PHYS580 Homework 1

Yicheng Feng
PUID: 0030193826

September 8, 2019

Before my solutions to the lab activities, I want to set up my typical **workflow**, which will also be used in this course. 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.
- `bicycling.h` and `bicycling.cxx` for the class `Bicycling` to solve problem (2).
- `projectile.h` and `projectile.cxx` for the class `Projectile` to solve problem (3).
- `projectile_3d.h` and `projectile_3d.cxx` for the class `Projectile3D` to solve problem (4).
- `hw1.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/hw1`.

(1) Problem 1.4 from the Giordano-Nakanishi book (p.16), with $\tau_A/\tau_B = 1/3, 1, \text{and} 3$. Make sure to include a discussion of any relationships among the time scales (τ_A, τ_B) and the time step you chose for the numerical work. Explore and interpret the results for various initial conditions such as $N_A(0)/\tau_A > N_B(0)/\tau_B$, or $N_A(0)/\tau_A < N_B(0)/\tau_B$.
[Optional: How would the differential equations change if B becomes A when it decays?]

Analytical solution:

The ODE set of this sequential decay $A \to B \to \dots$ is provided as:

$$\begin{aligned} \frac{\mathrm{d}N_A}{\mathrm{d}t} &= -\frac{N_A}{\tau_A}, \\ \frac{\mathrm{d}N_B}{\mathrm{d}t} &= \frac{N_A}{\tau_A} - \frac{N_B}{\tau_B}. \end{aligned} \tag{1}$$

It is straightforward to get the solution of the first equation

$$\int \frac{\mathrm{d}N_A}{N_A} = \frac{1}{\tau_A} \int \mathrm{d}t \quad \Rightarrow \quad N_A(t) = N_A(0) e^{-t/\tau_A}. \tag{2}$$

Then, the second differential equation is the set can be written as

$$\frac{\mathrm{d}N_B}{\mathrm{d}t} = \frac{N_A(0)}{\tau_A} e^{-t/\tau_A} - \frac{N_B}{\tau_B}. \tag{3}$$

Without loss of generality, we assume the format of $N_B(t)$ to be

$$N_B(t) = B(t) e^{-t/\tau_B}. \tag{4}$$

We plug this into Eq. 3 and get

$$\frac{\mathrm{d}B}{\mathrm{d}t} e^{-t/\tau_B} = \frac{N_A(0)}{\tau_A} e^{-t/\tau_A}. \tag{5}$$

There could be two cases: (a) $\tau_A = \tau_B$, (b) $\tau_A \neq \tau_B$. For case (a), it is easy to get

$$B(t) = N_B(0) + N_A(0) t/\tau_A \quad \Rightarrow \quad N_B(t) = \left[N_B(0) + N_A(0) t/\tau_A\right] e^{-t/\tau_B}. \tag{6}$$

For case (b), we can also get $B(t)$ from the following integral

$$\begin{aligned} B(t) =& \frac{N_A(0)}{\tau_A} \int e^{-t(\frac{1}{\tau_A} - \frac{1}{\tau_B})} \mathrm{d}t = N_B(0) + \frac{N_A(0)}{\tau_A/\tau_B - 1} \left(e^{-t(\frac{1}{\tau_A} - \frac{1}{\tau_B})} - 1 \right) \\ N_B(t) =& N_B(0) e^{-t/\tau_B} + \frac{N_A(0)}{\tau_A/\tau_B - 1} \left(e^{-t/\tau_A} - e^{-t/\tau_B} \right) \end{aligned} \tag{7}$$

This is a simple ODE set, and we use the Euler approximation with $t_n = n\Delta t$.

$$\begin{aligned} (N_A)_{n+1} &= (N_A)_n - \frac{\Delta t}{\tau_A} (N_A)_n, \\ (N_B)_{n+1} &= (N_B)_n + \frac{\Delta t}{\tau_A} (N_A)_n - \frac{\Delta t}{\tau_B} (N_B)_n. \end{aligned} \tag{8}$$

The plots and relevant code will be attached later.

Discussion:

As for the relationship between **the time step Δt and the mean lifes (τ_A and τ_B)**, we know the global error of Euler approximation is $\mathcal{O}(\Delta t)$, so the smaller Δt means smaller error. To test this, we

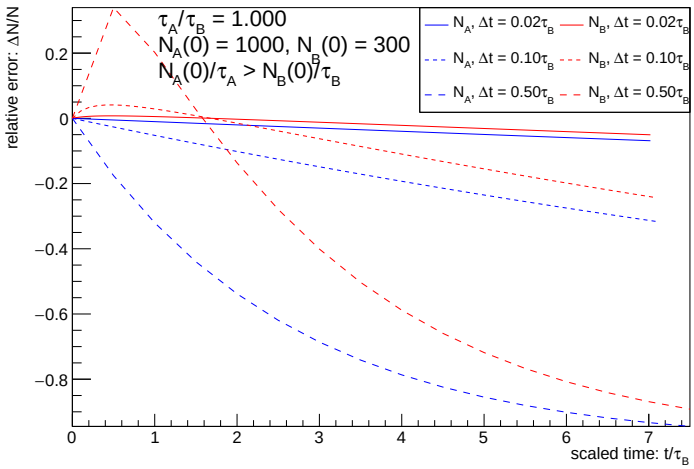

Figure 1: Relative error with various Δt depending on time.

use various $\Delta t = 0.02\tau_B, 0.1\tau_B,$ and $0.5\tau_B$ with other conditions the same: $\tau_A/\tau_B = 1$, $N_A(0) = 1000$, and $N_B(0) = 300$. Figure 1 shows the results as expected.

Figure 2 shows the exact nuclei number (dash line) and the approximate nuclei number by Euler method (solid line) of two different types A (blue) and B (red). We can see the Euler approximation is very close to the exact result, so the numerical approach we use in this problem is suitable.

The plots have various τ_A/τ_B values 1/3 (top), 1 (mid), and 3 (bottom); various initial condition $N_A(0) = 1000, N_B(0) = 300$ (left), and $N_A = 300, N_B = 1000$. Then the left plots have $N_A(0)/\tau_A > N_B(0)/\tau_B$, while the right $N_A(0)/\tau_A < N_B(0)/\tau_B$. In terms of the **initial conditions**, if we have $N_A(0)/\tau_A > N_B(0)/\tau_B$, then N_B increases at $t = 0$ and then decreases; if we have $N_A(0)/\tau_A < N_B(0)/\tau_B$, then N_B always decreases along time.

"*[Optional: How would the differential equations change if B becomes A when it decays?]*"

If B can also become A when it "decays", te differential equations will change into:

$$\begin{aligned} \frac{\mathrm{d}N_A}{\mathrm{d}t} &= -\frac{N_A}{\tau_A} + \frac{N_B}{\tau_B}, \\ \frac{\mathrm{d}N_B}{\mathrm{d}t} &= +\frac{N_A}{\tau_A} - \frac{N_B}{\tau_B}. \end{aligned} \tag{9}$$

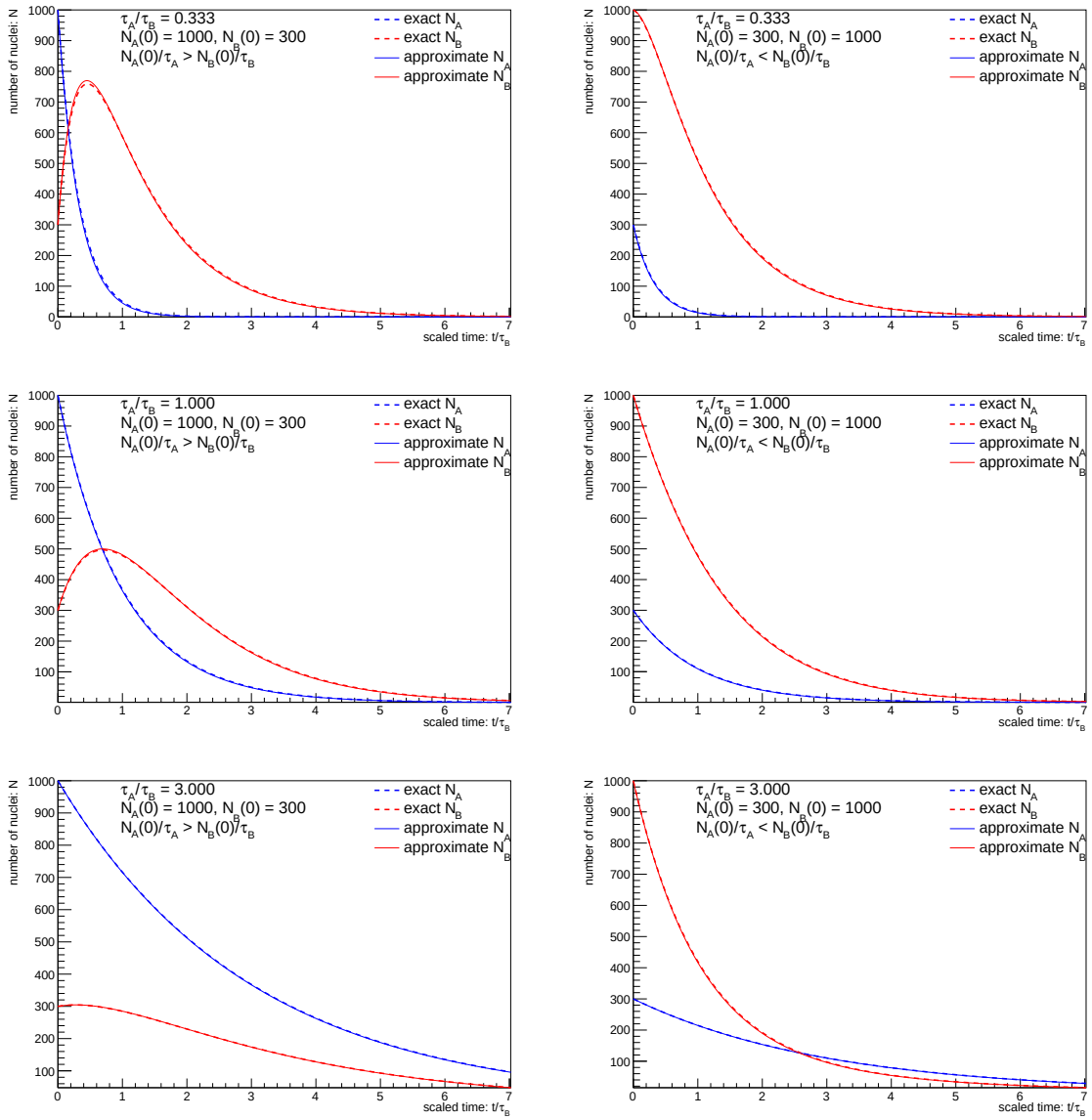

Figure 2: The plots show the exact nuclei number (dash line) and the approximate nuclei number (real line) of two different types A (blue) and B (red). The plots have various τ_A/τ_B values 1/3 (top), 1 (mid), and 3 (bottom); various initial condition $N_A(0) = 1000, N_B(0) = 300$ (left), and $N_A = 300, N_B = 1000$. Then the left plots have $N_A(0)/\tau_A > N_B(0)/\tau_B$, while the right $N_A(0)/\tau_A < N_B(0)/\tau_B$. The time step is $\Delta t = 0.02\tau_B$.

Relevant code:

In the file `runge_kutta.cxx`:

```
1 //------------------------------------------------------------------------//
2
3 void RungeKutta::cal_rk1() {
4         if(!check()) {
5                 cout << "WARNING: check() not passed, no calculation!" << endl;
6                 return;
7         }
8         //cout << "check() passed" << endl;
```

```
9
10          _n_stps = 0;
11
12          double t = _t_start;
13          vector<double> x = _x_start;
14
15          vector<double> F1(_n_eqns);
16
17          vector<double> x1(_n_eqns);
18
19          double t1;
20
21          int nfv = _fv.size(); // nfv = _n_eqns
22          while(!_stop(t, x)) {
23                  // Fill into output
24                  _t.push_back(t);
25                  for(int ifv=0; ifv<nfv; ifv++) {
26                          _x[ifv].push_back(x[ifv]);
27                  }
28                  _n_stps ++;
29
30                  // F1
31                  t1 = t;
32                  for(int ifv=0; ifv<nfv; ifv++) {
33                          x1[ifv] = x[ifv];
34                  }
35                  for(int ifv=0; ifv<nfv; ifv++) {
36                          F1[ifv] = _fv[ifv](t1, x1);
37                  }
38
39                  t += _dt;
40                  for(int ifv=0; ifv<nfv; ifv++) {
41                          x[ifv] += F1[ifv]*_dt;
42                  }
43          }
44
45          _t.push_back(t);
46          for(int ifv=0; ifv<nfv; ifv++) {
47                  _x[ifv].push_back(x[ifv]);
48          }
49          _n_stps ++;
50  }
51
52  //------------------------------------------------------------------------//
```

In the file **sequential_decay.cxx**,

```
1   //------------------------------------------------------------------------//
2
3   double SequentialDecay::_tau_A_B = 1;
4
5   //------------------------------------------------------------------------//
6
7   bool SequentialDecay::stop(double t, const vector<double> &x) {
8           return t>7;
9   }
10
11  //------------------------------------------------------------------------//
12
13  double SequentialDecay::f_N_A(double t, const vector<double> &x) {
14          return -x[0]/_tau_A_B;
```

```
15 }
16
17 //------------------------------------------------------------------------//
18
19 double SequentialDecay::f_N_B(double t, const vector<double> &x) {
20         return x[0]/_tau_A_B-x[1];
21 }
22
23 //------------------------------------------------------------------------//
24
25 double SequentialDecay::f_N_A_exact(double t) {
26         return _N_A_start*exp(-t/_tau_A_B);
27 }
28
29 //------------------------------------------------------------------------//
30
31 double SequentialDecay::f_N_B_exact(double t) {
32         if(_tau_A_B == 1) {
33                 return (_N_B_start + _N_A_start*t/_tau_A_B)*exp(-t);
34         } else {
35                 return _N_A_start/(1-_tau_A_B)*(exp(-t)-exp(-t/_tau_A_B))+_N_B_start*exp(-t);
36         }
37 }
38
39 //------------------------------------------------------------------------//
40
41 SequentialDecay::SequentialDecay() {
42         _dt = 0.02;
43         _t_start = 0;
44         _N_A_start = 1000;
45         _N_B_start = 0;
46         _x_start.push_back(_N_A_start);
47         _x_start.push_back(_N_B_start);
48 }
49
50 //------------------------------------------------------------------------//
51
52 void SequentialDecay::cal() {
53         if(!check()) {
54                 cout << "ERROR: check() not passed! no calculation!" << endl;
55         }
56
57         RungeKutta rk(_n_eqns, _dt, _t_start, _x_start);
58         rk.set_stop(stop);
59         rk.load_f(f_N_A);
60         rk.load_f(f_N_B);
61
62         rk.cal_rk1();
63
64         _t = rk.get_t();
65         _x = rk.get_x();
66
67         _N_A = _x[0];
68         _N_B = _x[1];
69
70         _n_stps = rk.get_n_stps();
71 }
72
73 //------------------------------------------------------------------------//
74
```

```
75  void SequentialDecay::cal_exact() {
76          double t = _t_start;
77          vector<double> x;
78          x.push_back(_N_A_start);
79          x.push_back(_N_B_start);
80
81          _n_stps_exact = 0;
82
83          _t_exact.clear();
84          _N_A_exact.clear();
85          _N_B_exact.clear();
86          _x_exact.clear();
87
88          while(!stop(t, x)) {
89                  _t_exact.push_back(t);
90                  _N_A_exact.push_back(x[0]);
91                  _N_B_exact.push_back(x[1]);
92                  _n_stps_exact ++;
93
94                  t += _dt;
95                  x[0] = f_N_A_exact(t);
96                  x[1] = f_N_B_exact(t);
97          }
98          _t_exact.push_back(t);
99          _N_A_exact.push_back(x[0]);
100         _N_B_exact.push_back(x[1]);
101         _n_stps_exact ++;
102
103         _x_exact.push_back(_N_A_exact);
104         _x_exact.push_back(_N_B_exact);
105 }
106
107 //-------------------------------------------------------------------------//
108
109 void SequentialDecay::cal_error() {
110         if(_N_A.size() != _N_A_exact.size() || _N_B.size() != _N_B_exact.size()) {
111                 cout << "ERROR: cal_error(): vector size not matched!" << endl;
112                 cout << _N_A.size() << " != " << _N_A_exact.size() << endl;
113                 return;
114         }
115
116         for(int i=0; i<_N_A.size(); i++) {
117                 if( _t[i] != _t_exact[i]) {
118                         cout << "ERROR: time not match!" << endl;
119                 }
120                 //double error = fabs(_N_A[i] - _N_A_exact[i])/_N_A_exact[i];
121                 double error = (_N_A[i] - _N_A_exact[i])/_N_A_exact[i];
122                 _N_A_error.push_back(error);
123         }
124
125         for(int i=0; i<_N_B.size(); i++) {
126                 //double error = fabs(_N_B[i] - _N_B_exact[i])/_N_B_exact[i];
127                 double error = (_N_B[i] - _N_B_exact[i])/_N_B_exact[i];
128                 _N_B_error.push_back(error);
129         }
130
131         _x_error.push_back(_N_A_error);
132         _x_error.push_back(_N_B_error);
133 }
```

(2) *[textbook page 24, problem 2.2]*
Investage the effect of varying both the rider's power and frontal area on the ultimate velocity. In particular, for a rider in the middle of a pack, the effective frontal area is about 30 percent less than for a rider at the front. How much less energy does a rider in the pack expend than does one at the front, assuming they both move at a velocity of 13 m/s?

If we only focus on the ultimate velocity, this problem could be solved analytically with the model provided in the textbook:

$$\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{P}{mv} - \frac{1}{2m} C\rho A v^2. \tag{10}$$

When it reaches the ultimate velocity v_u, the velocity doesn't change which means $\mathrm{d}v/\mathrm{d}t = 0$

$$0 = \frac{P}{mv_u} - \frac{1}{2m} C\rho A v_u^2 \quad \Rightarrow \quad v_u = \left(\frac{2P}{C\rho A}\right)^{1/3}. \tag{11}$$

We can see that both increasing the rider's power and decreasing the frontal area can make the ultimate velocity higher. If we set $P = 400$ W, $C = 0.5$, $A = 0.33$ m^2, and $\rho = 1.292$ kg m^{-3}, then we can estimate the ultimate velocity

$$v_u = \left(\frac{2P}{C\rho A}\right)^{1/3} = \left(\frac{2 \times 400 \text{ W}}{0.5 \times 1.292 \text{ kg m}^{-3} \times 0.33 \text{ m}^2}\right)^{1/3} \approx 15.5 \text{ m/s}, \tag{12}$$

which is reasonable.

We can also use the Euler approximation to get the numerical solution of Eq. 10. We can see the results as expected above.

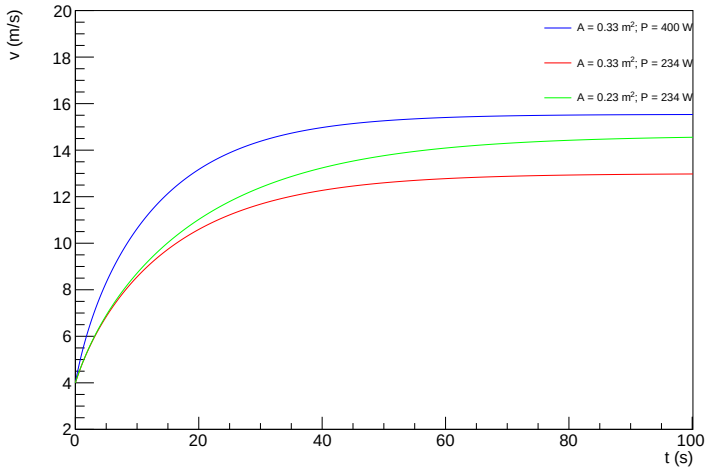

Figure 3: The bicycling velocity depending on the time with various front area A and power P.

Given the ultimate velocity $v_u = 13$ m/s, we can also calculate the power P from Eq. 11. The rider on the front has the front area $A_f = 0.33$ m^2, while the rider in the pack has the effective front area $A_p = 0.7A_f = 0.231$ m^2.

$$\begin{aligned} P_f &= \frac{C\rho A_f v_u^3}{2} = 0.5 \times 0.5 \times 1.292 \text{ kg m}^{-3} \times 0.33 \text{ m}^2 \times (13 \text{ m/s})^3 = 234 \text{ W}, \\ P_p &= \frac{C\rho A_p v_u^3}{2} = 0.5 \times 0.5 \times 1.292 \text{ kg m}^{-3} \times 0.241 \text{ m}^2 \times (13 \text{ m/s})^3 = 164 \text{ W}. \end{aligned} \tag{13}$$

Because P is proportional to A, the rider in the pack can spare 30% power than the rider on the front.

Relevant code:
For the Euler method, the code has been listed previously in problem (1), page 4.
In the file `bicycling.cxx`:

```
1  //--------------------------------------------------------------------------//
2
3  double Bicycling::_C = 0.5;
4  double Bicycling::_P = 400;
5  double Bicycling::_A = 0.33;
6  double Bicycling::_m = 70;
7  double Bicycling::_rho = 1.292;
8
9  //--------------------------------------------------------------------------//
10
11 Bicycling::Bicycling() {
12         _C = 0.5;
13         _P = 400;
14         _A = 0.33;
15         _m = 70;
16         _rho = 1.292;
17
18         _dt = 0.1;
19         _t_start = 0;
20         _v_start = 4;
21         _x_start.push_back(_v_start);
22 }
23
24 //--------------------------------------------------------------------------//
25
26 double Bicycling::f_v(double t, const vector<double> &x) {
27         return _P/_m/x[0] - 0.5/_m*_C*_rho*_A*x[0]*x[0];
28 }
29
30 //--------------------------------------------------------------------------//
31
32 bool Bicycling::stop(double t, const vector<double> &x) {
33         return t>100;
34 }
35
36 //--------------------------------------------------------------------------//
37
38 bool Bicycling::check() const {
39         if(_n_eqns != _x_start.size()) {
40                 cout << "ERROR: equation number not match!" << endl;
41                 return false;
42         }
43
44         return true;
45 }
46
47 //--------------------------------------------------------------------------//
48
49 void Bicycling::cal() {
50         if(!check()) {
51                 cout << "ERROR: check() not passed, no calculation!" << endl;
52                 return;
53         }
54
```

```
55          RungeKutta rk(_n_eqns, _dt, _t_start, _x_start);
56          rk.set_stop(stop);
57          rk.load_f(f_v);
58
59          rk.cal_rk1();
60
61          _t = rk.get_t();
62          _x = rk.get_x();
63
64          _n_stps = rk.get_n_stps();
65  }
66
67  //-------------------------------------------------------------------------//
```

(3) Use the adiabatic model of the air density (2.24) to calculate the cannon shell trajectory, and compare with the results found using the isothermal model (2.23). Aslo, one can further incorporate the effects of the variation of the ground temperature (seasonal changes) by replacing B_2 by $B_2^{\rm ref}(T_0/T_{\rm ref})^\alpha$, where $B_2^{\rm ref}$ is the value of B_2 at a reference temperature $T_{\rm ref}$ and T_0 is the actual ground temperature. The value quoted in the text is appropriate for $T = 300$ K. In particular, how much effect will the adiabatic model have on the maximum range and the launch angle to achieve it? How much do they vary from a cold day in winter to a hot summer day?

There are two models of air drag mentioned in this problem: (a) air drag with the isothermal model of air density, and (b) air drag with the adiabatic model. To make it clear, here lists the formulas of them:

$$\begin{aligned} &\text{(a) } \vec{F}_{\rm drag} = -B_2 e^{-mgy/(kT_0)} v^2 \hat{v} = -B_2 e^{-y/Y_0} v^2 \hat{v}, \\ &\text{(b) } \vec{F}_{\rm drag} = -B_2 (1 - ay/T_0)^{1/(\gamma-1)} v^2 \hat{v}, \end{aligned} \tag{14}$$

where B_2 is a just constant regarding the air density on the ground (ρ_0). If we take the effects of the variation of the ground temperature, the format of (b) will become

$$\begin{aligned} \vec{F}_{\rm drag} =& -B_2 (1 - ay/T_0)^{1/(\gamma-1)} v^2 \hat{v} = -B_2^{\rm ref} \left(\frac{T_0}{T_{\rm ref}}\right)^\alpha (1 - ay/T_0)^{1/(\gamma-1)} v^2 \hat{v} \\ =& -B_2^{\rm ref} \left(\frac{T_0 - ay}{T_{\rm ref}}\right)^{1/(\gamma-1)} v^2 \hat{v} \end{aligned} \tag{15}$$

It is easy to see that **the lower ground temperature T_0 can make the magnitude of drag smaller.**

In this problem, we will use Euler approximation. We set $Y_0 = 10000$ m, $B_2/m = 4 \times 10^{-5}$ m^{-1}, $T_0 = T_{\rm ref} = 300$ K, $\gamma = 1.4$, and $a = 6.5 \times 10^{-3}$K/m. We can get the blue (isothermal model) and red (adiabatic model) curves in Fig. 4. The temperature 300 K is roughly the typical summer temperature in West Lafayette. We pick the temperature 270 K as the typical winter temperature here, and then get the green curve for the modified adiabatic model. We can see with the same initial condition the winter curve (green) has longer range than the summer (red), which is expected by the Eq. 15.

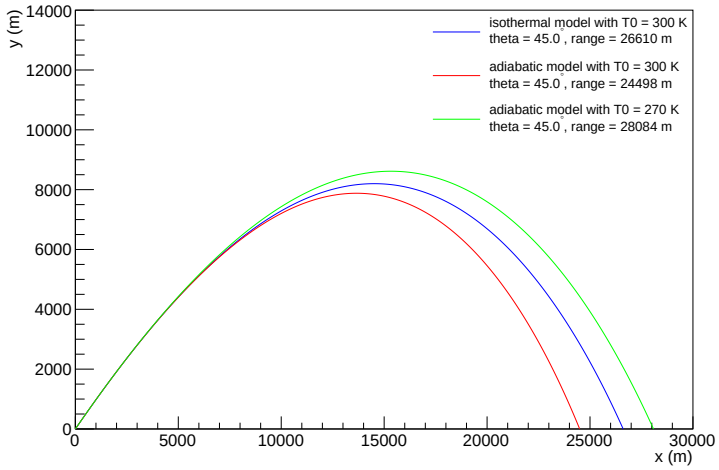

Figure 4: The projectile trajectories calculated from different models with fixed projection angle $\theta = 45°$.

We use the bisection method to search for the maximum range of the three cases above, up to the accuracy of 0.1° of the projection angle. We can see the following properties from Fig. 5.

- With the same ground temperature $T_0 = 300$ K, the adiabatice model (red) have small maximum range (24521 m) than the isothermal model (blue) (26622 m), by a fraction of 7.9%, and smaller angle (43.6°) to achieve it than the isothermal model (45.9°) by a fraction of 5.0%.
- Again, we can see that the winter curve (green) has longer maximum range (28085 m) than the summer curve (red) (24521 m) by a fraction of 14.5%. The winter curve (green) also has larger launch angle (44.7°) than the summer curve (red) (43.6°) by a fraction of 2.5%.

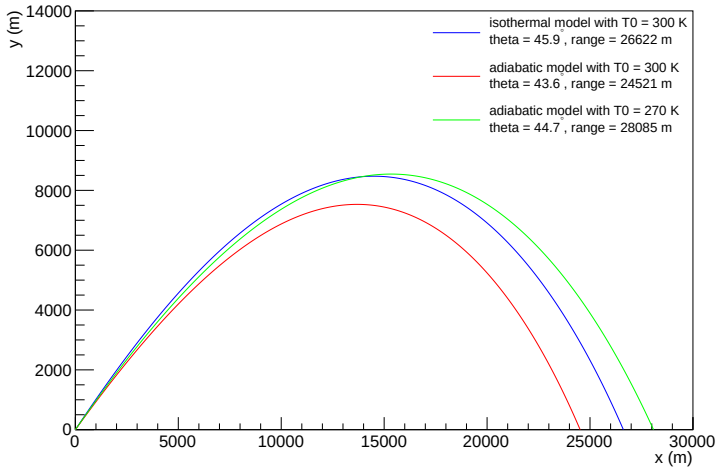

Figure 5: The projectile trajectories calculated from different models with maximum ranges.

Relevant code:
For the Euler method, the code has been listed previously in problem (1), page 4.
In the file `projectile.cxx`:

```
1  //---------------------------------------------------------------------------//
2
3  // static variables
4  double Projectile::_g     = 9.8;
5  double Projectile::_B2_m  = 4e-5;
6  double Projectile::_Y0    = 1e4;
7  double Projectile::_a     = 6.5e-3;
8  double Projectile::_T0    = 300;
9  double Projectile::_T_ref= 300;
10 double Projectile::_gamma= 1.4;
11
12 //---------------------------------------------------------------------------//
13
14 // x[0]: x; x[1]: y; x[2]: vx; x[3]: vy
15 double Projectile::f_x(double t, const vector<double> &x) {
16         // dx/dt = vx
17         return x[2];
18 }
19
20 //---------------------------------------------------------------------------//
21
```

```
22 double Projectile::f_y(double t, const vector<double> &x) {
23         // dy/dt = vy
24         return x[3];
25 }
26
27 //------------------------------------------------------------------------//
28
29 double Projectile::f_vx_no_drag(double t, const vector<double> &x) {
30         double drag = 0;
31         return -drag*x[2];
32 }
33
34 //------------------------------------------------------------------------//
35
36 double Projectile::f_vy_no_drag(double t, const vector<double> &x) {
37         double drag = 0;
38         return -drag*x[3]-_g;
39 }
40
41 //------------------------------------------------------------------------//
42
43 double Projectile::f_vx_constant_drag(double t, const vector<double> &x) {
44         double drag = _B2_m*sqrt(x[2]*x[2]+x[3]*x[3]);
45         return -drag*x[2];
46 }
47
48 //------------------------------------------------------------------------//
49
50 double Projectile::f_vy_constant_drag(double t, const vector<double> &x) {
51         double drag = _B2_m*sqrt(x[2]*x[2]+x[3]*x[3]);
52         return -drag*x[3]-_g;
53 }
54
55 //------------------------------------------------------------------------//
56
57 double Projectile::f_vx_isothermal_drag(double t, const vector<double> &x) {
58         double drag = _B2_m*sqrt(x[2]*x[2]+x[3]*x[3])*exp(-x[1]/_Y0);
59         return -drag*x[2];
60 }
61
62 //------------------------------------------------------------------------//
63
64 double Projectile::f_vy_isothermal_drag(double t, const vector<double> &x) {
65         double drag = _B2_m*sqrt(x[2]*x[2]+x[3]*x[3])*exp(-x[1]/_Y0);
66         return -drag*x[3]-_g;
67 }
68
69 //------------------------------------------------------------------------//
70
71 double Projectile::f_vx_adiabatic_drag(double t, const vector<double> &x) {
72         double alpha = 1/(_gamma-1);
73         double drag = _B2_m*sqrt(x[2]*x[2]+x[3]*x[3])*pow(1-_a*x[1]/_T0, alpha);
74         drag *= *pow(_T0/_T_ref, alpha);
75         return -drag*x[2];
76 }
77
78 //------------------------------------------------------------------------//
79
80 double Projectile::f_vy_adiabatic_drag(double t, const vector<double> &x) {
81         double alpha = 1/(_gamma-1);
```

```
82          double drag = _B2_m*sqrt(x[2]*x[2]+x[3]*x[3])*pow(1-_a*x[1]/_T0, alpha);
83          drag *= pow(_T0/_T_ref, alpha)
84          return -drag*x[3]-_g;
85  }
86
87  //------------------------------------------------------------------------//
88
89  bool Projectile::stop(double t, const vector<double> &x) {
90          return x[1]<0;
91  }
92
93  //------------------------------------------------------------------------//
94
95  bool Projectile::check() const {
96          if(_mode!=0 && _mode!=1 && _mode!=2 && _mode!=3) {
97                  cout << "ERROR: invalid mode!" << endl;
98                  cout << "0: no drag; 1: constant drag; " << flush;
99                  cout << "2: isothermal drag; 3: adiabatic drag." << endl;
100                 return false;
101         }
102
103         if(_rk_order!=1 && _rk_order!=2 && _rk_order!=4) {
104                 cout << "ERROR: invalid order of RK method!" << endl;
105                 cout << "1: RK1 (Euler); 2: RK2; 4: RK4." << endl;
106                 return false;
107         }
108
109         return true;
110 }
111
112 //------------------------------------------------------------------------//
113
114 void Projectile::cal() {
115         if(!check()) return;
116
117         RungeKutta rk(_n_eqns, _dt, _t_start, _c_start);
118         rk.load_f(f_x);
119         rk.load_f(f_y);
120         if(_mode == 0) {
121                 rk.load_f(f_vx_no_drag);
122                 rk.load_f(f_vy_no_drag);
123         } else if(_mode == 1) {
124                 rk.load_f(f_vx_constant_drag);
125                 rk.load_f(f_vy_constant_drag);
126         } else if(_mode == 2) {
127                 rk.load_f(f_vx_isothermal_drag);
128                 rk.load_f(f_vy_isothermal_drag);
129         } else if(_mode == 3) {
130                 rk.load_f(f_vx_adiabatic_drag);
131                 rk.load_f(f_vy_adiabatic_drag);
132         } else {
133                 cout << "ERROR: invalid mode!" << endl;
134                 cout << "0: no drag; 1: constant drag; " << flush;
135                 cout << "2: isothermal drag; 3: adiabatic drag." << endl;
136         }
137
138         rk.set_stop(stop);
139         if(_rk_order == 1) {
140                 rk.cal_rk1();
141         } else if(_rk_order == 2) {
```

```
142                 rk.cal_rk2();
143         } else if(_rk_order == 4) {
144                 rk.cal_rk4();
145         } else {
146                 cout << "ERROR: invalid order of RK method!" << endl;
147                 cout << "1: RK1 (Euler); 2: RK2; 4: RK4." << endl;
148         }
149
150         _t = rk.get_t();
151         _x = rk.get_x();
152         _n_stps = rk.get_n_stps();
153 }
154
155 //------------------------------------------------------------------------//
156
157 double Projectile::cal_range() {
158         int n = _n_stps - 1;
159         //cout << "y[n-1] = " << _x[1][n-1] << "; y[n] = " << _x[1][n] << endl;
160         _range = ( _x[1][n]*_x[0][n-1] - _x[1][n-1]*_x[0][n] )/( _x[1][n] - _x[1][n-1] );
161         return _range;
162 }
163
164 //------------------------------------------------------------------------//
165
166 double Projectile::search_theta_for_max_range(double v = 700) {
167         double dtheta = 32;
168         double theta = 45;
169         double theta_tmp_left;
170         double theta_tmp_right;
171         //double v = 700;
172         double range = -1;
173         double range_tmp_left;
174         double range_tmp_right;
175
176         do {
177                 theta_tmp_left = theta - dtheta;
178                 _c_start[2] = v*cos(theta_tmp_left*M_PI/180);
179                 _c_start[3] = v*sin(theta_tmp_left*M_PI/180);
180                 cal();
181                 range_tmp_left = cal_range();
182
183                 theta_tmp_right = theta + dtheta;
184                 _c_start[2] = v*cos(theta_tmp_right*M_PI/180);
185                 _c_start[3] = v*sin(theta_tmp_right*M_PI/180);
186                 cal();
187                 range_tmp_right = cal_range();
188
189                 if(range_tmp_left <= range && range_tmp_right <= range) {
190                         dtheta /= 2.0;
191                         continue;
192                 }
193                 if(range_tmp_left > range) {
194                         range = range_tmp_left;
195                         theta = theta_tmp_left;
196                 }
197                 if(range_tmp_right > range) {
198                         range = range_tmp_right;
199                         theta = theta_tmp_right;
200                 }
201         } while(dtheta>0.05);
```

```
202
203         cout << "left uncertainty: " << range-range_tmp_left << endl;
204         cout << "right uncertainty: " << range-range_tmp_right << endl;
205
206         return theta;
207 }
208
209 //-------------------------------------------------------------------------//
```

(4) In all calculations of cannon shots so far, we neglected the fact that the projectiles are launched from and measured in the rotating reference frame of Earth. Taking rotation into account would add a term $-2\vec{\omega} \times \vec{v}$ to the apparent acceleration in Earths frame of reference (due to the Coriolis force), making even the spinless cannon problem 3-dimensional. Estimate the effect of the Coriolis force on the trajectory of a typical cannonball launched toward southeast from Lafayette (latitude $40°25'N$) with $v_0 = 700$ m/s at $\theta = 45$ degrees with respect to the horizontal.

We first set up the directions of the local axes. The initial projection velocity is inside the xy-

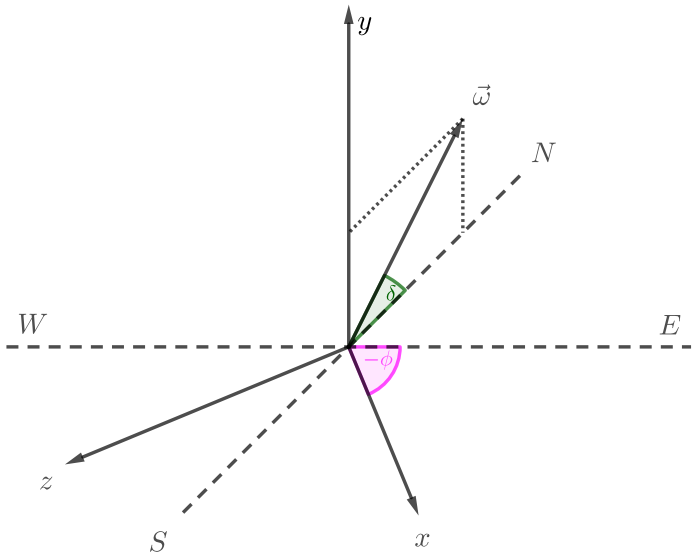

Figure 6: The relationship between the local axes and the directions of the earth.

plane. The y-axis is the vertical direction. The $\vec{\omega}$ shows the angular velocity of the earth spin, $\omega = 2\pi/24/3600 \approx 7.27 \times 10^{-5}$ rad/s. The δ is equal to the latitude, $\delta = 40°25'$. The azimuthal angle of projection is $-\phi = 45°$. The launch angle (not shown in the figure) is between the x- and y-axis, $\theta = 45°$. The magnitude of the initial velocity is 700 m/s. Now, we can write the x-, y-, and z-components of $\vec{\omega}$.

$$\begin{aligned} \omega_x =& \omega \cos\delta \sin\phi, \\ \omega_y =& \omega \sin\delta, \\ \omega_z =& -\omega \cos\delta \cos\phi. \end{aligned} \tag{16}$$

The total acceleration of the cannon would be

$$\vec{a} = -\vec{g} + \vec{F}_{\text{drag}}/m - 2\vec{\omega} \times \vec{v}, \tag{17}$$

which can also be written into components

$$\begin{aligned} a_x =& (F_{\text{drag}})_x/m - 2(\omega_y v_z - \omega_z v_y), \\ a_y =& (F_{\text{drag}})_y/m - 2(\omega_z v_x - \omega_x v_z) - g, \\ a_z =& (F_{\text{drag}})_z/m - 2(\omega_x v_y - \omega_y v_z). \end{aligned} \tag{18}$$

We use the Euler approximation to solve the ODE set numerically,

$$\frac{\mathrm{d}\vec{x}}{\mathrm{d}t} = \vec{v}, \qquad \frac{\mathrm{d}\vec{v}}{\mathrm{d}t} = \vec{a}. \tag{19}$$

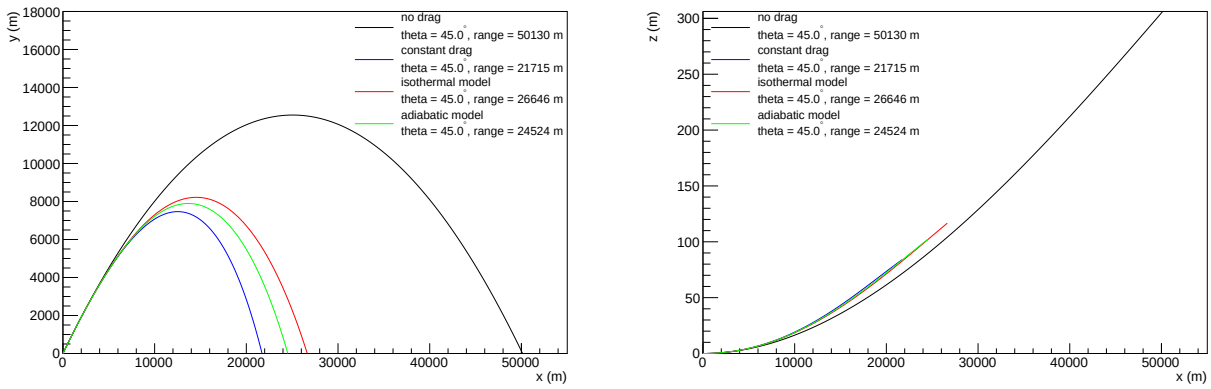

Figure 7: The xy- (left pad) and xz-projection (right pad) of the trajectories with different drag models. The displayed range is the length of the trajectory projection to the x-axis. The time step is $\Delta t = 0.0005$ s.

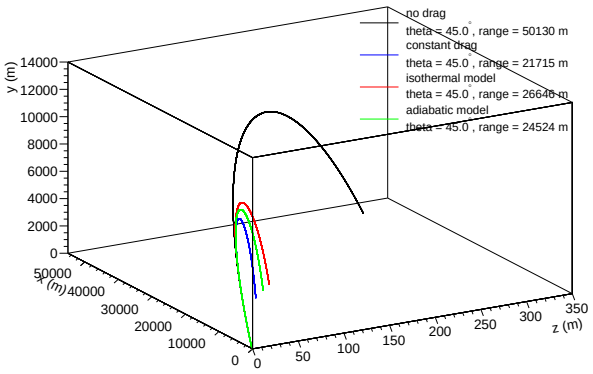

Figure 8: The 3D plot of the trajectories with different drag models. The displayed range is the length of the trajectory projection to the x-axis. The time step is $\Delta t = 0.0005$ s.

Figure 7 shows the xy- (left pad) and xz-projection (right pad) of the trajectories with different drag models, and Figure 8 shows the 3D plots of them. In this cannon model, we can see that the deviation in the z-direction due to the Coriolis force is small compared with the range in x-direction. The fraction is about $0.5 \sim 0.6\%$ in each drag model.

Relevant code:
For the Euler method, the code has been listed previously in problem (1), page 4.
In the file `projectile3d.cxx`:

```
1  inline void Projectile3D::set_omega_polar(double w, double la, double phi) {
2          _wx = +w*cos(la*M_PI/180)*sin(phi*M_PI/180);
3          _wy = +w*sin(la*M_PI/180);
4          _wz = -w*cos(la*M_PI/180)*cos(phi*M_PI/180);
5  }
6
7  //-------------------------------------------------------------------------//
8
9  // static variables
10 double Projectile3D::_g    = 9.8;
```

```
11 double Projectile3D::_B2_m = 4e-5;
12 double Projectile3D::_Y0   = 1e4;
13 double Projectile3D::_a    = 6.5e-3;
14 double Projectile3D::_T0   = 300;
15 double Projectile3D::_T_ref= 300;
16 double Projectile3D::_gamma= 1.4;
17 double Projectile3D::_wx   = 0;
18 double Projectile3D::_wy   = 0;
19 double Projectile3D::_wz   = 0;
20
21 //------------------------------------------------------------------------//
22
23 // x[0]: x; x[1]: y; x[2]: z; x[3]: vx; x[4]: vy; x[5]: vz
24 double Projectile3D::f_x(double t, const vector<double> &x) {
25         // dx/dt = vx
26         return x[3];
27 }
28
29 //------------------------------------------------------------------------//
30
31 double Projectile3D::f_y(double t, const vector<double> &x) {
32         // dy/dt = vy
33         return x[4];
34 }
35
36 //------------------------------------------------------------------------//
37
38 double Projectile3D::f_z(double t, const vector<double> &x) {
39         // dz/dt = vz
40         return x[5];
41 }
42
43 //------------------------------------------------------------------------//
44
45 double Projectile3D::f_vx_no_drag(double t, const vector<double> &x) {
46         double drag = 0;
47         return -drag*x[3]-2*(_wy*x[5]-_wz*x[4]);
48 }
49
50 //------------------------------------------------------------------------//
51
52 double Projectile3D::f_vy_no_drag(double t, const vector<double> &x) {
53         double drag = 0;
54         return -drag*x[4]-_g-2*(_wz*x[3]-_wx*x[5]);
55 }
56
57 //------------------------------------------------------------------------//
58
59 double Projectile3D::f_vz_no_drag(double t, const vector<double> &x) {
60         double drag = 0;
61         return -drag*x[5]-2*(_wx*x[4]-_wy*x[3]);
62 }
63
64 //------------------------------------------------------------------------//
65
66 double Projectile3D::f_vx_constant_drag(double t, const vector<double> &x) {
67         double drag = _B2_m*sqrt(x[3]*x[3]+x[4]*x[4]+x[5]*x[5]);
68         return -drag*x[3]-2*(_wy*x[5]-_wz*x[4]);
69 }
70
```

```
71  //------------------------------------------------------------------------//
72
73  double Projectile3D::f_vy_constant_drag(double t, const vector<double> &x) {
74          double drag = _B2_m*sqrt(x[3]*x[3]+x[4]*x[4]+x[5]*x[5]);
75          return -drag*x[4]-_g-2*(_wz*x[3]-_wx*x[5]);
76  }
77
78  //------------------------------------------------------------------------//
79
80  double Projectile3D::f_vz_constant_drag(double t, const vector<double> &x) {
81          double drag = _B2_m*sqrt(x[3]*x[3]+x[4]*x[4]+x[5]*x[5]);
82          return -drag*x[5]-2*(_wx*x[4]-_wy*x[3]);
83  }
84
85  //------------------------------------------------------------------------//
86
87  double Projectile3D::f_vx_isothermal_drag(double t, const vector<double> &x) {
88          double drag = _B2_m*sqrt(x[3]*x[3]+x[4]*x[4]+x[5]*x[5])*exp(-x[1]/_Y0);
89          return -drag*x[3]-2*(_wy*x[5]-_wz*x[4]);
90  }
91
92  //------------------------------------------------------------------------//
93
94  double Projectile3D::f_vy_isothermal_drag(double t, const vector<double> &x) {
95          double drag = _B2_m*sqrt(x[3]*x[3]+x[4]*x[4]+x[5]*x[5])*exp(-x[1]/_Y0);
96          return -drag*x[4]-_g-2*(_wz*x[3]-_wx*x[5]);
97  }
98
99  //------------------------------------------------------------------------//
100
101 double Projectile3D::f_vz_isothermal_drag(double t, const vector<double> &x) {
102         double drag = _B2_m*sqrt(x[3]*x[3]+x[4]*x[4]+x[5]*x[5])*exp(-x[1]/_Y0);
103         return -drag*x[5]-2*(_wx*x[4]-_wy*x[3]);
104 }
105
106 //------------------------------------------------------------------------//
107
108 double Projectile3D::f_vx_adiabatic_drag(double t, const vector<double> &x) {
109         double alpha = 1/(_gamma-1);
110         double drag = _B2_m*sqrt(x[3]*x[3]+x[4]*x[4]+x[5]*x[5])*pow(1-_a*x[1]/_T0, alpha);
111         drag *= *pow(_T0/_T_ref, alpha);
112         return -drag*x[3]-2*(_wy*x[5]-_wz*x[4]);
113 }
114
115 //------------------------------------------------------------------------//
116
117 double Projectile3D::f_vy_adiabatic_drag(double t, const vector<double> &x) {
118         double alpha = 1/(_gamma-1);
119         double drag = _B2_m*sqrt(x[3]*x[3]+x[4]*x[4]+x[5]*x[5])*pow(1-_a*x[1]/_T0, alpha);
120         drag *= *pow(_T0/_T_ref, alpha);
121         return -drag*x[4]-_g-2*(_wz*x[3]-_wx*x[5]);
122 }
123
124 //------------------------------------------------------------------------//
125
126 double Projectile3D::f_vz_adiabatic_drag(double t, const vector<double> &x) {
127         double alpha = 1/(_gamma-1);
128         double drag = _B2_m*sqrt(x[3]*x[3]+x[4]*x[4]+x[5]*x[5])*pow(1-_a*x[1]/_T0, alpha);
129         drag *= *pow(_T0/_T_ref, alpha);
130         return -drag*x[5]-2*(_wx*x[4]-_wy*x[3]);
```

```
131 }
132
133 //------------------------------------------------------------------------//
134
135 bool Projectile3D::stop(double t, const vector<double> &x) {
136         return x[1]<0;
137 }
138
139 //------------------------------------------------------------------------//
140
141 void Projectile3D::cal() {
142         if(!check()) return;
143
144         RungeKutta rk(_n_eqns, _dt, _t_start, _c_start);
145         rk.load_f(f_x);
146         rk.load_f(f_y);
147         rk.load_f(f_z);
148         if(_mode == 0) {
149                 rk.load_f(f_vx_no_drag);
150                 rk.load_f(f_vy_no_drag);
151                 rk.load_f(f_vz_no_drag);
152         } else if(_mode == 1) {
153                 rk.load_f(f_vx_constant_drag);
154                 rk.load_f(f_vy_constant_drag);
155                 rk.load_f(f_vz_constant_drag);
156         } else if(_mode == 2) {
157                 rk.load_f(f_vx_isothermal_drag);
158                 rk.load_f(f_vy_isothermal_drag);
159                 rk.load_f(f_vz_isothermal_drag);
160         } else if(_mode == 3) {
161                 rk.load_f(f_vx_adiabatic_drag);
162                 rk.load_f(f_vy_adiabatic_drag);
163                 rk.load_f(f_vz_adiabatic_drag);
164         } else {
165                 cout << "ERROR: invalid mode!" << endl;
166                 cout << "0: no drag; 1: constant drag; " << flush;
167                 cout << "2: isothermal drag; 3: adiabatic drag." << endl;
168         }
169
170         rk.set_stop(stop);
171
172         if(_rk_order == 1) {
173                 rk.cal_rk1();
174         } else if(_rk_order == 2) {
175                 rk.cal_rk2();
176         } else if(_rk_order == 4) {
177                 rk.cal_rk4();
178         } else {
179                 cout << "ERROR: invalid order of RK method!" << endl;
180                 cout << "1: RK1 (Euler); 2: RK2; 4: RK4." << endl;
181         }
182
183         _t = rk.get_t();
184         _x = rk.get_x();
185         _n_stps = rk.get_n_stps();
186 }
187
188 //------------------------------------------------------------------------//
189
190 double Projectile3D::cal_range() {
```

```
191         int n = _n_stps - 1;
192         //cout << "y[n-1] = " << _x[1][n-1] << "; y[n] = " << _x[1][n] << endl;
193         _range = ( _x[1][n]*_x[0][n-1] - _x[1][n-1]*_x[0][n] )/( _x[1][n] - _x[1][n-1] );
194         return _range;
195 }
```

(5) (Order of magnitude checks.) On p.28 of the Giordano-Nakanishi book, the air drag coefficient for a large cannon shell is said to be $B_2/m \approx 4 \times 10^{-5}$ m^{-1}. On p.38, the magnitude of the Magnus term in baseball is stated to be $S_0/m \approx 4.1 \times 10^{-4}$. Furthermore, p.45 gives an estimate $S_0\omega/m \approx 0.25$ s^{-1} for the golf ball, and the next page (p.46, Problem 2.24) says that for a ping-pong ball $S_0/m \approx 0.040$. Argue about the orders of magnitude of these values, and justify them if you can. If needed, refer to the official specifications for the various balls (see, e.g., the document BallSpecs.pdf posted in the Supplemental Materials section of the course home page). If you think that any of the above numbers in the text are not justified, then state why that is so.

Cannon shell We assume the cannon shell is a sphere with radius $R = 0.1$ m, and therefore the drag coefficient C is about 0.5. We assume it is made of iron, so its density is $\rho = 7.9 \times 10^3$ kg m^{-3}. Then, we can esitmate B_2/m

$$\frac{B_2}{m} = \frac{C\rho_{\text{air}}A}{2V\rho} = \frac{C\rho_{\text{air}}\pi R^2}{8\pi R^3\rho/3} = \frac{1.5 \times 1.292 \text{ kg m}^{-3}}{8 \times 0.1 \text{ m} \times 7.9 \times 10^3 \text{ kg m}^{-3}} \approx 3 \times 10^{-4}\text{m}^{-1}. \tag{20}$$

However, the actual shape of the shell could be streamlined, so its drag coefficient can be much smaller $C \approx 0.04 \sim 0.09$. If we take the real shape into consideration, the estimated B_2/m can be one order smaller, $2.4 \times 10^{-5} \sim 5.4 \times 10^{-5}$ m^{-1}. The given value 4×10^{-5} m^{-1} is inside this range.

Baseball, golf, ping-pong I think I am not able to estimate S_0 directly, because they only come from the experience. However, we can compare the S_0/m values of those different balls.

The mass of baseball is about $m_b = 0.145$ kg, ping-pong is about $m_p = 0.0027$ kg, and golf is $m_g = 0.0459$ kg.

$$\frac{(S_0/m)_b}{(S_0/m)_p} = \frac{4.1 \times 10^{-4}}{0.04} \approx 0.01 \quad \text{and} \quad \frac{m_p}{m_b} = \frac{0.0027 \text{ kg}}{0.145 \text{ kg}} \approx 0.019 \tag{21}$$

We can see the S_0 of baseball and ping-pong should be similar.

We can therefore get the anguler velocity of the golf.

$$\omega = \left(\frac{S_0\omega}{m}\right)_g \frac{m_g}{m_p} \left(\frac{m}{S_0}\right)_p = 0.25 \text{ s}^{-1} \times \frac{0.0459 \text{ kg}}{0.0027 \text{ kg}}/0.04 \approx 100 \text{ rad/s} \tag{22}$$

which seems reasonable.