MODEL ROCKET SIMULATION

WITH DRAG ANALYSIS

by

Brent Cannon

Submitted to the Department of Physics and Astronomy in partial fulfillment

of graduation requirements for the degree of

Bachelor of Science

Brigham Young University

April 2004

Abstract

When designing and constructing a model rocket, forehand knowledge of the performance is extremely helpful. With such knowledge, measures can be taken to ensure the rocket is recovered safely and has an optimal flight. I have created a software tool that allows the user to simulate their rocket design before physically building it. Written in Java, this simulator makes use of the fourth-order Runge-Kutta method as well as cubic spline interpolation. A method of estimating drag coefficients is also implemented.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1. Introduction

1.1 Background

Since its introduction in 1957, model rocketry has grown to become a widely practiced hobby. Rocketry is a family friendly activity. It is safe and is a great way for children to gain an understanding of science from an early age. Since most rocketeers do not have an in-depth understanding of calculus and physics, I have created a software tool that allows anybody to predict the flight path of their rocket. This software is easy to use, and it is informative enough for the user to increase their understanding of rocket mechanics.

1.2 Motivation

Three characteristics of a rocket's performance are important to know before it is constructed. These three characteristics are the maximum velocity, maximum altitude, and the time when that altitude will be reached. Knowing these values and constructing the rocket accordingly can increase the probability of a safe flight.

First, we will consider the maximum velocity. When an object moves through a viscous medium, drag forces oppose the object's movement. These drag forces are proportional to the velocity squared. Typical velocities at burnout fall within the range of 30 to 200 meters-per-second (m/s). The drag force at 200 m/s is almost 45 times the force at 30 m/s! Clearly, fast-flying rockets require sturdy construction techniques to withstand such forces, while slow rockets can be made of lightweight materials and with minimum reinforcements.

The next factor is the maximum altitude. Most rockets use a recovery method that increases the drag force as the rocket falls toward the ground. The most common recovery device is a parachute. As the rocket coasts down, it can be carried by the wind. As the maximum altitude increases, a larger area may be required to safely recover it.

Finally, we must consider the time at which the maximum altitude will be reached. One must be careful to time the ejection of the parachute correctly. If the parachute is deployed while the rocket is traveling quickly, the rocket will be jolted to a stop and can be damaged. Ideally, the parachute should be deployed at apogee, when the vertical velocity of the rocket is zero. When timed in this manner, the parachute will deploy gently and the rocket will coast down safely. The timing can be adjusted by selecting a motor with an appropriate delay.

1.3 Description of Simulator

I wrote my rocket simulator in the Java programming language. The code was compiled with the Java 2 Standard Edition SDK version 1.4.2. I utilized the Swing technology to build an applet that can run in any java-enabled wed browser. I have tested the program with Internet Explorer and Mozilla.

The applet can run in two modes, design and launch. In design mode, the user can enter parameters for the rocket, as well as select the desired motor. In launch mode, the flight path is animated and significant values are displayed.

1.4 Limitations

I made several assumptions to simplify the computations. During the rocket flight, I assume that the angle of attack is zero. This is somewhat unrealistic because air movement not parallel to the rocket's trajectory causes the rocket to pitch about its center of gravity. However, if the rocket has a stable design, the restoring force from induced drag will return the rocket to zero angle of attack rapidly [1]. Angle of attack is a factor in determining the lift drag coefficient [2]. Because of uncertainty in real-world conditions, I will assume it to be zero.

Figure 1. The angle between the vertical axis and the rocket's trajectory is the angle of attack.

I also assumed that the propellant is consumed at a constant rate. This translates to a linear decrease in mass during the boost phase. While this might not be completely accurate, there is no published data for the instantaneous propellant mass of these motors.

Finally, I did not account for the effects of supersonic flight. Although many high-power model rockets are capable of supersonic flight, with velocities exceeding 350 meters per second, this simulation only addresses subsonic flight using motors readily available to the public.

Chapter 2. Methods

2.1 Flight Mechanics

There are three phases in a rocket's flight: boost, coast, and recovery. During the boost phase, the rocket is influenced by thrust from the motor, gravity, and drag. During the coast and recovery phase, there is no thrust, so only gravity and drag are relevant. Any time the forces on an object are unbalanced, the object is accelerated. We can therefore see the net acceleration on the rocket by adding the forces in each direction.

$$a = \frac{F_{thrust} + F_{drag} + F_{gravity}}{m} = \frac{T - \frac{1}{2} C_D \rho A v |v|}{m} - g$$

We now know the acceleration. In the equation above, T represents the thrust from the rocket propellant, C_D is the drag coefficient, ρ is air density, v is velocity, A is the cross-sectional area, m is mass, and g is the acceleration due to gravity, which for this computation, we'll assume to be constant with altitude [3]. The flight path is computed with a fourth-order Runge-Kutta numerical integration routine. First, however, we need to determine some of the values that will be used in the computation.

2.2 Thrust Computation

Most model rocket motor manufacturers publish thrust curves for their motors. These curves show the instantaneous thrust during the boost period. Integrating the area under the curve gives the impulse of the motor. The manufacturer of the motors I used for this simulation, provide figures displaying the curve for each motor, but less than two-dozen points along the curve (the number varied for each motor). During the altitude computation, five hundred points per second are used, which requires a grid up to fifty

times finer than the grid provided in the motor documentation. To fill in the missing data, I used a cubic spline interpolation method. This method gives a third order polynomial that describes the curve between any two adjacent points [4]. The polynomials are then used to create a thrust curve much finer than the one published.

2.3 Drag Analysis

The best way to determine the drag coefficient of an object is to collect experimental data in a controlled environment, such as a wind tunnel or water channel. Since most rocketeers do not have these devices at their disposal to evaluate their designs, I used a set of empirical equations to estimate the coefficient of drag.

Several drag prediction methodologies can be used in the design of rockets. They all determine the total drag by adding the expected drag of individual rocket components. They also use empirical formulas to estimate the drag coefficients. Of the methodologies I considered, all but one were created with missile designers in mind.

The method I used was designed specifically for model rockets, and is based on data collected using model rockets. I chose to implement this method because it accounts for several characteristics unique to model rockets. Model rockets commonly have a three, four, or five fin configuration. The missile drag predictions were based on fins in multiples of two. Model rockets generally have a launch lug that guides the rocket along a rail as the rocket is taking off. Launch lugs are not commonly used on missiles or large rockets, so the missile methods did not discuss them. Finally, the model rocket drag method made suggestions for finding a skin-friction coefficient that is realistic for typical model rocket finishes and speeds.

The total drag coefficient of a rocket is estimated by finding the drag coefficients of the individual components, adding the coefficients, and adjusting for interference between the components [5]. For this analysis, the total drag is the sum of the drag coefficients of the nose cone, body tube, base, fins, launch hardware, and the interference. The equations for calculating the drag coefficients of each component are found in Table 1.

Component	Value	Symbols
$C_{D_{Total}}$	$C_{D_N} + C_{D_{BT}} + C_{D_B} + C_{D_F} + C_{D_L} + C_{D_I}$	A_{BT} = cross-sectional area of body tube, m^2 A_{LL} = cross-sectional area of launch Lug, m^2 C_{Ave} = average fin chord, m C_R = root fin chord, m C_{D_B} = base drag coefficient, dimensionless $C_{D_{BT}}$ = body tube drag coefficient, dimensionless C_{D_F} = fin drag coefficient, dimensionless C_{D_I} = interference drag coefficient, dimensionless C_{D_L} = launch lug drag coefficient, dimensionless C_{D_N} = nose drag coefficient, dimensionless C_F = skin-friction coefficient, dimensionless S_{BN} = wetted surface area of body and nose, m^2 S_F = wetted surface area of fins, m^2 S_{LL} = wetted surface area of launch lug, m^2 d = body diameter, m L = length of body and nose, m n = number of fins, dimensionless t = fin thickness, m v = velocity, m/s Δy = boundary layer thickness, m μ = viscosity, m^2/s ρ = air density, kg/m^3
$C_{D_N} + C_{D_{BT}}$	$1.02\, C_F \left(1 + \frac{1.5}{(L/d)^{3/2}}\right) \frac{S_{BT}}{A_{BT}}$	
C_{D_B}	$\frac{0.029}{\sqrt{C_{D_N} + C_{D_{BT}}}}$	
C_{D_F}	$2\, C_F \left(1 + 2\frac{t}{c_{Ave}}\right) \frac{S_F}{A_{BT}}$	
C_{D_L}	$\frac{1.2\, A_{LL} + 0.0045\, S_{LL}}{S_{BT}}$	
C_{D_I}	$C_F \left(1 + 2\frac{t}{c_{Ave}}\right) \frac{c_R}{S_{BT}} d\, n$	
C_F	$\frac{2\mu}{\rho v \Delta y}$	

Table 1. Equations for estimating drag coefficients [5,6,7].

Wetted surface area refers to the area of the portion of a component that comes in contact with air. Portions of surfaces that are joined to other components should not be included. For example, the edge of a fin that comes in contact with the body tube should be subtracted from the total surface of the fin.

When two components are joined together, the total drag of the system of components is not simply the sum of the two original coefficients. The flow of fluid around the new configuration is different. This change is called interference drag. This change can be positive or negative, depending on the configuration used [8].

Since it is based on the surface finish of the rocket, C_F is not something the users of my program will be able to measure. Instead, I used a typical value, which is 75% of the way between the C_F of a fully laminar and fully turbulent flow [7].

2.4 Altitude Computation

Now that the values for the computation are known, the problem can be solved using a fourth-order Runge-Kutta method modified to accommodate a system of equations. The equations the simulator solves are:

$$\dot{x} = v$$

$$\ddot{x} = a = \frac{T - \frac{1}{2} C_D \rho A v |v|}{m} + g$$

$$ic: x_0 = 0, v_0 = 0$$

The Runge-Kutta method solves the equations for values of position and velocity. Using this iterative method, the position x_{i+1} at any time t_{i+1} is a function of the position x_i at the previous time step t_i. Applying these values to the previous equation gives the acceleration at the same time step [9].

After calculating the altitudes, I used the maximum altitude to determine how long the rocket would take to touch down. I computed this time with the Runge-Kutta algorithm, using the maximum altitude as the beginning altitude. Thrust was held at zero and I used the parachute area and coefficient of drag [8].

2.5 Graphical User Interface

When I created the user interface, I did not use a visual layout editor. The entire program was coded manually. Although this might not have been the fastest way to create the software, I feel that it gave me the most control over the layout and over the background processes.

To accept input, I used a combination of text fields, buttons, and choice menus. For each field and button, I created a listener based on the ActionListener object in the Java standard class library. Each listener was programmed to update the appropriate variables and call supporting methods.

Output is given through the use of text fields and graphics. The text fields were simple to update, but the graphics were challenging at times. The Java graphics engine uses a handful of methods that allow the user to display basic shapes, lines, and arcs. To place some graphical elements in the correct positions, I defined a pixel, (baseX, baseY), in the middle the object to be drawn. Using this point as a reference, I was able to calculate the relative location of each element.

To display the large altitude graph on the Launch screen, I had to modify the altitude array. The graph is 320 pixels tall, so I multiplied each altitude point by $\frac{320}{altitude_{\max}}$. Being only 720 pixels wide, the graph cannot accommodate every point in the altitude

array. I therefore selected 720 evenly spaced points to plot. Finally, since computer pixels use an integer mapping, I cast the altitude values as integers. The output is red during the burn stage and yellow during the coast stage.

To create the dial indicators on the Launch screen, I located a point at the center of each circle. The line representing the needle began at that point and ran to the point $\left(xCenter + L\sin\left(\frac{2\pi}{10} value\right), yCenter - L\cos\left(\frac{2\pi}{10} value\right)\right)$. L is the length of the needle, and *value* is the value of an individual digit in the number to be displayed.

Chapter 3. Conclusions

3.1 Future Research Ideas

The next step that I am going to take in the development of this simulator will be to test it against real world data. Although expensive, recording altimeters are available. These altimeters sample the altitude periodically throughout the flight. The user can then export the values to a computer file. I would like to collect such data, plot both the calculated and collected data, and determine how close my calculations are to a real flight.

A weak link in the drag computation is the coefficient of friction calculation. Since it is based on average velocities and expected boundary layers, it doesn't account for the actual finish of the rocket. Modifying the calculation to incorporate the surface imperfection size of typical finishes and then calculating the coefficient of friction for several velocities might produce more acceptable results.

Earlier, I discussed the angle of attack. I chose not to include this variable in my computation because of the uncertainty of real-world conditions. A future improvement could involve implementing a wind simulator to allow for different angles of attack. A related improvement would be to calculate the stability of the rocket. Stability can be calculated using the Barrowman Equations to compare the spatial relation between the center of gravity and the center of pressure.

3.2 Conclusions

In testing the simulator, I have found that it returns an unrealistic drag coefficient when the user enters extreme dimensions. This is easy to see by entering a fin chord value of 400m. The user is responsible for giving realistic input. In general, L/d ratios between 6 and 15 worked well.

The simulator I have created is a good introduction to the flight dynamics of a model rocket. Due to limitations in the drag calculation and the lack of a dynamic environment, is more useful as a learning tool than as an engineering tool. Although it can assist in the evaluation of designs, there is no substitute for gathering experimental data.

References

[1] G. H. Stine, *Handbook of Model Rocketry*, 6th ed. (Wiley, New York, 1994).

[2] J. M. Simon and W. B. Blake, "Missile Datcom: High angle of attack capabilities," AIAA-99-4258, p. 3.

[3] A. Miele, *Flight Mechanics, Volume 1, Theory of Flight Paths*, (Addison-Wesley, Reading, MA, 1962), p. 403.

[4] R. L. Burden and J. D. Faires, *Numerical Analysis*, 6th ed. (Brooks/Cole, Pacific Grove, CA, 1997), pp.143-158.

[5] G. Merrill, *Principles of Guided Missile Design*, (D. Van Nostrand Company, Inc, Princeton, NJ, 1958), p. 215 .

[6] G. M. Gregorek, *Aerodynamic Drag of Model Rockets*, (Estes Industries, Penrose, CO, 1970), pp. 2-51.

[7] J. Moran, *An Introduction to Theoretical and Computational Aerodynamics*, (Wiley, New York, 1984), pp. 198-199.

[8] S. F. Hoerner, *Fluid-Dynamic Drag: Practical Information on Aerodynamic Drag and Hydrodynamic Resistance*, (Hoerner, Midland Park, NJ, 1965), pp. 2.4-3.28, 13.23-13.24.

[9] W. Cheney and D. Kincaid, *Numerical Mathematics and Computing*, (Brooks/Cole, Pacific Grove, CA, 1999), pp. 387-388.

[10] J. N. Nielsen, *Missile Aerodynamics*, (McGraw-Hill, New York, Toronto, London, 1960), pp. 261-341.

[11] J. Lewis and W. Loftus, *Java Software Solutions, Foundations of Program Design*, 2nd ed. (Addison-Wesley, Reading, MA, 2000).

Appendixes

Appendix A. User Guide

Introduction

RocketSimulator is a Java applet that enables you to experiment with the parameters of a model rocket and view the results. The most important features are its ability to calculate the coefficient of drag and plot the flight path.

Running RocketSimulator

To run the program, the following files must be placed in the same directory:

rocketSimulator.class
rocketSimulator$animationControl.class
rocketSimulator$bhTextFieldListener.class
rocketSimulator$ButtonEventListener.class
rocketSimulator$bwTextFieldListener.class
rocketSimulator$cdTextFieldListener.class
rocketSimulator$chdTextFieldListener.class
rocketSimulator$fhTextFieldListener.class
rocketSimulator$fwTextFieldListener.class
rocketSimulator$GenericWindowListener.class
rocketSimulator$llTextFieldListener.class
rocketSimulator$mTextFieldListener.class
rocketSimulator$nhTextFieldListener.class
rocketSimulator$scaleTextFieldListener.class
rocketSimulator.html

Open rocketSimulator.html in a java-enabled web browser. If the program does not run, Sun's Java runtime environment needs to be installed. J2RE v.1.4.2.04 for Windows is included on the RocketSimulator CD. Other versions can be obtained at www.sun.com.

Program Operation

RocketSimulator runs in two modes, Design and Launch. The default mode is Design. To change from one mode to another, simply click the Design or Launch buttons.

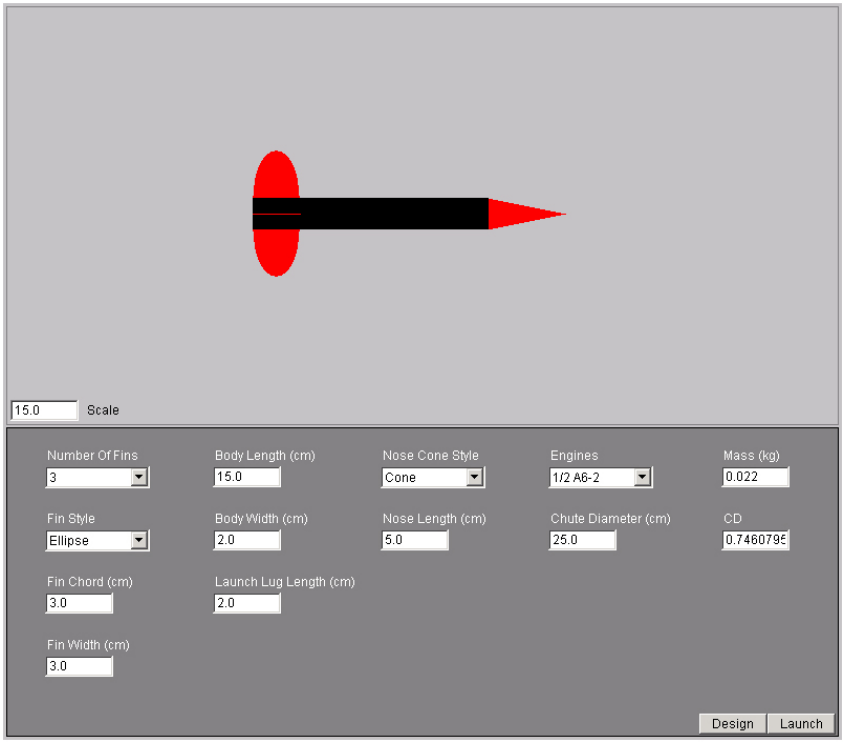

Figure 2. Design mode screen

In Design mode, the user can adjust the size and shape of the rocket shape. To modify the value of a text field, place the cursor in that field, type a new value, and press enter. **The value will not be updated if you do not press enter.** All of the text fields are editable. The CD (coefficient of drag) field also functions as an output. As values are changed, the drag coefficient is automatically updated. If you would like to use a different drag coefficient than the one that is calculated, you should enter that value after all other fields are filled in (if you modify a field after entering the drag coefficient, the CD field will be updated and your desired drag coefficient will be discarded). The scale

field can be adjusted if necessary to change the size of the rocket image. It will not, however, modify the values in the bottom portion of the screen.

Drop down menus allow you to choose from a set of predetermined inputs. To change the value, simply click on the arrow and scroll down to the value you want.

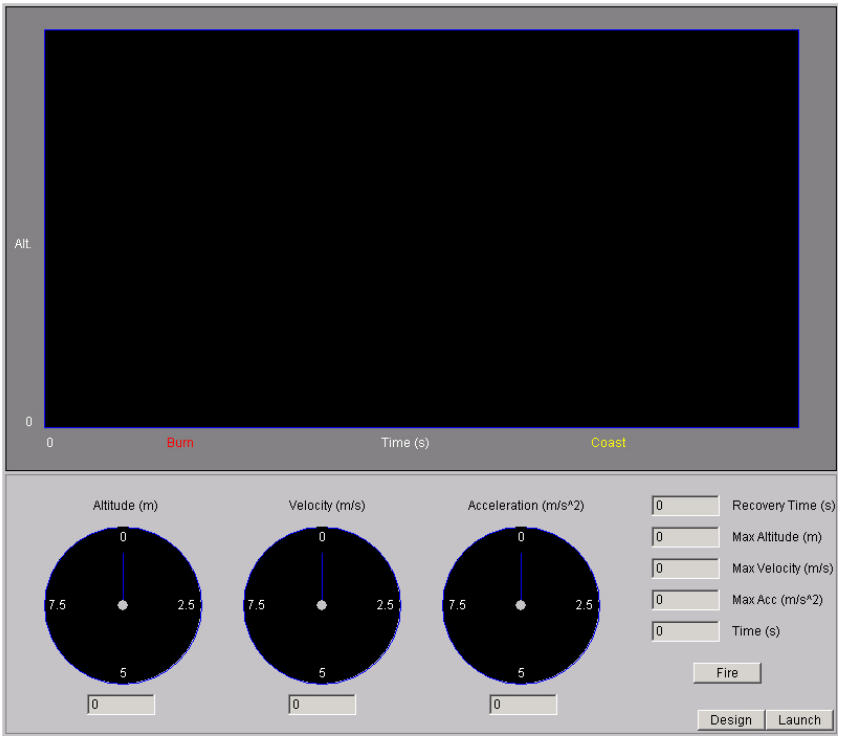

Figure 3. Launch mode initial screen

You can test your design in Launch mode. To begin the simulation, press Fire. The altitude will begin to be plotted on the graph (see Figure 4). The line will be red during the burn phase of the rocket's flight, and yellow during the coast phase. The maximum altitude is displayed on the vertical axis, and the parachute deployment time is on the horizontal axis.

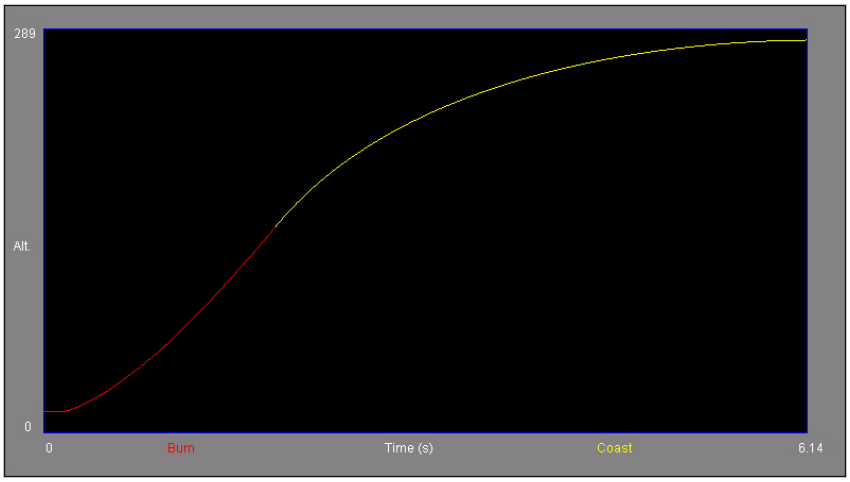

Figure 4. Launch mode graph

As the graph is being plotted, the indicators in the lower portion of the screen are updated (see Figure 5). The dial indicators are read in the same way that an analog watch is read. Instead of reading using seconds, minutes, and hours, the dials indicate portions of powers of ten. Using the altitude indicator on Figure 5 as an example, the value that is represented is approximately 289.7 *m*. The yellow needle points to 2.897, the green needle points to 8.97, and the blue needle points to 9.7.

The Recovery Time field represents the amount of time in seconds that the rocket will take to touch down after the parachute is deployed. Max Altitude, Max Velocity, and Max Acc display the current maximum values reached for their respective fields. The Time field indicates the current time during the animation.

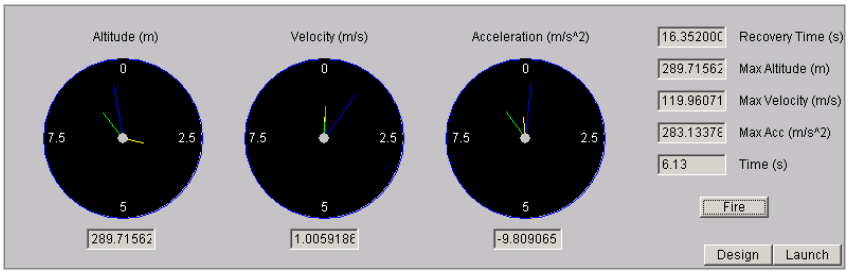

Figure 5. Launch mode indicators

To exit the program, simply close the browser window. No data will be saved for future sessions.

Appendix B. Source Code

```
 1: import java.applet.Applet;
 2: import java.awt.*;
 3: import java.awt.event.*;
 4: import javax.swing.Timer;
 5: import javax.swing.*;
 6:
 7: public class rocketSimulator extends java.applet.Applet implements MouseListener, MouseMotio
    nListener
 8: {
 9:   double bodyWidthVal = 2; // cm
10:   double bodyHeightVal = 15; // cm
11:   double launchLugLengthVal = 2; // cm
12:   double noseHeightVal = 5; // cm
13:   double finWidthVal = 3; // cm
14:   double finChordVal = 3; // cm
15:   int numFinVal = 3;
16:   double finArea = 2*0.785*finChordVal*finWidthVal/10000;
17:   int finType = 0;  // ellipse
18:
19:   int noseType = 0; // cone
20:   int mode = 1;       // design
21:   double pi = 3.145926;
22:
23:   int baseX = 350; // center X point positioning rocket
24:   int baseY = 200; // center Y point positioning rocket
25:
26:   double scale = 15;
27:   int drawBodyHeight = (int)(scale*bodyHeightVal);
28:   int drawBodyWidth = (int)(scale*bodyWidthVal);
29:   int drawNoseHeight = (int)(scale*noseHeightVal);
30:   int drawFinChord = (int)(scale*finChordVal);
31:   int drawFinWidth = (int)(scale*finWidthVal);
32:
33:   int animationIndex = 0;
34:
35:   // VALUES FOR RUNGE KUTTA COMPUTATION
36:   double rho = 1.29; // kg/m^3
37:   double g = 9.8; // m/s^2
38:   double cd = 0.4;
39:   double chuteDrag = 1;
40:   double chuteDiameterVal = 25; // cm
41:   double chuteArea = pi*chuteDiameterVal*chuteDiameterVal/10000; // m^2
42:   double A = (bodyWidthVal/200)*(bodyWidthVal/200)*pi + numFinVal*finWidthVal/100*0.003; //
    m^2
43:   double mRocket = 0.022; // kg
44:   double mCasing = 0.0099; // kg
45:   double mFuel = 0.0026; // kg
46:   double burnTime = 0.33; // s
47:   double coastTime = 1.74; // s
48:   double timeUp = burnTime+coastTime; // s
49:   double recoveryTimeVal = 0; // s
50:   double apogee = 0; // m
51:   double pointsPerSecond = 500;
52:
53:   //POSITION ARRAYS
54:   int numPoints = (int)(timeUp*pointsPerSecond);
55:   int burnPoints = (int)(burnTime*pointsPerSecond);
56:   double[] T = new double[numPoints];
57:   double[] X = new double[numPoints];
58:   double[] V = new double[numPoints];
59:   double[] Acc = new double[numPoints];
60:
61:   //INITIAL THRUST VALUES
62:   double[] thrustTime = {0,0.031,0.064,0.096,0.124,0.149,0.172,0.196,0.210,0.225,0.235,0.244
    ,0.254,0.263,0.269,0.279,0.290,0.297,0.306,0.314,0.330};
63:   double[] thrustPoints = {0,0.404,1.258,2.263,3.467,4.720,6.023,7.027,7.528,7.860,7.482,6.6
    83,5.685,4.487,4.087,3.039,1.790,1.042,0.593,0.344,0.000};
64:   double[] thrustCurve = new double[numPoints];
65:
66:   //PLOTTING ARRAYS
```

```
67:   int delay = 50; // timer delay
68:   boolean[] graph = new boolean[720];
69:   double[] gX = new double[720]; // position
70:   double[] gV = new double[720]; // velocity
71:   double[] gA = new double[720]; // acceleration
72:   int[] gXInt = new int[720];     // position values to plot
73:   double[] gT = new double[720];// plotTime
74:
75:   //VALUES FOR DRAWING GUAGES
76:   double v1x = 0;
77:   double v10x = 0;
78:   double v100x = 0;
79:   double x1x = 0;
80:   double x10x = 0;
81:   double x100x = 0;
82:   double x1000x = 0;
83:   double a1x = 0;
84:   double a10x = 0;
85:   double a100x = 0;
86:
87:   Choice engines = new Choice();
88:   Choice bodyStyle = new Choice();
89:   Choice noseStyle = new Choice();
90:   Choice numFin = new Choice();
91:   Choice finStyle = new Choice();
92:
93:   Button design = new Button("Design");
94:   Button launch = new Button("Launch");
95:   Button fire = new Button("Fire");
96:
97:   Label engineLabel = new Label ("Engines");
98:   Label bodyLabel = new Label ("Body Styles");
99:   Label noseLabel = new Label ("Nose Cone Styles");
100:  Label noseConeLabel = new Label ("Nose Cone Style");
101:  Label bodyWidthLabel = new Label ("Body Width (cm)");
102:  Label bodyHeightLabel = new Label ("Body Length (cm)");
103:  Label launchLugLengthLabel = new Label ("Launch Lug Length (cm)");
104:  Label noseHeightLabel = new Label ("Nose Length (cm)");
105:  Label noseWidthLabel = new Label ("Nose Width (cm)");
106:  Label finWidthLabel = new Label ("Fin Width (cm)");
107:  Label finChordLabel = new Label ("Fin Chord (cm)");
108:  Label chuteDiameterLabel = new Label ("Chute Diameter (cm)");
109:  Label numFinLabel = new Label ("Number Of Fins");
110:  Label finStyleLabel = new Label ("Fin Style");
111:  Label scaleLabel = new Label ("Scale");
112:  Label massLabel = new Label ("Mass (kg)");
113:  Label cdLabel = new Label ("CD");
114:  Label altitudeLabel = new Label ("Altitude (m)");
115:  Label maxAltitudeLabel = new Label ("Max Altitude (m)");
116:  Label velocityLabel = new Label ("Velocity (m/s)");
117:  Label maxVelocityLabel = new Label ("Max Velocity (m/s)");
118:  Label accelerationLabel = new Label ("Acceleration (m/s^2)");
119:  Label maxAccelerationLabel = new Label ("Max Acc (m/s^2)");
120:  Label timeLabel = new Label ("Time (s)");
121:  Label recoveryTimeLabel = new Label ("Recovery Time (s)");
122:  Label gL1 = new Label ("0");
123:  Label gL2 = new Label ("0");
124:  Label gL3 = new Label ("");
125:  Label gL4 = new Label ("");
126:  Label gL5 = new Label ("Alt.");
127:  Label gL6 = new Label ("Time (s)");
128:  Label gL7 = new Label ("Burn");
129:  Label gL8 = new Label ("Coast");
130:  Label dL1 = new Label ("0");
131:  Label dL2 = new Label ("2.5");
132:  Label dL3 = new Label ("5");
133:  Label dL4 = new Label ("7.5");
134:  Label dL5 = new Label ("0");
135:  Label dL6 = new Label ("2.5");
136:  Label dL7 = new Label ("5");
```

```
137:   Label dL8 = new Label ("7.5");
138:   Label dL9 = new Label ("0");
139:   Label dL10 = new Label ("2.5");
140:   Label dL11 = new Label ("5");
141:   Label dL12 = new Label ("7.5");
142:
143:
144:   TextField bodyWidth = new TextField (Double.toString(bodyWidthVal), 5);
145:   TextField bodyHeight = new TextField (Double.toString(bodyHeightVal), 5);
146:   TextField launchLugLength = new TextField (Double.toString(launchLugLengthVal), 5);
147:   TextField noseHeight = new TextField (Double.toString(noseHeightVal), 5);
148:   TextField finWidth = new TextField (Double.toString(finWidthVal), 5);
149:   TextField chuteDiameter = new TextField (Double.toString(chuteDiameterVal), 5);
150:   TextField finChord = new TextField (Double.toString(finChordVal), 5);
151:   TextField scaleField = new TextField (Double.toString(scale), 5);
152:   TextField mass = new TextField (Double.toString(mRocket), 5);
153:   TextField cdField = new TextField (Double.toString(cd), 5);
154:   TextField altitude = new TextField ("0", 7);
155:   TextField maxAltitude = new TextField ("0", 7);
156:   TextField velocity = new TextField ("0", 7);
157:   TextField maxVelocity = new TextField ("0", 7);
158:   TextField acceleration = new TextField ("0", 7);
159:   TextField maxAcceleration = new TextField ("0", 7);
160:   TextField time = new TextField ("0",7);
161:   TextField recoveryTime = new TextField ("0",7);
162:
163:
164:   Timer timer;
165:
166:   public void init()
167:   {
168:     // DEFINE GUI LAYOUT //
169:     setLayout(null);
170:
171:     // DESIGN ELEMENTS //
172:     massLabel.setBounds(685,420,65,20);
173:     massLabel.setBackground(Color.gray);
174:     massLabel.setForeground(Color.white);
175:     mass.setBounds(685,440,65,20);
176:
177:     cdLabel.setBounds(685,480,23,20);
178:     cdLabel.setBackground(Color.gray);
179:     cdLabel.setForeground(Color.white);
180:     cdField.setBounds(685,500,65,20);
181:
182:     numFinLabel.setBounds(40,420,95,20);
183:     numFinLabel.setBackground(Color.gray);
184:     numFinLabel.setForeground(Color.white);
185:     numFin.setBounds(40,440,100,20);
186:
187:     finStyleLabel.setBounds(40,480,95,20);
188:     finStyleLabel.setBackground(Color.gray);
189:     finStyleLabel.setForeground(Color.white);
190:     finStyle.setBounds(40,500,100,20);
191:
192:     finChordLabel.setBounds(40,540,85,20);
193:     finChordLabel.setBackground(Color.gray);
194:     finChordLabel.setForeground(Color.white);
195:     finChord.setBounds(40,560,65,20);
196:
197:     finWidthLabel.setBounds(40,600,85,20);
198:     finWidthLabel.setBackground(Color.gray);
199:     finWidthLabel.setForeground(Color.white);
200:     finWidth.setBounds(40,620,65,20);
201:
202:     bodyHeightLabel.setBounds(200,420,110,20);
203:     bodyHeightLabel.setBackground(Color.gray);
204:     bodyHeightLabel.setForeground(Color.white);
205:     bodyHeight.setBounds(200,440,65,20);
206:
```

```
207:      bodyWidthLabel.setBounds(200,480,95,20);
208:      bodyWidthLabel.setBackground(Color.gray);
209:      bodyWidthLabel.setForeground(Color.white);
210:      bodyWidth.setBounds(200,500,65,20);
211:
212:      launchLugLengthLabel.setBounds(200,540,145,20);
213:      launchLugLengthLabel.setBackground(Color.gray);
214:      launchLugLengthLabel.setForeground(Color.white);
215:      launchLugLength.setBounds(200,560,65,20);
216:
217:      noseConeLabel.setBounds(360,420,100,20);
218:      noseConeLabel.setBackground(Color.gray);
219:      noseConeLabel.setForeground(Color.white);
220:      noseStyle.setBounds(360,440,100,20);
221:
222:      noseHeightLabel.setBounds(360,480,100,20);
223:      noseHeightLabel.setBackground(Color.gray);
224:      noseHeightLabel.setForeground(Color.white);
225:      noseHeight.setBounds(360,500,65,20);
226:
227:      engineLabel.setBounds(520,420,50,20);
228:      engineLabel.setBackground(Color.gray);
229:      engineLabel.setForeground(Color.white);
230:      engines.setBounds(520,440,100,20);
231:
232:      chuteDiameterLabel.setBounds(520,480,120,20);
233:      chuteDiameterLabel.setBackground(Color.gray);
234:      chuteDiameterLabel.setForeground(Color.white);
235:      chuteDiameter.setBounds(520,500,65,20);
236:
237:      scaleLabel.setBounds (78,377,36,20);
238:      scaleLabel.setBackground(Color.lightGray);
239:      scaleField.setBounds (7,377,65,20);
240:
241:      // LAUNCH ELEMENTS //
242:      altitudeLabel.setBounds (85,470,75,20);
243:      altitudeLabel.setBackground(Color.lightGray);
244:      altitude.setBounds (81,660,65,20);
245:      altitude.setEditable(false);
246:
247:      velocityLabel.setBounds (272,470,75,20);
248:      velocityLabel.setBackground(Color.lightGray);
249:      velocity.setBounds (273,660,65,20);
250:      velocity.setEditable(false);
251:
252:      accelerationLabel.setBounds (443,470,120,20);
253:      accelerationLabel.setBackground(Color.lightGray);
254:      acceleration.setBounds (465,660,65,20);
255:      acceleration.setEditable(false);
256:
257:      maxAltitudeLabel.setBounds (695,500,100,20);
258:      maxAltitudeLabel.setBackground(Color.lightGray);
259:      maxAltitude.setBounds (620,500,65,20);
260:      maxAltitude.setEditable(false);
261:
262:      maxVelocityLabel.setBounds (695,530,100,20);
263:      maxVelocityLabel.setBackground(Color.lightGray);
264:      maxVelocity.setBounds (620,530,65,20);
265:      maxVelocity.setEditable(false);
266:
267:      maxAccelerationLabel.setBounds (695,560,100,20);
268:      maxAccelerationLabel.setBackground(Color.lightGray);
269:      maxAcceleration.setBounds (620,560,65,20);
270:      maxAcceleration.setEditable(false);
271:
272:      timeLabel.setBounds (695,590,90,20);
273:      timeLabel.setBackground(Color.lightGray);
274:      time.setBounds (620,590,65,20);
275:      time.setEditable(false);
276:
```

```
277:     recoveryTimeLabel.setBounds (695,470,100,20);
278:     recoveryTimeLabel.setBackground(Color.lightGray);
279:     recoveryTime.setBounds (620,470,65,20);
280:     recoveryTime.setEditable(false);
281:
282:     gL1.setBounds (20,390,20,20);
283:     gL1.setBackground(Color.gray);
284:     gL1.setForeground(Color.white);
285:     gL2.setBounds (40,410,20,20);
286:     gL2.setBackground(Color.gray);
287:     gL2.setForeground(Color.white);
288:     gL3.setBounds (10,20,30,20);
289:     gL3.setBackground(Color.gray);
290:     gL3.setForeground(Color.white);
291:     gL4.setBounds (750,410,30,20);
292:     gL4.setBackground(Color.gray);
293:     gL4.setForeground(Color.white);
294:     gL5.setBounds (10,220,30,20);
295:     gL5.setBackground(Color.gray);
296:     gL5.setForeground(Color.white);
297:     gL6.setBounds (360,410,80,20);
298:     gL6.setBackground(Color.gray);
299:     gL6.setForeground(Color.white);
300:     gL7.setBounds (155,410,40,20);
301:     gL7.setBackground(Color.gray);
302:     gL7.setForeground(Color.red);
303:     gL8.setBounds (560,410,40,20);
304:     gL8.setBackground(Color.gray);
305:     gL8.setForeground(Color.yellow);
306:     dL1.setBounds (110,500,10,20);
307:     dL1.setBackground(Color.black);
308:     dL1.setForeground(Color.white);
309:     dL2.setBounds (165,565,20,20);
310:     dL2.setBackground(Color.black);
311:     dL2.setForeground(Color.white);
312:     dL3.setBounds (110,630,10,20);
313:     dL3.setBackground(Color.black);
314:     dL3.setForeground(Color.white);
315:     dL4.setBounds (42,565,20,20);
316:     dL4.setBackground(Color.black);
317:     dL4.setForeground(Color.white);
318:     dL5.setBounds (300,500,10,20);
319:     dL5.setBackground(Color.black);
320:     dL5.setForeground(Color.white);
321:     dL6.setBounds (355,565,20,20);
322:     dL6.setBackground(Color.black);
323:     dL6.setForeground(Color.white);
324:     dL7.setBounds (300,630,10,20);
325:     dL7.setBackground(Color.black);
326:     dL7.setForeground(Color.white);
327:     dL8.setBounds (232,565,20,20);
328:     dL8.setBackground(Color.black);
329:     dL8.setForeground(Color.white);
330:     dL9.setBounds (490,500,10,20);
331:     dL9.setBackground(Color.black);
332:     dL9.setForeground(Color.white);
333:     dL10.setBounds (545,565,20,20);
334:     dL10.setBackground(Color.black);
335:     dL10.setForeground(Color.white);
336:     dL11.setBounds (490,630,10,20);
337:     dL11.setBackground(Color.black);
338:     dL11.setForeground(Color.white);
339:     dL12.setBounds (424,565,20,20);
340:     dL12.setBackground(Color.black);
341:     dL12.setForeground(Color.white);
342:
343:     fire.setBounds (660,630,65,20);
344:
345:     // COMMON ELEMENTS //
346:     launch.setBounds (729,675,65,20);
```

```
347:     design.setBounds (664,675,65,20);
348:
349:     // INITIALIZE GUI IN DESIGN MODE //
350:     add(numFin);
351:     add(finStyle);
352:     add(finChord);
353:     add(finWidth);
354:     add(bodyHeight);
355:     add(bodyWidth);
356:     add(launchLugLength);
357:     add(noseStyle);
358:     add(noseHeight);
359:     add(engines);
360:     add(chuteDiameter);
361:     add(bodyHeightLabel);
362:     add(bodyWidthLabel);
363:     add(launchLugLengthLabel);
364:     add(noseConeLabel);
365:     add(noseHeightLabel);
366:     add(numFinLabel);
367:     add(finStyleLabel);
368:     add(finChordLabel);
369:     add(finWidthLabel);
370:     add(engineLabel);
371:     add(chuteDiameterLabel);
372:     add(mass);
373:     add(cdField);
374:     add(massLabel);
375:     add(cdLabel);
376:
377:     noseStyle.add("Cone");
378:     noseStyle.add("Ogive");
379:     noseStyle.add("Round");
380:     numFin.add("3");
381:     numFin.add("4");
382:     numFin.add("5");
383:     finStyle.add("Ellipse");
384:     finStyle.add("Rectangular");
385:     finStyle.add("Tapered");
386:     finStyle.add("Swept");
387:     engines.add("1/2 A6-2");
388:     engines.add("A8-3");
389:     engines.add("A8-5");
390:     engines.add("B4-2");
391:     engines.add("B4-4");
392:     engines.add("B4-6");
393:     engines.add("B6-2");
394:     engines.add("B6-4");
395:     engines.add("B6-6");
396:     engines.add("C5-3");
397:     engines.add("C6-3");
398:     engines.add("C6-5");
399:     engines.add("C6-7");
400:     add(design);
401:     add(launch);
402:     add(scaleLabel);
403:     add(scaleField);
404:
405:     // BUTTON EVENT LISTENERS //
406:     ButtonEventListener buttonListener = new ButtonEventListener();
407:     design.addActionListener(buttonListener);
408:     launch.addActionListener(buttonListener);
409:     fire.addActionListener(buttonListener);
410:
411:     // TEXT FIELD EVENT LISTENERS //
412:     bwTextFieldListener bwListener = new bwTextFieldListener();
413:     bodyWidth.addActionListener(bwListener);
414:     bhTextFieldListener bhListener = new bhTextFieldListener();
415:     bodyHeight.addActionListener(bhListener);
416:     llTextFieldListener llListener = new llTextFieldListener();
```

```
417:     launchLugLength.addActionListener(llListener);
418:     nhTextFieldListener nhListener = new nhTextFieldListener();
419:     noseHeight.addActionListener(nhListener);
420:     fhTextFieldListener fhListener = new fhTextFieldListener();
421:     finChord.addActionListener(fhListener);
422:     fwTextFieldListener fwListener = new fwTextFieldListener();
423:     finWidth.addActionListener(fwListener);
424:     chdTextFieldListener chdListener = new chdTextFieldListener();
425:     chuteDiameter.addActionListener(chdListener);
426:     cdTextFieldListener cdListener = new cdTextFieldListener();
427:     cdField.addActionListener(cdListener);
428:     mTextFieldListener mListener = new mTextFieldListener();
429:     mass.addActionListener(mListener);
430:     scaleTextFieldListener scaleListener = new scaleTextFieldListener();
431:     scaleField.addActionListener(scaleListener);
432:
433:     // MOUSE EVENT LISTENERS //
434:     addMouseListener(this);
435:     addMouseMotionListener(this);
436:
437:     // ANIMATION CONTROLS //
438:     timer = new Timer(delay, new animationControl());
439:
440:     computeCD();
441:   }// End init
442:
443:   public void paint(Graphics g)
444:   {
445:     if (mode == 1) // DESIGN //
446:     {
447:       g.setColor(Color.lightGray);
448:       g.fillRect(3,3,793,397);
449:
450:       g.setColor(Color.gray);
451:       g.drawRect(3,3,793,397);
452:
453:       g.setColor(Color.gray);
454:       g.fillRect(3,403,793,293);
455:
456:       g.setColor(Color.black);
457:       g.drawRect(3,403,793,293);
458:
459:       // DRAW NOSE CONE //
460:       g.setColor (Color.red);
461:       if (noseType == 0) // CONE //
462:       {
463:         int leftPt = baseY-drawBodyWidth/2;
464:         int rightPt = baseY+drawBodyWidth/2;
465:         int centerPt = baseY;
466:         int topPt = baseX+drawBodyHeight/2+drawNoseHeight;
467:         int bottomPt = baseX+drawBodyHeight/2;
468:
469:         int[] ypoints = {leftPt, rightPt, centerPt};
470:         int[] xpoints = {bottomPt, bottomPt, topPt};
471:         g.fillPolygon(xpoints,ypoints,3);
472:       }// End if
473:       else
474:       {
475:         if (noseType == 1) // OGIVE //
476:         {
477:           g.fillOval(baseX+drawBodyHeight/2-drawNoseHeight,baseY-drawBodyWidth/2,drawNoseHei
    ght*2,drawBodyWidth);
478:         }// End if
479:         else
480:         {
481:           if (noseType == 2) // ROUND //
482:           {
483:             g.fillOval(baseX+(drawBodyHeight-drawBodyWidth)/2,baseY-drawBodyWidth/2,drawBody
    Width,drawBodyWidth);
484:           }// End if
```

```
485:         }//end else
486:       }//end else
487:
488:       // DRAW FINS //
489:       if (finType == 0) // ELLIPTICAL //
490:       {
491:         g.fillOval(baseX-drawBodyHeight/2,baseY-drawBodyWidth/2-drawFinWidth,drawFinChord,2*
     drawFinWidth);
492:         g.fillOval(baseX-drawBodyHeight/2,baseY+drawBodyWidth/2-drawFinWidth,drawFinChord,2*
     drawFinWidth);
493:         g.drawLine(baseX-drawBodyHeight/2,baseY,baseX-drawBodyHeight/2+drawFinChord,baseY);
494:       }//end if
495:       else
496:       {
497:         if (finType == 1) // RECTANGULAR //
498:         {
499:           g.fillRect(baseX-drawBodyHeight/2,baseY-drawBodyWidth/2-drawFinWidth,drawFinChord,
     2*drawFinWidth);
500:           g.fillRect(baseX-drawBodyHeight/2,baseY+drawBodyWidth/2-drawFinWidth,drawFinChord,
     2*drawFinWidth);
501:           g.drawLine(baseX-drawBodyHeight/2,baseY,baseX-drawBodyHeight/2+drawFinChord,baseY)
     ;
502:         }//end if
503:         else
504:         {
505:           if (finType == 2) // TAPERED //
506:           {
507:             int[] leftFinXPoints = {baseX-drawBodyHeight/2, baseX-drawBodyHeight/2+drawFinCh
     ord, baseX-drawBodyHeight/2+(3*drawFinChord/4), baseX-drawBodyHeight/2+drawFinChord/4};
508:             int[] leftFinYPoints = {baseY-drawBodyWidth/2, baseY-drawBodyWidth/2,baseY-drawB
     odyWidth/2-drawFinWidth, baseY-drawBodyWidth/2-drawFinWidth};
509:             g.fillPolygon(leftFinXPoints,leftFinYPoints,4);
510:             int[] rightFinXPoints = {baseX-drawBodyHeight/2, baseX-drawBodyHeight/2+drawFinC
     hord, baseX-drawBodyHeight/2+(3*drawFinChord/4), baseX-drawBodyHeight/2+drawFinChord/4};
511:             int[] rightFinYPoints = {baseY+drawBodyWidth/2, baseY+drawBodyWidth/2,baseY+draw
     BodyWidth/2+drawFinWidth,baseY+drawBodyWidth/2+drawFinWidth};
512:             g.fillPolygon(rightFinXPoints,rightFinYPoints,4);
513:             g.drawLine(baseX-drawBodyHeight/2,baseY,baseX-drawBodyHeight/2+drawFinChord,base
     Y);
514:           }//end if
515:           else
516:           {
517:             if (finType == 3) // SWEPT //
518:             {
519:               int[] leftFinXPoints = {baseX-drawBodyHeight/2, baseX-drawBodyHeight/2+drawFin
     Chord, baseX-drawBodyHeight/2, baseX-drawBodyHeight/2-drawFinChord};
520:               int[] leftFinYPoints = {baseY-drawBodyWidth/2, baseY-drawBodyWidth/2,baseY-dra
     wBodyWidth/2-drawFinWidth, baseY-drawBodyWidth/2-drawFinWidth};
521:               g.fillPolygon(leftFinXPoints,leftFinYPoints,4);
522:               int[] rightFinXPoints = {baseX-drawBodyHeight/2, baseX-drawBodyHeight/2+drawFi
     nChord, baseX-drawBodyHeight/2, baseX-drawBodyHeight/2-drawFinChord};
523:               int[] rightFinYPoints = {baseY+drawBodyWidth/2, baseY+drawBodyWidth/2,baseY+dr
     awBodyWidth/2+drawFinWidth,baseY+drawBodyWidth/2+drawFinWidth};
524:               g.fillPolygon(rightFinXPoints,rightFinYPoints,4);
525:               g.drawLine(baseX-drawBodyHeight/2,baseY,baseX-drawBodyHeight/2+drawFinChord,ba
     seY);
526:             }//end if
527:           }//end else
528:         }//end else
529:       }//end else
530:
531:       // DRAW BODY //
532:       g.setColor (Color.black);
533:       g.fillRect(baseX-drawBodyHeight/2, baseY-drawBodyWidth/2, drawBodyHeight, drawBodyWidt
     h);
534:       g.setColor (Color.red);
535:       if (finType == 3)
536:       {
537:         g.drawLine(baseX-drawBodyHeight/2-drawFinChord,baseY,baseX-drawBodyHeight/2+drawFinC
     hord,baseY);
```

```
538:       }//end if
539:       else
540:       {
541:         g.drawLine(baseX-drawBodyHeight/2,baseY,baseX-drawBodyHeight/2+drawFinChord,baseY);
542:       }//end else
543:     }//end if mode
544:
545:     if (mode == 2) // LAUNCH //
546:     {
547:       g.setColor(Color.gray);
548:       g.fillRect(3,3,793,443);
549:
550:       g.setColor(Color.black);
551:       g.drawRect(3,3,793,443);
552:
553:       g.setColor(Color.lightGray);
554:       g.fillRect(3,450,793,247);
555:
556:       g.setColor(Color.black);
557:       g.fillOval(40,500,150,150);
558:       g.fillOval(230,500,150,150);
559:       g.fillOval(420,500,150,150);
560:
561:       g.setColor(Color.gray);
562:       g.drawRect(3,450,793,247);
563:
564:       g.setColor(Color.blue);
565:       g.drawOval(40,500,150,150);
566:       g.drawOval(230,500,150,150);
567:       g.drawOval(420,500,150,150);
568:
569:       g.setColor(Color.yellow);
570:       g.drawLine(115,575,(int)(115+Math.sin(x100x*2*3.14159/10)*20),(int)(575-Math.cos(x100x
     *2*3.14159/10)*20));
571:       g.drawLine(305,575,(int)(305+Math.sin(v100x*2*3.14159/10)*20),(int)(575-Math.cos(v100x
     *2*3.14159/10)*20));
572:       g.drawLine(495,575,(int)(495+Math.sin(a100x*2*3.14159/10)*20),(int)(575-Math.cos(a100x
     *2*3.14159/10)*20));
573:
574:       g.setColor(Color.green);
575:       g.drawLine(115,575,(int)(115+Math.sin(x10x*2*3.14159/10)*30),(int)(575-Math.cos(x10x*2
     *3.14159/10)*30));
576:       g.drawLine(305,575,(int)(305+Math.sin(v10x*2*3.14159/10)*30),(int)(575-Math.cos(v10x*2
     *3.14159/10)*30));
577:       g.drawLine(495,575,(int)(495+Math.sin(a10x*2*3.14159/10)*30),(int)(575-Math.cos(a10x*2
     *3.14159/10)*30));
578:
579:       g.setColor(Color.blue);
580:       g.drawLine(115,575,(int)(115+Math.sin(x1x*2*3.14159/10)*50),(int)(575-Math.cos(x1x*2*3
     .14159/10)*50));
581:       g.drawLine(305,575,(int)(305+Math.sin(v1x*2*3.14159/10)*50),(int)(575-Math.cos(v1x*2*3
     .14159/10)*50));
582:       g.drawLine(495,575,(int)(495+Math.sin(a1x*2*3.14159/10)*50),(int)(575-Math.cos(a1x*2*3
     .14159/10)*50));
583:
584:       g.setColor(Color.lightGray);
585:       g.fillOval(110,570,10,10);
586:       g.fillOval(300,570,10,10);
587:       g.fillOval(490,570,10,10);
588:
589:       g.setColor(Color.black);
590:       g.fillRect(40,25,720,380);
591:
592:       g.setColor(Color.blue);
593:       g.drawRect(40,25,720,380);
594:
595:       g.setColor(Color.red);
596:       for(int i = 0; i < graph.length-1; i++)
597:       {
598:         if (gT[i] < burnTime)
```

```
599:          {
600:            g.setColor(Color.red);
601:          }//end if
602:          else
603:          {
604:            g.setColor(Color.yellow);
605:          }//end else
606:          if(graph[i] == true)
607:          {
608:            g.drawLine(40+i,385-gXInt[i],40+i+1,385-gXInt[i+1]);
609:          }
610:        }//end for
611:      }//end if mode
612:    }// End paint
613:
614:    public void mouseClicked(MouseEvent e)
615:    {
616:      System.out.println("(" + e.getX() + "," + e.getY() + ")");
617:    }// End mouseClicked
618:
619:    public void mousePressed(MouseEvent e){}// End mousePressed
620:    public void mouseDragged(MouseEvent e){}// End mouseDragged
621:    public void mouseReleased(MouseEvent event){}// End mouseReleased
622:    public void mouseEntered(MouseEvent event){}// End mouseEntered
623:    public void mouseExited(MouseEvent event){}// End mouseExited
624:    public void mouseMoved(MouseEvent event){}// End mouseMoved
625:
626:    public class GenericWindowListener
627:    {
628:      public void windowClosing(WindowEvent e)
629:      {
630:        System.exit(0);
631:      }//end windowClosing
632:    }//end GenericWindowListener
633:
634:    class animationControl implements ActionListener
635:    {
636:      public void actionPerformed(ActionEvent e)
637:      {
638:        int control2 = 0;
639:        if (animationIndex < graph.length)
640:        {
641:          altitude.setText(Double.toString(gX[animationIndex]));
642:          velocity.setText(Double.toString(gV[animationIndex]));
643:          acceleration.setText(Double.toString(gA[animationIndex]));
644:          time.setText(Double.toString(gT[animationIndex]));
645:
646:          if (Double.parseDouble (maxVelocity.getText()) < gV[animationIndex])
647:          {
648:            maxVelocity.setText(Double.toString(gV[animationIndex]));
649:          }//end if
650:
651:          if (Double.parseDouble (maxAltitude.getText()) < gX[animationIndex])
652:          {
653:            maxAltitude.setText(Double.toString(gX[animationIndex]));
654:          }//end if
655:
656:          if (Double.parseDouble(maxAcceleration.getText()) < gA[animationIndex])
657:          {
658:            maxAcceleration.setText(Double.toString(gA[animationIndex]));
659:          }//end if
660:
661:          double tempX = gX[animationIndex];
662:          double tempV = gV[animationIndex];
663:          double tempA = gA[animationIndex];
664:
665:          // MASKS TO FIND EACH DIGIT //
666:          x1000x = tempX/1000;
667:          x100x = (tempX - 1000*(int)(x1000x/100))/100;
668:          x10x = (tempX - 100*(int)(x100x/100))/10;
```

```
669:         x1x = tempX - 10*(int)(x10x/10);
670:         v100x = tempV/100;
671:         v10x = (tempV - 100*(int)(v100x/100))/10;
672:         v1x = tempV - 10*(int)(v10x/10);
673:         a100x = tempA/100;
674:         a10x = (tempA - 100*(int)(a100x/100))/10;
675:         a1x = tempA - 10*(int)(a10x/10);
676:
677:         graph[animationIndex] = true;
678:         repaint();
679:         animationIndex++;
680:       }//end if
681:       else
682:       {
683:         timer.stop();
684:       }//end else
685:
686:     }//end actionPerformed
687:   }//end animationControl
688:
689:   class bwTextFieldListener implements ActionListener
690:   {
691:     public void actionPerformed(ActionEvent e)
692:     {
693:       bodyWidthVal = Double.parseDouble (e.getActionCommand());
694:       drawBodyWidth = (int)(scale * bodyWidthVal);
695:       A = (bodyWidthVal/2000)*(bodyWidthVal/2000)*pi + numFinVal*finWidthVal/1000*0.003;
696:       repaint();
697:       computeCD();
698:     }//end actionPerformed
699:   }//end TextFieldListener
700:
701:   class bhTextFieldListener implements ActionListener
702:   {
703:     public void actionPerformed(ActionEvent e)
704:     {
705:       bodyHeightVal = Double.parseDouble (e.getActionCommand());
706:       drawBodyHeight = (int)(scale * bodyHeightVal);
707:       repaint();
708:       computeCD();
709:     }//end actionPerformed
710:   }//end TextFieldListener
711:
712:   class llTextFieldListener implements ActionListener
713:   {
714:     public void actionPerformed(ActionEvent e)
715:     {
716:       launchLugLengthVal = Double.parseDouble (e.getActionCommand());
717:       repaint();
718:       computeCD();
719:     }//end actionPerformed
720:   }//end TextFieldListener
721:
722:   class nhTextFieldListener implements ActionListener
723:   {
724:     public void actionPerformed(ActionEvent e)
725:     {
726:       noseHeightVal = Double.parseDouble (e.getActionCommand());
727:       drawNoseHeight = (int)(scale * noseHeightVal);
728:       repaint();
729:       computeCD();
730:     }//end actionPerformed
731:   }//end TextFieldListener
732:
733:   class fhTextFieldListener implements ActionListener
734:   {
735:     public void actionPerformed(ActionEvent e)
736:     {
737:       finChordVal = Double.parseDouble (e.getActionCommand());
738:       drawFinChord = (int)(scale * finChordVal);
```

```
739:       if(finType == 0)//e
740:       {
741:         finArea = 2*0.785*finChordVal*finWidthVal/10000;
742:       }//end if
743:       else
744:       {
745:         if(finType == 1 || finType == 3)//r,s
746:         {
747:           finArea = 2*finChordVal*finWidthVal/10000;
748:         }//end if
749:         else
750:         {   //t
751:           finArea = 3*(finChordVal+finWidthVal)*finWidthVal/10000;
752:         }//end else
753:       }//end else
754:       repaint();
755:       computeCD();
756:     }//end actionPerformed
757:   }//end TextFieldListener
758:
759:   class fwTextFieldListener implements ActionListener
760:   {
761:     public void actionPerformed(ActionEvent e)
762:     {
763:       finWidthVal = Double.parseDouble (e.getActionCommand());
764:       drawFinWidth = (int)(scale * finWidthVal);
765:       A = (bodyWidthVal/2000)*(bodyWidthVal/2000)*pi + numFinVal*finWidthVal/1000*0.003;
766:       if(finType == 0)//e
767:       {
768:         finArea = 2*0.785*finChordVal*finWidthVal/10000;
769:       }//end if
770:       else
771:       {
772:         if(finType == 1 || finType == 3)//r,s
773:         {
774:           finArea = 2*finChordVal*finWidthVal/10000;
775:         }//end if
776:         else
777:         {   //t
778:           finArea = 3*(finChordVal+finWidthVal)*finWidthVal/10000;
779:         }//end else
780:       }//end else
781:       repaint();
782:       computeCD();
783:     }//end actionPerformed
784:   }//end TextFieldListener
785:
786:   class chdTextFieldListener implements ActionListener
787:   {
788:     public void actionPerformed(ActionEvent e)
789:     {
790:       chuteDiameterVal = Double.parseDouble (e.getActionCommand());
791:       chuteArea = pi*chuteDiameterVal*chuteDiameterVal/1000000;
792:     }//end actionPerformed
793:   }//end TextFieldListener
794:
795:   class cdTextFieldListener implements ActionListener
796:   {
797:     public void actionPerformed(ActionEvent e)
798:     {
799:       cd = Double.parseDouble (e.getActionCommand());
800:     }//end actionPerformed
801:   }//end TextFieldListener
802:
803:   class mTextFieldListener implements ActionListener
804:   {
805:     public void actionPerformed(ActionEvent e)
806:     {
807:       mRocket = Double.parseDouble (e.getActionCommand());
808:       repaint();
```

```
809:       computeCD();
810:     }//end actionPerformed
811:   }//end TextFieldListener
812:
813:   class scaleTextFieldListener implements ActionListener
814:   {
815:     public void actionPerformed(ActionEvent e)
816:     {
817:       scale = Double.parseDouble (e.getActionCommand());
818:       drawBodyHeight = (int)(scale * bodyHeightVal);
819:       drawBodyWidth = (int)(scale * bodyWidthVal);
820:       drawNoseHeight = (int)(scale * noseHeightVal);
821:       drawFinChord = (int)(scale * finChordVal);
822:       drawFinWidth = (int)(scale * finWidthVal);
823:       repaint();
824:     }//end actionPerformed
825:   }//end TextFieldListener
826:
827:   public boolean action(Event e, Object arg)
828:   {
829:     if(e.target.equals(engines))
830:     {
831:       if(engines.getSelectedIndex() == 0)
832:       {
833:         double[] tt = {0,0.031,0.064,0.096,0.124,0.149,0.172,0.196,0.210,0.225,0.235,0.244,0
    .254,0.263,0.269,0.279,0.290,0.297,0.306,0.314,0.330};
834:         thrustTime = tt;
835:         double[] tc = {0,0.404,1.258,2.263,3.467,4.720,6.023,7.027,7.528,7.860,7.482,6.683,5
    .685,4.487,4.087,3.039,1.790,1.042,0.593,0.344,0.000};
836:         thrustPoints = tc;
837:         mCasing = 0.0099 ; // kg
838:         mFuel = 0.0026; // kg
839:         burnTime = 0.33; // s
840:         coastTime = 1.74; // s
841:       }//end 1/2 A6
842:       if(engines.getSelectedIndex() == 1 || engines.getSelectedIndex() == 2)
843:       {
844:         double[] tt = {0,0.041,0.084,0.127,0.166,0.192,0.206,0.226,0.236,0.247,0.261,0.277,0
    .306,0.351,0.405,0.467,0.532,0.589,0.632,0.652,0.668,0.684,0.703,0.730};
845:         thrustTime = tt;
846:         double[] tc = {0,0.512,2.115,4.358,6.794,8.588,9.294,9.730,8.845,7.179,5.063,3.717,3
    .205,2.884,2.499,2.371,2.307,2.371,2.371,2.243,1.794,1.153,0.448,0.000};
847:         thrustPoints = tc;
848:         mCasing = 0.0102 ; // kg
849:         mFuel = 0.0033; // kg
850:         burnTime = 0.73; // s
851:         if(engines.getSelectedIndex() == 1)
852:         {
853:           coastTime = 2.25; // s
854:         }//end if
855:         else
856:         {
857:           coastTime = 4.45; // s
858:         }//end else
859:       }//end A8
860:       if(engines.getSelectedIndex() == 3 || engines.getSelectedIndex() == 4 || engines.getSe
    lectedIndex() == 5)
861:       {
862:         double[] tt = {0,0.020,0.040,0.065,0.085,0.105,0.119,0.136,0.153,0.173,0.187,0.198,0
    .207,0.226,0.258,0.326,0.422,0.549,0.665,0.776,0.863,0.940,0.991,1.002,1.010,1.030};
863:         thrustTime = tt;
864:         double[] tc = {0,0.418,1.673,4.076,6.690,9.304,11.496,12.750,11.916,10.666,9.304,7.2
    14,5.645,4.809,4.182,3.763,3.554,3.345,3.345,3.345,3.345,3.449,3.449,2.404,1.254,0.000};
865:         thrustPoints = tc;
866:         mCasing = 0.01; // kg
867:         mFuel = 0.006; // kg
868:         burnTime = 1.03; // s
869:         if(engines.getSelectedIndex() == 3)
870:         {
871:           coastTime = 1.12; // s
```

```
872:         }//end if
873:         else
874:         {
875:           if(engines.getSelectedIndex() == 4)
876:           {
877:             coastTime = 3.65; // s
878:           }
879:           else
880:           {
881:             coastTime = 5.29; // s
882:           }
883:         }//end else
884:       }//end B4
885:       if(engines.getSelectedIndex() == 6 || engines.getSelectedIndex() == 7 || engines.getSe
     lectedIndex() == 8)
886:       {
887:         double[] tt = {0,0.023,0.057,0.089,0.116,0.148,0.171,0.191,0.200,0.209,0.230,0.255,0
     .305,0.375,0.477,0.580,0.671,0.746,0.786,0.802,0.825,0.860};
888:         thrustTime = tt;
889:         double[] tc = {0,0.688,2.457,4.816,7.274,9.929,12.140,11.695,10.719,9.240,7.667,6.48
     8,5.505,4.816,4.620,4.620,4.521,4.226,4.325,3.145,1.572,0.000};
890:         thrustPoints = tc;
891:         mCasing = 0.0097; // kg
892:         mFuel = 0.0056; // kg
893:         burnTime = 0.86; // s
894:         if(engines.getSelectedIndex() == 6)
895:         {
896:           coastTime = 1.53; // s
897:         }//end if
898:         else
899:         {
900:           if(engines.getSelectedIndex() == 7)
901:           {
902:             coastTime = 3.68; // s
903:           }//end if
904:           else
905:           {
906:             coastTime = 5.44; // s
907:           }//end else
908:         }//end else
909:       }//end B6
910:       if(engines.getSelectedIndex() == 9)
911:       {
912:         double[] tt = {0,0.042,0.107,0.159,0.210,0.233,0.270,0.289,0.303,0.326,0.401,0.550,0
     .802,1.026,1.291,1.524,1.683,1.702,1.730};
913:         thrustTime = tt;
914:         double[] tc =  {0,2.195,9.118,16.213,21.850,18.407,13.677,9.793,7.092,5.065,4.390,3.
     883,3.714,3.883,3.883,4.221,4.221,2.195,0.000};
915:         thrustPoints = tc;
916:         mCasing = 0.0093; // kg
917:         mFuel = 0.0113; // kg
918:         burnTime = 1.73; // s
919:         coastTime = 2.92; // s
920:       }//end C5
921:       if(engines.getSelectedIndex() == 10 || engines.getSelectedIndex() == 11 || engines.get
     SelectedIndex() == 12)
922:       {
923:         double[] tt = {0,0.031,0.092,0.139,0.192,0.209,0.231,0.248,0.292,0.370,0.475,0.671,0
     .702,0.723,0.850,1.063,1.211,1.242,1.303,1.468,1.656,1.821,1.834,1.847,1.860};
924:         thrustTime = tt;
925:         double[] tc = {0,0.946,4.826,9.936,14.090,11.446,7.381,6.151,5.489,4.921,4.448,4.258
     ,4.542,4.164,4.448,4.353,4.353,4.069,4.258,4.353,4.448,4.448,2.933,1.325,0.000};
926:         thrustPoints = tc;
927:         mCasing = 0.0094; // kg
928:         mFuel = 0.0108; // kg
929:         burnTime = 1.86; // s
930:         if(engines.getSelectedIndex() == 10)
931:         {
932:           coastTime = 2.47; // s
933:         }//end if
```

```
934:         else
935:         {
936:           if(engines.getSelectedIndex() == 11)
937:           {
938:             coastTime = 4.28; // s
939:           }
940:           else
941:           {
942:             coastTime = 6.24; // s
943:           }
944:         }//end else
945:       }//end C6
946:
947:       timeUp = burnTime+coastTime; // s
948:       numPoints = (int)(timeUp*pointsPerSecond);
949:       burnPoints = (int)(burnTime*pointsPerSecond);
950:       thrustCurve = new double[numPoints];
951:
952:       //POSITION ARRAYS
953:       T = new double[numPoints];
954:       X = new double[numPoints];
955:       V = new double[numPoints];
956:       Acc = new double[numPoints];
957:     }//end if
958:     else
959:     {
960:       if(e.target.equals(numFin))
961:       {
962:         numFinVal = Integer.parseInt((String)arg);
963:       }//end if
964:       else
965:       {
966:         if(e.target.equals(noseStyle))
967:         {
968:           noseType = noseStyle.getSelectedIndex();
969:           if (noseType == 2)
970:           {
971:             remove(noseHeight);
972:             remove(noseHeightLabel);
973:           }//end if
974:           else
975:           {
976:             add(noseHeight);
977:             add(noseHeightLabel);
978:           }//end else
979:           repaint();
980:         }//end if
981:         else
982:         {
983:           if(e.target.equals(finStyle))
984:           {
985:             finType = finStyle.getSelectedIndex();
986:             if(finType == 0)//e
987:             {
988:               finArea = 2*0.785*finChordVal*finWidthVal/10000;
989:             }//end if
990:             else
991:             {
992:               if(finType == 1 || finType == 3)//r,s
993:               {
994:                 finArea = 2*finChordVal*finWidthVal/10000;
995:               }//end if
996:               else
997:               { //t
998:                 finArea = 3*(finChordVal+finWidthVal)*finWidthVal/10000;
999:               }//end else
1000:            }//end else
1001:            repaint();
1002:          }//end if
1003:        }//end else
```

```
1004:       }//end else
1005:     }//end else
1006:
1007:     computeCD();
1008:     return true;
1009:   }//end action
1010:
1011:   class ButtonEventListener implements ActionListener
1012:   {
1013:     public void actionPerformed(ActionEvent e)
1014:     {
1015:       if(e.getActionCommand().equals("Design"))
1016:       {
1017:         timer.stop();
1018:         graph = new boolean[720];
1019:         mode = 1;
1020:
1021:         remove(altitudeLabel);
1022:         remove(maxAltitudeLabel);
1023:         remove(velocityLabel);
1024:         remove(accelerationLabel);
1025:         remove(maxVelocityLabel);
1026:         remove(maxAccelerationLabel);
1027:         remove(timeLabel);
1028:         remove(recoveryTimeLabel);
1029:         remove(altitude);
1030:         remove(maxAltitude);
1031:         remove(velocity);
1032:         remove(acceleration);
1033:         remove(maxVelocity);
1034:         remove(maxAcceleration);
1035:         remove(time);
1036:         remove(recoveryTime);
1037:         remove(fire);
1038:         remove(gL1);
1039:         remove(gL2);
1040:         remove(gL3);
1041:         remove(gL4);
1042:         remove(gL5);
1043:         remove(gL6);
1044:         remove(gL7);
1045:         remove(gL8);
1046:         remove(dL1);
1047:         remove(dL2);
1048:         remove(dL3);
1049:         remove(dL4);
1050:         remove(dL5);
1051:         remove(dL6);
1052:         remove(dL7);
1053:         remove(dL8);
1054:         remove(dL9);
1055:         remove(dL10);
1056:         remove(dL11);
1057:         remove(dL12);
1058:         add(numFin);
1059:         add(finStyle);
1060:         add(finChord);
1061:         add(finWidth);
1062:         add(bodyHeight);
1063:         add(bodyWidth);
1064:         add(launchLugLength);
1065:         add(noseStyle);
1066:         add(noseHeight);
1067:         add(engines);
1068:         add(chuteDiameter);
1069:         add(bodyHeightLabel);
1070:         add(bodyWidthLabel);
1071:         add(launchLugLengthLabel);
1072:         add(noseConeLabel);
1073:         add(noseHeightLabel);
```

```
1074:         add(numFinLabel);
1075:         add(finStyleLabel);
1076:         add(finChordLabel);
1077:         add(finWidthLabel);
1078:         add(engineLabel);
1079:         add(chuteDiameterLabel);
1080:         add(mass);
1081:         add(cdField);
1082:         add(massLabel);
1083:         add(cdLabel);
1084:         add(design);
1085:         add(launch);
1086:         add(scaleField);
1087:         add(scaleLabel);
1088:       }//end if
1089:       if(e.getActionCommand().equals("Launch"))
1090:       {
1091:         mode = 2;
1092:
1093:         recoveryTime.setText("0");
1094:         maxAltitude.setText("0");
1095:         altitude.setText("0");
1096:         maxVelocity.setText("0");
1097:         velocity.setText("0");
1098:         maxAcceleration.setText("0");
1099:         acceleration.setText("0");
1100:         recoveryTime.setText("0");
1101:         time.setText("0");
1102:
1103:         remove(launchLugLength);
1104:         remove(launchLugLengthLabel);
1105:         remove(finStyleLabel);
1106:         remove(finStyle);
1107:         remove(bodyWidth);
1108:         remove(bodyHeight);
1109:         remove(noseHeight);
1110:         remove(finWidth);
1111:         remove(finChord);
1112:         remove(numFin);
1113:         remove(chuteDiameter);
1114:         remove(chuteDiameterLabel);
1115:         remove(bodyWidthLabel);
1116:         remove(bodyHeightLabel);
1117:         remove(noseWidthLabel);
1118:         remove(noseHeightLabel);
1119:         remove(finWidthLabel);
1120:         remove(finChordLabel);
1121:         remove(numFinLabel);
1122:         remove(engineLabel);
1123:         remove(noseConeLabel);
1124:         remove(mass);
1125:         remove(cdField);
1126:         remove(massLabel);
1127:         remove(cdLabel);
1128:         remove(scaleField);
1129:         remove(scaleLabel);
1130:         remove(bodyLabel);
1131:         remove(bodyStyle);
1132:         remove(noseStyle);
1133:         remove(engines);
1134:         add(altitudeLabel);
1135:         add(maxAltitudeLabel);
1136:         add(velocityLabel);
1137:         add(accelerationLabel);
1138:         add(maxVelocityLabel);
1139:         add(maxAccelerationLabel);
1140:         add(timeLabel);
1141:         add(recoveryTimeLabel);
1142:         add(altitude);
1143:         add(maxAltitude);
```

```
1144:         add(velocity);
1145:         add(acceleration);
1146:         add(maxVelocity);
1147:         add(maxAcceleration);
1148:         add(time);
1149:         add(recoveryTime);
1150:         add(fire);
1151:         add(gL1);
1152:         add(gL2);
1153:         add(gL5);
1154:         add(gL6);
1155:         add(gL7);
1156:         add(gL8);
1157:         add(dL1);
1158:         add(dL2);
1159:         add(dL3);
1160:         add(dL4);
1161:         add(dL5);
1162:         add(dL6);
1163:         add(dL7);
1164:         add(dL8);
1165:         add(dL9);
1166:         add(dL10);
1167:         add(dL11);
1168:         add(dL12);
1169:       }//end if
1170:       if(e.getActionCommand().equals("Fire"))
1171:       {
1172:         gX = new double[720]; // position
1173:         gV = new double[720]; // velocity
1174:         gA = new double[720]; // acceleration
1175:         gXInt = new int[720]; // integer position for plotting
1176:         gT = new double[720]; // time
1177:
1178:         if (timer.isRunning())
1179:         {
1180:           timer.stop();
1181:         }//end if
1182:         else
1183:         {
1184:           for (int i = 0; i < numPoints; i++)
1185:           {
1186:             T[i] = i /pointsPerSecond;
1187:           }//end for
1188:
1189:           rungeKutta();
1190:           recoveryTimeVal = recovery(apogee);
1191:           recoveryTime.setText(Double.toString(recoveryTimeVal));
1192:
1193:           double rescale = (double)numPoints/720;
1194:
1195:           for (int i = 0; i < 720; i++)
1196:           {
1197:             int index = (int)(i*rescale);
1198:             gX[i] = X[index];
1199:             gV[i] = V[index];
1200:             gA[i] = Acc[index];
1201:             gT[i] = T[index];
1202:           }//end for
1203:
1204:           // SCALE THE GRAPH //
1205:           double max = 0;
1206:           for (int i = 0; i < gX.length; i++)
1207:           {
1208:             if (gX[i] > max)
1209:             {
1210:               max = gX[i];
1211:             }//end if
1212:           }//end for
1213:           double ratio = max/350;
```

```
1214:
1215:           for (int i = 0; i < gX.length; i++)
1216:           {
1217:             gXInt[i] = (int)(gX[i]/ratio);
1218:           }//end for
1219:
1220:           graph = new boolean[720];
1221:           maxVelocity.setText(Double.toString(0));
1222:           maxAltitude.setText(Double.toString(0));
1223:           maxAcceleration.setText(Double.toString(0));
1224:           v1x = 0;
1225:           v10x = 0;
1226:           v100x = 0;
1227:           x1x = 0;
1228:           x10x = 0;
1229:           x100x = 0;
1230:           x1000x = 0;
1231:           a1x = 0;
1232:           a10x = 0;
1233:           a100x = 0;
1234:           animationIndex = 0;
1235:
1236:           timer.start();
1237:         }//end else
1238:       }//end if
1239:       repaint();
1240:     }//end actionPerformed
1241:   }// end ButtonEventListener
1242:
1243:   public void computeCD()
1244:   {
1245:     double bl = bodyHeightVal/100;
1246:     double bd = bodyWidthVal/100;
1247:     double nl = noseHeightVal/100;
1248:     double l = (bodyHeightVal + noseHeightVal)/100;
1249:     double sBT = pi*(bd/2)*(bd/2);
1250:     double cr = finChordVal/100;
1251:     double ct = finChordVal/200;
1252:     double fw = finWidthVal/100;
1253:     double ll = launchLugLengthVal/100;
1254:     System.out.println("bl "+bl);
1255:     System.out.println("bd " + bd);
1256:     System.out.println("nl " + nl);
1257:     System.out.println("l" + l);
1258:     System.out.println("sBT " + sBT);
1259:     System.out.println("cr " + cr);
1260:     System.out.println("ct" + ct);
1261:     System.out.println("fw " + fw);
1262:     System.out.println("ll " + ll);
1263:
1264:     double sf;
1265:     if (finType == 0)//elliptical
1266:     {
1267:       sf = 2*fw*cr*0.785 + cr*bd;
1268:     }// end if
1269:     else
1270:     {
1271:       if(finType == 2)//tapered
1272:       {
1273:         sf = fw*(cr+ct) + cr*bd;
1274:       }//end if
1275:       else//rectangular and swept
1276:       {
1277:         sf = 2*fw*cr + cr*bd;
1278:       }//end else
1279:     }//end else
1280:
1281:     double swb = bl/bd*4;
1282:     double swn;
1283:     if (noseType == 0)//conical
```

```
1284:     {
1285:       swn = Math.sqrt(bd*bd/4 + nl*nl)*2/bd;
1286:     }// end if
1287:     else
1288:     {
1289:       if(noseType == 2)//ogive
1290:       {
1291:         swn = 2.666*nl/bd;
1292:       }//end if
1293:       else//round
1294:       {
1295:         swn = 2;
1296:       }//end else
1297:     }//end else
1298:     double swt = swn+swb; // sw/sBT
1299:
1300:     double velL = 50;
1301:     double velH = 200;
1302:     double viscosity = 0.0000173; //Ns/m^2
1303:     double bll = .000025; //m
1304:     double blt = .0000025; //m
1305:     double twl = viscosity*velL/bll;
1306:     double twt = viscosity*velH/blt;
1307:     double cfl = twl/2/rho/velL/velL;
1308:     double cft = twt/2/rho/velH/velH;
1309:     double cf = cfl+(cft-cfl)*0.75;
1310:     double finThickness = 0.00238125; // m (3/32 in)
1311:     double tc = finThickness/cr;
1312:     double sLL = 0.0000032429; // .004318m od, .00381m id
1313:     double sLLw = sLL + pi*(0.004318 + 0.00381)*ll;
1314:     double cdNoseAndBody = 1.02 * cf * (1 + 1.5/Math.sqrt((bl/bd)*(bl/bd)*(bl/bd))) * swt;
1315:     double cdBase = 0.029/Math.sqrt(cdNoseAndBody);
1316:     double cdOFs = 2*cf*(1+2*finThickness/cr);
1317:     double cdFin = cdOFs*sf/sBT;
1318:     double cdInt = cdOFs*cr/sBT*bd/2*numFinVal;
1319:     double cdLL = (1.2*sLL+0.0045*sLLw)/sBT;
1320:
1321:     cd = cdNoseAndBody + cdBase + cdFin + cdInt + cdLL;
1322:     cdField.setText(Double.toString(cd));
1323:   }//end computeCD
1324:
1325:   public void computeThrust()
1326:   {
1327:     //SET THRUST POINTS IN thrustCurve //
1328:     int n = thrustPoints.length-1;
1329:     double[] h = new double[n];
1330:     double[] alpha = new double[n-1];
1331:     double[] l = new double[n+1];
1332:     double[] mu = new double[n+1];
1333:     double[] z = new double[n+1];
1334:     double[] b = new double[n];
1335:     double[] c = new double[n+1];
1336:     double[] d = new double[n];
1337:
1338:     l[0] = 0;
1339:     mu[0] = 0;
1340:     z[0] = 0;
1341:     l[n] = 1;
1342:     z[n] = 0;
1343:     c[n] = 0;
1344:
1345:     for (int i = 0; i < n; i++)
1346:     {
1347:       h[i] = thrustTime[i+1] - thrustTime[i];
1348:     }//end for
1349:     for (int i = 1; i < n-1; i++)
1350:     {
1351:       alpha[i] = 3*((thrustPoints[i+1] - thrustPoints[i])/h[i] - (thrustPoints[i] - thrustPo
      ints[i-1])/h[i-1]);
1352:     }//end for
```

```
1353:     for (int i = 1; i < n-1; i++)
1354:     {
1355:       l[i] = 2*(thrustTime[i+1] - thrustTime[i-1]) - h[i-1]*mu[i-1];
1356:       mu[i] = h[i]/l[i];
1357:       z[i] = (alpha[i] - h[i-1]*z[i-1])/l[i];
1358:     }//end for
1359:     for (int i = n-2; i >= 0; i--)
1360:     {
1361:       c[i] = z[i] - mu[i]*c[i+1];
1362:       b[i] = (thrustPoints[i+1]-thrustPoints[i])/h[i] - h[i]*(c[i+1] + 2*c[i])/3;
1363:       d[i] = (c[i+1] - c[i])/3/h[i];
1364:     }//end for
1365:
1366:     for (int j = 0; j < burnPoints; j++)
1367:     {
1368:       for (int i = 0; i < n; i++)
1369:       {
1370:         if (T[j] > thrustTime[i] && T[j] < thrustTime[i+1] )
1371:         {
1372:           thrustCurve[j] = thrustPoints[i] + b[i]*(T[j]-thrustTime[i]) + c[i]*(T[j]-thrustTi
      me[i])*(T[j]-thrustTime[i]) + d[i]*(T[j]-thrustTime[i])*(T[j]-thrustTime[i])*(T[j]-thrustTim
      e[i]);
1373:         }//end if
1374:       }//end for
1375:     }//end for
1376:
1377:     for (int i = burnPoints; i < numPoints; i++)
1378:     {
1379:       thrustCurve[i] = 0;
1380:     }//end for
1381:
1382:   }//end computeThrust
1383:
1384:   public void rungeKutta()
1385:   {
1386:     computeThrust();
1387:     double h = 1/pointsPerSecond;
1388:
1389:     // equations to be solved
1390:     // x (t)
1391:     // x'(t) = v(t)
1392:     // v'(t) = - v(t)^2 * (cd*rho*A/2/(mRocket-t*dmdt)) + thrust*t/(mRocket-dmdt*t) - g
1393:     // x(0) = 0
1394:     // v(0) = 0
1395:
1396:     X[0] = 0;
1397:     V[0] = 0;
1398:     Acc[0] = 0;
1399:     double dm = mFuel/burnPoints;
1400:     double[] mass = new double[numPoints];
1401:
1402:     for (int i = 0; i < numPoints; i++)
1403:     {
1404:       if (i < burnPoints)
1405:       {
1406:         mass[i] = mRocket + mCasing + mFuel - dm*i;
1407:       }//end if
1408:       else
1409:       {
1410:         mass[i] = mRocket + mCasing;
1411:       }//end else
1412:     }//end for
1413:
1414:     double xa,xb,xc,xd,va,vb,vc,vd;
1415:     double thrust = thrustPoints[0];
1416:     apogee = 0;
1417:
1418:     for (int i = 0; i < numPoints - 1; i++)
1419:     {
1420:       xa = h*fx(T[i],X[i],V[i]);
```

```
1421:       va = h*fv(T[i],X[i],V[i], mass[i], thrustCurve[i]);
1422:
1423:       xb = h*fx(T[i] + h/2,X[i] + h*xa/2,V[i] + h*va/2);
1424:       vb = h*fv(T[i] + h/2,X[i] + h*xa/2,V[i] + h*va/2, mass[i], thrustCurve[i]);
1425:
1426:       xc = h*fx(T[i] + h/2,X[i] + h*xb/2,V[i] + h*vb/2);
1427:       vc = h*fv(T[i] + h/2,X[i] + h*xb/2,V[i] + h*vb/2, mass[i], thrustCurve[i]);
1428:
1429:       xd = h*fx(T[i] + h,X[i] + h*xc,V[i] + h*vc);
1430:       vd = h*fv(T[i] + h,X[i] + h*xc,V[i] + h*vc, mass[i], thrustCurve[i]);
1431:
1432:       X[i+1] = X[i] + (xa + 2*xb + 2*xc + xd)/6;
1433:       V[i+1] = V[i] + (va + 2*vb + 2*vc + vd)/6;
1434:
1435:       if (X[i+1] > apogee)
1436:       {
1437:         apogee = X[i+1];
1438:       }//end if
1439:
1440:       Acc[i+1] = (thrustCurve[i]*T[i+1]-Math.abs(V[i+1])*V[i+1]*cd*rho*A/2)/mass[i+1] - g;
1441:
1442:       if (X[i] < 0)
1443:       {
1444:         X[i+1] = 0;
1445:         Acc[i+1] = 0;
1446:         if (V[i+1] < 0)
1447:         {
1448:           V[i+1] = 0;
1449:         }//end if
1450:       }//end if
1451:     }//end for
1452:
1453:     gL4.setText(Double.toString(timeUp));
1454:     gL3.setText(Integer.toString((int)apogee));
1455:     add(gL3);
1456:     add(gL4);
1457:
1458:   }//end rungeKutta
1459:
1460:   public double fx(double t, double x, double v)
1461:   {
1462:     return v;
1463:   }//end fx
1464:
1465:   public double fv(double t, double x, double v, double m, double thrust)
1466:   {
1467:     double h = 1/pointsPerSecond;
1468:     return -Math.abs(v)*v*cd*rho*A/2/m - g + thrust/m;
1469:   }//end fv
1470:
1471:   public double recovery(double alt)
1472:   {
1473:     double h = 1/pointsPerSecond;
1474:     double Xr = X[numPoints-1];
1475:     double Vr = V[numPoints-1];
1476:     double Tr = 0;
1477:
1478:     double mass = mRocket+mCasing;
1479:     double xa,xb,xc,xd,va,vb,vc,vd;
1480:
1481:     if (Xr > 0)
1482:     {
1483:       int i = 0;
1484:       while (Xr > 0)
1485:       {
1486:         xa = h*fxr(Tr,Xr,Vr);
1487:         va = h*fvr(Tr,Xr,Vr, mass);
1488:
1489:         xb = h*fxr(Tr + h/2,Xr + h*xa/2,Vr + h*va/2);
1490:         vb = h*fvr(Tr + h/2,Xr + h*xa/2,Vr + h*va/2, mass);
```

```
1491:
1492:         xc = h*fxr(Tr + h/2,Xr + h*xb/2,Vr + h*vb/2);
1493:         vc = h*fvr(Tr + h/2,Xr + h*xb/2,Vr + h*vb/2, mass);
1494:
1495:         xd = h*fxr(Tr + h,Xr + h*xc,Vr + h*vc);
1496:         vd = h*fvr(Tr + h,Xr + h*xc,Vr + h*vc, mass);
1497:
1498:         Xr = Xr + (xa + 2*xb + 2*xc + xd)/6;
1499:         Vr = Vr + (va + 2*vb + 2*vc + vd)/6;
1500:
1501:         if (Xr <= 0)
1502:         {
1503:           return Tr;
1504:         }//end if
1505:
1506:         Tr += h;
1507:
1508:       }//end for
1509:     }//end if
1510:     return 0;
1511:   }//end recovery
1512:
1513:   public double fxr(double t, double x, double v)
1514:   {
1515:     return v;
1516:   }//end fxr
1517:
1518:   public double fvr(double t, double x, double v, double m)
1519:   {
1520:     double h = 1/pointsPerSecond;
1521:     return -Math.abs(v)*v*chuteDrag*rho*chuteArea/2/m - g;
1522:   }//end fvr
1523:
1524: }// End class rocketSimulator
```