4
Object-Oriented Programming: Kinematics $\odot$

In this chapter we provide examples of object-oriented programming (OOP) using the C++ language. Even though this subject fits in well with our earlier discussion of programming principles, we have put it off until now and marked it as optional (the $\odot$) since we do not use explicit OOP in the projects and because it is difficult for the new programmer. However, we suggest that everyone read through Section 4.2.1.

4.1
Problem: Superposition of Motions

The isotropy of space implies that motion in one direction is independent of motion in other directions. So, for example, when a soccer ball is kicked, we have acceleration in the vertical direction and simultaneous, yet independent, uniform motion in the horizontal direction. In addition, Galilean invariance (velocity independence of Newton's laws of motion) tells us that when an acceleration is added in to uniform motion, the distance covered due to the acceleration adds to the distance covered by uniform motion.

Your **problem** is to use the computer to describe motion in such a way that velocities and accelerations in each direction are treated as separate entities or objects, independent of motion in other directions. In this way the problem is viewed consistently by both the programming philosophy and the basic physics.

4.2
Theory: Object-Oriented Programming

We will analyze this problem from an object-oriented programming viewpoint. While the *objects* in OOP are often graphical on a computer screen, the objects in our problem are the motions in each dimension. By reading through and running the programs in this project, you should become familiar with the concepts of OOP.

Computationyal Physics. Problem Solving with Computers (2nd edn).
Rubin H. Landau, Manuel José Páez, Cristian C. Bordeianu
Copyright © 2007 WILEY-VCH Verlag GmbH & Co. KGaA, Weinheim
ISBN: 978-3-527-40626-5

4.2.1 OOP Fundamentals

Object-oriented programming (OOP) has a precise definition. The concept is general and the *object* can be a component of a program with the properties of *encapsulation*, *abstraction*, *inheritance*, and *polymorphism* (to be defined shortly). Of interest to us is OOP's programming paradigm, which aims to simplify writing large programs by providing a framework for reusing components developed and tested in previous problems. A true object-oriented language has four characteristics [5,6]:

Encapsulation: The data and the *methods* used to produce or access the data are encapsulated into an entity called an *object.* For our 1D problem, we take the data as the initial position and velocity of the soccer ball, and the object as the solution $x(t)$ of the equations of motion that gives the position of the ball as a function of time. As part of the OOP philosophy, data are manipulated only via distinct *methods*.

Abstraction: Operations applied to objects must give expected results according to the nature of the objects. For example, summing two matrices always gives another matrix. The programmer can in this way concentrate on solving the problem rather than on details of implementation.

Inheritance: Objects inherit characteristics (including code) from their ancestors, yet may be different from their ancestors. In our problem, motion in two dimensions inherits the properties of 1D motion in each of two dimensions, and accelerated motion inherits the properties of uniform motion.

Polymorphism: Different objects may have *methods* with the same name, yet the method may differ for different objects. Child objects may have *member* functions with the same name but properties differing from those of their ancestors. In our problem, a member function *archive*, which contains the data to be plotted, will be redefined depending on whether the motion is uniform or accelerated.

4.3 Theory: Newton's Laws, Equation of Motion

Newton's second law of motion relates the force vector **F** acting on a mass m to the acceleration vector **a** of the mass:

$$\mathbf{F} = m\mathbf{a} \tag{4.1}$$

When the vectors are resolved into Cartesian components, each component yields a second-order differential equation:

$$F_i = m\frac{d^2x_i}{dt^2} \quad (i = 1, 2, 3) \tag{4.2}$$

If the force in the x direction vanishes, $F_x = 0$, the equation of motion (4.2) has a solution corresponding to uniform motion in the x direction with a constant velocity v_{0x}:

$$x = x_0 + v_{0x}t \tag{4.3}$$

Equation (4.3) is the *base* or *parent* object in our project. If the force in the y direction also vanishes, then there also will be uniform motion in the y direction:

$$y = y_0 + v_{0y}t \tag{4.4}$$

In our project we consider uniform x motion as a parent and view the y motion as a child.

Equation (4.2) tells us that a constant force in the x direction causes a constant acceleration a_x in that direction. The solution of the x equation of motion with uniform acceleration is

$$x = x_0 + v_{0x}t + \tfrac{1}{2}a_x t^2 \tag{4.5}$$

For projectile motion without air resistance, we usually have no x acceleration and a negative y acceleration due to gravity, $a_y = -g = -9.8\,\mathrm{m/s^2}$. The y equation of motion is then

$$y = y_0 + v_{0y}t - \tfrac{1}{2}gt^2 \tag{4.6}$$

We define this accelerated y motion to be a child to the parent uniform x motion.

4.4 OOP Method: Class Structure

The *class structure* we use to solve our problem contains the objects:

Parent class Um1D:	1D uniform motion for given initial conditions,
Child class Um2D:	2D uniform motion for given initial conditions,
Child class Am2d:	2D accelerated motion for given initial conditions.

The *member functions* include

x:	position after time t,
archive:	creator of a file of position versus time.

For our projectile motion, *encapsulation* is the combination of the initial conditions (x_0, v_{x0}) with the member functions used to compute $x(t)$. Our member functions are the creator of the class of uniform 1D motion `Um1D`, its destructor ∼ `Um1D`, and the creator `x(t)` of a file of x as a function of time t. *Inheritance* is the child class `Um2D` for uniform motion in both x and y directions, it being created from the parent class `Um1D` of 1D uniform motion. *Abstraction* is present (although not used powerfully) by the simple addition of motion in the x and y directions. Polymorphism is present by having the member function that creates the output file be different for 1D and 2D motions. In our implementation of OOP, the class `Am2D` for accelerated motion in two dimensions inherits uniform motion in two dimensions (which, in turn, inherits uniform 1D motion), and adds to it the attribute of acceleration.

4.5 Implementation: Uniform 1D Motion, unim1d.cpp

For 1D motion we need a program that outputs positions along a line as a function of time, (x, t). For 2D motion we need a program that outputs positions in a plane as a function of time (x, y, t). Time varies in discrete steps of $\Delta t =$`delt`$= T/N$, where the total time T and the number of steps N are input parameters. We give here program fragments that can be pasted together into a complete C++ program (the complete program is on the diskette).

Our parent class `Um1D` of uniform motion in one dimension contains

x00:	the initial position,
delt:	the time step,
vx0:	the initial velocity,
time:	the total time of the motion,
steps:	the number of time steps.

To create it, we start with the C++ headers:

```
#include <stdio.h>                    /* Input-output libe */
#include <stdlib.h>                           /* Math libe */
```

The encapsulation of the data and the member functions is achieved via `Class Um1D`:

4.5.1 Uniform Motion in 1D, Class Um1D

```
class Um1D  {                               /* Create base class
*/
  public:
  double x00,delt,vx,time;       /* Initial position , velocity , dt */
  int steps;                                           /* Time steps */
  Um1D(double x0,double dt,double vx0,double ttot); /* Constructor */
  ~Um1D(void);                                   /* Class Destructor */

  double x(double tt);                                      /* x(t) */
    void archive();                       /* send  x vs t to file */
};
```

Next, the variables `x0`, `delt`, `vx`, and `time` are initialized by the constructor of the class `Um1D`:

```
Um1D::Um1D(double x0,double dt,double vx0,double ttot) {
                                             /* Constructor Um1D */
  x00 = x0;
  delt = dt;
  vx = vx0;
  time = ttot;
  steps = ttot/delt;
}
```

After that, we make the destructor of the class, which also prints the time when the class is destroyed:

```
Um1D::~Um1D(void)  {                 /* Destructor of class Um1D  */
  printf("Class Um1D destroyed \ n");
}
```

Given the initial position x_0, the member function returns the position after time dt:

```
double Um1D::x(double tt) {                            /* x=x0+dt*v
*/
  return x00+tt*vx;
}
```

The algorithm is implemented in a member routine, and the positions and times are written to the file `Motion1D.dat`:

```
void Um1D::archive()  {                       /* Produce x vs t file
*/
  FILE  *pf;
  int i;
  double xx, tt;
  if ( (pf = fopen("Motion1D.dat","w+")) == NULL ) {
    printf("Could not open file \ n");
    exit(1);
  }
  tt = 0.0;
  for ( i = 1; i <= steps; i++ ) {
    xx = x(tt);                              /* Computes x=x0+t*v */

    fprintf(pf," %f  %f \n",tt,xx);
    tt = tt+delt;
  }
  fclose(pf);
}
```

The main program defines an object (class) `unimotx` of type `Um1D` and gives initial numeric values to the data:

```
main() {
  double inix, inivx, dtim, ttotal;
  inix = 5.0;
  dtim = 0.1;
  inivx = 10.0;
  ttotal = 4.0;
  Um1D unimotx(inix, dtim, inivx, ttotal);     /* Class constructor */
  unimotx.archive();                          /* Produce y vs x file */
}
```

4.5.2 Implementation: Uniform Motion in 2D, Child Um2D, unimot2d.cpp

The first part of the program is the same as before. We now make the child class `Um2D` from the class `Um1D`.

```
#include <stdio.h> #include <stdlib.h> class Um1D {
/* Base class created */
  public:
  double x00, delt, vx, time;    /* Initial conditions, parameters */
  int steps;                                          /* Time step */
                    /* constructor */
  Um1D(double x0, double dt, double vx0, double ttot);
  ~Um1D(void);                                  /* Class destructor */

  double x(double tt);                                /* x=xo+v*t */
    void archive();                           /* Send x vs t to file */
};
                                              /* Um1D Constructor */
  Um1D::Um1D(double x0, double dt, double vx0, double ttot) {
    x00 = x0;
    delt = dt;
    vx = vx0;
    time = ttot;
    steps = ttot/delt;
  }
  Um1D::~Um1D(void)  {                     /* Class Um1D destructor */
    printf("Class Um1D destroyed \n");
  }
  double Um1D::x(double tt)  {                        /* x=x0+dt*v */
```

```
    return x00+tt*vx;
  }
  void Um1D::archive() {                          /* Produce x vs t file */
    FILE  *pf;
    int i;
    double xx, tt;
    if ( (pf = fopen("Motion1D.dat", "w+"))==NULL ) {
      printf("Could not open file \ n");
    exit(1);
    }
    tt = 0.0;
    for ( i = 1; i <= steps; i++ ) {
      xx = x(tt);                                  /* computes x=x0+t*v */
      fprintf(pf,"%f  %f \ n", tt, xx);
      tt = tt+delt;
    }
    fclose(pf);
}
```

4.5.3
Class Um2D: Uniform Motion in 2D

To include another degree of freedom, we define a new class that inherits the x component of uniform motion from the parent `Um1D`, as well as the y component of uniform motion from the parent `Um1D`. The new data for the class are the initial y position `y00` and the velocity in the y direction `vy0`. A new member `y` is included to describe the y motion. Note, in making the constructor of the `Um2D` class, that our interest in the data y versus x leads to the member `archive` being redefined. This is polymorphism in action.

```
class Um2D : public Um1D {                    /* Child class , parent Um1D
*/
  public:                             /* Data accessible to other code */
  double y00, vy;           /* member functions  accessible to all */
  Um2D(double x0,double dt,double vx0,double ttot,double y0,double
       vy0);
  ~Um2D(void);                             /* destructor of Um2D class */

  double y(double tt);                           /* Added motion for 2D */
    void archive();                              /* redefinition for 2D */
};
```

Observe how the `Um2D` constructor initializes the data in `Um1D`, `y00`, and `vy`:

```
Um2D::Um2D(double x0, double dt, double vx0, double tott,
      double y0, double vy0):
Um1D(x0,dt,vx0,tott)  {
   y00 = y0;
   vy  = vy0;
}
```

The destructor of the new class is

```
Um2D::~Um2D(void) {
  printf("Class Um2D is destroyed \ n");
}
```

The new member of class `Um2D` accounts for the y motion:

```
double Um2D::y(double tt)  {
  return y00+tt*vy;
}
```

The new member function for two dimensions contains the data of the y and x positions, and incorporates the polymorphism property of the objects:

```
void Um2D::archive()  {                    /* Uniform motion in 2D
*/
  FILE  *pf;
  int i;
  double xx, yy, tt;
  if ( (pf = fopen("Motion2D.dat","w+") ) == NULL )  {
   printf("Could not open file \ n");
   exit(1);
  }
  tt = 0.0;
  for (i = 1; i <= steps; i++) {
    xx = x(tt);                          /* uses member function x */
    yy = y(tt);                        /* add the second dimension */
    fprintf(pf,"%f  %f \ n", yy, xx);
    tt = tt + delt;
  }
  fclose(pf);
}
```

The differences with the previous `main` program are the inclusion of the y component of the motion and the constructor `unimotxy` of class type `Um2D`:

```
main() {
  double inix, iniy, inivx, inivy, dtim, ttotal;
  inix = 5.0;
  dtim = 0.1;
  inivx = 10.0;
  ttotal = 4.0;
  iniy = 3.0;
  inivy = 8.0;
                                              /* class constructor */
 Um2D unimotxy(inix, dtim, inivx, ttotal, iniy, inivy);
  unimotxy.archive();                  /* To obtain file of y vs x */
}
```

4.5.4 Implementation: Projectile Motion, Child Accm2D, accm2d.cpp

Consider now the problem of the motion of a soccer ball kicked at $(x, y) = (0, 0)$ with initial velocity (v_{0x}, v_{0y}) =(14,14) m/s. This is, of course, motion of a projectile in a uniform gravitational field which we know is described by a parabolic trajectory. We define a new child class `Accm2D` derived from the parent class `Um2D` that adds acceleration to the motion. The first part of the program is the same as the one for uniform motion in two dimensions, but now there is an extension with the class `Accm2D`:

```
#include <stdio.h> #include <stdlib.h> class Um1D  {
/* Base class created */
  public:
  double x00, delt, vx,time;      /* Initial conditions, parameters */
  int steps;                           /* Time steps to write in file */
                /* Constructor */
  Um1D(double x0, double dt, double vx0, double ttot);
  ~Um1D(void);                                      /* Destructor */
  double x(double tt);                              /* x=xo+v* dt */

  void archive();                            /* send x vs t to file */
};
                     /* Constructor */
  Um1D::Um1D(double x0, double dt, double vx0, double ttot)  {
    x00 = x0;
    delt = dt;
    vx = vx0;
    time =  ttot;
    steps = ttot/delt;
  }

  Um1D::~Um1D(void)  {             /* Destructor of the class Um1D */
    printf("Class Um1D destroyed \ n");
  }

  double Um1D::x(double tt)  {                         /* x=x0+dt*v
      */
    return x00+tt*vx;
  }

  void Um1D::archive()   {              /* Produce  file of  x vs t */
    FILE  *pf;
    int i;
    double xx, tt;
    if ( (pf = fopen("Motion1D.dat","w+")) == NULL )  {
      printf("Could not open file \ n ");
      exit(1);
    }
    tt=0.0;
    for (i = 1; i< = steps; i++)  {         /* x=x0+dt*v, change x0 */
      xx = x(tt);
      fprintf(pf,"%f  %f \ n ",tt,xx);
      tt = tt+delt;
    }
    fclose(pf);
  }

  class Um2D : public Um1D {        /* Child class Um2D, parent Um1D */
    public:                      /* Data: code; functions: all members */
    double y00,vy;
                                              /* constructor of Um2D class */
   Um2D(double x0, double dt, double vx0, double ttot, double y0,
        double vy0);
    ~Um2D(void);                            /* destructor of Um2D class */
    double y(double tt);
    void archive();                           /* redefine member for 2D */
  };

                                               /* Construct class Um2D */
 Um2D::Um2D(double x0, double dt, double vx0, double ttot,
            double y0, double vy0):Um1D(x0, dt, vx0, ttot)  {
    y00 = y0;
    vy = vy0;
  }

 Um2D::~Um2D(void) {
    printf("Class Um2D is destroyed \ n ");
  }

  double Um2D::y(double tt) {
    return y00+tt*vy;
  }

  void Um2D::archive() {
    FILE  *pf;
    int i;
    double xx, yy, tt;
                                              /*  now in 2D, still uniform */
    if ( (pf = fopen("Motion2D.dat","w+")) == NULL ) {
      printf("Could not open file \ n ");
      exit(1);
    }
    tt = 0.0;
    for ( i = 1; i <= steps; i++ )    {
      xx = x(tt);                           /* uses member function x */
      yy = y(tt);                            /* adds second dimension */
      fprintf( pf,"%f  %f \ n ",yy,xx );
      tt = tt+delt;
    }
    fclose(pf);
}
```

4.5.5 Accelerated Motion in Two Directions

A child class `Accm2D` is created from the parent class `Um2D`. It will inherit uniform motion in two dimensions and add acceleration in both the x and y directions. It has member functions:

1. Constructor of class.
2. Destructor of class.
3. A new member `xy` that gives the x and y components of acceleration.
4. An *archive* to override the member function of same name in the parent class for 2D uniform motion.

```
class Accm2D : public Um2D  {
  public:
  double ax, ay;
  Accm2D(double x0, double dt, double vx0, double ttot, double y0,
         double vy0, double accx, double accy);
  ~Accm2D(void);
  void xy(double *xxac, double *yyac, double tt);
  void archive();
};
```

Observe the method used to initialize the classes `Am2d` and `Um2D`:

```
Accm2D::Accm2D(double x0, double dt, double vx0, double ttot,
            double y0, double vy0, double accx, doubleaccy):
        Um2D(x0, dt, vx0, ttot, y0, vy0)  {
  ax = accx;
  ay = accy;
}

Accm2D::~Accm2D(void) {
  printf(" Class Accm2D destroyed \ n ");
}
```

Next we introduce a member function to show the inheritance of the parent class functions `xpox` and `y` and to include the two components of acceleration:

```
void Accm2D::xy(double *xxac, double *yyac, double tt)  {
  double dt2;
  dt2 = 0.5*tt*tt;
  *xxac = x(tt) + ax*dt2;
  *yyac = y(tt) + ay*dt2;
}
```

To override *archive* (which creates the data file), we redefine *archive* to take into account the acceleration:

```
void Accm2D::archive()   {
  FILE  *pf;
  int i;
  double tt, xxac, yyac;
  if ( (pf = fopen("Motion.dat", "w+") ) == NULL )  {
    printf("Could not open file \ n ");
    exit(1);
  }
  tt = 0.0;
  for(i = 1;i<=steps;i++)  {
    xy(&xxac, &yyac, tt);
    fprintf(pf,"%f  %f \ n ",xxac, yyac);
    tt = tt + delt;
  }
  fclose(pf);
}
```

Next a file is produced with the y and x positions of the ball as functions of time:

```
main()  {
  double inix, iniy, inivx, inivy, aclx, acly, dtim, ttotal;
  inix = 0.0;
  dtim = 0.1;
  inivx = 14.0;
  ttotal = 4.0;
  iniy = 0.0;
  inivy = 14.0;
  aclx = 0.0;
  acly = −9.8;
  Accm2D acmo2d(inix, dtim, inivx, ttotal, iniy, inivy, aclx, acly);
  printf("   \ n ");
  printf("    \ n ");
  acmo2d.archive();
}
```

4.6 Assessment: Exploration, shms.cpp

The superposition of independent simple harmonic motion in each of two dimensions can be studied with OOP. Define a class `ShmX` for harmonic motion in the x direction:

$$x = A_x \sin(\omega_x t + \phi_x) \tag{4.7}$$

where A_x is the amplitude, ω_x is the angular frequency, and ϕ_x is the phase. Define another class `Shmy` for independent harmonic motion in the y direction:

$$y = A_y \sin(\omega_y t + \phi_y) \tag{4.8}$$

Now employ the concept of *multiple inheritance* to define a child class `ShmXY` of both `ShmX` and `ShmY`. It should have a member function to write a file with the x and y positions at several times (which can then be used to plot Lissajous figures). To obtain multiple inheritance use `class ShmXY : public ShmX, public ShmY`. To obtain the constructor, use something like this:

```
ShmXY::ShmXY(double Axi, double delx, double wx, double tx,
  double dtx, double Ayi, double dely, double wy, double ty, double
      dty)
  : ShmX(Axi, delx, wx, tx, dtx), ShmY(Ay, dely, wy, ty, dty)
```