Using Computer Simulations to Test the Viability of a Spring Powered Launcher

Simon Abrelat

May 2019

002129-0004

IB Physics HL IA

Words:

Contents

1 Introduction

In the design process, there needs to be a way to vet interesting and alternative ideas. On my robotics team, one way to do this with computer simulations. To scale a 13 inch (33 cm) vertical step the idea of a spring powered launch was brought up, so its efficacy had to be determined. The idea was to angle the robot and have a pogo-inspired launcher spring out from the back of the robot. The goal of this simulation is to approximately test if this design possible, and, if its theoretically possible, if the materials are reasonable.

1.1 Design Constraints

This robot is designed for the FIRST Robotics Competition (FRC), so there are a lot of rules that inform what is possible. There must be a padded bumper that is no more than 3 inch (7.6 cm) off of the ground. This greatly limits the angle we are able to tilt the robot, for this simulation a max angle of 50°. There is also a weight limit of 150 lbs (69 kg). This weight will be used for the simulation to prove that the design can work in a variety of situations and is more consistent. There are space constraints, but they are not being factored in for this simulation since those are more design specific implementation details over physic constraints.

2 Simulation

This simulation has 2 sections: acceleration and flight. The acceleration uses the spring to force the robot up. The flight is the projectile motion of the robot in the air. Given the design constraints (Section 1.1), there are simulation constants like a weight of 69 kg and angle of 50°.

2.1 Spring Acceleration

Acceleration has to be calculated from the force of the spring. The spring force is approximated by Hooke's Law (1). Hooke's Law states the force is proportional to the distance of compression of the spring. The slope of this proportion would then be the spring constant which is measured in Newtons per meter. So for each meter compressed the spring exerts that many Newtons of force.

$$F_s = -kx \tag{1}$$

F_s: Force of the spring, N

k: Spring constant, $\frac{N}{m}$

x: The displacement of the spring, m

The force is then used to describe the acceleration using Newton's second law (2). The vertical acceleration from the spring is then reduced by the

force due to gravity giving the net acceleration on the object. This net force is described by equation 3.

$$\vec{F} = m\vec{a} \tag{2}$$

$\vec{F}$: Force vector, N

m: Mass, kg

$\vec{a}$: Acceleration vector, $\frac{m}{s^2}$

$$ma = -kx - mg \tag{3}$$

a: Net acceleration, $\frac{m}{s^2}$

k: Spring constant, $\frac{N}{m}$

m: Mass, kg

g: Acceleration due to gravity, $9.81\frac{m}{s^2}$

The acceleration of the object decreases the distance which then decreases the force and the net force is decreased by gravity (4), full derivation in Equation 9. This generates a cyclical decay in force. For this simulation, this second degree differential equation would be approximated using Euler's method.

$$\ddot{x} = \frac{-kx}{m} - g \tag{4}$$

Equation 4 is calculated by solving the equation in repeating small steps. For this, the equation gives you acceleration given a position, the

acceleration determines the velocity which moves the position as described in Algorithm 1.

Algorithm 1 Acceleration Solver

Require: $v, a = 0$
Require: $g = 9.81$
Require: $t = 0.001$
Require: $x =$ starting displacement
Require: $k =$ spring constant
  while $x \geq 0$ **do**
    $a \leftarrow \frac{-kx}{m} - g$
    $v \leftarrow v + a * \Delta t$
    $x \leftarrow x + v * \Delta t$
  end while

2.2 Projection Motion

There are two different ways to simulate the projectile motion: iterative and formulaic. The iterative method uses a series of differential equations following the assumptions of projectile motion. The assumptions are the gravity pushes down on the vertical axis and there is no force opposing horizontal motion.

$$\begin{aligned} \dot{x} &= \cos(\theta)v \\ \dot{y} &= v_v \\ \dot{v_v} &= g \end{aligned} \tag{5}$$

θ: initial angle

v: initial velocity

v_v: vertical velocity

Much like the Hooke's equation solver, Algorithm 1, the approximator for this motion uses Euler's method to iterate through time and accumulate toward the answer, as shown in Algorithm 2. Euler's method is a way to solve differential or integral equations by summing the output of your equation by some arbitrarily small Δt, this gets the 'area under the curve'. Then the algorithm iterates the position using the velocity and the vertical component of the velocity by the acceleration from gravity.

Algorithm 2 Projectile Motion Approximator

Require: $g = -9.81$
Require: $t = 0.001$
Require: $v =$ starting velocity
Require: $\theta =$ starting angle
Require: $v_v = \sin(\theta)v$
  while $x \geq 0$ **do**
    $y \leftarrow y + \cos(\theta)v * \Delta t$
    $v_v \leftarrow v_v + g * \Delta t$
    $x \leftarrow y + v_v * \Delta t$
  end while

This method is similar to the prior method used in Algorithm 1. The problem with it is that not only the value accumulates but any error. The benefit is greater control and visibility into the program, as well as the more intuitive explanation because you need to describe that happens in a single moment. In more complex simulations, this method is better because it allows for drag and rotational forces, but for this proof-of-concept those are not required. That means that we should use the formulaic approach. The problem with the formulaic approach is with the vertical step which cuts off the motion of the object sooner than expecting and trying to get close to an

optimal angle is difficult. This means you need additions to the general projectile motion equations, and it looses some of its generality in the process. Another problem is that if the pogo does not behave as the equations predict, invalid answers are returned. In normal projectile motion, the hight is governed in Equation 6, and the distance is Equation 7. When incorporating the step, the new distance equation is Equation 8.

$$y = \frac{v_o^2 \sin^2(\phi)}{2g} \tag{6}$$

$$x = \frac{v_o^2 \sin(2\phi)}{g} \tag{7}$$

$$x = (\frac{v_o \cos(\phi)}{g})[v_o \sin(\phi) + \sqrt{v_o^2 \sin^2(\phi) - 2gz}\,] \tag{8}$$

v_o: initial velocity

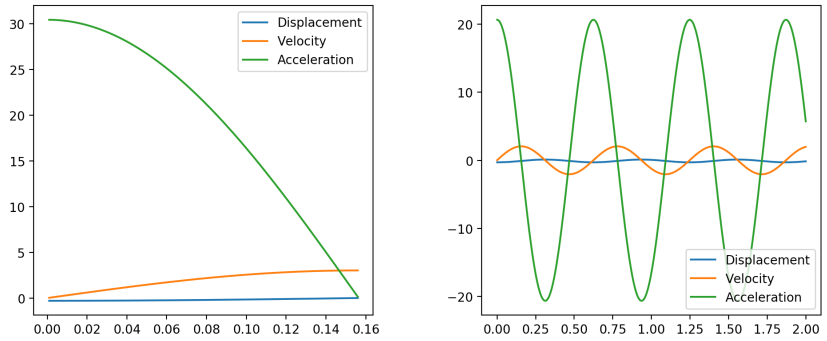
ϕ: initial angle

z: step height

3 Results

The cyclical relationship between the acceleration and displacement, described in Section 2.1, makes sense because of simple harmonic motion. For the purposes of a single action pogo, the simulation ends when the displacement is equal to zero, Figure 1a, but this is only a section of simple harmonic motion, Figure 1b.

Figure 1: Simple Harmonic Motion

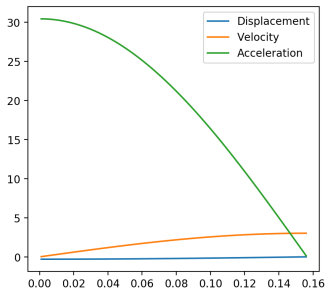


(a) Simulation to $x = 0$ (b) Simulation to $t = 2$

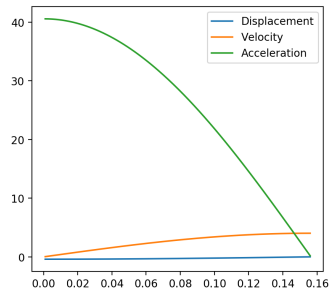
Given the parameters shown above, a mass of 69 kg, spring constant of 7000 N/m, and displacement of -.3 m, the robot would be launched 0.465 m, or around 18.3 inches, if vertical. Then, at an angle of 50° the robot is able to fly 0.127 m vertically and .506 m horizontally. To find the lowest spring constant, the spring constant would increase by some increment until the robot gets launched 0.33 m vertically. The lowest spring constant candidate is 13120 ± 10 N/m and it would launch the robot .678 m horizontally. Then to include a buffer, if the step height were 15 inches, or .38 m, the spring constant would need to be 14630 ± 10 N/m and it would travel .780 m horizontally.

For the possible mechanism, it would be best to have the longest throw as possible. When the displacement is increased, but time to $x = 0$ remains the same, Fig 2. That means that the overall power is more efficiently added by increasing the displacement. When the spring constant increases the time decreases, but the force increases, shown in Fig 3.

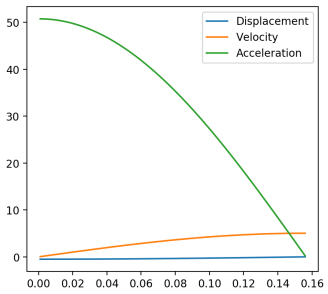
Figure 2: Varying Displacement



(a) $x = -0.3$

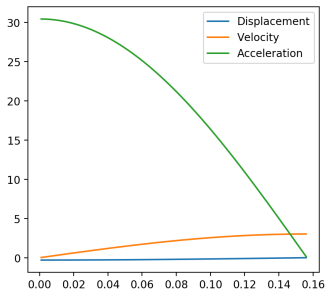


(b) $x = -0.4$

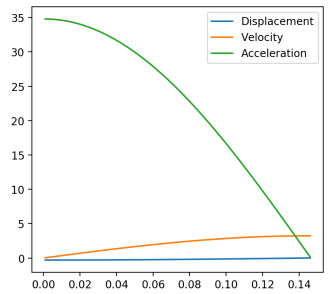


(c) $x = -0.5$

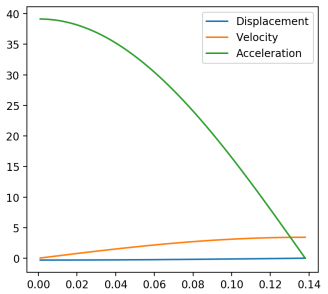
Figure 3: Varying Spring Constant



(a) $x = -0.3$



(b) $x = -0.4$



(c) $x = -0.5$

This result is intuitive especially when looking at real pogo sticks. They have relatively low weight springs but they run the length of the stick, so they have a very long throw. The problem is that we would have at most 18 inches of throw. The springs that can launch the robot given this relatively short displacement would take up too much space to make sense.

4 Evaluation

The accuracy and precision of this simulation could be improved. The model used is overly simplistic since it does not take slippage on the ground, deviance from Hooke's law, and other inefficiencies. The model presented to the team to prove if this mechanism was viable was the same as the one presented but half of the power was lost to mechanical inefficiencies. Beyond an optimistic model, the numerical approach for solving these equations yields, at least for me, a more intuitive way of describing the mechanics at the cost of accuracy. The way to mitigate error would be to decrease the change in time, or increase the number of iterations, which eventually accumulate significant floating point error. For computers, decimal numbers are difficult to define and when doing successive operations on decimal numbers will eventually cause a small error that propagates through the system. Because of this, computer simulations will never be perfect representations of reality, but they can get very close.

5 Conclusion

The design proposed is preposterous. It is mathematically possible, but for all intents and purposes entirely unfeasible. The power of the springs and their required range of motion would take up the majority of the robots

internal volume, and the tension these springs would be under would pose a major safety risk. Even if we made this robot it would not be allowed to play.

6 Appendix

List of Figures

Pogo Equation Derivation

$$
\begin{aligned}
F &= ma \\
F &= -kx \\
\sum F &= -kx - mg\cos(\theta) \\
ma &= -kx - mg\cos(\theta) \\
m\ddot{x} &= -kx - mg\cos(\theta) \\
m(\ddot{x} + g\cos(\theta)) &= -kx \\
\ddot{x} + g\cos(\theta) &= \frac{-kx}{m} \\
\ddot{x} &= \frac{-kx}{m} - g\cos(\theta)
\end{aligned} \tag{9}
$$

Code

Listing 1: Spring Acceleration

```
import matplotlib.pyplot as plt
from typing import List
import numpy as np
```

```
from math import sin, cos, hypot
from math import radians as rad

# Simulation Constants
dt: float = 1e-3  # time between physics iterations

# Physics Constants
m: float = 69  # mass: kg
k: float = 9000  # spring constant: N/m
g: float = 9.81  # gravitational accel: m/s^2
theta: float = rad(90)  # the angle the robot luaches at

# Spring data recording
x: float = -0.3 # inital compression
v: float = 0  # initial velocity
ta: List[float] = []  # time
xa: List[float] = []  # position
va: List[float] = []  # velocity
aa: List[float] = []  # acceleration

def sign(x: float) -> float:
    return -1 if x < 0 else 1

def hookesAcc(x: float, k: float, m: float) -> float:
    return ((-k * x) / m) - g

def hookesAccAngle(x: float, k: float, m: float, t: float) -> float:
    a = ((-k * x) / m) - (g * cos(t))
    return a

def spring(x: float, k: float, m: float) -> float:
    global ta, xa, va, aa, v
    i = 0
```

```
    while(x < 0):
        i += 1
        a = hookesAccAngle(x, k, m, theta)
        v += a * dt
        x += v * dt

        t = i * dt

        ta.append(t)
        xa.append(x)
        va.append(v)
        aa.append(a)
    return (v)

v = spring(x, k, m)
print(ta[-1])
plt.plot(ta, xa, label="Displacement")
plt.plot(ta, va, label="Velocity")
plt.plot(ta, aa, label="Acceleration")
plt.legend()
plt.show()
```

Listing 2: Formulatic Projectile Motion

```
import matplotlib.pyplot as plt
from typing import List
import numpy as np
from math import sin, cos, sqrt
from math import radians as rad

# Simulation Constants
g: float = 9.81  # acceleration due to gravity
dt: float = .001  # time between physics iterations

```

```
# Initial Values
v: float = 1  # the initial velocity
theta: float = rad(70)  # the initial angle
vv: float = sin(theta) * v  # verticle velocity
vh: float = cos(theta) * v  # horizontal velocity
z: float =  0.81 #0.6985

# Data Recording
x: float = 0  # projectile x
y: float = 0  # projectile y
xa: List[float] = []  # projectile x
ya: List[float] = []  # projectile y
ym: float = 0  # max Y

def height(): return (v ** 2 * sin(theta) ** 2) / (2 * g)
def distance(): return (v**2 * sin(2*theta)) / g
def stepDistance():
    det = (v**2 * sin(theta)**2) - (2 * g * z)
    if(det < 0): return -1
    return ((v * cos(theta)) / g) *\
           ((v * sin(theta)) + sqrt(det))

h: float = -1 #distance
while(h == -1):
    v += .01
    h = stepDistance()

print(h, v)
#plt.plot(xa, ya)
#plt.show()
```

Listing 3: Full Pogo Code

```
import matplotlib.pyplot as plt
```

```
from typing import List
import numpy as np
from math import sin, cos, hypot, sqrt
from math import radians as rad

# Simulation Constants
dt: float = 1e-3  # time between physics iterations

# Physics Constants
m: float = 69   # mass: kg
k: float = 7000  # spring constant: N/m
g: float = 9.81  # gravitational accel: m/s^2
theta: float = rad(45)  # the angle the robot luaches at
z: float = 0.38 # step height

# Spring data recording
x: float = -0.3  # inital compression
v: float = 0  # placeholder
h: float = -1  # placeholder

def hookesAcc(x: float, k: float, m: float) -> float:
    return ((-k * x) / m) - g

def hookesAccAngle(x: float, k: float, m: float, t: float) -> float:
    return ((-k * x) / m) - (g * cos(t))

def spring(x: float, k: float, m: float) -> float:
    v: float = 0.0
    while(x < 0):
        a = hookesAccAngle(x, k, m, theta)
        v += a * dt
        x += v * dt
    return v
```

```
35
36 def height(): return (v ** 2 * sin(theta) ** 2) / (2 * g)
37 def distance(): return (v**2 * sin(2*theta)) / g
38 def stepDistance(v: float):
39     det = (v**2 * sin(theta)**2) - (2 * g * z)
40     if(det < 0): return -1
41     return ((v * cos(theta)) / g) *\
42            ((v * sin(theta)) + sqrt(det))
43
44 def isValid(x: float, k: float) -> bool:
45     return -1 if stepDistance(spring(x, k, m)) == -1 else 1
46
47 size: int = 2
48
49 P = np.zeros((size, size))
50 for i in range(size):
51     for j in range(size):
52         P[i][j] = isValid(i * -0.3, j * 15000)
53
54 plt.imshow(P, cmap=plt.cm.coolwarm, interpolation='bilinear')
55 plt.colorbar() # side colorbar for reference
56 plt.show()
57 #  v = spring(x, k, m)
58 #  print(hookesAcc(x, k, m), hookesAccAngle(x, k, m, rad(00)))
59 #  print(height(), distance())
60 #  while(h == -1):
61     #  k += 10
62     #  v = spring(x, k, m)
63     #  h = stepDistance(v)
64
65 #  print(h, k)
```