コンピューターシミュレーション
プログラム解説書

3年15組　38番　島岡　望

1.プログラム概要

天体同士が万有引力で影響しあって運動を行うシミュレーションを作成した。

2.開発環境/実行環境

マシン: PC
OS: Ubuntu 18.04.2 64bit
言語: Java 11.0.2
実行環境: openjdk version "11.0.2"

3.ファイル構成

- OrbitSim.java

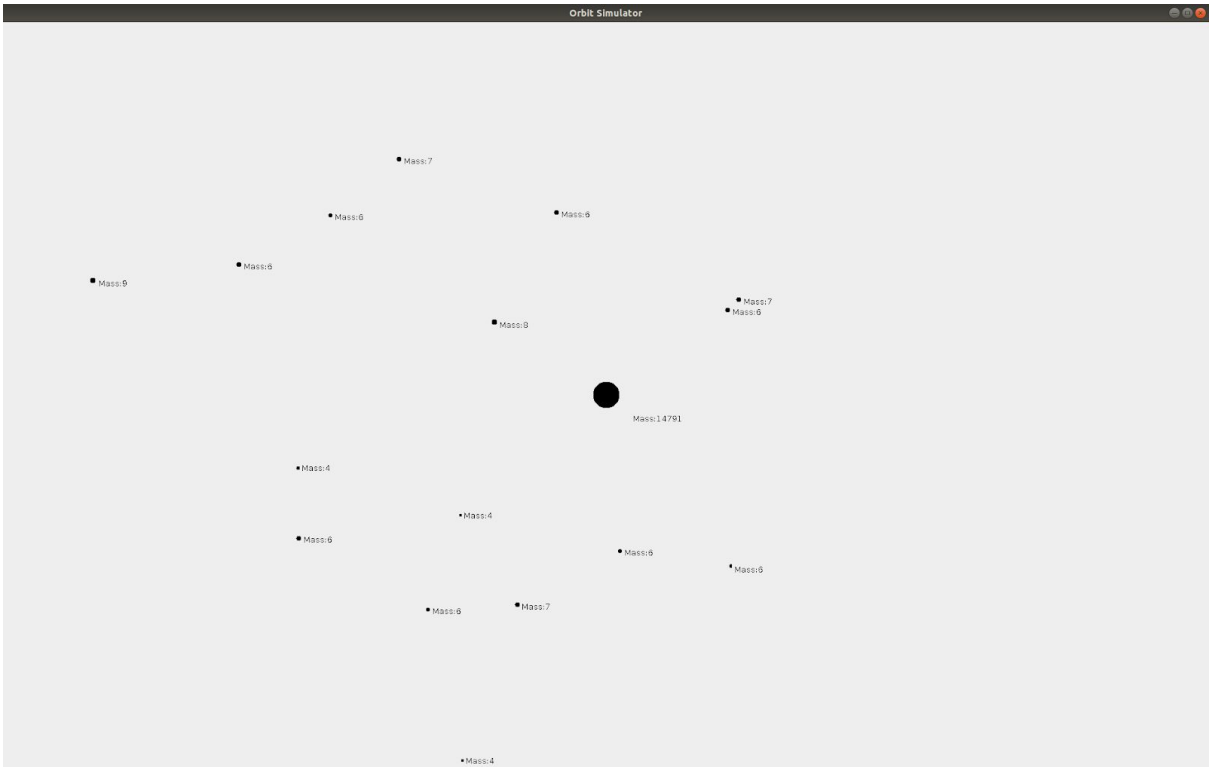
4.コンパイル方法

ソースディレクトリのあるディレクトリで
$ javac OrbitSim.java

5.実行方法

コンパイルしたディレクトリで
$ java OrbitSim

6.実行結果

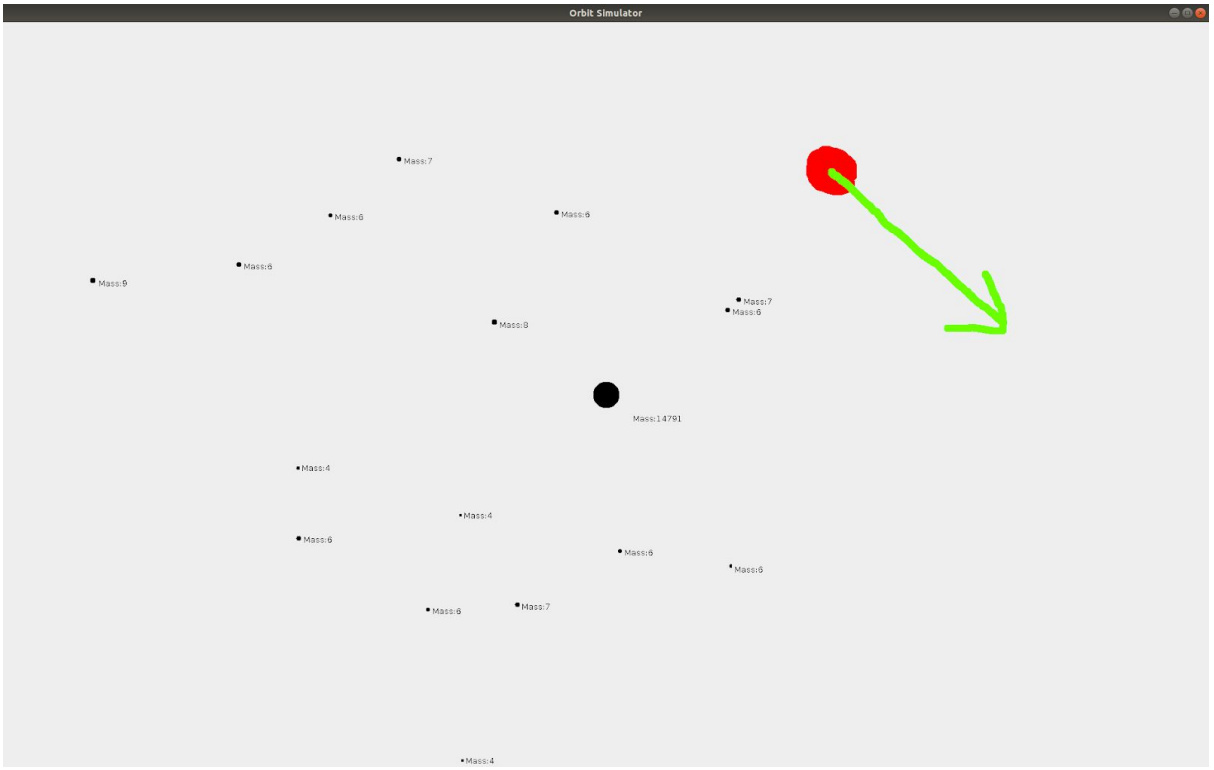


7.プログラム操作説明

プログラムを実行すると、ウィンドウが表示される。

- ドラッグで新しい天体（質点）を追加することができる。ここで、ドラッグを始めた位置で天体の位置が決まり、ドラッグした方向と距離によって天体の速度が決定され、ドラッグしていた時間によって質量がきまる。
- 初めて質点を追加する場合は、ウィンドウのどこをクリックしても画面中央に表示される。
- 質点はウィンドウ上に円として表示され、その質量によって円の大きさが変わり、質量が大きければ大きいほど大きい円として表示される。

例えば、以下の画像で赤い場所に中央の質点に対して円軌道をとる質点を追加したい場合は、赤い中央でドラッグを開始し、緑の矢印にそってドラッグを続け、矢印の先端（やじり）でボタンを離してドラッグを完了すればよい。その時ドラッグしていた時間で質量が決まる。



トリック1. 初めて追加する天体はできるだけ質量の大きい方がよいが、大きすぎると公転速度が極端に高くなってしまうので、質量は30000ぐらいがよい。
トリック2. 中央の天体に近づくほど軌道を描くために必要な速度が大きくなるので長い距離ドラッグする。
トリック3. 衛星はなるべく質量の少ないほうがよい。

8.原理

ニュートンの万有引力の法則

$$F = G\frac{Mm}{r^2}$$

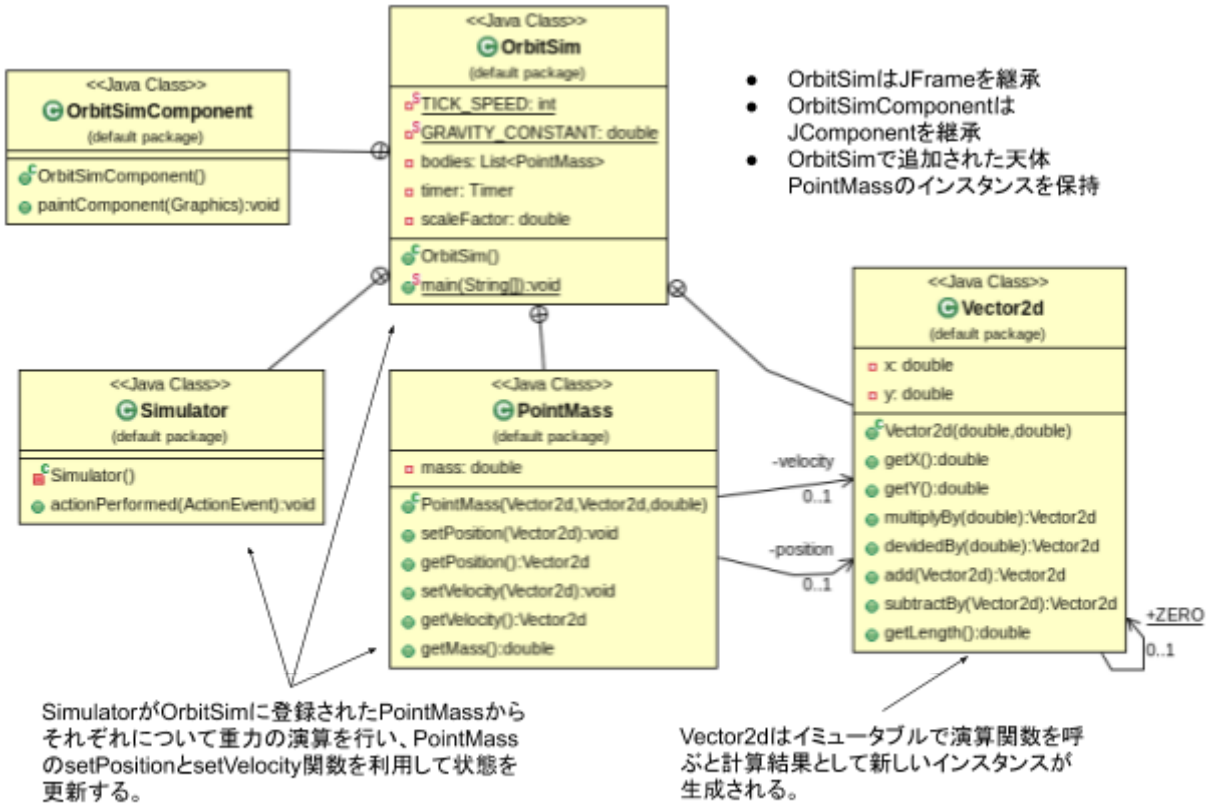
のベクトル形式（2に働く力）

$$F_{21} = G\frac{Mm}{\|r_{12}\|^3}r_{12}$$

とF=maを使って加速度を計算し、充分小さい時間で速度に加算していく。

9.プログラムコード説明

9.1.クラス図



9.2.クラス説明

9.2.1.OrbitSim

`` class OrbitSim extends JFrame { private static int TICK_SPEED = 1; private static double GRAVITY_CONSTANT = 60000; private List<PointMass> bodies = new ArrayList<>(); private Timer timer; private double scaleFactor = 2; public OrbitSim() { super("Orbit Simulator"); setSize(500, 500); setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); setLayout(new BorderLayout()); add(new OrbitSimComponent(), BorderLayout.CENTER); // Setting up timer 	JFrameを継承したクラスで、ウィンドウを表す。 bodiesリストに追加された天体のリストが格納されている。 OrbitSimComponentが最大化されてほしいので、レイアウトにはBorderLayoutを利用し、CENTERに追加する。 javax.swing.Timerを利用して定期的に重力計算を行うようスケジュールする。

`` timer = new Timer(TICK_SPEED, new Simulator()); this.timer.start(); } public static void main(String[] args) { new OrbitSim().setVisible(true); } } ```	mainでは、OrbitSimのインスタンス（ウィンドウ）を生成し、可視をtrueにすることでそれを表示する。

9.2.2.Vector2d

`` private static class Vector2d { public static Vector2d ZERO = new Vector2d(0, 0); private double x; private double y; public Vector2d(double x, double y) { this.x = x; this.y = y; } public double getX() { return x; } public double getY() { return y; } public Vector2d multiplyBy(double factor) { return new Vector2d(x*factor, y*factor); } public Vector2d devidedBy(double factor) { return new Vector2d(x/factor, y/factor); } public Vector2d add(Vector2d vector2d) { return new Vector2d(x + vector2d.x, y + vector2d.y); } public Vector2d subtractBy(Vector2d vector2d) { return new Vector2d(x - vector2d.x, y - vector2d.y); } public double getLength() { return Math.sqrt(Math.pow(x, 2) + Math.pow(y, ```	OrbitSimの内部クラス。 2次元実ベクトルを表すクラス。 いくつかの対ベクトルまたはスカラ演算が関数として定義されており、イミュータブルなので、自身のインスタンスのとの演算結果は新しいVector2dのインスタンスを生成して返す。 getLengthはこのベクトルの大きさ（内積の平方根）を返す。

2)); } }	

9.2.3.PointMass

private static class PointMass { private Vector2d position; private Vector2d velocity; private double mass; public PointMass(Vector2d position, Vector2d velocity, double mass) { this.position = position; this.velocity = velocity; this.mass = mass; } public void setPosition(Vector2d position) { this.position = position; } public Vector2d getPosition() { return position; } public void setVelocity(Vector2d velocity) { this.velocity = velocity; } public Vector2d getVelocity() { return velocity; } public double getMass() { return mass; } }	OrbitSimの内部クラス。 天体（質点）を表すクラス。 天体の位置、速度、質量を保持する。 インスタンスを生成するときは、この全てを与えなければならない。 位置と速度はシミュレーションのため、セッターをあるので、のちに変更できる。

9.2.4. Simulator

private class Simulator implements ActionListener { @Override public void actionPerformed(ActionEvent arg0) { synchronized (bodies) { // Save copy of bodies List<PointMass> copied = new ArrayList<>(bodies);	OrbitSimの内部クラス。 シミュレーションを実際に行うクラス。 javax.swing.Timerで利用する

```
            for (PointMass pointMass : copied) {
                // We want to know combined force applied
to this point mass
                Vector2d pos = pointMass.getPosition();
                Vector2d velDelta = Vector2d.ZERO;

                for (PointMass affect : copied) {
                    if (pointMass == affect)
                        continue;

                    // Calculate force of this object
"affect" applies to this object
                    // Acceleration of Object 1 = G * M1 /
||^p2 - ^p1||^3 * (^p2 - ^p1)
                    Vector2d affPos =
affect.getPosition();
                    double affMass = affect.getMass();
                    Vector2d distance =
affPos.subtractBy(pos);

                    // ^p2 - ^p1
                    Vector2d localAccel =
affPos.subtractBy(pos);
                    // G*M1
                    localAccel =
localAccel.multiplyBy(GRAVITY_CONSTANT*affMass);
                    // ||^p2 - ^p1||^3
                    localAccel =
localAccel.devidedBy(Math.pow(distance.getLength(), 3));
                    // Multiply by tick speed
                    localAccel =
localAccel.multiplyBy(TICK_SPEED).devidecBy(1000);

                    // Add it to delta
                    velDelta = velDelta.add(localAccel);
                }

                // Replace velocity and position
                Vector2d newVel =
pointMass.getVelocity().add(velDelta);
                Vector2d newPos =
pointMass.getPosition().add(newVel);
                pointMass.setVelocity(newVel);
                pointMass.setPosition(newPos);
            }
        }

        repaint();
    }
}
```

ため、TimerActionListenerを実装する。

タイマーによって一定間隔でactionPerformedが呼ばれる。

全ての天体について、それ以外の天体がその天体に及ぼす加速度（力/自身の質量）の合成し、それをシミュレーション速度（タイマーの間隔）で掛け算する。タイマー間隔で掛け算するのは、表した加速度が1sあたりの加速度であるからである。

そうして求められた速度の変化量を、天体の速度に加算する。

さらに、その速度に基づいて位置を決定する。

10.考察

- 単純な微分方程式なら、シミュレーション時間（密度）を充分小さくとれば、かなり近い結果が得られる。
- O(n^2)のアルゴリズムでも、天体が少ないなら相当な速度で実行できる。
- ベクトルクラスはゲームなどでよく実装されているが、2変数を扱うよりも計算のイメージがつきやすい。例えば、平行移動するためにx,yを別々で扱うよりも、ベクトルとして足し算を実行した方が、コードから操作を簡単にイメージできるのでよい。今回、相対速度や相対位置を多用したが、ベクトルの幾何学的な概念を考えることによって簡単に実装できた。

11.感想

- コードを書き始めた時はここまでうまく動くとは思わなかったが、シミュレーション時間を小さくとることでほとんど微分に近い結果が出ることが分かって非常に興味深かった。
- Javaを久しぶりに使ったので、MouseAdaptorがもっと読みやすくできるのではないかと思ったが、方法を忘れてしまった。

12.参考文献

- **Newton’s law of universal gravitation - Vector form - Wikipedia**
  https://en.wikipedia.org/wiki/Newton%27s_law_of_universal_gravitation
- **ObjectAid UML Explorer - Class diagramの生成に利用**
  https://objectaid.com/