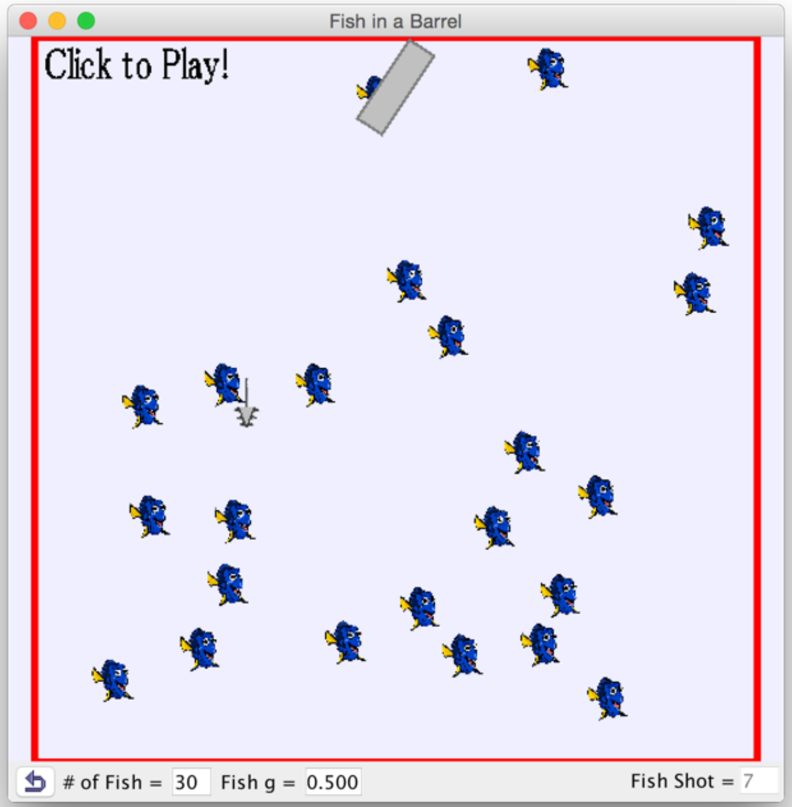
Midterm Game Project: Fish in a Barrel



# of Fish = 30 Fish g = 0.500 Fish Shot = 7

Hello Fisherman,

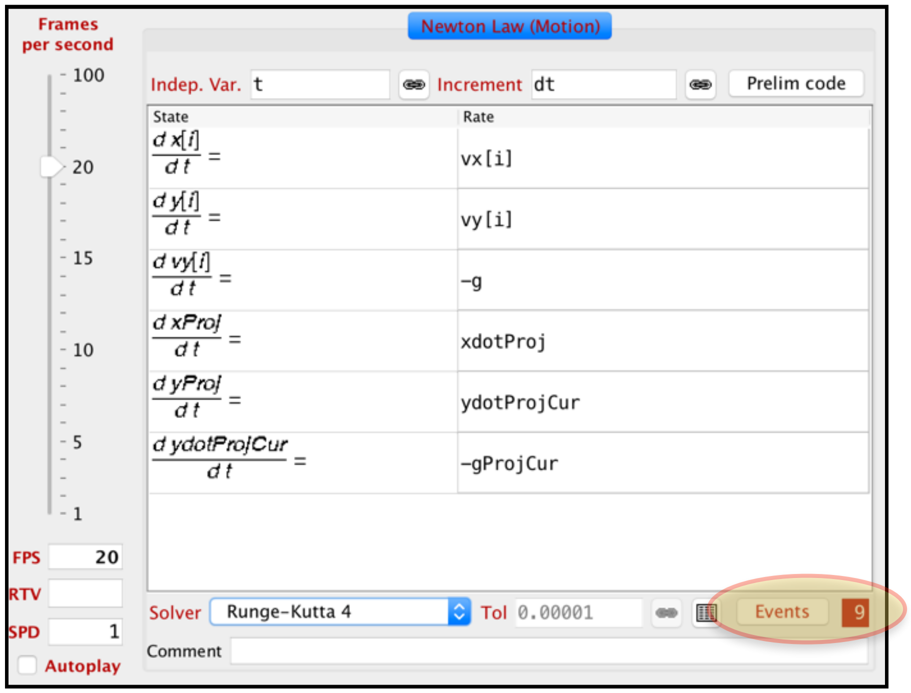
We've all heard that saying, "it's like shooting fish in a barrel" used to represent a task that's easy and undemanding. However, not many of those people have actually tried to shoot fish in a barrel, so, I ask, who are they to make such bold and brash comparisons.[1]

Here I present you with a rather simple game, but one of those games that is tirelessly addicting such as snake, Tetris, or brick breaker. It is bold to place "Fish in a Barrel" among such canonical games but once you've played for yourself it will be hard to disagree. Although a few decades too late, “Fish in a Barrel” is the type of game that would be played on Ataris all across the globe.

In all seriousness though, the purpose of creating this game was to test both my EJS knowledge, my understanding of Java syntax, as well as my understanding of some of the most basic facets of classical physics. The physics behind this simulation are fairly simply, reflecting Newton’s laws of motion, the relation between position, velocity, acceleration and time derivatives. Using EJS’s ODE based evolution option, it proved quite simple to relate the many

[1] Whether or not this phenomena is due to PETA and other activists I cannot say.

different coordinate and condition sets to a single independent time variable. The evolution page can be seen below:



Indep. Var. t Increment dt

State	Rate
$\frac{d\,x[i]}{d\,t} =$	vx[i]
$\frac{d\,y[i]}{d\,t} =$	vy[i]
$\frac{d\,vy[i]}{d\,t} =$	-g
$\frac{d\,xProj}{d\,t} =$	xdotProj
$\frac{d\,yProj}{d\,t} =$	ydotProjCur
$\frac{d\,ydotProjCur}{d\,t} =$	-gProjCur

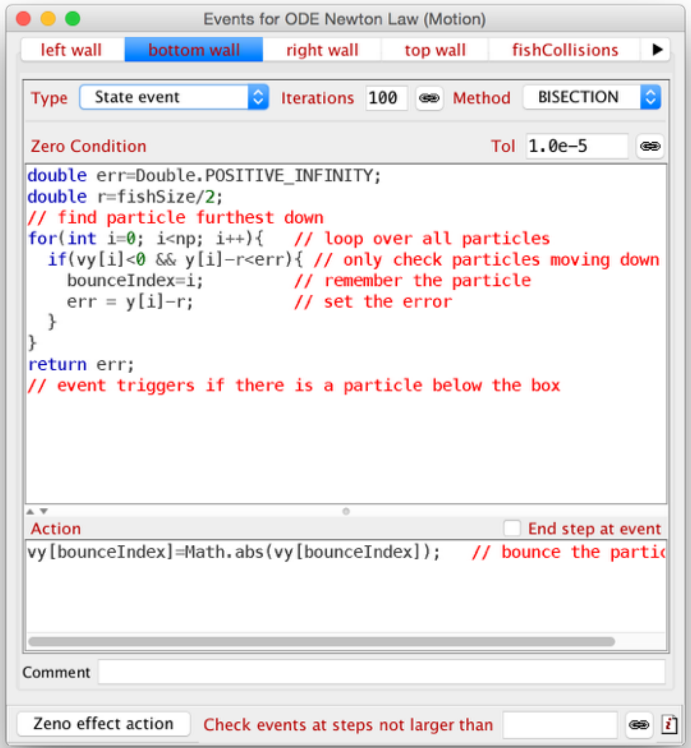
FPS 20
RTV
SPD 1
Autoplay
Solver Runge-Kutta 4 Tol 0.00001
Comment

On the evolution page, two sets of coordinates can be seen. The first three states involve arrays of x's, y's, and y velocities. The ability of EJS's ODE Solver to be able to implement it's evolutions over arrays (essentially saving a great deal of manual for looping) makes it no big task to create a set of objects such as has been done with the fish in "Fish in a Barrel." The evolution page essentially iterates through every coordinate in each array variable seen above, changing each according to the specified rate, each coordinate maintaining a designated fish/object using it's index number. The fish may look chaotic while the game is being played, but don't be fooled, those first three states in the above ODE page are all that is necessary to progress those objects.

The spear projectile, the bottom three states, progresses nearly the same as the the fish object array, but represents only a single object. Much of the meat of the code falls under the Events option that is offered by EJS's ODE solver.

This simulation uses nine events total:
4 to prevent the fish from escaping the box (all four walls)
1 to govern fish to fish collisions
1 to judge if a fish has been hit
1 to reload the harpoon for the user
2 to allow the user to bounce the harpoon of the left and right walls of the box/barrel

On the next page, for example, is a screenshot of the event that prevents fish from falling out the bottom of the box display.



Type: State event Iterations: 100 Method: BISECTION

Zero Condition Tol: 1.0e-5

```
double err=Double.POSITIVE_INFINITY;
double r=fishSize/2;
// find particle furthest down
for(int i=0; i<np; i++){   // loop over all particles
  if(vy[i]<0 && y[i]-r<err){ // only check particles moving down
    bounceIndex=i;          // remember the particle
    err = y[i]-r;           // set the error
  }
}
return err;
// event triggers if there is a particle below the box
```

Action End step at event

```
vy[bounceIndex]=Math.abs(vy[bounceIndex]);    // bounce the partic
```

Check events at steps not larger than

This event is fairly indicative of how most of the events included in this program function, although not as complicated as, say, the fishCollisions event that runs in O(N^2) time. The program iterates over every fish object and checks for the case that a fish has a negative velocity in the y direction (moving down) and is passing the zero or x axis. In this case and this case only, the "bottom wall" event will trigger and reassign the negative velocity to a positive velocity of the same magnitude. ***This represents an elastic collision in which kinetic energy is conserved, and it is worth mentioning that all collisions in "Fish in a Barrel" are elastic, as there is no energy lost to a damping factor as there would be in an inelastic collision. Thus, it is safe to say that "Fish in a Barrel" conserves energy.***

A feature worth mentioning is the implementation of two different gravities, within the same frame, one for the fish and one for the harpoon. The fish are acted upon by a much lighter gravity, while the projectile has a gravitational constant of greater value to evolve it's coordinates.

****Try changing the gravitational constant acting on the fish to catch those pesky evaders****

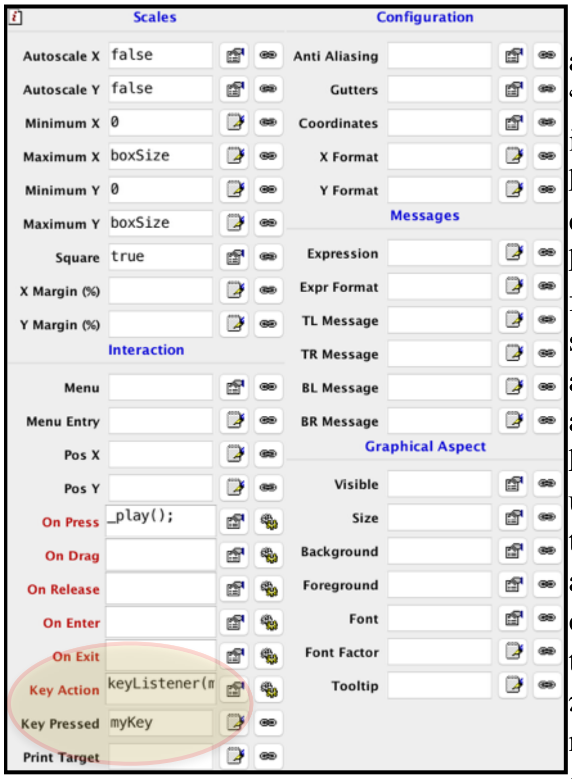
****Make the fish gravitational constant high enough and it's like shooting fish in a barrel****

The first trouble I ran into while coding for "Fish in a Barrel" was how to reload the projectile once it exited the box. I quickly realized that the best way to do this was to create a custom method that essentially acts as an initialization page for only the projectile. The

```
  //Resets Projectile Conditions
public void initializeProjectile () {
  yProj = 9.4;
  xProj = 5;
  xdotProj = 0.0;
  ydotProjCur = 0.0;
  gProjCur = 0.0;
  projVis = false;
}
```

initializeProjectile() function simple resets all of the projectiles coordinates to their initial values. The x and y coordinates again place the harpoon in the cannon, the the velocities and gravitation factor are set to 0 so that the harpoon will stay in place until the space bar is hit, and the visibility is set to false so that the user only sees the firing chamber of their cannon. This function is called in an event on the evolution page. Just as the fish bounce off the bottom wall, the harpoon re-initializes as soon as it crosses the boundary of the bottom wall. This workaround is fairly simple, but gave me pause initially as I wondered how would be the best way to continually use one object, as it would be challenging and a waste of computing power to use a set of objects to perform this feat.

A second piece of coding that gave me pause was the how I would implement key commands to actually change characteristics within the program such as aiming the cannon/ harpoon and firing the projectile. However, upon discovering the “Key Pressed” and “Key Action” attributes in he drawing frame, how to do this became more obvious.



Scales	
Autoscale X	false
Autoscale Y	false
Minimum X	0
Maximum X	boxSize
Minimum Y	0
Maximum Y	boxSize
Square	true
X Margin (%)	
Y Margin (%)	
Interaction	
Menu	
Menu Entry	
Pos X	
Pos Y	
On Press	_play();
On Drag	
On Release	
On Enter	
On Exit	
Key Action	keyListener(m
Key Pressed	myKey
Print Target	

Configuration	
Anti Aliasing	
Gutters	
Coordinates	
X Format	
Y Format	
Messages	
Expression	
Expr Format	
TL Message	
TR Message	
BL Message	
BR Message	
Graphical Aspect	
Visible	
Size	
Background	
Foreground	
Font	
Font Factor	
Tooltip	

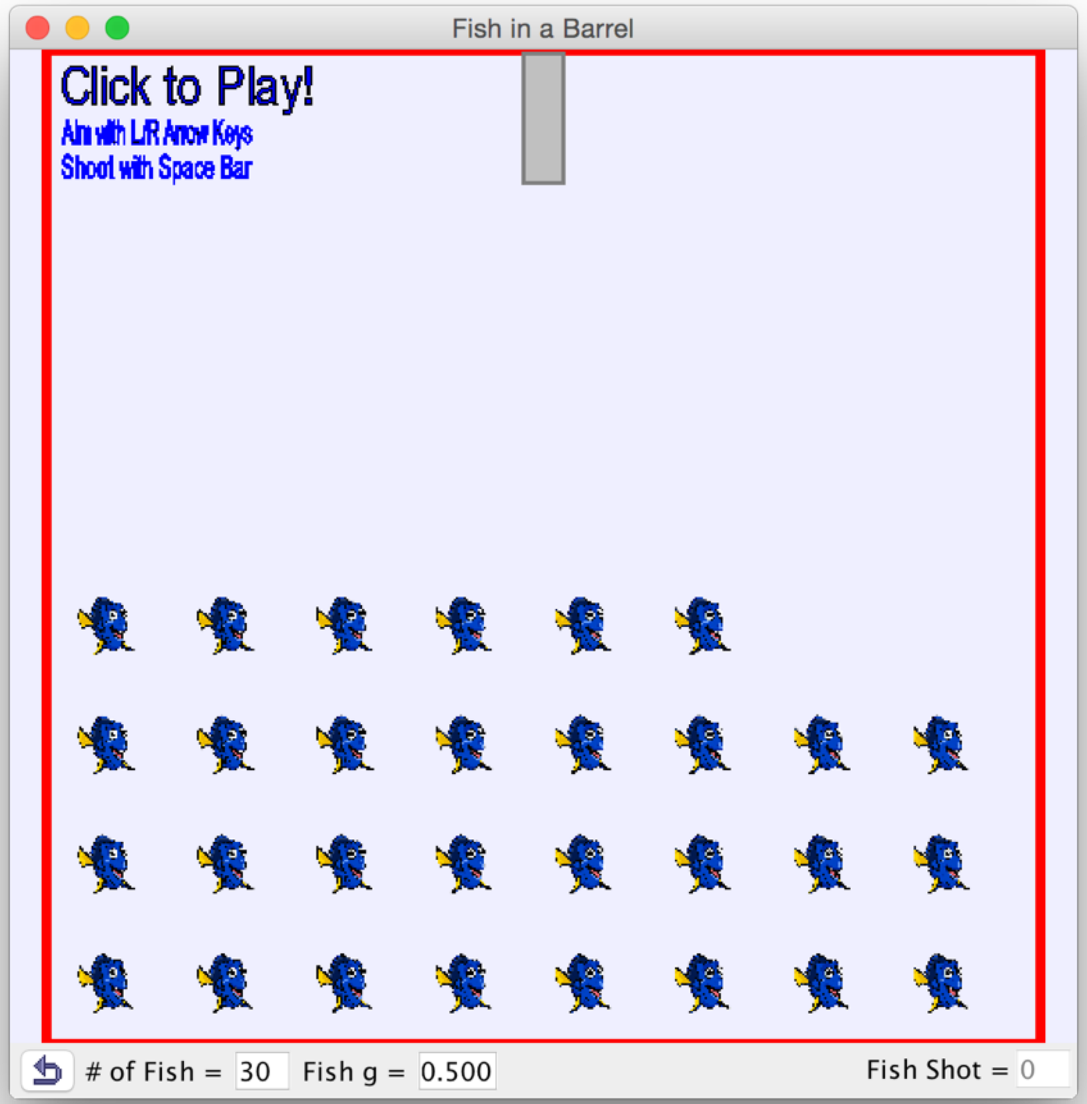
```
public void keyListener (int key) {

  if (key == 32){
    projVis= true;
    ydotProjCur = ydotProjNext;
    gProjCur = gProjNext;
    xdotProj = 5*Math.sin(rightAngle-angle);
    }

  if (key == 39 && angle > Math.PI){
    angle -= Math.PI/15;
    }
  if (key == 37 && angle < 2*Math.PI){
     angle += Math.PI/15;
    }
```

I created a myKey variable that would be assigned an integer, upon pressing a key on the keyboard, the “Key Pressed” option in the drawing frame immediately assigns the integer value of the pressed key to myKey. At this point, the “Key Action” attribute comes into play, making a call to my custom function keyListener() that takes in myKey as a parameter. This function (pictured below) is simply a string of if statements to check if the key that was pressed matches any of the specified integers. This allows me to map any key on the keyboard to a certain if statement in keyListener() and then write any sort of command under the if statement. This allows me assign shooting the projectile to the space bar, changing the visibility and other conditions as can be seen below, and assign changing the angle of the cannon and harpoon trajectory based on the arrow keys (in increments of π/15). This method is very straightforward far as mapping certain keys on the keyboard to implement certain commands within the program, and I believe that the code is quire readable. Also worth mentioning, is that this keyListener() function provides a user or “Fish in a Barrel” aficionado an easy way to add other features to keys on the keyboard or change the control scheme with only the simplest changes.

The possibilities are endless, think of the new features that can be added to “Fish in a Barrel.”

I sincerely hope you enjoy playing “Fish in a Barrel” and that we the human race will begin to gain a better understanding of the level of difficulty that really is involved with, as the saying goes, shooting a fish in a barrel.

Happy Fishing!



Credits:

This model was created by Jack Taylor using the Easy Java Simulations (EJS) version 4.1 authoring and modeling tool. Some of the code and visualization techniques were drawn from the EJS programs "*Bouncing Balls Simple Collisions*" and "*Truck Drawing*" designed by Dr. Wolfgang Christian which can be found in the Davidson College EJS library.

You can examine and modify a compiled EJS model if you run the model (double click on the model's jar file), right-click within a plot, and select "Open Ejs Model" from the pop-up menu. You must, of course, have EJS installed on your computer.

Information about Ejs is available at: <**http://www.um.es/fem/Ejs/**> and in the OSP comPADRE collection <**http://www.compadre.org/OSP/**>.