PHYS580 Homework 3

Yicheng Feng
PUID: 0030193826

October 13, 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.
- `relativity_precession.h` and `relativity_precession.cxx` for the class `RelativityPrecession` to solve the Mercury precession due to relativity.
- `two_planet_orbit.h` and `two_planet_orbit.cxx` for the class `TwoPlanetOrbit` to solve the problem that two planets motion in solar system.
- `capacitor_2d.h` and `capacitor_2d.cxx` for the class `Capacitor2D` to solve the 2D capacitor problem.
- `charge_dist_3d.h` and `charge_dist_3d.cxx` for the class `ChargeDist3D` to solve the 3D charge distribution problem.
- `nintegrate_1d.h` and `nintegrate_1d.cxx` for the class `NIntegrate1D` to solve the 1D numerical integration.
- `hw3.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/hw3`.

(1) *[Problem 4.10 (p.113)]* Calculate the precession of the perihelion of Mercury, following the approach described in this section.

Physics explanation:

The Runge-Kutta 4^{th}-order is used for this problem. Time step is set to $\Delta t = 0.0001$ yr. The unit "yr" is the Earth year, and "AU" is the distance between the Earth and the Sun.

The force law predicted by general relativity is

$$F_G \approx \frac{GM_S M_M}{r^2}\left(1 + \frac{\alpha}{r^2}\right), \tag{1}$$

and the predicted value is $\alpha \approx 1.1 \times 10^{-8}$ AU^2, which is too small to be directly used. Therefore, we input various values of α and get the corresponding perihelion precession angular velocity. The α scan includes α values from 0 AU^2 to 0.0035 AU^2 (7 cases), 3 of them are shown in Fig. 1 and Fig. 2. After several envolutions, we can calculate the precession rate of each α by linear fitting in Fig. 2.

Then, we can get the relationship between the precession rate and the input α by linear fitting of the 7 cases of α. From Fig. 3, we can see the precession rate is proportional to α by a slope $k = 11993.3$ degree/yr/AU^2. Finally, we can use the actual α and the calculated k to get

$$\alpha \times k \approx 1.1 \times 10^{-8}\ \text{AU}^2 \times 11993.3\ \text{degree/yr/AU}^2 = 47.4933\ \text{arcsecond/century}, \tag{2}$$

which is consistent with the experimental result ~ 43 arcsecond/century.

Plots:

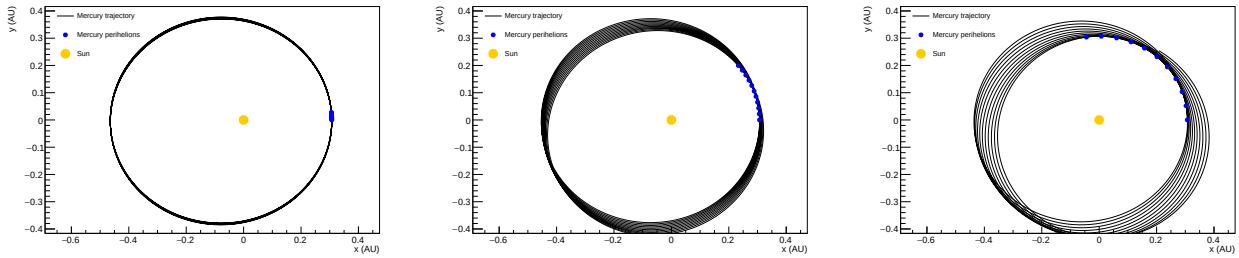

Figure 1: The position of Mercury during the precession with various α (left: 0.0002 AU^2; middle: 0.0015 AU^2; right: 0.0035 AU^2).

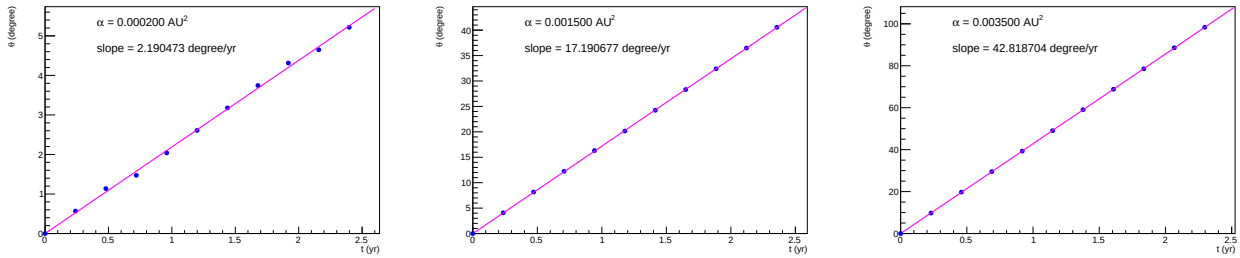

Figure 2: The angular position of Mercury during the precession with various α (left: 0.0002 AU^2; middle: 0.0015 AU^2; right: 0.0035 AU^2).

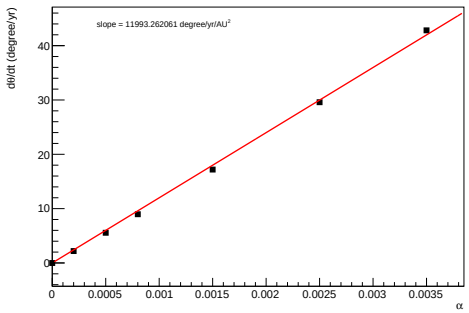

Figure 3: The relationship between precession rate and the input α.

Relevant code:

For the Runge-Kutta 4$^{\text{th}}$-order approximation:

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

void RungeKutta::cal_rk4() {
        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> F3(_n_eqns);
        vector<double> F4(_n_eqns);

        vector<double> x1(_n_eqns);
        vector<double> x2(_n_eqns);
        vector<double> x3(_n_eqns);
        vector<double> x4(_n_eqns);

        double t1, t2, t3, t4;

        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);
                }
                // F3
                t3 = t + 0.5*_dt;
                for(int ifv=0; ifv<nfv; ifv++) {
                        x3[ifv] = x[ifv] + F2[ifv]*0.5*_dt;
                }
                for(int ifv=0; ifv<nfv; ifv++) {
```

```
                        F3[ifv] = _fv[ifv](t3, x3);
                }
                // F4
                t4 = t + _dt;
                for(int ifv=0; ifv<nfv; ifv++) {
                        x4[ifv] = x[ifv] + F3[ifv]*_dt;
                }
                for(int ifv=0; ifv<nfv; ifv++) {
                        F4[ifv] = _fv[ifv](t4, x4);
                }

                t += _dt;
                for(int ifv=0; ifv<nfv; ifv++) {
                        x[ifv] += (F1[ifv]/6 + F2[ifv]/3 + F3[ifv]/3 + F4[ifv]/6)*_dt;
                }
        }

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

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

For the ODE set of the precession problem

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

double RelativityPrecession::f_x(double t, const vector<double> &x) {
        return x[2];
}

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

double RelativityPrecession::f_y(double t, const vector<double> &x) {
        return x[3];
}

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

double RelativityPrecession::f_vx(double t, const vector<double> &x) {
        double r = sqrt(x[0]*x[0]+x[1]*x[1]);
        return -_GM*x[0]/(r*r*r)*(1+_alpha/(r*r));
}

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

double RelativityPrecession::f_vy(double t, const vector<double> &x) {
        double r = sqrt(x[0]*x[0]+x[1]*x[1]);
        return -_GM*x[1]/(r*r*r)*(1+_alpha/(r*r));
}

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

For the initial conditions of the Mercury

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

RelativityPrecession::RelativityPrecession() {
```

```
        _a = 0.38665;
        _e = 0.206;
        _mp = 3.30e23/1.99e30;
        _GM = 4*M_PI*M_PI;
        _t_end = 5;
        _alpha = 0;

        _t_start = 0;
        _x_start = _a*(1-_e);
        _y_start = 0;
        _vx_start = 0;
        _vy_start = sqrt(_GM*(1+_e)/(1-_e)/_a);
        _c_start.push_back(_x_start);
        _c_start.push_back(_y_start);
        _c_start.push_back(_vx_start);
        _c_start.push_back(_vy_start);

        _energy = 0.5*_mp*(_vy_start*_vy_start + _vx_start*_vx_start);
        _energy += -_GM*_mp/sqrt(_x_start*_x_start + _y_start*_y_start);

        _alg = 0;
        _dt = 0.001;
}

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

For the position (xy and θ) of the Mercury perihelions

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

void RelativityPrecession::cal_perihelion() {
        for(int i=0; i<_n_stps; i++) {
                _rr.push_back(_x[0][i]*_x[0][i]+_x[1][i]*_x[1][i]);
        }
        _perihelion_t.push_back(_t[0]);
        _perihelion_x.push_back(_x[0][0]);
        _perihelion_y.push_back(_x[1][0]);
        _perihelion_theta.push_back(0);

        for(int i=1; i<_n_stps-1; i++) {
                if(_rr[i]<=_rr[i-1] && _rr[i]<_rr[i+1]) {
                        _perihelion_t.push_back(_t[i]);
                        _perihelion_x.push_back(_x[0][i]);
                        _perihelion_y.push_back(_x[1][i]);
                        double tmp_theta = atan2(_x[1][i], _x[0][i]);
                        _perihelion_theta.push_back(tmp_theta);
                }
        }
}

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

(2) *[Problem 4.14 (p.118)]* Simulate the orbits of Earth and Moon in the solar system by writing a program that accurately tracks the motions of both as they move about the Sun. Be careful (1) the different time scales present in this problem and (2) the correct initial velocities (i.e., set the initial velocity of Moon taking into account the motion of Earth around which it orbits).
[Also run your code for a hypothetical Moon that has either the same mass as Earth (e.g., binary planets), or a rather elliptical orbit.]

Physics explanation:

The Runge-Kutta 4^{th}-order is used for this problem. Time step is set to $\Delta t = 0.0001$ yr. The unit "yr" is the Earth year, and "AU" is the distance between the Earth and the Sun.

In this problem (**real case**), we think the orbit of the Earth is circular, which is almost the real case. Then the velocity of the Earth is 2π AU/yr. We set the initial condition to be that

- The Sun is fixed at position $(x, y) = (0, 0)$.
- The Earth is at position $(1, 0)$ with velocity $(v_x, v_y) = (0, 2\pi)$.
- The Moon is at position $(1 + 2.570 \times 10^{-3}, 0)$, because the distance between the Earth and the Moon is $R_{EM} = 2.570 \times 10^{-3}$ AU.
- In the frame of the Earth motion, which is non-inertial, the period of the Moon is $T_M = 29.53/365.2422$ yr $= 0.08085$ yr, so the initial relative velocity should be $v'_M = 2\pi R_{EM}/T_M = 0.199724$ AU/yr. Then

$$v_M = \frac{v_E}{R_E}(R_E + R_{EM}) + v'_M = 6.299333 + 0.199724 \text{ AU/yr} = 6.499057 \text{ AU/yr}, \tag{3}$$

  where the first term comes from the non-inertial frame and the second term comes from the relative velocity. The initial velocity of the Moon is $(0, 6.499057)$.

To better display the results, the distance between the Earth and the Moon is enlarged by a factor of 10. The result of this real case is shown in Fig. 4. We can see there are roughly 12 periods of the Moon in 1 period of the Earth (12 months).

For the **binary planets case**, the initial conditions are

- The Moon is set to have the same mass of the Earth $M_M = M_E$.
- The initial position of the Earth is $(0.9998, 0.02)$. The initial position of the Moon is $(0.9998, -0.02)$. The distance between them is $R_{EM} = 0.04$ AU.
- The velocity of them relative to the mass center of the binary system should be

$$\frac{GM_EM_M}{R_{EM}^2} = \frac{M_E v'^2_E}{R_{EM}/2} = \frac{M_M v'^2_M}{R_{EM}/2} \quad \Rightarrow \quad v'_E = v'_M = \sqrt{\frac{GM_E}{2R_{EM}}} \approx 0.0384938 \text{ AU/yr}. \tag{4}$$

  Therefore the initial velocity of the Earth is $(0.0384938, 2\pi)$, and the Moon is $(-0.0384938, 2\pi)$

The result of this binary case is shown in Fig. 5.

For the **elliptical Moon orbit case**, the initial conditions are similar to the real case, except the initial velocity of the moon. To make the elliptical Moon orbit, we need to increase the relative velocity a bit $v'_{EM} \to 1.2v'_{EM}$, so the initial velocity of the Moon should be

$$v_M = \frac{v_E}{R_E}(R_E + R_{EM}) + 1.2v'_M = 6.299333 + 1.2 * 0.199724 \text{ AU/yr} = 6.5390018 \text{ AU/yr}, \tag{5}$$

in the positive-y direction. The result is shown in the Fig. 6. To better display this, we again enlarge R_{EM} by a factor of 10.

Plots:

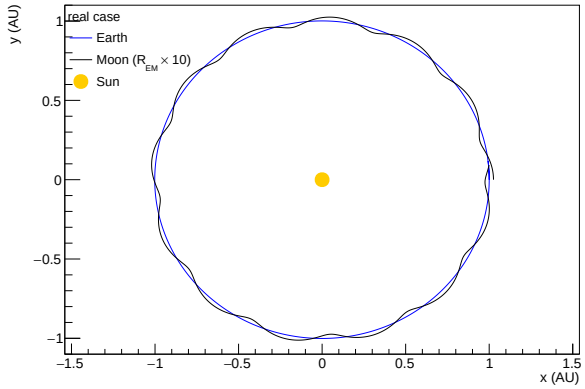

Figure 4: The trajectories of the Earth and the Moon in the real case ($R_{EM} \times 10$).

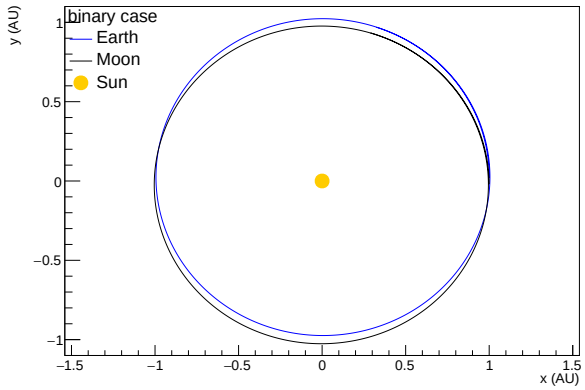

Figure 5: The trajectories of the Earth and the Moon in the binary case ($R_{EM} \times 1$).

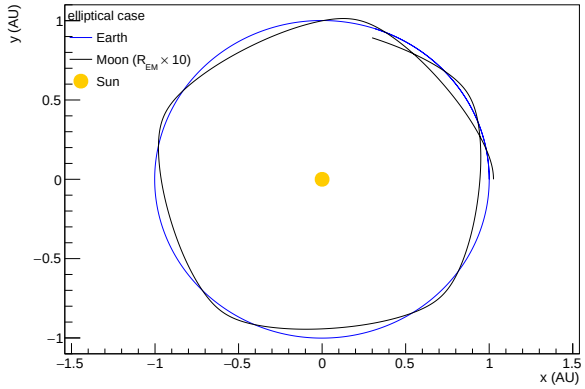

Figure 6: The trajectories of the Earth and the Moon in the elliptical Moon orbit case ($R_{EM} \times 10$).

Relevant code:

For the Runge-Kutta 4^{th}-order approximation, it is the same to that in problem (1).

For the ODE sets of the problem

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

double TwoPlanetOrbit::f_x1(double t, const vector<double> &x) {
        return x[4];
}

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

double TwoPlanetOrbit::f_y1(double t, const vector<double> &x) {
        return x[5];
}

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

double TwoPlanetOrbit::f_x2(double t, const vector<double> &x) {
        return x[6];
}

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

double TwoPlanetOrbit::f_y2(double t, const vector<double> &x) {
        return x[7];
}

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

double TwoPlanetOrbit::f_vx1(double t, const vector<double> &x) {
        double rEJ = sqrt((x[0]-x[2])*(x[0]-x[2])+(x[1]-x[3])*(x[1]-x[3]));
        return -_GM*x[0]/pow(x[0]*x[0]+x[1]*x[1], 1.5) - _GM*_m2*(x[0]-x[2])/rEJ/rEJ/rEJ;
}

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

double TwoPlanetOrbit::f_vy1(double t, const vector<double> &x) {
        double rEJ = sqrt((x[0]-x[2])*(x[0]-x[2])+(x[1]-x[3])*(x[1]-x[3]));
        return -_GM*x[1]/pow(x[0]*x[0]+x[1]*x[1], 1.5) - _GM*_m2*(x[1]-x[3])/rEJ/rEJ/rEJ;
}

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

double TwoPlanetOrbit::f_vx2(double t, const vector<double> &x) {
        double rEJ = sqrt((x[0]-x[2])*(x[0]-x[2])+(x[1]-x[3])*(x[1]-x[3]));
        return -_GM*x[2]/pow(x[2]*x[2]+x[3]*x[3], 1.5) - _GM*_m1*(x[2]-x[0])/rEJ/rEJ/rEJ;
}

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

double TwoPlanetOrbit::f_vy2(double t, const vector<double> &x) {
        double rEJ = sqrt((x[0]-x[2])*(x[0]-x[2])+(x[1]-x[3])*(x[1]-x[3]));
        return -_GM*x[3]/pow(x[2]*x[2]+x[3]*x[3], 1.5) - _GM*_m1*(x[3]-x[1])/rEJ/rEJ/rEJ;
}

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

For the initial condition of the real case

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

TwoPlanetOrbit::TwoPlanetOrbit() {
        // planet 1: Earth
        // planet 2: Moon

        _a1 = 1;
        _e1 = 0;//0.017;
        _m1 = 5.97e24/1.99e30;
        _a2 = 1+2.570e-3;
        _e2 = 0.;
        _m2 = 3.69397e-8;;

        _GM = 4*M_PI*M_PI;
        _t_end = 2;

        _t_start = 0;
        _x1_start = _a1*(1-_e1);
        _y1_start = 0;
        _x2_start = _a2*(1-_e2);
        _y2_start = 0;
        _vx1_start = 0;
        _vy1_start = sqrt(_GM*(1+_e1)/(1-_e1)/_a1);
        _vx2_start = 0;
        _vy2_start = _vy1_start*_a2/_a1+2*M_PI/29.53*365.2422*2.570e-3;
        _c_start.push_back(_x1_start);
        _c_start.push_back(_y1_start);
        _c_start.push_back(_x2_start);
        _c_start.push_back(_y2_start);
        _c_start.push_back(_vx1_start);
        _c_start.push_back(_vy1_start);
        _c_start.push_back(_vx2_start);
        _c_start.push_back(_vy2_start);

        _alg = 4;
        _dt = 0.0001;
}

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

For the initial conditionss of various cases

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

void two_planet_orbit_plot(int mod) {

        TString str_mods[3] = {"real", "binary", "elliptical"};
        double m2s[3] = {3.69397e-8, 3.0027e-6, 3.69397e-8};
        double rdf = 0.02;
        double x1s[3] = {1, sqrt(1-rdf*rdf), 1};
        double x2s[3] = {1+2.570e-3, sqrt(1-rdf*rdf), 1+2.570e-3};
        double y1s[3] = {0, rdf, 0};
        double y2s[3] = {0, -rdf, 0};
        double v1xs[3] = {0, 2*M_PI*sqrt(0.25/rdf*3.0027e-6), 0};
        double v2xs[3] = {0, -2*M_PI*sqrt(0.25/rdf*3.0027e-6), 0};
        double v1ys[3] = {2*M_PI, 2*M_PI, 2*M_PI};
        double v2ys[3] = {2*M_PI*(1+2.570e-3)+2*M_PI/29.53*365.2422*2.570e-3,
        2*M_PI,
        2*M_PI*(1+2.570e-3)+2*M_PI/29.53*365.2422*2.570e-3*1.2};
        double t_ends[3] = {1.02, 1.2, 1.2};

```

```
        TString str_mod = str_mods[mod];
        double m2 = m2s[mod];
        double x1 = x1s[mod];
        double x2 = x2s[mod];
        double y1 = y1s[mod];
        double y2 = y2s[mod];
        double v1x = v1xs[mod];
        double v2x = v2xs[mod];
        double v1y = v1ys[mod];
        double v2y = v2ys[mod];
        double t_end = t_ends[mod];

        TwoPlanetOrbit tpo;
        tpo.set_m2(m2);
        tpo.set_x1_start(x1);
        tpo.set_x2_start(x2);
        tpo.set_y1_start(y1);
        tpo.set_y2_start(y2);
        tpo.set_vx1_start(v1x);
        tpo.set_vx2_start(v2x);
        tpo.set_vy1_start(v1y);
        tpo.set_vy2_start(v2y);
        tpo.set_t_end(t_end);
        tpo.cal();

        // ... plot code ...

}

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

(3) *[Problem 5.7 (p.143)]* Write two programs to solve the capacitor problem of Figures 5.6 and 5.7, one using the Jacobi method and one using the SOR algorithm. For a fixed accuracy (as set by the convergence test) compare the number of iterations, N_{iter}, that each algorithm requires a function of the number of grid elements, L. Show that for the Jacobi method $N_{\text{iter}} \sim L^2$, while with SOR $N_{\text{iter}} \sim L$.

Physics explanation:

We let the edge of the square to be l and the center of it to be the origin. The left plate is set to $x = -0.15l$ and symmetrical in y direction with potential $V = 1$, whereas the right plate is set to $x = 0.15l$ and symmetrical in y direction with potential $V = -1$. The average accuracy is set to be 1×10^{-6}.

For the **Jacobi relaxation**, the grid edges d are scanned from $0.005l$ to $0.05l$. The results of $d = 0.01l$ is shown in Fig. 7.

d/l	0.05	0.04	0.02	0.01	0.005
number of divisions in each direction L	21	27	51	101	201
number of iteration steps N_{iter}	241	351	985	3138	9388

The relationship between N_{iter} and L is also plotted in Fig. 8. We can see the shape looks like a parabola, which means $N_{\text{iter}} \sim L^2$.

For the **SOR relaxation**, we suppose the old potential is $V_{\text{old}}(i, j)$. We use the Gauss-Seidel relaxation and get $V*(i, j)$. The new potential would be

$$V_{\text{new}}(i, j) = V_{\text{old}}(i, j) + \alpha \left(V*(i, j) - V_{\text{old}}(i, j) \right), \tag{6}$$

where α is the relaxation factor. From the textbook, we choose the optimal relaxation factor

$$\alpha = \frac{2}{1 + \pi/L}. \tag{7}$$

The SOR approach does not always converge, so I can only pick the cases with not too much grids, d from $0.04l$ to $0.1l$. The relationship between N_{iter} and L is also plotted in Fig. 10. We can see the shape looks like a straight line, which means $N_{\text{iter}} \sim L$.

d/l	0.1	0.08	0.05	0.04
number of divisions in each direction L	11	13	21	27
number of iteration steps N_{iter}	49	66	128	185

Plots:

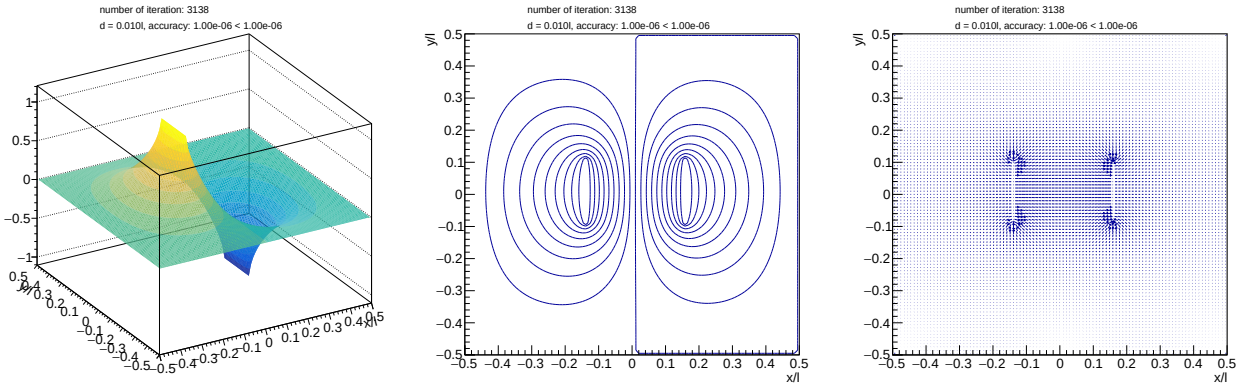

Figure 7: Jacobi relaxation results for the potential (left pad, middle pad) and the electric field (right pad) with grid size $d = 0.01l$.

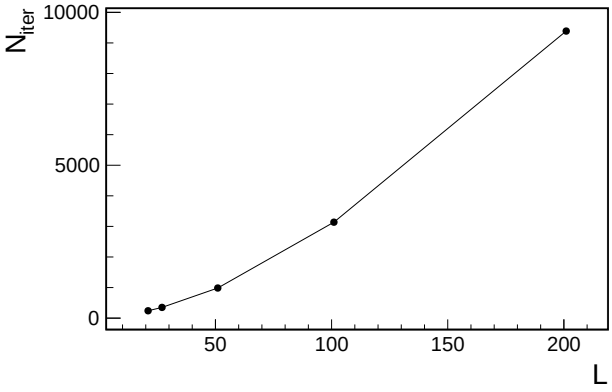

Figure 8: Jacobi relaxation, the relationship between N_{iter} and L.

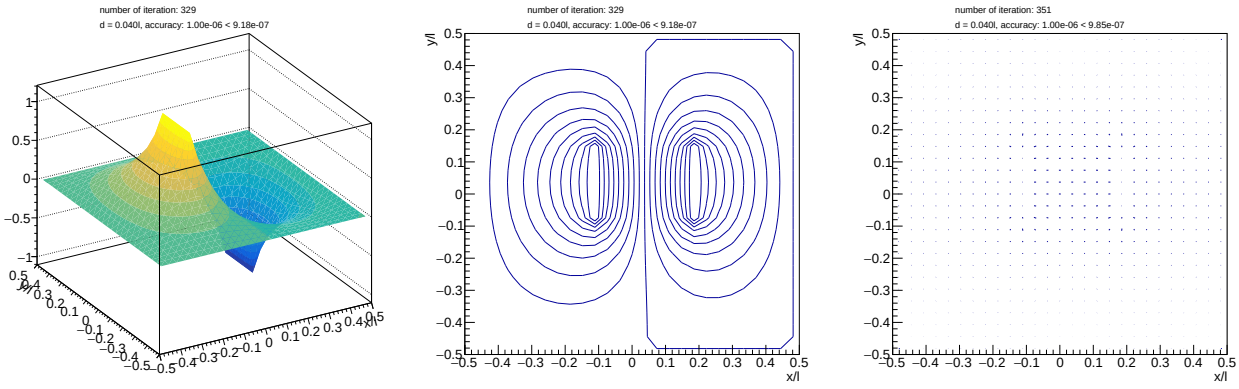

Figure 9: SOR relaxation results for the potential (left pad, middle pad) and the electric field (right pad) with grid size $d = 0.04l$.

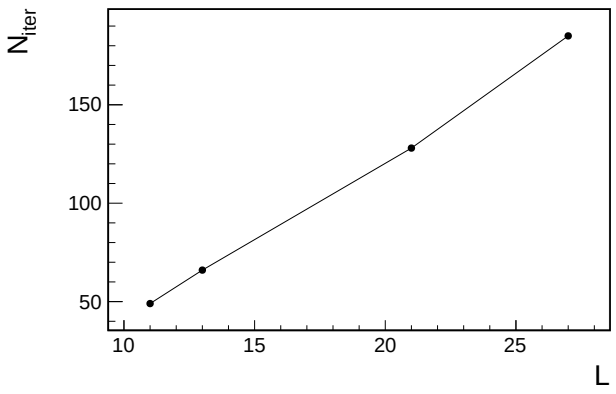

Figure 10: SOR relaxation, the relationship between N_{iter} and L.

Relevant code:

For Jacobi relaxation

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

void Capacitor2D::cal_once_Jacobi() {

        if(!check()) return;

        vector< vector<double> > Vnew = _V;
        _tmp_acc = 0;

        for(int j=1; j<_N-1; j++) {
                for(int i=1; i<_N-1; i++) {
                        if(_V[i][j]==1 || _V[i][j]==-1) continue;
                        Vnew[i][j] = (_V[i-1][j] + _V[i+1][j] + _V[i][j-1] + _V[i][j+1] + _rho[i][j]);
                        Vnew[i][j] *= 0.25;
                        _tmp_acc += fabs(Vnew[i][j] - _V[i][j]);
                }
        }

        _tmp_acc = _tmp_acc/_N/_N;

        _V = Vnew;

        _n_iter ++;
}

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

For SOR relaxation

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

void Capacitor2D::cal_once_SOR() {

        if(!check()) return;

        vector< vector<double> > Vold = _V;
        _tmp_acc = 0;

        for(int j=1; j<_N-1; j++) {
```

```
                for(int i=1; i<_N-1; i++) {
                        if(_V[i][j]==1 || _V[i][j]==-1) continue;
                        _V[i][j] = (_V[i-1][j] + _V[i+1][j] + _V[i][j-1] + _V[i][j+1] + _rho[i][j]);
                        _V[i][j] *= 0.25;
                }
        }

        for(int j=1; j<_N-1; j++) {
                for(int i=1; i<_N-1; i++) {
                        if(_V[i][j]==1 || _V[i][j]==-1) continue;
                        _V[i][j] = _alpha*_V[i][j] + (1-_alpha)*Vold[i][j];
                        _tmp_acc += fabs(Vold[i][j] - _V[i][j]);
                }
        }

        _tmp_acc = _tmp_acc/_N/_N;

        _n_iter ++;
}

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

(4) *[Problem 5.8 (p.147)]* Extend our treatment of a point charge in a metal box to deal with wht case in which the charge is located near one face of the box. Study how the equipotential contours are affected by the proximity of a grounded surface (the face of the box).

Physics explanation:

The grounded metal box is described as following: $-0.5l < x < 0.5l$, $-0.5l < y < 0.5l$, and $-0.5l < z < 0.5l$. The position of the point charge $Q/\varepsilon_0 = 1$ is $(x, y, z) = (-0.4l, 0, 0)$.

The edge of grid is $d = 0.05l$, so the number of grid in each cordinate is $L = 21$. We use the Jacobi relaxation for this problem. The potential and electric field distribution of the plane $z = 0$ are shown in Fig. 11.

From the middle plot, we can see the equipotential contours are dense on the left of the point charge (close to the wall), whereas sparse on the right (far from the wall). The outer contours are not any close to a right circule.

At the regime close to the point charge, the contours still show the grid symmetry (like squares).

Plots:

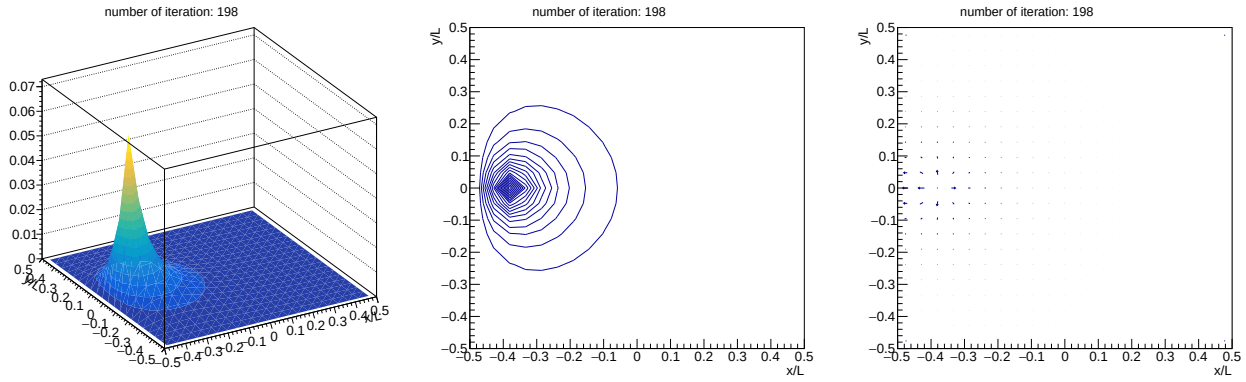

Figure 11: Jacobi relaxation results for the potential (left pad, middle pad) and the electric field (right pad) with grid size $d = 0.05l$. The plots are for the plane $z = 0$.

Relevant code:

For the input charge distribution

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

void ChargeDist3D::set_rho(double x1, double x2, double y1, double y2,
double z1, double z2, double r) {
        int i1 = int((x1+0.5)/_d+0.5);
        int i2 = int((x2+0.5)/_d+0.5);
        int ni = fabs(i2-i1) + 1;
        int j1 = int((y1+0.5)/_d+0.5);
        int j2 = int((y2+0.5)/_d+0.5);
        int nj = fabs(j2-j1) + 1;
        int k1 = int((z1+0.5)/_d+0.5);
        int k2 = int((z2+0.5)/_d+0.5);
        int nk = fabs(k2-k1) + 1;
        for(int k=k1; k<=k2; k++) {
                for(int j=j1; j<=j2; j++) {
                        for(int i=i1; i<=i2; i++) {
                                _rho[k][j][i] = r/ni/nj/nk;
                        }
                }
```

```
        }
}

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

For the Jacobi relaxation

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

void ChargeDist3D::cal_once_Jacobi() {

        if(!check()) return;

        vector< vector< vector<double> > > Vnew = _V;
        _tmp_acc = 0;

        for(int i=1; i<_N-1; i++) {
                for(int j=1; j<_N-1; j++) {
                        for(int k=1; k<_N-1; k++) {
                                Vnew[i][j][k] = (_V[i-1][j][k] + _V[i+1][j][k] + _V[i][j-1][k]
                                + _V[i][j+1][k] + _V[i][j][k-1] + _V[i][j][k+1] + _rho[i][j][k])/6;
                                _tmp_acc += fabs(Vnew[i][j][k] - _V[i][j][k]);
                        }
                }
        }

        _tmp_acc = _tmp_acc/_N/_N/_N;

        _V = Vnew;

        _n_iter ++;
}

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

(5) *[Problem E.2 (p.510)]* Calculate the period of a non-linear oscillator described by

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -\sin\theta \tag{8}$$

by numerically integrating

$$f(\theta) = \sqrt{8}\int_0^{\theta_m} \frac{\mathrm{d}\theta}{\sqrt{\cos\theta - \cos\theta_m}} \tag{9}$$

for several values of the maximum angle θ_m, using the trapezoidal rule, Simpson's rule, or the Romberg integration method.

Physics explanation:

We will use the trapezoidal rule and Simpson's rule in this problem.

The trapezoidal rule is

$$\int_a^b f(x)\mathrm{d}x \approx \sum_{k=1}^{N} \frac{f(x_{k-1}) + f(x_k)}{2}\Delta x_k = \frac{\Delta x}{2}\left(f(x_0) + 2f(x_1) + 2f(x_2) + \ldots + 2f(x_{N-1}) + f(x_N)\right). \tag{10}$$

Simpson's rule is

$$\int_a^b f(x)\mathrm{d}x \approx \frac{\Delta x}{3}\left(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + 2f(x_4) + \ldots + 4f(x_{N-1}) + f(x_N)\right). \tag{11}$$

When $\theta = 0$ or $\theta = \theta_m$, the function $f(\theta) \to \infty$, which make the numerical integration not doable. To avoid this, we change the limits a little bit by 10^{-8} (integration from 10^{-8} to $\theta_m - 10^{-8}$).

θ_m		0.05	0.1	0.5	1.0	2.0	3.0
exact T		6.28417	6.28711	6.38279	6.69998	8.34975	16.1555
trapezoidal	T_n	6.28251	6.28710	6.38871	6.71085	8.37326	16.2429
	$\lvert T_n - T\rvert/T$	2.64×10^{-4}	1.65×10^{-6}	9.27×10^{-4}	1.62×10^{-3}	2.81×10^{-3}	5.81×10^{-3}
Simpson's	T_n	6.28196	6.28615	6.38588	6.70633	8.36422	16.2144
	$\lvert T_n - T\rvert/T$	3.51×10^{-4}	1.53×10^{-4}	4.84×10^{-4}	9.48×10^{-4}	1.73×10^{-3}	3.64×10^{-3}

Generally, the Simpson integration is closer to the exact value, but it is not necessarily always the case.

Relevant code:

For the trapezoidal rule

```
1  //------------------------------------------------------------------------//
2
3  void NIntegrate1D::cal_trapzoidal() {
4
5          double dx = (_x2 - _x1)/_n;
6          double wt;
7
8          _output = 0;
9
10         for(int i=0; i<=_n; i++) {
11                 if(i==0 || i==_n) {
12                         wt = 1;
13                 } else {
14                         wt = 2;
```

```
                }

                double x = _x1 + i*dx;
                _output += wt*_f(x);
        }

        _output *= 0.5*dx;
}

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

For the Simpson's rule

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

void NIntegrate1D::cal_Simpson() {

        if(_n%2 == 1) _n ++;

        double dx = (_x2 - _x1)/_n;
        double wt;

        _output = 0;

        for(int i=0; i<=_n; i++) {
                if(i==0 || i==_n) {
                        wt = 1;
                } else {
                        wt = i%2==0?2:4;
                }

                double x = _x1 + i*dx;
                _output += wt*_f(x);
        }

        _output *= dx/3.0;
}

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

For the function $f(\theta)$

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

double real_oscillator_period(double x) {

        return sqrt(8.0)/sqrt(cos(x)-cos(G_MX));
}

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

void nintegrate_1d_cal() {

        map<double,double> exact;
        exact[0.05] = 6.28417;
        exact[0.1]  = 6.28711;
        exact[0.5]  = 6.38279;
        exact[1.0]  = 6.69998;
        exact[2.0]  = 8.34975;
        exact[3.0]  = 16.1555;
        double T = exact[G_MX];
```

```
20
21          NIntegrate1D ni1d(1e-8,G_MX-1e-8, real_oscillator_period);
22          ni1d.set_n(1000000);
23          double output;
24
25          ni1d.set_alg(0);
26          output = ni1d.cal();
27          cout << "trapezoidal" << endl;
28          cout << "xm = " << G_MX << ", T = " << output << ", err = " << fabs(output-T)/T << endl;;
29
30          ni1d.set_alg(1);
31          output = ni1d.cal();
32          cout << "Simpson" << endl;
33          cout << "xm = " << G_MX << ", T = " << output << ", err = " << fabs(output-T)/T << endl;;
34
35  }
36
37  //-------------------------------------------------------------------------//
```