NPSL-2D : A 2D Newtonian Physics Simulation Language

Glenn Barney
gb2174@columbia.edu

0) Introduction

Physical simulations of basic Newtonian physics are relatively tedious to run in a typical programming language. Simple physical laws require modeling objects, collision rules and detection, basic geometric arithmetic, time monitoring, and linear equation processing. The goal of NPSL-2D is to provide the constructs to quickly and simply simulate basic 2D physical interactions. Students studying physics, artists wanted to create animations, and scientists/engineers wanting to run simulations for data purposes can use the language for a quick and easy way to model Newtonian mechanics.

1) Deployment

NPSL-2D code will be interpreted into Java code which will take advantage of the Swing and Java2D APIs to generate a graphical simulation that is animated in front of the user. This way the code can be run anywhere a virtual machine is installed, and since it uses the core JAVA API, no additional Jar files should be needed. There will be a lot of Java code generation to model hit detection, time monitoring and the overall world and the objects contained.

2) Scope of Physics Model

Abilities and Limitations

NPSL-2D supports the modeling of basic forces represented through vectors acting on object of basic geometric shapes such as rectangles, triangles, and circles. That is velocity, acceleration, and force all will have a vector that includes a value and a direction. Each simulation has a timeline, starting at t=0 and simulating until a defined stop time. External forces affecting the entire world, called global forces (such as gravity), and specific forces acting on a single object, called object event forces (such as an initial velocity on a ball with a set mass), can be modeled. Besides these types, forces can be normal or contact forces, which act when an object is in contact (collision, sliding) on another. All forces can be conditional, set as a global or object event force that is applied under certain rules. A Normal force has a reference to the object it is acting on, and contact forces have a reference to the object it is acting on plus all objects it is in contact with. For example, air resistance is a normal global force proportional to an object's up/down (y direction) velocity. Another example: friction is a contact global force applied to all objects that is dependent on a predefined force (gravity) and a force from object interaction. Friction acts if there is a gravitational force on an object that is touching a second object and a component of force perpendicular to gravity acting on it from the second object.

Basic elastic and inelastic collisions can be modeled. Inelastic collisions can be modeled through a loss of energy due to kinetic energy transfer proportional to a basic coefficient of restitution based on a property set on each object. Complex forces such as buoyant forces of liquids lift of airplanes, and elastic or spring objects will not be supported[1]. Every object will be treated as unbreakable and intractable. Objects can be attached at a specific point by string, and string can be broken through a specific command at a certain time in the animation.

Finally, in order to run a simulation beyond simple global force interaction (say dropping a ball under the force of gravity), the programmer is able to introduce object event forces into the system at a given set time or as a reactive response. For example, two identical balls sitting on a table two lengths apart to collide after an object event force pushes one of them into the other with an initial momentum. Another use of an object event force could be a special "pushing ball" force defined on a unique object that activates on collision. When something comes in contact with the "pushing ball", it releases an additional force besides the natural equal and opposite force on collision.

Lastly, NPSL-2D I does not support full interactions between triangles and rectangles as objects. Instead these objects are placed as Walls and are automatically treated as grounded. Future releases of the code can support triangle and rectangle objects, which are much harder to implement as they can rotate (torque) and stack.

Interactivity, Events, logging[2]:
The entire simulation will be non-interactive, the programmer defines the environment, including objects and forces, then runs the simulation. The simulation will display on the screen and also allow for logging. Logging can be set on every object defined, and is output in table format, outputting the time, each logged object's name, location, acting forces, velocity, momentum, and location. The user can set full or partial logging, and the logging interval in seconds, deepening on which of these she wishes to log.

3) Simulation Running

The programmer will define the setup of his simulation, first with objects and their placement in the world. Forces are added (global, and object event), and then a call to animate the entire system is executed. In this way, NPSL-2D is a very procedural language. Although there are objects with properties in NPSL-2D, the built in runtime animation algorithm will largely execute the functionality of object interaction.

[1] Well, spring forces have basic force equations proportional to a constant, so this may be doable by the end of the semester. The tricky part would probably be animating the spring, so I will declare them out of scope for now.

[2] Logging can possibly be a todo that will be implemented in a future release.

The world is defined in terms of Cartesian coordinates, and will display a default view of pixels set by the user. There will be a normal Cartesian grid with an x and y axis. Objects in motion outside of the animation window will continue to exist and interact with the environment, although hidden to the user's view. Each tick on the grid will represent one meter. In fact, all units will be metric, with mass in kilograms. The default bottom left corner of the view window is (0,0).

Declaring and Defining a Simulation

Objects are given a location and shape, which must be of type **rectangle**, **triangle**, or **circle**. Properties of the object can be set at any time on the object such as friction coefficient and elasticity coefficient for inelastic collisions. Objects also are grounded if they are immobile under any circumstances.

Object event and **global** forces are effects defined and applied to each object they act on at each interval in time. Forces can **depend** on certain other forces (friction **depends** on gravity), which must be defined or the complier will throw and error.

4) Language Syntax

(eserved Words are bolded below in all charts and sections)

The language is strongly typed and declaration order matters such that objects and forces that depend on others must be declared after the ones they depend on (although this could change depending on how much time I have to code a declaration order independent system).

define is used to define object references to complex data and primitive data.
Examples :

```
define num as Number;
num = 123.456;
define myBall as Circle {
        mass = 5;
        radius = 2;
        centerX = 400;
        centerY = 300;
}
```

<reference> **instanceof** <type> returns true if reference is of the specific type.
Example :

myBall **instanceof** Circle returns Boolean true.

Dot notation access object's fields or functions. All fields and functions are public sharable data.

Data Types

Complex Data Types			
Name	Base Type	Required Fields	Functions

Object	Object	id	setElasticityCoef(object, ratio)
			setStaticFrictionCoef(Object, ratio)
			setKineticFrictionCoef(Object, ratio)
			getElasticityCoef(object)
			getStaticFrictionCoef(Object)
			getStaticFrictionCoef(Object)
			attach(Object)
			detach(Object)

Circle	Movable Object	mass (Number) radius (Number) centerX (Number) centerY (Number) velocity (Number) direction(Number)	
Rectangle	Fixed Object	length (Number) width (Number) centerX (Number) centerY (Number)	
Triangle	Fixed Object	pt1X (Number) pt1Y (Number) pt2X (Number) pt2Y (Number) pt3X (Number) pt3Y (Number)	
GlobalForce	Force	direction (Number) acceleration (Number) target(Object)	

GlobalColForce	Force	direction (Number)
		acceleration (Number)
		target(Object)
		touchingMe(Object List)	

ObjEventForce	Force	direction (Number) acceleration (Number) target(Object)	applyForce(Object, time)
ObjEventColForce	Force	direction (Number)	applyForce(Object)

		acceleration (Number)
		target(Object)
		touchingMe(Object List)
			isTouching(Object)

Note ObjEventForce is a one time force that is applied to an object at a certain time, while ObjEventColForce is applied to an object whenever it collides with another.

Defining forces can use the dependsOn keyword. Forces can also access target, which represents the target Object this force is acting on. Each global force is calculated once for every Object in the world (with target having a reference to the Object the force is currently acting on). Here's an example :

```
define gravity as GlobalForce  {
        direction = DOWN;
        acceleration = 9.8;

        log target.id;
}
```

foreach is a special keyword that acts on a touchingMe list. Inside a force declaration is the *only* place a touchingMe list (or any list for that matter) is used and therefore foreach is valid only in this scope. touchingMe is kept up to date internally as the simulated.

```
//define a static function intersect that calculates the points
//of intersection between two objects

//define a static function calcAngle that finds the angle of a certain
//point in radians

define friction as GlobalColForce  dependsOn gravity {
        direction =0;
        acceleration = 0;

        foreach X in touchingMe
                If (x instanceof Wall)
                        define xIntersect as Int;
                        define yIntersect as Int;
                        xIntersect = calcXIntersect(target, x);
                        yIntersect = calcYIntersect(target, y);
                        define angle as Number;
                        angle = calcAngle(xIntersect, YIntersect);
                        direction = direction + gravity.direction * cos(angle);
                        If (target.velocity > 0)
                                acceleration = acceleration + gravity.acceleration *
                                getStaticFrictionCoef(X);
                        Else
                                acceleration = acceleration + gravity.acceleration *
                                getKineticFrictionCoef(X);
                        End

                Else
                        //do nothing
                End
        End

}
```

Primitive Data Types		
Name	Base Type	Built in Functions
Number	double	asInt() (eturns new Int with the same value)
Int	int	asNumber (returns new Number with the same value)
Boolean	boolean	

Constants

Name	Type	Value
PI	Number	3.14159265
LEFT	Number	0
UP	Number	PI / 2
RIGHT	Number	PI
DOWN	Number	3* PI / 2

Special constant:
World – Sets the view of the world for the user using setBounds(Number, Number). Has an animate() function to let the compiler know you are done defining your objects.

Operators

Math (operates on Number and Int)	
+	Addition
-	Minus
*****	Multiplication
/	Division
(	Open Parenthesis

)	Close Parenthesis
Relational	
<	Less than
>	Greater than
=	Assign
==	Equal to
Logical Operator	
&	Logical And
|	Logical OR

Control Flow

Note: Whitespace is ignored.

```
If (test)
  Statement
Else
  Statement
End

While (test)
        Statement
End

For (init;test;statement)
        Statement
End
```

Functions

Utility functions are supported to allow users to do mathematical manipulations for defining force rules and the attributes they depend on. User defined functions are all “static”. That is they are utility and are not associated with any object.

To declare a function:

```
ret_type Function foo (param_type 1, pram_type 2 ) {
        <statement>;
        <statement>;
        return variable; //optional if returning void.
}
```

Functions can return void for no return value

To call a function, just call one. Example

```
Int Function integerPlus (Int first, Int second) {
        define int as Int;
        int = first + second;
        return ( int)
}

define anInt as Int;
define firstParam as Int;
define secondParam as Int;
firstParam = 1;
secondParam = 2;
anINt = integetPlus(firstParam, secondParam);
```

Comments, delimeters
-Whitespace is ignored.
-Comments are line only and begin with //
-; is used as a end of statement token

Built in functions
Math basic built in functions will exist such as
Number cos Function (Number radian) ; //returns cosine of radian
Number sin Function (Number radian) ; //returns sin of radian

5)Code/Syntax example

To demonstrate the NPSL-2's modeling abilities, I'll walk through modeling two classic physics examples. Direction is a number in radians.

Example 1: Drop a ball

```
World.setBounds(800,600);

define gravity as GlobalForce {
        direction = DOWN;
        acceleration = 9.8;
}

define myBall as Circle {
        mass = 5;
        grounded = false;
        radius = 2;
        centerX = 400;
        centerY = 300;
}

define groundBar as Rectangle {
        mass = 100;
        centerX = 400;
        centerY = 50;
        width = 200;
```

```
	height = 100;
}

myBall.setElasticityCoef(groundBar, .85);
myBall.setStaticFrictionCoef(groundBar, .2);
myBall.setKineticFrictionCoef(groundBar, .15);
//if elasticity, friction coefficients are not set, they default to 0 (no friction,
//purely elastic collision)

World.simulate();
```

One can model a classic “swinging balls” example, which features several polished steel balls hung in a straight line in contact with each other. Momentum of the lifted ball will carry to the last one upon impact, and it will swing back causing the first one to lift and swing up. The user of the language can decided to run the simulation elastically with no air resistance for a animation that runs in perpetuity, or run it inelastically so that the simulation eventually comes to rest. Note the attach code attaches, through string, from one object’s center to another.

```
World.setBounds(800,600);

define gravity as GlobalForce {
	direction = DOWN;
	acceleration = 9.8;
}

define anchor1 as Rectangle {
	mass = 100;
	centerX = 400;
	centerY = 50;
	width = 200;
	height = 100;
}

define myBall as Circle {
	mass = 5;
	grounded = false;
	radius = 2;
	centerX = 400;
	centerY = 300;
}

anchor1.attach(myBall1);
```

```
define anchor2 as Rectangle {
	mass = 100;
	centerX = 450;
	centerY = 50;
	width = 200;
	height = 100;
}

define myBall2 as Circle {
	mass = 5;
	grounded = false;
	radius = 2;
	centerX = 450;
	centerY = 300;
}
anchor2.attach(myBall2);


define push as ObjectEventForce (myball1, 5) {
	direction = LEFT;
	acceleration = 10;
}


//elastic coefficients default to 1 if not set
//friction coefficients deftault to 0 if not set
World.simulate();
```