Oscillations and Waves Exam 1 — Naval Battle

Jeremy's Solution

Feb. 14, 2025

Very nice! Sorry about the rendering issue that was a headache. See comments. 20/20

You only have work to do in Parts 1, 2, 4, 5, 6, and 8. *The biggest part is Part 5. Glance ahead to that part so you know where you are going, and then get started on Part 1.*

---

1. Warmup — Using NestList[] (2 pts)

(a) Write a super-simple function that doubles whatever it gets and returns that as its result. I have started the function for you:

In[135]:=

```
doubler[valueToDouble_] := valueToDouble * 2
```

Of course, you can leave out the asterisk, because Mma knows that juxtaposition means multiplication. It is a matter of style when to use it.

(b) Repeatedly call the function you just wrote using NestList[]. Start with 1 as the original value. After NestList does 5 calls of **doubler[]**, **NestList[]** should return {1, 2, 4, 8, 16, 32}. **NestList[]** takes three arguments that I have called rooster, pig, and rabbit. That is what you are fixing up:

In[136]:=

```
NestList[doubler, 1, 5]
```

Perfect.

Out[136]=

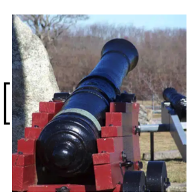
```
{1, 2, 4, 8, 16, 32}
```

---

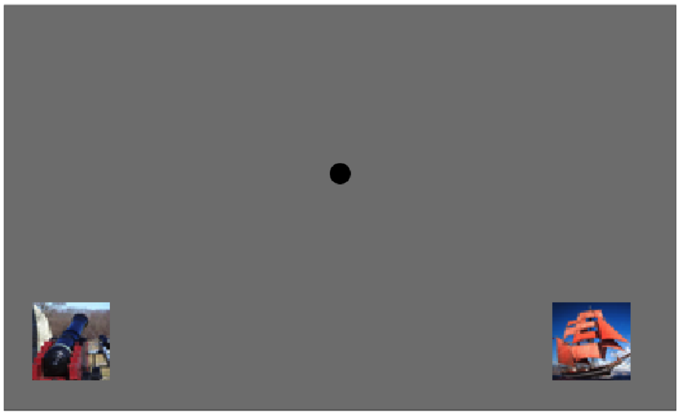
2. Naval Battle Graphics (3 pts)

In[137]:=

sailingShip = ImageResize[, {40, 40}];

cannon = ImageResize[, {40, 40}];

The goal of Part 2 is to make a graphic that looks likes this:



After trying everything else (quitting and restarting, deleting and re-rendering, etc.), the only reason I can imagine is that yours was not working is because there was a bug in Mma version 14.1 or earlier.

If you can find a middle of the night time and the internet is quiet enough to do another multi-gigabyte download of 14.2, I have to imagine that using that the rendering issue will clear up.

Everyone else (including me) is using 14.2.

You will be starting with the `cannonballGraphic[]` function below.

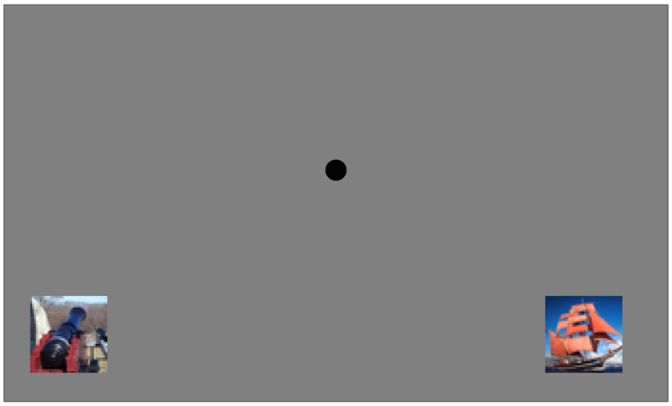
(a) Add a line that insets sailingShip to position {6.0,0.0}.

(b) Add one point, with the position specified by the argument *cannonballPosition*. *Your point should be styled to have a point size of 0.03.*

In[139]:=

```
cannonballGraphic[cannonballPosition_] := Graphics[{
   (* the first line makes a gray rectangle *)
   {EdgeForm[Thin], Gray, Polygon[{{-1, -0.8}, {7, -0.8}, {7, 4}, {-1, 4}}]},
   (* Don't mess with the next line -- that puts the cannon in *)
   Inset[cannon, {-0.2, 0.0}],
   (* (a) now you add a one-
    liner that puts the sailing ship in the graphics at (6.0, 0.0) *)
   Inset[sailingShip, {6.0, 0.0}],        Perfect.
   (* (b) then add a one-
    liner that puts the cannonball in at cannonballPosition *)
   Style[Point[cannonballPosition], PointSize[0.03]]           Perfect.
   (* and styles the point to have point size 0.03! *)
  }]
cannonballGraphic[{3, 2}]
```

Out[140]=



Your code renders perfectly on my machine.

3. Initial Conditions

There is nothing for you to do in Part 3 yet! You will be coming back to it at the very end, but do glance through it,
especially the three-line comment towards the end.

In[141]:=

```
muzzleVelocity = 0.3; (* cannonball muzzle velocity in miles / second *)
muzzleAngle = 60 °; (* you will be adjusting this -- initially it is set to 60° *)
mass = 100; (* a 100 pound cannonball *)
initialx = 0.0;
initialy = 0.3;
initialVx = muzzleVelocity Cos[muzzleAngle];
initialVy = muzzleVelocity Sin[muzzleAngle];
tInitial = 0.0;
tFinal = 100.0;

(* This is the first time you have ever
 seen a problem with both x and y coordinates *)
(* we need t, the x position, the y position,
the x velocity and the y velocity *)
(* in cc[[1]], cc[[2]], cc[[3]], cc[[4]], and cc[[5]]. *)
initialConditions = N[{tInitial, initialx, initialy, initialVx, initialVy}];
```

4. Forces on the Cannonball — Getting Acceleration (3 pts)

In[151]:=

```
dragCoefficient = 12.0;
(* the units of the drag coefficient are a screwball unit *)
(* similar to but not precisely pounds/(mile/second)^2 *)
dragFx[vx_, vy_] := -dragCoefficient vx Sqrt[vx^2 + vy^2]
dragFy[vx_, vy_] := -dragCoefficient vy Sqrt[vx^2 + vy^2]
forceOfGravity[] := -mass 0.007
(* gravity in miles/second^2 is very small because a mile is a big unit *)
```

In this problem there is an x-component and a y-component to the motion, and so we need to define an acceleration in the x-direction and an acceleration in the y-direction. What you are about to code is:

$a_x = F_x / m$ where F_x is the drag force's x-component I have given you above
$a_y = F_y / m$ where F_y is the sum of the drag force's y-component plus the force of gravity

(a) Code the acceleration in the x direction.

In[155]:=

```
ax[vx_, vy_] := (dragFx[vx, vy]) / mass
```

Perfect.

(b) Code the acceleration in the y direction (include the drag force's y-component and the force of gravity):

In[156]:=

```
ay[vx_, vy_] := (dragFy[vx, vy] + forceOfGravity[]) / mass
```

Perfect.

5. Implementing Second-Order Runge-Kutta (8 pts)

In[157]:=

```
steps = 5000;
deltaT = (tFinal - tInitial) / steps;
```

Your job is to finish implementing **rungeKutta2[]** below. To make implementation easier, I am going to go straight to the midpoint version of Runge-Kutta ($\lambda = 1/2$). Then the Second-Order Runge-Kutta equations simplify a bunch. Also, notice that in Part 4(a) and 4(b) ax and ay only depended on v_x and v_y. So that makes your Second-Order Runge-Kutta easier to implement too! Here are all seven equations you will be implementing:

$$v_x{}^* = v_x(t_i) + a_x(v_x(t_i), v_y(t_i)) \cdot \frac{\Delta t}{2}$$

$$v_y{}^* = v_y(t_i) + a_y(v_x(t_i), v_y(t_i)) \cdot \frac{\Delta t}{2}$$

$$t_{i+1} = t_i + \Delta t$$

$$v_x(t_{i+1}) = v_x(t_i) + a_x\left(v_x{}^*, v_y{}^*\right) \cdot \Delta t$$

$$v_y(t_{i+1}) = v_y(t_i) + a_y\left(v_x{}^*, v_y{}^*\right) \cdot \Delta t$$

$$x(t_{i+1}) = x(t_i) + \left(v_x(t_i) + v_x(t_{i+1})\right) \frac{\Delta t}{2}$$

$$y(t_{i+1}) = y(t_i) + \left(v_y(t_i) + v_y(t_{i+1})\right) \frac{\Delta t}{2}$$

In[159]:=

```
rungeKutta2[cc_] := (
  currentTime = cc[[1]];
  currentx = cc[[2]];
  (* What is missing here!? *)
  currenty = cc[[3]];
  currentVx = cc[[4]];
  (* And what is missing here!? *)
  currentVy = cc[[5]];
  (* Your main work is the next seven lines: *)
  vxStar = currentVx + ax[currentVx, currentVy] * deltaT/2;
  vyStar = currentVy + ay[currentVx, currentVy] * deltaT/2;
  newTime = currentTime + deltaT;
  newVx = currentVx + ax[vxStar, vyStar] * deltaT;
  newVy = currentVy + ay[vxStar, vyStar] * deltaT;
  newx = currentx + (currentVx + newVx) * deltaT/2;
  newy = currenty + (currentVy + newVy) * deltaT/2;
  (* Do not mess with the rest of this stuff. *)
  (* It stops the cannonball from going off the right edge of the *)
  (* graphic, and also stops it from going below the water. *)
  newx = If[newx ≥ 6.7, 6.7, newx];
  newy = If[newy ≤ -0.2, -0.2, newy];
  {newTime, newx, newy, newVx, newVy}
 )
```

Perfect, although stylistically I would leave out the asterisks that are required in other computer languages.

```
(* As a test, your function should return *)
(* {0.018,0.00269913,0.304674,0.149903, 0.259513} *)
(* when given the initial conditions. *)
rungeKutta2[initialConditions]
```

Out[160]=

```
{0.02, 0.00299892, 0.305193, 0.149892, 0.259481}
```

6. Computing and Collecting the Results (2 pts)

You are going to call `NestList[]` on your `rungeKutta2` functions, with `initialConditions` as the original value, and make **NestList[]** do `steps` calls of the function.

(a) Fix up the call to `NestList[]`.

(b) After `NestList[]` does all the hard work, you also need to do the right thing with `Transpose[]` to get positions to be a list of all the {x, y} pairs.

Can't remember what to do? Go back to Part 1(b) and look at what you did in that warmup problem.

In[161]:=

```
(* fix up the NestList call *)
results = NestList[rungeKutta2, initialConditions, steps];
transposedResults = Transpose[results];
times = transposedResults[[1]];
xPositions = transposedResults[[2]];
yPositions = transposedResults[[3]];
(* assemble xPositions and yPositions into a bunch of points *)
positions = Transpose[{xPositions, yPositions}];
```

Perfect! This is where the computer does the thousands of steps of work :)

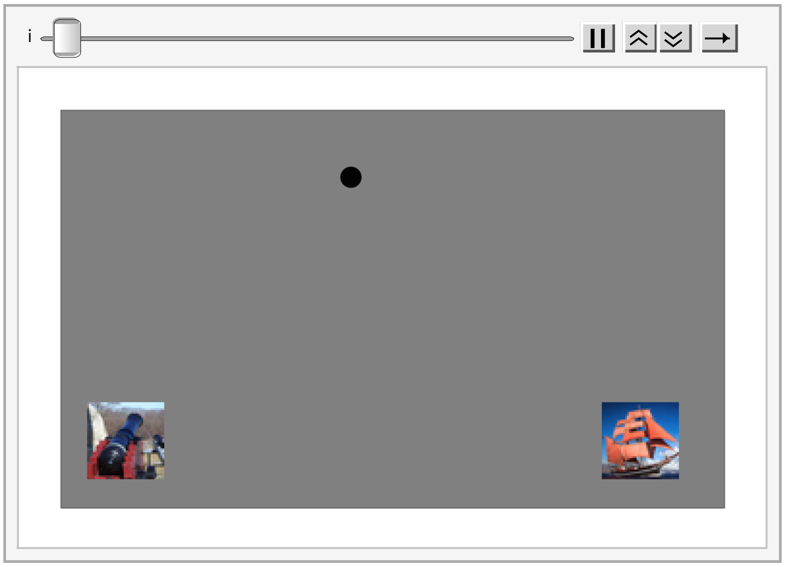
7. Animating the Results

There is nothing for you to do in this part. If everything has gone well, you will see an animation.

In[167]:=

```
Animate[cannonballGraphic[positions[[i]]], {i, 1, steps, 1}]
```

Out[167]=



8. Initial Conditions — Adjusting the Muzzle Angle (2 pts)

Now you get to go back to Part 3 and do something. You are going to adjust the muzzle angle.

Try every 10° from 10° to 60°. That's six different re-executions of the notebook.

For which angles does the cannonball do a broadside into the ship? (I find two such angles.)

(a) Low angle that causes the best broadside: 20° (nearest 10°)

Perfect!

(b) High angle that causes the best broadside: 50° (nearest 10°)

9. Game Over

Thank you for playing!

I hope that was educational and fun!