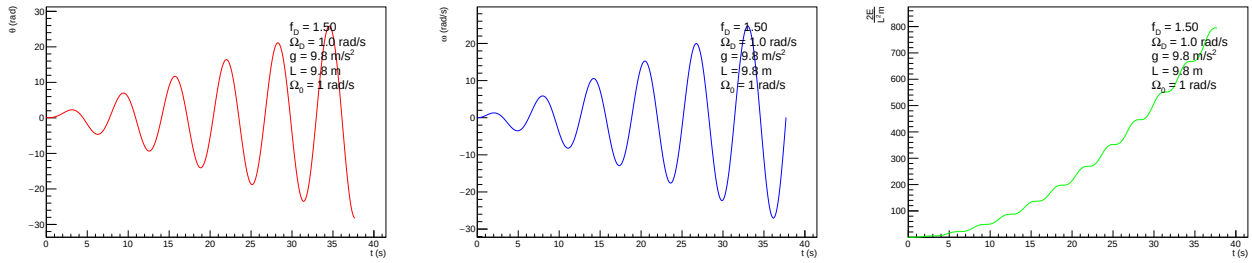

Figure 11: The θ, ω and E depending on time t with RK2 approximation. The drive force $f_D = 1.5$, and drive frequency $\Omega_D = \sqrt{g/l}$, **resonance** case.

Relevant code:

The input functions for the simplified pendulum ODE set:

```
//------------------------------------------------------------------------//

// x[0]: theta; x[1]: omega
double Pendulum::f_theta(double t, const vector<double> &x) {
        // dtheta/dt = omega
        return x[1];
}

//------------------------------------------------------------------------//

double Pendulum::f_omega_simplified(double t, const vector<double> &x) {
        return -_g/_l*x[0] - _q*x[1] + _F*sin(_O*t+_P);
}

//------------------------------------------------------------------------//

double Pendulum::f_energy_simplified(double t, const vector<double> &x) {
        // 2E/m/l^2
        return _g/_l*x[0]*x[0]+x[1]*x[1];
}

//------------------------------------------------------------------------//
```

(3) Finally, take the physical oscillator with unapproximated nonlinear equation of motion (i.e., the full $\sin\theta$ term) , but without damping and driving force. Investigate the dependences, if any, of the period (if periodic) and wave form on the amplitude of oscillation. Compare with the exact result for the period:

$$T = 4\sqrt{\frac{l}{g}}K(\sin(\theta_m/2)) = 2\pi\sqrt{\frac{l}{g}}\left(1 + \frac{1}{16}\theta_m^2 + \frac{11}{3072}\theta_m^4 + \dots\right) \tag{1}$$

Here, the right-hand side explicitly shows the first three terms of $K(a)$, the complete elliptic integral of 1^{st} kind, when expanded into powers of the oscillation amplitude θ_m.

Plots:

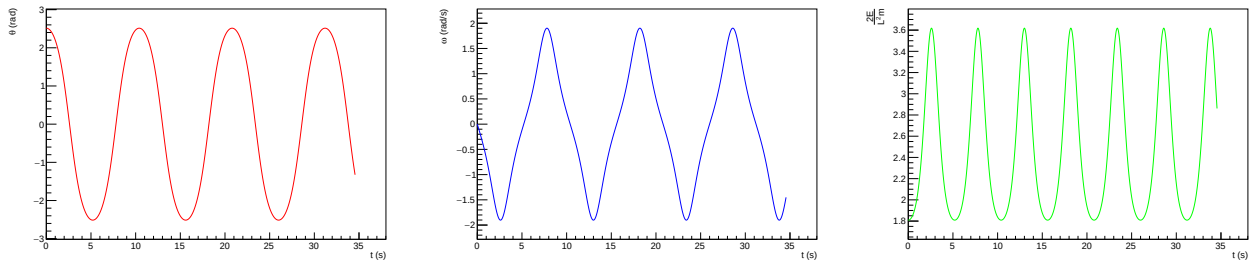

Figure 12: The θ, ω and E depending on time t with RK2 approximation. Initial condition: $\theta_0 = 0.8\pi$, $\omega_0 = 0$.

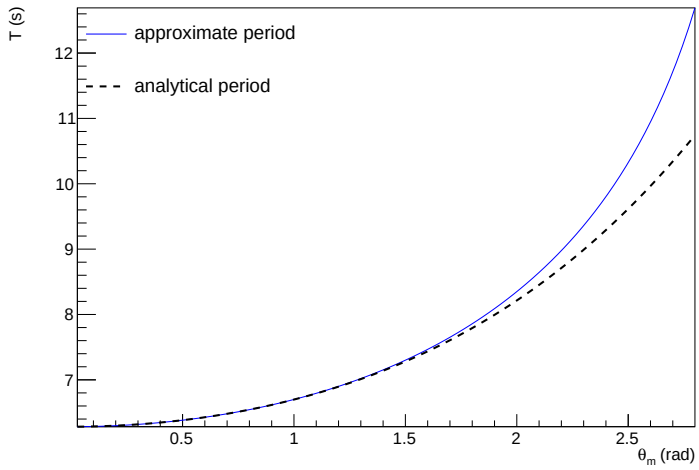

Figure 13: The RK2 approximate period and the analytical period.

Physics explanation:

The physics pendulum still has the periodical motion, whose period is depending on the maximum angle θ_m. However, the dependence on time is not simply sin cos shape.

When the θ_m is small, the approximate period and the analytical estimation are very close. However, when the θ_m is large, they differ from each other. This is because the analytical estimation is only to the fourth order of θ_m, and the residual terms can make a huge difference when the θ_m is large.

It is easy to think the period diverges when $\theta_m = \pi$, so the finite Taylor expansion must go wrong near $\theta_m = \pi$.

Relevant code:

The input functions for the physics pendulum ODE set:

```
//------------------------------------------------------------------------//

// x[0]: theta; x[1]: omega
double Pendulum::f_theta(double t, const vector<double> &x) {
        // dtheta/dt = omega
        return x[1];
}

//------------------------------------------------------------------------//

double Pendulum::f_omega_physics(double t, const vector<double> &x) {
        return -_g/_l*sin(x[0]) - _q*x[1] + _F*sin(_O*t+_P);
}

//------------------------------------------------------------------------//

double Pendulum::f_energy_physics(double t, const vector<double> &x) {
        // 2E/m/l^2
        return _g/_l*(1-cos(x[0]))+x[1]*x[1];
}

//------------------------------------------------------------------------//
```

Calculate the period from the numerical approximation and analytical estimation:

```
//------------------------------------------------------------------------//

double Pendulum::cal_period() const {
        double period = -1;
        vector<double> t0s;
        double t0;
        for(int i=0; i<_n_stps-1; i++) {
                if(_x[0][i]*_x[0][i+1] > 0) continue;
                t0 = (_x[0][i+1]*_t[i]-_x[0][i]*_t[i+1])/(_x[0][i+1]-_x[0][i]);
                t0s.push_back(t0);
        }

        if(t0s.size()>=3) {
                period = t0s[2]-t0s[0];
        }

        return period;
}

//------------------------------------------------------------------------//

double Pendulum::cal_analytical_period() const {
        double period = -1;
        double energy = _energy[0];
        double theta_max = acos(1-energy*_l/_g);
        if(_mode == 0) {
                period = 2*M_PI*sqrt(_l/_g);
        } else if(_mode == 1) {
                period = 2*M_PI*sqrt(_l/_g);
```

```
                period *= 1 + pow(theta_max,2)/16 + pow(theta_max,4)*11/3072;
                //period *= 1 + pow(theta_max,2)/16 + pow(theta_max,4)*11/3072 + pow(theta_max,6)*173/
        }

        return period;
}
```