Modeling and Analyzing 2D Soft Body Interactions using Spring-Mass Systems

PHY 380 - Spring 2025

Yashab Yadav
Department of Physics, Lehigh University

Contents

1 Introduction

The spring-mass model is a well-established and computationally manageable approach for simulating soft bodies. It uses systems of coupled ODEs that can be integrated with solvers from the OSP library. Unlike the more computationally demanding Finite Element Method, this model allows rapid iteration, easier debugging, and better integration with OSP's visualization tools.

This approach enables meaningful physical analysis (e.g., energy dissipation, deformation dynamics, fluctuation behavior).

The general goal of this project is to use basic physics principles to come up with a model and apply it to produce physically (or at least visully) accurate soft body simulations. I'll stick to the object oriented practices and problem solving approach discussed in the class.

2 Utils Package

The simulation is restricted to two dimensions and in spirits of being object oriented, I wrote a utility package to work with 2D vectors that can be scaled upon in the future.

2.1 Vec2D

Rather than using two variables for each vector (e.g. x and y for position), I made an object `vec2`, containing both the components. The class file contains methods for standard vector operations, for example the dot product:

```
public double dot(vec2 other){
   return
      (this.x*other.x)+(this.y*other.y);
}
```

The entire `Vec2D.java` file can be found in code listings.

The OSP differential equation interfaces are written to work with scalers and cannot work with `vec2` objects. That is, if I want to solve a vector differential equation, say:

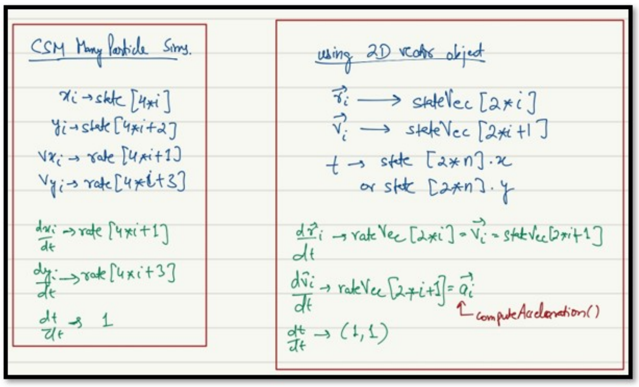
$$\frac{d\vec{r}}{dt} = A\vec{r} \tag{1}$$

where $\vec{\mathbf{r}} = r_x\hat{i} + r_y\hat{j}$ I would need the differential equation separately for r_x and r_y:

$$\frac{dr_x}{dt} = Ar_x \tag{2}$$

$$\frac{dr_y}{dt} = Ar_y \tag{3}$$

To directly use the differential equation pipline of OSP, I modified the classes: `AbstractODESolver, ODE, ODESolver, Verlet` to work directly with `vec2` type objects. These modified classes are also a part of the Utils package and are listed in codeListings section.



CSM Many Particle Sims.

$x_i \to$ state [4*i]
$y_i \to$ state [4*i+2]
$Vx_i \to$ rate [4*i+1]
$Vy_i \to$ rate [4*i+3]

$\frac{dx_i}{dt} \to$ rate [4*i+1]
$\frac{dy_i}{dt} \to$ rate [4*i+3]
$\frac{dt}{dt} \to 1$

using 2D vectors object

$\vec{r}_i \longrightarrow$ stateVec [2*i]
$\vec{V}_i \longrightarrow$ stateVec [2*i+1]
$t \to$ state [2*n].x
or state [2*n].y

$\frac{d\vec{r}_i}{dt} \to$ rateVec [2*i] = $\vec{V}_i$ = stateVec[2*i+1]
$\frac{d\vec{v}_i}{dt} \to$ rateVec [2*i+1] = $\vec{a}_i$
↳ computeAcceleration()
$\frac{dt}{dt} \to (1,1)$

3 SoftBall2 class

An arrangement of springs and masses to mimic soft body behaviour

3.1 Initialization

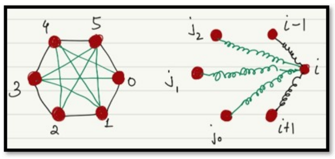


Figure 1: Schematic of the `SoftBall2` class showing the arrangement of springs and 6 masses (left) and the springs connecting node 0 (right).

To initialze a spring mass arrangement to resemble a soft 'ball', similiar to the drawing in 1, the number of surface nodes, radius of the soft ball, and the coordinate of the center are passed via `softBall2()` constructor.

To calculate the coordinate of each surface node, the angular separation between the surface nodes is required, which is given by trivial circle geometry:

$$\theta = \frac{2\pi}{n} \tag{4}$$

and then parametric equation of the circle can be used to get the cartesian coordinates each node:

$$x_i = R\cos(\theta_i) \tag{5}$$
$$y_i = R\sin(\theta_i) \tag{6}$$

which is implemented in the softBall2 class as:

```
for(int i=0; i<this.n;i++){
        currMassCoordinate = new vec2(
            (R*Math.cos((i)*2*Math.PI/n)),
            (R*Math.sin((i)*2*Math.PI/n))
            );
        massList.add(currMassCoordinate);
        this.stateVec[2*i].assign(
            currMassCoordinate
            );
    }
```

Once the soft ball calss is initialized, the equilibrium lengths are calculated using a nested loop and assigned to a 2D matrix. The angular separation between i^{th} and j^{th} node is calculated using:

$$\theta_{ij} = \frac{2\pi}{n}|i-j| \tag{7}$$

and then the equilibrium length of the spring joining i^{th} node with j^{th} node can be calculated from this angle:

$$l_{ij} = 2R\sin\frac{\theta_{ij}}{2} \tag{8}$$

This is implemented in the softBall2 class as:

```
for(int i = 0; i < n; i++) {
        for(int j = 0; j < n; j++) {
            double ang = ((j - i) * 2.0 *
                Math.PI / n);
            allSpringsEquilibriumLen[i][j]
                = Math.abs( R * 2.0 *
                Math.sin(ang / 2.0));
        }
}
```

3.2 Forces and constraints

In addition to gravity, a given nodes experiences forces from springs of stiffness k from it's nearest neighbors, and forces from springs of stiffness k_2 from it's non-immediate neighbors. The expressions of these forces on i^{th} surface node are:

$$\begin{aligned} f_i^{\text{neighbors}} &= (-k\,\Delta l_{i,i+1} + c\,|\Delta\vec{v}_{i,i+1}|)\hat{n}_{i,i+1} \\ &\quad +(-k\,\Delta l_{i,i-1} + c\,|\Delta\vec{v}_{i,i-1}|)\hat{n}_{i,i-1} \end{aligned} \tag{9}$$

$$\begin{aligned} f_i^{\text{non-neighbors}} &= \sum_j(-k_2\,\Delta l_{i,j} \\ &\qquad + c_2\,|\Delta\vec{v}_{i,j}|)\hat{n}_{i,j} \end{aligned} \tag{10}$$

where c and c_2 are velocity dependent damping factors of springs k and $k2$ respectively, and the sum j goes over all the surface nodes $\forall\ j \notin i, i-1, i+1$.

The spring force is computed at every timestep for all the surface nodes in `computeAccelaration()` method by invoking the `calcSpringForceVector()` method, which calculates the forces mentioned in (9) and (10):

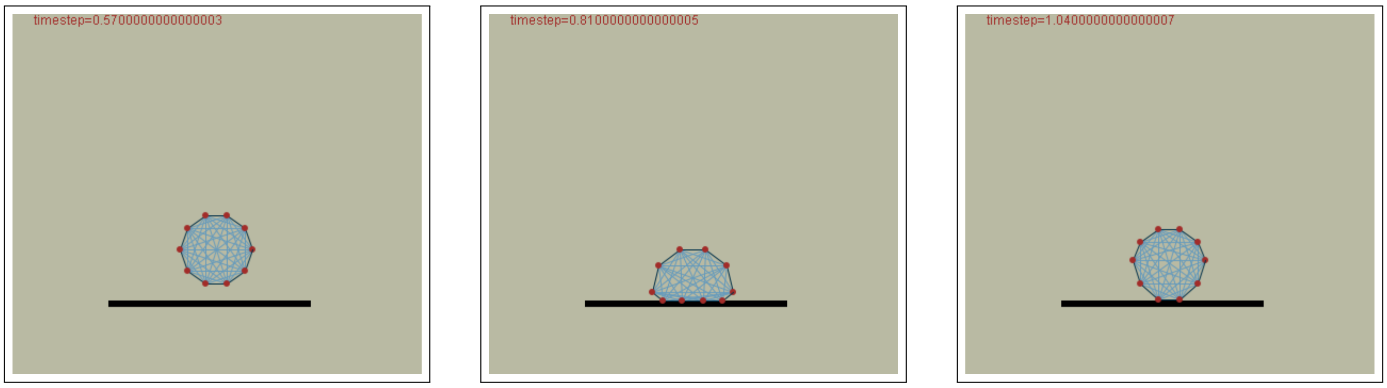
```
private vec2 calcSpringForceVector(double
    eqbmSpringLen, double currSpringLen,
    double k, vec2 unitVecAlongSpring,
    vec2 vel_i,vec2 vel_j, double
    springDampingFactor){
        double delL =
            (currSpringLen-eqbmSpringLen);
            //compression in the spring
        //return
            unitVecAlongSpring.scalerMult(delL);
        //not sure about the signs yet,
            will try to both positive and
            negative

        if(Math.abs(delL)<10E-6){
            return new vec2(0,0);
        }
        else{
            double relativeVelocityMag =
                (vel_i.x - vel_j.x) *
                unitVecAlongSpring.x +
                (vel_i.y - vel_j.y) *
                unitVecAlongSpring.y;

            double dampingTerm =
                springDampingFactor *
                relativeVelocityMag;
            return
                unitVecAlongSpring.scalerMult(k
                * delL - dampingTerm);
        }
    }
```

An additional constraint is collision with ground, which is applied in the `applyGroundCollision` method as:

```
public void applyGroundCollision(double
    groundY, double bounceDamping, double
    friction) {
    for (int i = 0; i < this.n; i++) {
        vec2 pos = stateVec[2*i];
        vec2 vel = stateVec[2*i+1];
        if (pos.y < groundY) {
            pos.y = groundY;
            if (vel.y < 0) vel.y *=
                -bounceDamping;
            vel.x *= friction; // friction
                in x direction to stop
                nodes from sliding in the
                horizontal direction due to
                the surface force
        }
    }
}
```

3.2.1 Visual Analysis

One of most common applications of soft bodies is in video games and movies, so it makes sense to start the analysis by the animating the interaction of soft body with ground or animate a bounce, so to speak.


timestep=0.5700000000000003
timestep=0.8100000000000005
timestep=1.0400000000000007

(a) Before collision (b) During collision (c) Post collision

Figure 2: Stages of soft body evolution during bounce simulation

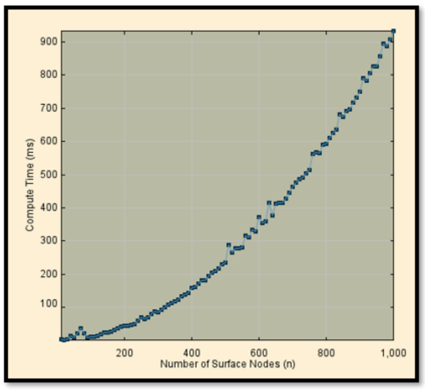
As can be seen in animation snapshots in 2, the behaviour of the softBall in the rendered animation resembles a real world soft body.

3.2.2 Compute speed with number of nodes

To start with quantitative analysis, it's a good idea to put the system to it's computational limits. I run the simulation for multiple `softBall2()` objects with varying number of surface nodes ranging from 10 to a 1000 using the loop:

```
double dt = 0.01;
int steps = 1000;

for (int n = 10; n <= steps; n += 10) {
    softBall2 ball = new
        softBall2(ballCenter, n, k, k2,
        kArea, R, b, c, c2, false);
    ball.setStepSize(dt);

    long startTime = System.nanoTime();
    for (int i = 0; i < steps; i++) {
        ball.applyGroundCollision(-3, 0.5,
           0.0);
        ball.step();
    }
    long endTime = System.nanoTime();
    double elapsedMs = (endTime -
        startTime) / 1e6;

    computeTimePlot.append(0, n,
        elapsedMs);
    System.out.printf("n = %d\tTime = %.3f
        ms%n", n, elapsedMs);
}
```

The full code computeTimeNumNodesApp.java can be found in the code listings. The output plot below shows a non-linear scaling of compute time with the number of surface nodes, which is somewhat expected as there are $n^2 - n$ computations of force calculation at each timestep for a soft ball of n surface nodes (plus other calculations).



3.2.3 Effects of spring damping on area conservation

A trivial test for stability of elastic systems is to see how well they tend to return to their original configuration post deformation. To see how the soft body behaves with different types values of spring damping, I plotted the area vs time for different values of both c and $c2$.

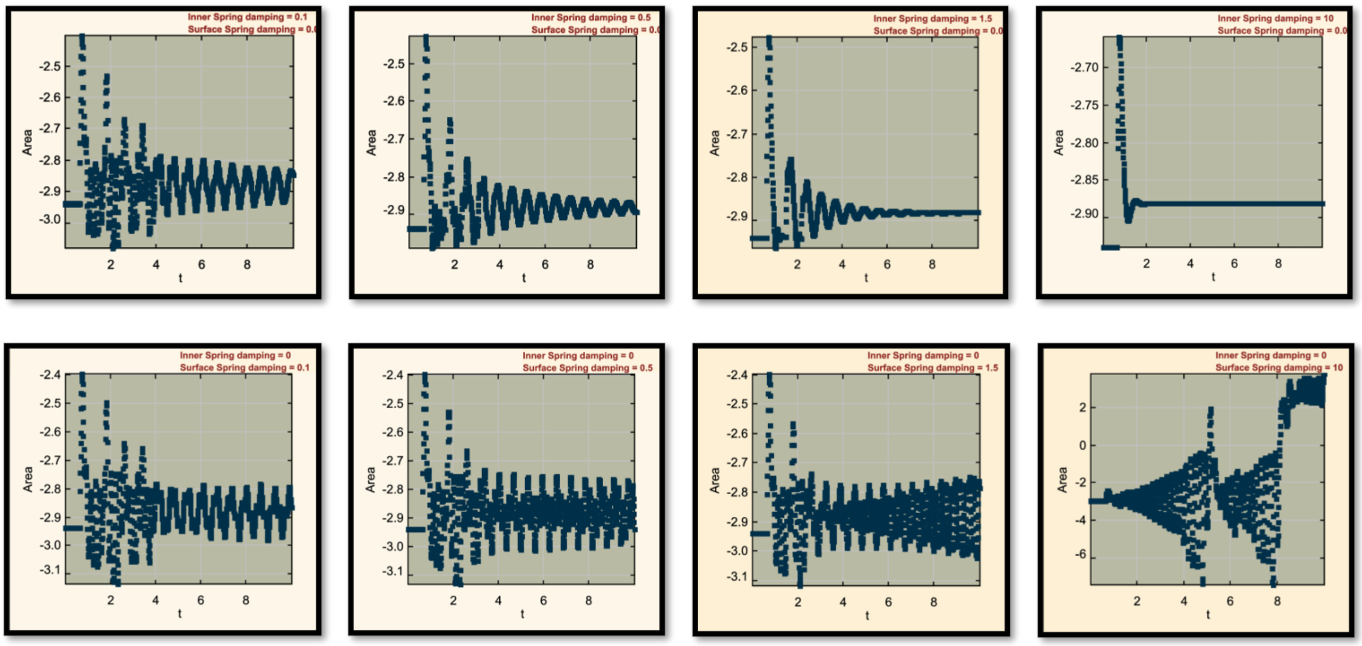


Figure 3: Area vs Time plot of a soft ball with n=10, $k = k2$ =100, $R = 1$ and groundY = -3

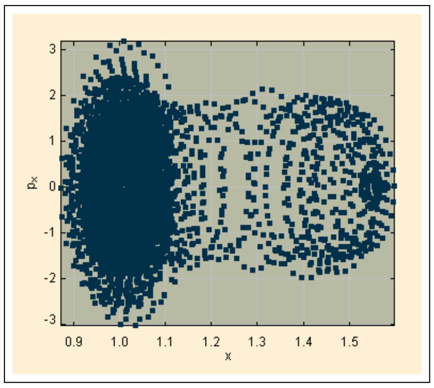
The first row of the figure 3 are the plots for no damping in surface springs, and different values of damping for inner springs. As can be concluded from the plots, the inner spring dampings increases the stability of the soft ball.

The second row of figure 3 are the plots for no damping in the inner springs, and different values of damping for the surface springs. The increase in damping of surface springs makes the soft body susceptible to unstable behaviour post collision with ground.

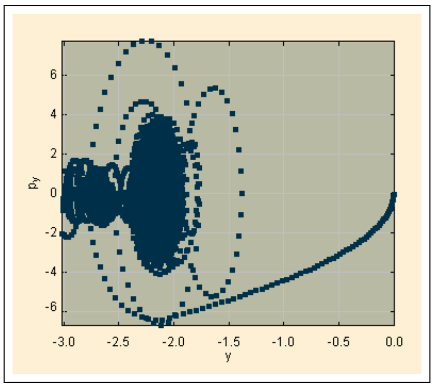
3.2.4 Phase Plots

To understand the energy fluctuations and verify energy conservation, I plot the phse plots of the x-component, y-component and the total momentum. The main loop of the `phasePlotsApp` is:

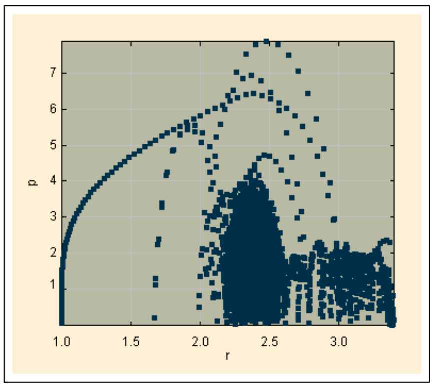
```
double dt = 0.01;
double steps = 10000;
for (int i = 0; i<steps; i++){
    ball.applyGroundCollision(-3, 0.5,
        0.5);
    ball.step();
    x = ball.getStateVec()[2*nodeIndex].x;
    vx =
        ball.getStateVec()[2*nodeIndex+1].x;

    y = ball.getStateVec()[2*nodeIndex].y;
    vy =
        ball.getStateVec()[2*nodeIndex+1].y;
    r = Math.sqrt(
        Math.pow((
        ball.getStateVec()[2*nodeIndex].x),
        2)
    +Math.pow((
        ball.getStateVec()[2*nodeIndex].y)
        ,2)
        );
    p = Math.sqrt(
        Math.pow(
        (ball.getStateVec()[2*nodeIndex+1].x),
        2)
        +Math.pow(
        (ball.getStateVec()[2*nodeIndex+1].y),
        2)
        );


    xPhaseSpace.append(1, x,vx);
    yPhaseSpace.append(1, y,vy);
    xyPhaseSpace.append(1, r,p);
}
```



(a)

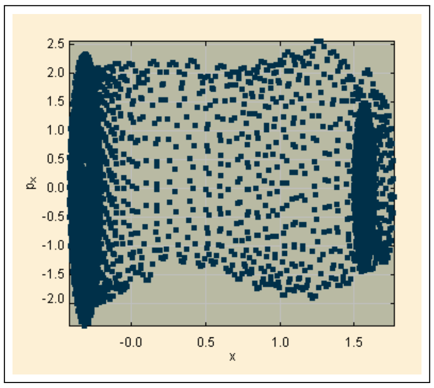


(b)

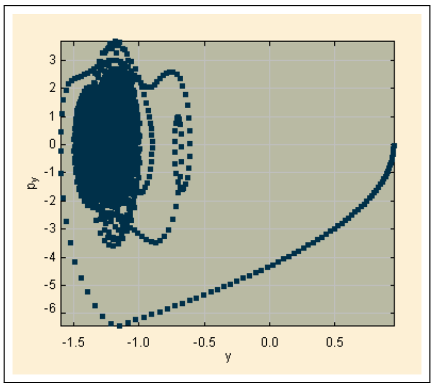


(c)

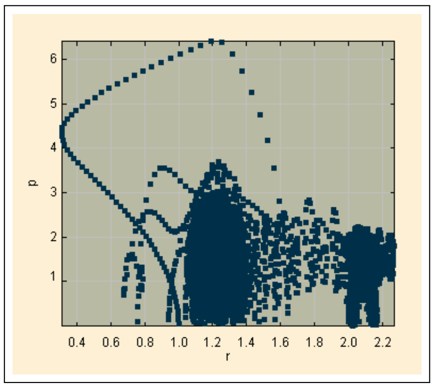
Figure 4: Phase plots of node 0, no damping



(a)

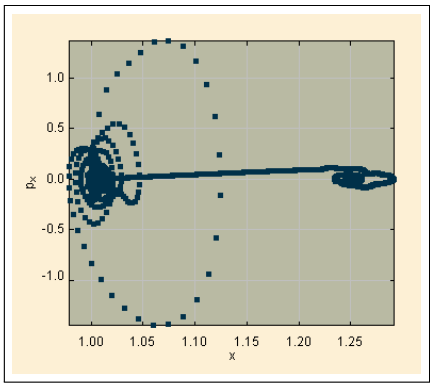


(b)

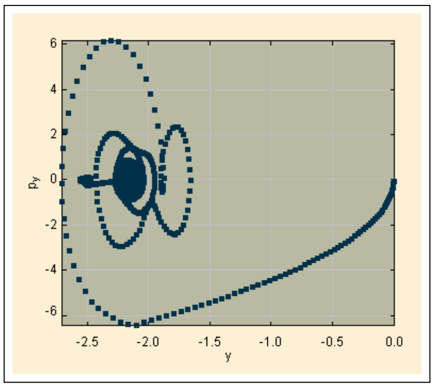


(c)

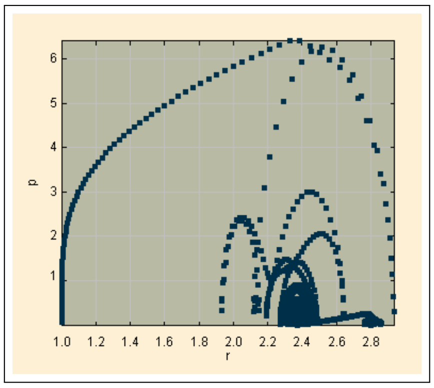
Figure 5: Phase plots of node 3, no damping



(a)

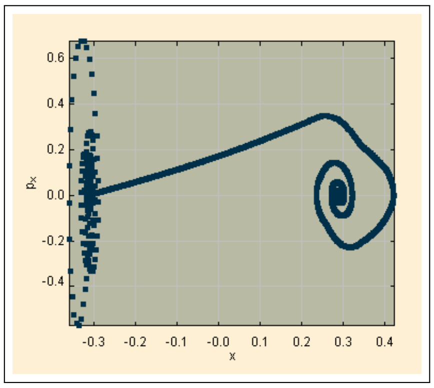


(b)

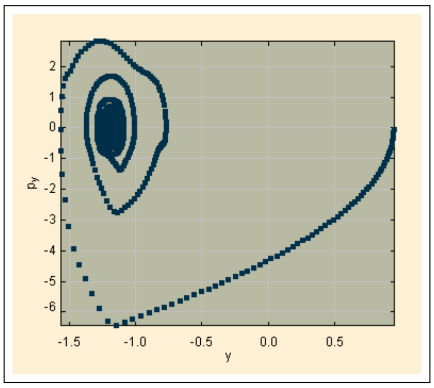


(c)

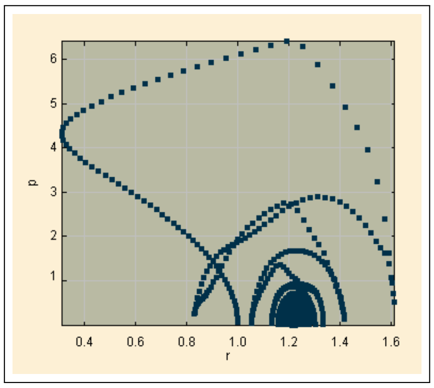
Figure 6: Phase plots of node 0, $c = 0.5$, $c_2 = 0.9$



(a)



(b)



(c)

Figure 7: Phase plots of node 3, $c = 0.5$, $c_2 = 0.9$

Figrue 6 shows the phase plots for node 0 (the right most surface node). All the plots seems to be show stability. The sperad for momentum in x seems to be larger, as compared to the y component. This is also true for node 3 is the node that makes contact with the ground. From the phase plots in figure 6 and 5, it can also be seen that the plots are attractors which is due to the fact that the collision with ground was completely inelastic causing energy loss in the soft ball.

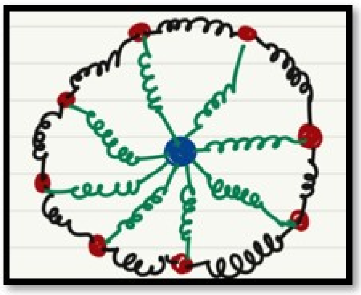
The phase plots in figures 6 and 7 converge to a point quicker as compared to no spring damping case. This is because in addition to inelastic collision with ground, the energy of the sofy ball is being disspated by damping of the springs of the soft body.

4 Conclusions

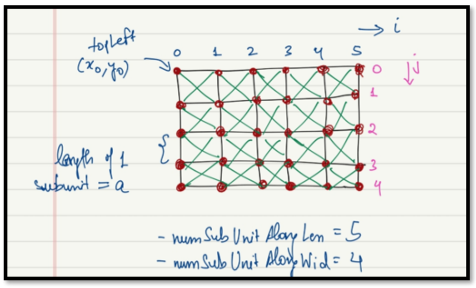
- A simple two dimensional soft body simulation using spring-mass arrangement was implemented.
- Learned and applied concepts of Object oriented programming to write reusable and scalable code.
- Analysed the compute time, stability and energy flow of the soft body simulation.

4.1 Future Goals

- Implement different configurations of springs mass nodes that the one used in this project (in ¡figure¿, for example) and study their dyanmics.
- Study collision of soft body with rigid bodies and other soft bodies by analyzing the time of collision.
- Three dimensional rigid body and rigid body rotation.
- Application to cell dynamics modelling.
- Finite Element Analysis method for implement soft body.



(a) Soft Ball with a center node



(b) Rectangular Soft Body

Figure 8: Other soft body configurations

5 Code Listings

5.1 Utils Package

5.1.1 vec2.java

```java
package utils;

public class vec2 {
public double x;
public double y;
public vec2(){
    this.x = 0;
    this.y = 0;
}
public vec2(double X, double Y){
    this.x = X;
    this.y = Y;
}
public vec2 add(vec2 other){
    return new vec2(this.x+other.x, this.y+other.y);
}
public void addSelf(vec2 other){
    this.x += other.x;
    this.y += other.y;
}
public vec2 addAll(vec2... others) {
    for (vec2 other : others) {
        this.x += other.x;
        this.y += other.y;
    }
    return this;
}
public vec2 sub(vec2 other){
    return new vec2(this.x-other.x, this.y-other.y);
}
public void subSelf(vec2 other){
    this.x -= other.x;
    this.y -= other.y;
}
public double dot(vec2 other){
    return (this.x*other.x)+(this.y*other.y);
}
public double mag(){
    return Math.sqrt(this.x*this.x + this.y*this.y);
}
public static double distanceFormula(vec2 a, vec2 b){
    return Math.sqrt(Math.pow((a.x-b.x),2)+Math.pow((a.y-b.y),2));
}
public vec2 scalerMult(double scaler){
        vec2 c = new vec2();
        c.x = scaler*this.x;
        c.y = scaler*this.y;
        return c;

    }
    public static vec2 unitVec2(vec2 a, vec2 b){
        vec2 c = new vec2();
        c.x = b.x - a.x;
        c.y = b.y - a.y;
        double mag = Math.sqrt(c.x*c.x+c.y*c.y);
        if (mag!=0){
            c.x /= mag;
            c.y /= mag;
        }
        return c;


    }
    public void assign(vec2 other){
        this.x = other.x;
        this.y = other.y;
    }
    public void assign(double other_x, double other_y){
        this.x = other_x;
        this.y = other_y;
    }
    public vec2 calcNormalToLineJoining(vec2 p1, vec2 p2){
        double lenp1p2 = distanceFormula(p1, p2);
        vec2 Np1P2 = new vec2(-(p2.y-p1.y),(p2.x-p1.x));
        return Np1P2.scalerMult(lenp1p2);
    }
    public static double areaOfTriangleBetweenThreePoints(vec2 p1, vec2 p2, vec2 p3){
        return 0.5*(p1.x*(p2.y-p3.y)+p2.x*(p3.y-p1.y)+p3.x*(p1.y-p2.y));
    }
}
```

5.1.2 AbstractODESolver2D.java

```
    package utils;


    public abstract class AbstractODESolver2D implements ODESolver2D{
    protected double stepSize = 0.1;
    protected int numEqn = 0;
    protected ODE2D ode;

    public AbstractODESolver2D(ODE2D var1) {
        this.ode = var1;
        this.initialize(0.1);
    }

    public abstract double step();

    public void setStepSize(double dt) {
        this.stepSize = dt;
    }

    public void initialize(double dt) {
        this.stepSize = dt;
        vec2[] stateVec = this.ode.getStateVec();
        if (stateVec == null) {
            this.numEqn = 0;
        } else {
            this.numEqn = stateVec.length;
        }

    }

    public double getStepSize() {
        return this.stepSize;
    }

    }
```

5.1.3 ODE2D

```
    package utils;

    public interface ODE2D {
        vec2[] getStateVec();
        void getRateVec(vec2[] state, vec2[] rate);

    }
```

5.1.4 ODESolver2D

```
    package utils;
    public interface ODESolver2D {
    void initialize(double stepSize);
    double step();
    void setStepSize(double stepSize);
    double getStepSize();
}
```

5.1.5 Verlet2D

```
    package utils;
    public class Verlet2D extends AbstractODESolver2D {
```

```java
    private vec2[] rate1; // acceleration at current position
    private vec2[] rate2; // acceleration at updated position
    private int rateCounter = -1;
    public Verlet2D(ODE2D ode) {
        super(ode);
    }
    @Override
    public void initialize(double stepSize) {
        super.initialize(stepSize);
        int N = (ode.getStateVec().length-1) / 2;
        int totalVecs = 2*N+1;
        rate1 = new vec2[totalVecs];
        rate2 = new vec2[totalVecs];
        for (int i = 0; i < totalVecs; i++) {
            rate1[i] = new vec2();
            rate2[i] = new vec2();
        }
        this.rateCounter = -1;
    }
    @Override
    public double step() {
        vec2[] state = ode.getStateVec();
        int N = (state.length - 1) / 2;
        rateCounter = 0;
        ode.getRateVec(state, rate1);
        double h = stepSize;
        double h2 = 0.5 * h * h;
        for (int i = 0; i < N; i++) {
            state[2*i].x += h * rate1[2*i].x + h2 * rate1[2*i + 1].x;
            state[2*i].y += h * rate1[2*i].y + h2 * rate1[2*i + 1].y;
        }
        rateCounter = 1;
        ode.getRateVec(state, rate2);
        rateCounter = 2;
        for (int i = 0; i < N; i++) {
            state[2*i + 1].x += 0.5 * h * (rate1[2*i + 1].x + rate2[2*i + 1].x);
            state[2*i + 1].y += 0.5 * h * (rate1[2*i + 1].y + rate2[2*i + 1].y);
        }
        int timeIndex = 2 * N;
        state[timeIndex].x += h;
        state[timeIndex].y += h;
        return stepSize;
    }
    public int getRateCounter() {
        return rateCounter;
    }
}
```

5.2 SoftBall package

5.2.1 softBall2

```
package SoftBall;
import java.util.ArrayList;
import org.opensourcephysics.display.Drawable;
import org.opensourcephysics.display.DrawingPanel;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.RenderingHints;
import java.awt.Color;
import utils.ODE2D;
import utils.vec2;
import utils.Verlet2D;
import utils.colorPalletes;

public class softBall2 implements Drawable, ODE2D{
public double x, y;
vec2 center = new vec2(x,y);
vec2 initVel;
public int n; //number of beads and springs that make the surface of the ball
public double R; //radius of the soft ball
ArrayList<vec2> massList = new ArrayList<>();
public double k; // spring constant for springs connecting the surface nodes
public double k2 = 25.0; // spring constant for"springs" connecting surface nodes
    and the center of the bead
public double cSurface;
public double cInner;
public double[][] allSpringsEquilibriumLen;

double m = 1; //"mass" of the surface nodes
vec2 currMassCoordinate;

double A0 = 0;//initial (equilibrium) area of the soft body
public double A;
double kArea = 50.0;//spring constant for area restoring force
double dampingFactor;
int pixRadius = 3;

double dt;
vec2 g = new vec2(0,-9.8);
vec2[] stateVec;
vec2[] acclrnVec;
Verlet2D odeSolver;

boolean displayForceVectors = false;



colorPalletes pallate = new colorPalletes();
ArrayList<Color> metro = pallate.Metro_Colors;
```

```
ArrayList<Color> azure = pallate.Color_Azure;
ArrayList<Color> kanagawa = pallate.Kanagawa;
ArrayList<Color> dataPlot1 = pallate.dataPlot1;

public softBall2(vec2 centerCoordinateVec2, int numBeads, double springConstBlue,
   double springConstGreen, double springConstArea, double ballRadius, double
   airDragFactor, double surfaceSpringDampingConst, double
   centerToSurfaceSpringDampingConst, boolean displayForceVectors){
   this.center = centerCoordinateVec2;
   this.n = numBeads;
   this.k = springConstBlue;
   this.k2 = springConstGreen;
   this.kArea = springConstArea;
   this.R = ballRadius;
   this.dampingFactor = airDragFactor;
   this.cSurface = surfaceSpringDampingConst;
   this.cInner = centerToSurfaceSpringDampingConst;
   this.displayForceVectors = displayForceVectors;

   initializeSoftBall();
   odeSolver = new Verlet2D(this);

}
public void initializeSoftBall(){
   this.stateVec = new vec2[2*(this.n)+1];//n position vectors for the surface
      nodes as well as n velocity vectors and 1 for the center of mass
   this.acclrnVec = new vec2[this.n];
   this.initializeStateVecObjectsWithZeros();
   this.initializeAcclrnVecObjectsWithZeros();
   for(int i=0; i<this.n;i++){
      currMassCoordinate = new vec2(
         (R*Math.cos((i)*2*Math.PI/n)), //x coordinate of the ith node from center
         (R*Math.sin((i)*2*Math.PI/n)) //y coordinate of the ith node from the
            center
         );
       massList.add(currMassCoordinate);
       this.stateVec[2*i].assign(currMassCoordinate);
   }


   // why was the second element n-1 before?
   this.allSpringsEquilibriumLen = new double[this.n][this.n];

   double[] xCoordArr = new double[this.n], yCoordArr = new double[this.n];
   for(int i=0; i<this.n;i++){
       xCoordArr[i] = this.stateVec[2*i].x;
       yCoordArr[i] = this.stateVec[2*i].y;
   }
   this.A0 = polygonAreaFromVerticesCoordinates(xCoordArr, yCoordArr, this.n);

   for(int i = 0; i < n; i++) {
       for(int j = 0; j < n; j++) {
```

```
                // the j-th neighbor of i on the circle has angular separation
                   (j+1)*2*pi/n
                double ang = ((j - i) * 2.0 * Math.PI / n);
                allSpringsEquilibriumLen[i][j] = Math.abs( R * 2.0 * Math.sin(ang / 2.0));
            }
        }


        this.odeSolver = new Verlet2D(this);
    }
    public void setStepSize(double dt){
        odeSolver.setStepSize(dt);
    }
    public void step(){
        odeSolver.step();
    }
    public vec2[] getStateVec(){
        return this.stateVec;
    }
    public vec2[] getAcclrnVec(){
        return this.acclrnVec;
    }
    public double calcCurrSpringLen(vec2 springNodeCoordinate1, vec2
       springNodeCoordinate2){
        vec2 p1 = springNodeCoordinate1;
        vec2 p2 = springNodeCoordinate2;

        return vec2.distanceFormula(p1, p2);
    }
    private vec2 calcUnitVecAlongSpring(vec2 springNodeCoordinate1, vec2
       springNodeCoordinate2){
        /**
         * coord1 -> coord2
         */
        vec2 p1 = springNodeCoordinate1;
        vec2 p2 = springNodeCoordinate2;

        return vec2.unitVec2(p1, p2);
    }
    private vec2 calcSpringForceVector(double eqbmSpringLen, double currSpringLen,
       double k, vec2 unitVecAlongSpring, vec2 vel_i,vec2 vel_j, double
       springDampingFactor){
        double delL = (currSpringLen-eqbmSpringLen); //compression in the spring
        //return unitVecAlongSpring.scalerMult(delL);
        //not sure about the signs yet, will try to both positive and negative

        if(Math.abs(delL)<10E-6){
            return new vec2(0,0);
        }
        else{
            double relativeVelocityMag = (vel_i.x - vel_j.x) * unitVecAlongSpring.x +
               (vel_i.y - vel_j.y) * unitVecAlongSpring.y;
            double dampingTerm = springDampingFactor * relativeVelocityMag;
            return unitVecAlongSpring.scalerMult(k * delL - dampingTerm);
        }
    }



    private double polygonAreaFromVerticesCoordinates(double[] xCoordArr, double[]
        yCoordArr, int numPoints) //https://www.mathopenref.com/coordpolygonarea2.html
        {
        double area = 0; // Accumulates area
        int j = numPoints-1;

        for (int i=0; i<numPoints; i++){
            area += (xCoordArr[j]+xCoordArr[i]) * (yCoordArr[j]-yCoordArr[i]);
            j = i; //j is previous vertex to i
        }
        return area/2;
        }



    private vec2 calcAreaConsvForceOnThisNode(int nodeIndex){
        int nNext = (nodeIndex == this.n-1 ? 0 : nodeIndex + 1);
        int nPrev = (nodeIndex == 0 ? this.n-1 : nodeIndex - 1);

        vec2 fArea = new vec2(this.stateVec[2*(nNext)].y-this.stateVec[2*(nPrev)].y,
            -(this.stateVec[2*(nNext)].x-this.stateVec[2*(nPrev)].x));

        return fArea.scalerMult((this.A-this.A0)*this.kArea/(2*this.A0));
    }


    private void initializeStateVecObjectsWithZeros(){
        for(int i=0;i<stateVec.length;i++){
            this.stateVec[i] = new vec2(0,0);
        }
    }
    private void initializeAcclrnVecObjectsWithZeros(){
        for(int i=0;i<acclrnVec.length;i++){
            this.acclrnVec[i] = new vec2(0,0);
        }
    }


    public void applyGroundCollision(double groundY, double bounceDamping, double
        friction) {
    for (int i = 0; i < this.n; i++) {
        vec2 pos = stateVec[2*i];
        vec2 vel = stateVec[2*i+1];
        if (pos.y < groundY) {
            pos.y = groundY;
            if (vel.y < 0) vel.y *= -bounceDamping;
            vel.x *= friction; // friction in x direction to stop nodes from sliding in
                the horizontal direction due to the surface force
            }
        }
    }

    public double getcInner(){
        return this.cInner;
    }
    public double getcSurface(){
        return this.cSurface;
    }


    public void computeAcceleration(){

        this.initializeAcclrnVecObjectsWithZeros();

        double[] xCoordArr = new double[this.n], yCoordArr = new double[this.n];
        for(int i=0; i<this.n;i++){
            xCoordArr[i] = this.stateVec[2*i].x;
            yCoordArr[i] = this.stateVec[2*i].y;
        }
        this.A = polygonAreaFromVerticesCoordinates(xCoordArr, yCoordArr, this.n);

        for(int i=0; i< this.n; i++){

            vec2 fiSpring = new vec2(0,0);


            int j = (i+1) % this.n; //next immediate neighbor
            int l = (i+this.n-1)%this.n;//prev immediate neighbor


            double springLenNext = calcCurrSpringLen(this.stateVec[2*i],
                this.stateVec[2*j]);
            double springLenPrev = calcCurrSpringLen(this.stateVec[2*i],
                this.stateVec[2*l]);


            vec2 nHatNext = calcUnitVecAlongSpring(this.stateVec[2*i],
                this.stateVec[2*j]);
            vec2 nHatPrev = calcUnitVecAlongSpring(this.stateVec[2*i],
                this.stateVec[2*l]);
            this.cInner = 0.0;
            this.cSurface = 0.0;
            fiSpring.addSelf(calcSpringForceVector(this.allSpringsEquilibriumLen[i][j],
                springLenNext, this.k, nHatNext, this.stateVec[2*i+1],
                this.stateVec[2*l+1], this.cSurface));
            fiSpring.addSelf(calcSpringForceVector(this.allSpringsEquilibriumLen[i][l],
                springLenPrev, this.k, nHatPrev, this.stateVec[2*i+1],
                this.stateVec[2*j+1], this.cSurface));


            for (int loopVar = 0; loopVar < this.n; loopVar++) {
                if (loopVar == j || loopVar == l || loopVar == i) continue; // skip
                   immediate neighbors

                double l_eqbm = this.allSpringsEquilibriumLen[i][loopVar];
                double l_curr = calcCurrSpringLen(this.stateVec[2*i],
                   this.stateVec[2*loopVar]);


                vec2 nHat = calcUnitVecAlongSpring(this.stateVec[2*i],
                   this.stateVec[2*loopVar]);
                vec2 vel_i = this.stateVec[2*i + 1];
                vec2 vel_j = this.stateVec[2*loopVar + 1];

                vec2 fij = calcSpringForceVector(l_eqbm, l_curr, this.k, nHat, vel_i,
                   vel_j, this.cInner);
                fiSpring.addSelf(fij);
            }
            fiSpring.addSelf(this.g);
            this.acclrnVec[i].assign(fiSpring.scalerMult(1/this.m));

            this.acclrnVec[i].addSelf(this.calcAreaConsvForceOnThisNode(i));
        }
    }
    public void getRateVec(vec2[] stateVec, vec2[] rateVec){
        if(odeSolver.getRateCounter()==1){
            this.computeAcceleration();
        }

        for(int i=0;i<this.n;i++){
            rateVec[2*i].assign(this.stateVec[2*i+1]);
            rateVec[2*i+1].assign(this.acclrnVec[i]);
        }
        rateVec[this.stateVec.length-1].assign(1.0,1.0);
    }
    public void draw(DrawingPanel panel, Graphics g){
        Graphics2D g2 = (Graphics2D) g;
        g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
        String timestep = "timestep=" +
            String.valueOf(this.stateVec[this.stateVec.length-1].x);
        g2.setColor(dataPlot1.get(3));
        g2.drawString(timestep, 20,10);
        g2.setColor(Color.BLACK);

        int rectWidth = panel.getWidth()-panel.xToPix(0.1);
```

```
        int rectHeight = panel.getHeight() / 50;
        g2.fillRect(panel.xToPix(-3), panel.yToPix(-3), rectWidth, rectHeight);

        //drawing surface nodes and connecting springs
        for(int i=0; i<this.n; i++){
            int xipix = panel.xToPix(this.stateVec[2*i].x);
            int yipix = panel.yToPix(this.stateVec[2*i].y);
            for(int j=i+1;j<this.n;j++){
                int next = (i+1)%this.n;
                int prev = (i-1+this.n)%this.n;
                if(j!= next && j != prev){ //j>i so that one spring between a given node
                    pair is drawn only once
                    int xjpix = panel.xToPix(this.stateVec[2*j].x);
                    int yjpix = panel.yToPix(this.stateVec[2*j].y);

                    g2.setColor(dataPlot1.get(7));
                    g2.drawLine(xipix, yipix, xjpix, yjpix);

                }
        }

            int k =(i+1)%this.n;
            int xkpix = panel.xToPix(this.stateVec[2*k].x);
            int ykpix = panel.yToPix(this.stateVec[2*k].y);

            g2.setColor(dataPlot1.get(6));
            g2.drawLine(xipix, yipix, xkpix, ykpix);

            g2.setColor(dataPlot1.get(3));
            g2.fillOval(xipix-pixRadius, yipix-pixRadius, 2*pixRadius, 2*pixRadius);

            if(displayForceVectors){
                int xPix = panel.xToPix(this.stateVec[2*i].x);
                int yPix = panel.yToPix(this.stateVec[2*i].y);

                double fx = this.acclrnVec[i].x;
                double fy = this.acclrnVec[i].y;

                double scale = Math.sqrt(fx*fx+fy*fy)*0.01; //scale length with magnitude
                int fxPix = panel.xToPix(this.stateVec[2*i].x + fx * scale);
                int fyPix = panel.yToPix(this.stateVec[2*i].y + fy * scale);
                g2.setColor(kanagawa.get(1));
                g2.drawLine(xPix, yPix, fxPix, fyPix);
            }
        }
    }
}
```

5.2.2 softBall2App

```
    package SoftBall;
```

```java
import utils.vec2;
import utils.colorPalletes;
import java.awt.Color;
import java.util.ArrayList;


import org.opensourcephysics.controls.AbstractSimulation;
import org.opensourcephysics.controls.SimulationControl;
import org.opensourcephysics.display.DrawingFrame;
import org.opensourcephysics.display.DrawingPanel;




public class softBall2App extends AbstractSimulation{
    vec2 ballCenter = new vec2();

    softBall2 ball2;

    DrawingFrame frame = new DrawingFrame();
    DrawingPanel animPanel = frame.getDrawingPanel();
    colorPalletes pallete = new colorPalletes();
    ArrayList<Color> kanagawa = colorPalletes.Kanagawa;
    ArrayList<Color> plotColors = colorPalletes.dataPlot1;

    public void initialize(){
    ball2 = null;
    frame.clearDrawables();
    frame.clearData();
    this.delayTime = 0;
    double dt = control.getDouble("dt");
    double x = control.getDouble("initial x");
    double y = control.getDouble("initial y");
    int n = control.getInt("Number of surface nodes");
    double k = control.getDouble("surface springs' stiffness");
    double k2 = control.getDouble("inner springs' stiffness");
    double kArea = control.getDouble("area consv spring constant");
    double R = control.getDouble("Radius");
    double b = control.getDouble("velocity dependent drag factor");
    double c = control.getDouble("surface spring damping constant");
    double c2 = control.getDouble("inner springs' stiffness");
    boolean dispFVecs = control.getBoolean("(bool) display force vectors");

    ballCenter.x = x;
    ballCenter.y = y;
    ball2 = new softBall2(ballCenter, n, k, k2, kArea, R, b, c, c2, dispFVecs);
    ball2.setStepSize(dt);

    frame.setPreferredMinMax(-5, 5, -5, 5);
    frame.setInteriorBackground(plotColors.get(1));
    frame.setSize(400,400);
```

```
        frame.setSquareAspect(true);
        frame.setVisible(true);
        frame.repaint();
        frame.addDrawable(ball2);
        }
        public void doStep(){
            ball2.applyGroundCollision(-3, 0.0, 0.0);
            ball2.step();
            frame.repaint();
        }
        public void reset(){
            control.setValue("dt",0.01);
            control.setValue("initial x", 0);
            control.setValue("initial y", 0);
            control.setValue("Number of surface nodes", 10);
            control.setValue("surface springs' stiffness", 100);
            control.setValue("Radius", 1);
            control.setValue("inner springs' stiffness",100);
            control.setValue("area consv spring constant",2.0);
            control.setValue("velocity dependent drag factor",0.2);
            control.setValue("surface spring damping constant",0.0);
            control.setValue("center to surface spring damping constant",0.0);
            control.setValue("(bool) display force vectors",false);
        }
        public static void main(String[] args){
            SimulationControl.createApp(new softBall2App());
        }
    }
```

5.3 analysisScripts Package

5.3.1 computeTimeNumNodesApp

```
package analysisScripts;

import SoftBall.softBall2;

import utils.vec2;
import utils.colorPalletes;

import org.opensourcephysics.frames.PlotFrame;
import org.opensourcephysics.controls.AbstractCalculation;
import org.opensourcephysics.controls.CalculationControl;
import java.awt.Color;
import java.util.ArrayList;

/**
* Computes the time taken to simulate fixed number of steps for softBall2 with
    varying number of surface nodes.
*/
public class computeTimeNumNodesApp extends AbstractCalculation {
```

```
PlotFrame computeTimePlot = new PlotFrame("Number of Surface Nodes (n)", "Compute
   Time (ms)", "Compute Time vs n");

public void calculate() {
   computeTimePlot.clearData();

   double dt = 0.01;
   int steps = 100;

   // simulation parameters
   vec2 ballCenter = new vec2(0, 0);
   double R = 1.0;
   double k = 100.0;
   double k2 = 100.0;
   double kArea = 2.0;
   double b = 1.0;
   double c = 0.5;
   double c2 = 0.9;

   colorPalletes pallate = new colorPalletes();
   ArrayList<Color> plotColors = pallate.dataPlot1;

   for (int n = 10; n <= 1000; n += 10) {
      softBall2 ball = new softBall2(ballCenter, n, k, k2, kArea, R, b, c, c2,
         false);
      ball.setStepSize(dt);

      long startTime = System.nanoTime();
      for (int i = 0; i < steps; i++) {
         ball.applyGroundCollision(-3, 0.5, 0.0);
         ball.step();
      }
      long endTime = System.nanoTime();
      double elapsedMs = (endTime - startTime) / 1e6;

      computeTimePlot.append(0, n, elapsedMs);
      System.out.printf("n = %d\tTime = %.3f ms%n", n, elapsedMs);
   }
   computeTimePlot.setInteriorBackground(plotColors.get(1));
   computeTimePlot.setBackground(plotColors.get(5));
   computeTimePlot.setSize(500,500);
   computeTimePlot.setVisible(true);
   computeTimePlot.setMarkerSize(0, 2);
   computeTimePlot.setConnected(true);
   computeTimePlot.setLineColor(0, plotColors.get(7));
   computeTimePlot.setMarkerColor(0, plotColors.get(6));
   control.calculationDone("Benchmark complete.");
}

public static void main(String[] args) {
   CalculationControl.createApp(new computeTimeNumNodesApp());
```

```
    }
}
```

5.3.2 phasePlotsApp

```
package analysisScripts;

import java.awt.Color;
import java.util.ArrayList;


import org.opensourcephysics.controls.AbstractCalculation;
import org.opensourcephysics.controls.CalculationControl;
import org.opensourcephysics.frames.PlotFrame;

import SoftBall.softBall2;
import utils.colorPalletes;
import utils.vec2;

public class phasePlotsApp extends AbstractCalculation{
PlotFrame xPhaseSpace = new PlotFrame("x", "p_x", "x phase plot");
PlotFrame yPhaseSpace = new PlotFrame("y", "p_y", "y phase plot");
PlotFrame xyPhaseSpace = new PlotFrame("r", "p", "phase plot");
colorPalletes pallate = new colorPalletes();
ArrayList<Color> plotColors = pallate.dataPlot1;



public void calculate() {
    xPhaseSpace.clearData();
    yPhaseSpace.clearData();
    xyPhaseSpace.clearData();

    xPhaseSpace.setSize(400,400);
    xPhaseSpace.setInteriorBackground(plotColors.get(1));
    xPhaseSpace.setBackground(plotColors.get(5));
    xPhaseSpace.setMarkerColor(1, plotColors.get(6)); // Set the color of the plot
    xPhaseSpace.setVisible(true);

    yPhaseSpace.setSize(400,400);
    yPhaseSpace.setInteriorBackground(plotColors.get(1));
    yPhaseSpace.setBackground(plotColors.get(5));
    yPhaseSpace.setMarkerColor(1, plotColors.get(6)); // Set the color of the plot
    yPhaseSpace.setVisible(true);

    xyPhaseSpace.setSize(400,400);
    xyPhaseSpace.setInteriorBackground(plotColors.get(1));
    xyPhaseSpace.setBackground(plotColors.get(5));
    xyPhaseSpace.setMarkerColor(1, plotColors.get(6)); // Set the color of the plot
    xyPhaseSpace.setVisible(true);
```

```java
        double dt = 0.01;
        double steps = 10000;
        // simulation parameters
        vec2 ballCenter = new vec2(0, 0);
        int n = 10;
        double R = 1.0;
        double k = 100.0;
        double k2 = 100.0;
        double kArea = 2.0;
        double b = 1.0;
        double c = 0.5;
        double c2 = 0.9;

        softBall2 ball = new softBall2(ballCenter, n, k, k2, kArea, R, b, c, c2, false);
        ball.setStepSize(dt);


        double r,p, x, vx, y, vy;
        int nodeIndex = 3;

        for (int i = 0; i<steps; i++){
            ball.applyGroundCollision(-3, 0.5, 0.5);
            ball.step();
            x = ball.getStateVec()[2*nodeIndex].x;
            vx = ball.getStateVec()[2*nodeIndex+1].x;

            y = ball.getStateVec()[2*nodeIndex].y;
            vy = ball.getStateVec()[2*nodeIndex+1].y;

            r =
                Math.sqrt(Math.pow((ball.getStateVec()[2*nodeIndex].x),2)+Math.pow((ball.getStateV
            p =
                Math.sqrt(Math.pow((ball.getStateVec()[2*nodeIndex+1].x),2)+Math.pow((ball.getState


            xPhaseSpace.append(1, x,vx);
            yPhaseSpace.append(1, y,vy);
            xyPhaseSpace.append(1, r,p);
        }

    }
    public static void main(String[] args){
        CalculationControl.createApp(new areaAndPhasePlotsApp());
    }
}
```