Task 1: Understand the implementation of the member functions.

What is the initial state of the ball when the object is created (instantiated)?
- the initial state can be found from the ball.cpp implementation file.
- r(0.1), x(0), y(0), vx(0.3), vy(-0.1), g(9.8), m(1), xmin(-1), xmax(1), ymin(-1), ymax(1)
- the implementation uses initialization list such that constant variables can be initialized.
- although there are no constant variables, gravity can be set as a constant, as it is unlikely to change during simulation

How does display() prints information on the screen?
- by putting the string into ostream - 'cout << '.
- the display output x followed by y, for example: '10 20'.

The ball is constrained to bounce in the box [−1, 1] × [−1, 1]. How is this handled by the integrator implemented in step()? Is the total (kinetic + potential) energy of the ball conserved by this integrator?
- the integrator integrates position from velocity and acceleration if the ball is within bound, if outside bound, velocity vector would simply flip its direction, mimic a bound action.
- total energy would be conserved, since there are no damping action, and conservation of energy. however, this is subject to numerical errors

Now look at test-ball.cpp and answer the following question: What do you expect the output of your program to be?
- it would be a stream of rows of x and y, for 100 lines.

Task 2: Running the program.

This will automatically compile and link the program to create and run the ball executable for you. A console window will pop up, displaying the output of the program. Does it look reasonable?
- reasonable.

```
jiahao@jiahao b16-lab-code-master % ./test-ball
0.01 -0.00877778
0.02 -0.0284444
0.03 -0.059
0.04 -0.100444
0.05 -0.152778
0.06 -0.216
0.07 -0.290111
0.08 -0.375111
0.09 -0.471
0.1 -0.577778
0.11 -0.695444
0.12 -0.824
0.13 -0.824
0.14 -0.695444
0.15 -0.577778
0.16 -0.471
0.17 -0.375111
0.18 -0.290111
0.19 -0.216
0.2 -0.152778
0.21 -0.100444
0.22 -0.059
```

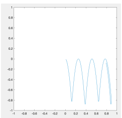
Task 3: Use redirection to save the textual output to a file.

ball.txt

```
1   0.01 -0.00877778
2   0.02 -0.0284444
3   0.03 -0.059
4   0.04 -0.100444
5   0.05 -0.152778
6   0.06 -0.216
7   0.07 -0.290111
8   0.08 -0.375111
9   0.09 -0.471
10  0.1 -0.577778
11  0.11 -0.695444
12  0.12 -0.824
13  0.13 -0.824
14  0.14 -0.695444
15  0.15 -0.577778
16  0.16 -0.471
17  0.17 -0.375111
18  0.18 -0.290111
```

Task 4: Load the output in MATLAB and visualise it.

Does the output seem reasonable?
- the output curve is parabolic, which matches the prediction that energy is conserved.
- the ball bounces at the bottom and at the right.
- there are missing points close to the bottom as the ball crosses the boundary, hence velocity is simply reversed at the next time step.
- however, after setting xlim and ylim to [-1, 1], there seems to be some missing points with collision to the right wall.
- After reviewing the code, this is due to the non zero radius of the ball.



Task 5: Member functions and separation of concerns.

How should the class be changed so that a user could be able to get and set the position of the ball?
- either move the double x and double y into public section
- or create Getter and Setter member function as public functions

The member functions of a class are often said to encode its "behaviour". Can you find a practical example demonstrating why separating the data representation from its behaviour is useful?
- if we want the ball to be bouncing around a circular object in free space (from gravity), it may be easier to have data representation in polar coordinates.
- with the behaviour separated from the data representation, the end user can still operate on this object without worrying about the unusual coordinates systems the ball is currently in.

Task 6: Understanding constructors.

The constructor Ball() does not allow the user of the class to choose a custom initial position or velocity of the ball. Design an alternative constructor that would allow to do so upon class instantiation.

```
Ball::Ball(double _x, double _y, double _vx, double _vy) : r(0.1), g(9.8), m(1), xmin(-1), xmax(1), ymin(-1), ymax(1) {
  x = _x;
  y = _y;
  vx = _vx;
  vy = _vy;
}
```

Task 7: Programming principles.

What is the distinction between a class declaration, definition, implementation, and instantiation?

- class declaration - is allocation of memory space to the variable defined, the memory size depends on the type of the variable
- class definition - is interface of the class stored in header file. it defines how other program can interact with it.
- class implementation - is the behaviour of the class stored in cpp file. it defines the internal processing of the data.
- class instantiation - is population of memory occupied by the class variable. default constructor is provided.

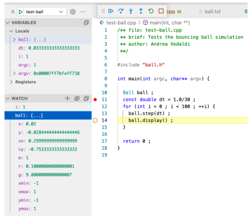
Task 8: Compiling the program.

```
jiahao@jiahao b16-lab-code-master % ./test-ball
0.01 -0.00877778
0.02 -0.0284444
0.03 -0.059
0.04 -0.100444
0.05 -0.152778
0.06 -0.216
0.07 -0.290111
0.08 -0.375111
0.09 -0.471
0.1 -0.577778
0.11 -0.695444
0.12 -0.824
0.13 -0.824
0.14 -0.695444
0.15 -0.577778
0.16 -0.471
0.17 -0.375111
0.18 -0.290111
0.19 -0.216
0.2 -0.152778
0.21 -0.100444
0.22 -0.059
```

Task 9: Try the debugger.

Try out the various options. In particular try to discover how to display the value of a variable (e.g. ball). Once you have done this, step through the program one line at a time examining how it changes during the for loop. Also try inserting a breakpoint and continuing execution to observe the program stopping at the breakpoint.

- to display value of the variable, add them in the WATCH tab.



Task 10: Understanding interfaces.

Can you replace the for loop in the main function defined in test-ball.cpp with the instruction run(ball, dt)?
- yes, because ball is inherited from simulation, and has provided implementation of step and display.

What would happen if Simulation did not declare step and display to be virtual? Would run still work as expected? If not, what would happen?
- since input type for the function is 'simulation', calling step and display will call the member function from simulation class if they are not declared virtual.
- run function would not work as expected.
- what happens depends on how step and display are implemented in 'simulation'.

Task 11: Complete the Mass class.

Start by completing the implementation Mass class by implementing the member functions getEnergy() and step().

```
double Mass::getEnergy(double gravity) const {

  // potential energy
  double potential = mass * gravity * position.y;

  // kinetic energy
  double kinetic = 0.5 * mass * velocity.norm2();

  // total energy
  double total = potential + kinetic;

  return total;

}
```

```
void Mass::step(double dt) {

  // new position and velocity
  // assuming constant acceleration
  Vector2 end_position = position + velocity * dt + 0.5 * force / mass * dt * dt;
  Vector2 end_velocity = velocity + force / mass * dt;

  // x direction
  if (xmin <= end_position.x - radius && end_position.x + radius <= xmax) {
    position.x = end_position.x;
    velocity.x = end_velocity.x;
  } else {
    velocity.x = - velocity.x;
  }

  // y direction
  if (ymin <= end_position.y - radius && end_position.y + radius <= ymax) {
    position.y = end_position.y;
    velocity.y = end_velocity.y;
  } else {
    velocity.y = - velocity.y;
  }

}
```

Task 12: Complete the Spring class. Once you are satisfied with your implementation of Mass, move to the implementation of Spring.

```
Vector2 Spring::getForce() const {
  // mass information
  Vector2 x1 = mass1 -> getPosition();
  Vector2 x2 = mass2 -> getPosition();
  Vector2 v1 = mass1 -> getVelocity();
  Vector2 v2 = mass2 -> getVelocity();

  // spring information
  double l = getLength();                 // spring length
  Vector2 u12 = 1/l * (x2 - x1);          // spring direction
  Vector2 v12 = dot((v2 - v1), u12) * u12; // contraction/expansion speed in spring direction

  // forces
  Vector2 F1 = stiffness * (l - naturalLength) * u12;
  Vector2 F2 = damping * v12;
  Vector2 F = F1 + F2;

  return F ;
}
```

```
class Spring {
public:
  Spring(Mass * mass1, Mass * mass2, double naturalLength, double stiff, double damping = 0.01) ;
  Mass * getMass1() const ;
  Mass * getMass2() const ;
  Vector2 getForce() const ;
  double getLength() const ;
  double getEnergy() const ;
  double getStiffness() const;

protected:

  Mass * mass1 ;
  Mass * mass2 ;

  double naturalLength ;
  double stiffness ;
  double damping ;

} ;
```

Task 13: Design and implement SpringMass.

Once Mass and Spring are complete, it is time to implement the simulation by constructing a class SpringMass that represents the simulated "world". In this case you are expected to work out part of the design of a class itself.

```
class SpringMass : public Simulation {
public:
  // constructor
  SpringMass();

  // add elements
  void addSpring(std::vector<Spring> more_springs);
  void addMass(Spring);

  // simulation
  void setGravity(double _gravity);
  void step(double dt) ;
  void display() ;
  void loadSample();

  // calculation
  double getEnergy() ;

protected:
  // data member
  std::vector<Spring> spring_list;
  std::vector<Mass * > mass_list;

  double gravity;

} ;
```

```
SpringMass::SpringMass() {
  gravity = EARTH_GRAVITY;
}

void SpringMass::addSpring(std::vector<Spring> more_springs) {
  for (std::vector<Spring>::iterator it = begin(more_springs); it != end (more_springs); ++it) {
    // add springs
    spring_list.push_back(*it) ;

    // add mass
    addMass(*it);
  }
}
void SpringMass::addMass(Spring _spring) {
  Mass * mass1 = _spring.getMass1();
  Mass * mass2 = _spring.getMass2();

  // append if not present
  if ( std::find(mass_list.begin(), mass_list.end(), mass1) == mass_list.end() ) {
    mass_list.push_back(mass1);
  }
  if ( std::find(mass_list.begin(), mass_list.end(), mass2) == mass_list.end() ) {
    mass_list.push_back(mass2);
  }
}

void SpringMass::loadSample() {
  // mass
  const double mass = 1 ;
  const double radius = 0.1 ;
  Mass * m1 = new Mass(Vector2(-0.5,0), Vector2(0, 0), mass, radius) ;
  Mass * m2 = new Mass(Vector2(+0.5,0), Vector2(1, 2), mass, radius) ; // need destructor

  // spring
  const double naturalLength = 0;
  const double stiff = 0;
  const double damping = 0;
  Spring spring1(m1, m2, naturalLength, stiff, damping) ;

  // spring vector
  std::vector<Spring> more_springs(1, spring1);

  // springmass
  addSpring(more_springs);
}

void SpringMass::setGravity(double _gravity) {
  gravity = _gravity;
}

void SpringMass::display() {
  // multiple mass per line
  for (std::vector<Mass *>::iterator it = begin(mass_list); it != end (mass_list); ++it) {
    Vector2 position = (*it) -> getPosition();
    std::cout << position.x << " " << position.y << " ";
  }
  // end line
  std::cout << std::endl;
}

double SpringMass::getEnergy() {
  double energy = 0 ;

  // mass
  for (std::vector<Mass *>::iterator it = mass_list.begin(); it != mass_list.end(); ++it) {
    energy += (*it)->getEnergy(gravity);
  }

  // spring
  for (std::vector<Spring>::iterator it = spring_list.begin(); it != spring_list.end(); ++it) {
    energy += (*it).getEnergy();
  }

  return energy ;
}
```

```
void SpringMass::step(double dt) {

  // set initial force
  Vector2 g(0, -gravity) ;
  for (std::vector<Mass *>::iterator it = mass_list.begin(); it != mass_list.end(); ++it) {
    (*it) -> setForce(g * (*it)->getMass());
  }

  // get spring force and add force
  for (std::vector<Spring>::iterator it = spring_list.begin(); it != spring_list.end(); ++it) {
    // get mass
    Mass * mass1 = (*it).getMass1();
    Mass * mass2 = (*it).getMass2();

    // get force
    Vector2 F1 = (*it).getForce();
    Vector2 F2 = -1 * F1;

    // add force to mass
    mass1 -> addForce(F1);
    mass2 -> addForce(F2);
  }

  // update
  for (std::vector<Mass *>::iterator it = mass_list.begin(); it != mass_list.end(); ++it) {
    (*it) -> step(dt);
  }

}
```

B16 Lab Logbook II

Task 14: Initialise and run the simulation.

Modify the file test-springmass.cpp to run your simulation. Write code to setup a new SpringMass instances with two masses and one spring (or a more complex configuration) using the interface that you just designed and implemented.

```cpp
int main(int argc, char** argv) {

  // mass
  const double mass = 0.1 ;
  const double radius = 0.2 ;
  Mass m1(Vector2(-0.5,0), Vector2(1, 0), mass, radius) ;
  Mass m2(Vector2(+0.5,0), Vector2(0.5, 0), mass, radius) ;

  // spring
  const double naturalLength = 1;
  const double stiff = 0;
  const double damping = 0;
  Spring spring1(&m1, &m2, naturalLength, stiff, damping) ;

  // spring vector
  std::vector<Spring> more_springs(1, spring1);


  // springmass
  SpringMass springmass;
  springmass.addSpring(more_springs);


  // simulation
  const double dt = 1.0/30 ;
  for (int i = 0 ; i < 400 ; ++i) {
    springmass.step(dt) ;
    springmass.display() ;
  }

  return 0 ;
}
```

```
jiahao@jiahao b16-lab-code-master % ./test-springmass
-0.466667 0 0.516667 0
-0.433333 -0.0018 0.533333 -0.0018
-0.4 -0.0054 0.55 -0.0054
-0.366667 -0.0108 0.566667 -0.0108
-0.333333 -0.018 0.583333 -0.018
-0.3 -0.027 0.6 -0.027
-0.266667 -0.0378 0.616667 -0.0378
-0.233333 -0.0504 0.633333 -0.0504
-0.2 -0.0648 0.65 -0.0648
-0.166667 -0.081 0.666667 -0.081
-0.133333 -0.099 0.683333 -0.099
-0.1 -0.1188 0.7 -0.1188
-0.0666667 -0.1404 0.716667 -0.1404
-0.0333333 -0.1638 0.733333 -0.1638
-9.71445e-17 -0.189 0.75 -0.189
0.0333333 -0.216 0.766667 -0.216
0.0666667 -0.2448 0.783333 -0.2448
0.1 -0.2754 0.783333 -0.2754
0.133333 -0.3078 0.766667 -0.3078
0.166667 -0.342 0.75 -0.342
0.2 -0.378 0.733333 -0.378
0.233333 -0.4158 0.716667 -0.4158
0.266667 -0.4554 0.7 -0.4554
```

Task 15: Redirect the program output to a file and visualise it in MATLAB.

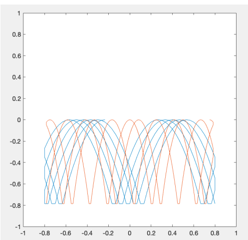
Try the same trick we used for the Ball simulation in order to redirect your program output to a file, load it in MATLAB, and visualise it. Note that in this case more than one mass per line of output should be included.

```matlab
load('springmass.txt');

hold on
mass1 = plot(springmass(1, 1), springmass(1, 2), 'o');
mass2 = plot(springmass(1, 3), springmass(1, 4), 'o');
xlim([-1, 1])
ylim([-1, 1])
axis square

for i = 2:size(springmass, 1)
    mass1.XData = springmass(i, 1);
    mass1.YData = springmass(i, 2);
    mass2.XData = springmass(i, 3);
    mass2.YData = springmass(i, 4);
    drawnow
end
```

```matlab
ball = load("springmass.txt");
plot(ball(:, 1), ball(:, 2), ball(:, 3), ball(:, 4))
axis square
xlim([-1, 1])
ylim([-1, 1])
```



Task 18: Understand how this is implemented by using inheritance.

Which objects are of type Simulation and which of type Drawable?
- BallDrawable is of type Ball which is of type Simulation. Thus BallDrawable is a simulation
- BallDrawable is of type Drawable.

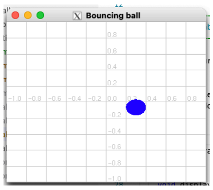
Which objects are creating an instance of Figure and when?
- BallDrawable creates an instance of Figure using initialization list when it is first constructed.

Which objects are added as Drawable to the figure and when?
- BallDrawable is added as Drawable to figure during its default construction.

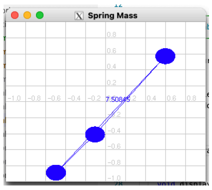
Which objects implement respectively: display, draw, and update?
- display is a virtual function in simulation, which is later overwritten in Ball, and now overwritten in BallDrawable.
- draw is implemented in both BallDrawable and in Figure. when draw() is called in figure, it loops over all objects it has and call the draw() function from those objects, in this case, the BallDrawable.
- update is implemented in Figure, which is called from display in BallDrawable. during 'run', BallDrawable is passed in as simulation, and its display() is called. since BallDrawable overwrites teh display() function, it instead calls update from Figure.

Which different purposes do these functions serve?
- BallDrawable() is the constructor of this class
- draw() calls the function in Figure to draw in figure.
- display() calls the update() function in figure to update the drawing.

Task 19: Compile and run code using the graphics library.



Task 20: Create a graphic version of SpringMass.




```cpp
class SpringMassDrawable : public SpringMass, public Drawable {

  private:
    Figure figure ;

  public:
    SpringMassDrawable() : figure("Spring Mass") {
      figure.addDrawable(this) ;
    }

    void draw() {

      // draw mass
      for (std::vector<Mass *>::iterator it = begin(mass_list); it != end (mass_list); ++it) {
        // position and radius
        Vector2 position = (*it)->getPosition();
        double r = (*it)->getRadius();

        // draw
        figure.drawCircle(position.x, position.y, r) ;
      }

        // draw spring
        for (std::vector<Spring>::iterator it = begin(spring_list); it != end (spring_list); ++it) {
          // position
          Mass * m1 = (*it).getMass1();
          Mass * m2 = (*it).getMass2();

          Vector2 p1 = m1->getPosition();
          Vector2 p2 = m2->getPosition();

          // thickness
          double thickness = (*it).getStiffness();

          // draw
          figure.drawLine(p1.x, p1.y, p2.x, p2.y, thickness) ;
        }


        // draw energy
        // x and y
        double x = 0;
        double y = 0;
        // energy
        double energy = getEnergy();
        std::stringstream ss;
        ss << energy;
        std::string energy_str = ss.str();

        figure.drawString(x, y, energy_str);

      }

    void display() {
      figure.update() ;
    }
} ;
```

```cpp
int main(int argc, char** argv) {
  // GLUT
  glutInit(&argc,argv);

  // run simulation
  SpringMassDrawable springmass;


  // springmass.loadSample();
  // mass
  const double mass = 1 ;
  const double radius = 0.1 ;
  Mass * m1 = new Mass(Vector2(-0.5,0), Vector2(0, 0), mass, radius) ;
  Mass * m2 = new Mass(Vector2(+0.5,0), Vector2(1, 2), mass, radius) ; // need destructor
  Mass * m3 = new Mass(Vector2(+0.5,0.5), Vector2(0, 0), mass, radius) ; // need destructor

  // spring
  const double naturalLength = 0.5;
  const double stiff = 1;
  const double damping = 1;
  Spring spring1(m1, m2, naturalLength, 0, 0) ;
  Spring spring2(m2, m3, naturalLength, 1, 0) ;
  Spring spring3(m3, m1, naturalLength, 1, 0) ;

  // spring vector
  std::vector<Spring> more_springs(1, spring1);
  more_springs.push_back(spring2);
  more_springs.push_back(spring3);

  // springmass
  springmass.addSpring(more_springs);

  run(&springmass, 1.0/240.0);

  // return
  return 0 ;
}
```

Task 21: Compute and display the energy of the spring-mass system.

What happens to the energy as the simulation progresses? Why?
- initially my implementation was incorrect. I used 1/2 instead of 1.0/2.0 when calculating kinetic energy as well as end position, this results in an fluctuating total energy (since kinetic is zero due to 1/2), and the mass is bouncing higher and higher (since end position formula is incorrect)
- after fixing this issue, the total energy accounting for the mass and the spring remains constant, when damping is set to zero. for non-zero damping in the spring, the total energy keeps decreasing.
- there is however, a slight increase in the total energy. investigation shows this increase in energy occurs from the potential energy of springs. may be numerical error

Now set the damping factor of the spring to zero. What happens now to the energy? Is this result correct? If not, what may be the problem?
- answered above.

note

- copy constructor seems to exist by default for custom class pointer. e.g. mass1(_mass)
- type safe language is really really good. it indicates an error as soon as it is typed. this stops the error from happening!
- can check if energy is conserved to see if simulation is running correctly
- visualization in MATLAB shows the ball bouncing higher and higher, something is wrong
- the error of increasing energy is because, 1/2 returns an integer which is 0, 1.0/2.0 returns 0.5. the former equation result in incorrect calculation of end position as well as incorrect kinetic energy.