Spring 2021 RTOS Project

Inverted Pendulum game

Version 0.4 (Config Parameter change for mass redistribution; Physics explanation)

April 12, 2021

Authored by: Jon Haines

Instructions on how to earn credit for delivering the right work product for this project during this semester can be found on Canvas.

**Basic idea:**

To model the physics behind a simple mechanics problem, with some environmental inputs that will affect the physics, with inputs from our SDK affecting the system and with the physical state represented in graphics and LEDs on our SDK.

<https://en.wikipedia.org/wiki/Inverted_pendulum>

<https://en.wikipedia.org/wiki/Pogo_stick>

One of the great advantages to our use of the SDK to simulate this is that we can have ideal conditions (no slippage where we don’t want it, no friction anywhere else; motors that have instant response and constant torque when we want it; configurable gravity; effectively-infinite K springs; etc).

**Variants suggested for consideration:**

1. Inverted pendulum (first figure, with MaxImpulse=0)
2. Inverted Pendulum with impulse kicker (first figure, with MaxImpulse>0)
3. Pogo Stick (second figure)
4. Pogo Stick with flywheel (third figure)



Baseline project expectations (if you want to get buy-in for your own idea, create 6 analogous elements):

* Physics that gets serviced every τphy.  (Part of your work in a later week on your project is going to be to push up the physics update frequency to find out when you no longer have uniform physics)
  + The Impulsive Inverted Pendulum is going to be a little weird. It will have magical impulse given to it (simply add momentum to the pendulum at the time of impulse, in the post’s direction), and upon landing will conserve the translational kinetic energy of the system, but has a perfectly inelastic collision in the y direction (adjust x velocity to maintain x momentum, and the y velocity at the bottom but not the top).
  + The pogo sticks have perfectly static friction between the earth and the post’s bottom end.
* CapSense Slider
  + Is used to adjust:
    - Force being applied to the mobile base (Variants 1,2)
    - Offset position of the mass on the “handlebar” of the pogo stick (Variant 3)
    - The flywheel’s angular momentum (Variant 4)
  + May be used as a 4-position slider, or you can use the interpolating feature of CAPSENSE\_GetSliderPosition function to get a much more precise, but sometimes more glitchy position estimate.
* Pushbutton controls and first LED
  + For the simple inverted pendulum:
    - The buttons turn down and up the gain for the Force on the bottom of the pendulum, per button push. The first LED is pulse width modulated to show the gain.
  + For all Impulse Kicker systems:
    - First button is used to pre-load the kicker based on how long the button is held between kicks, second button kicks. First LED is pulse width modulated to show how much the kicker is pre-loaded
* Second LED
  + For the simple inverted pendulum, and pogos without obstacles: light and stay lit until game is reset if Theta ever reaches +/- π/2. (The post has hit the ground)
  + For pogos with obstacles: light and stay lit until game is reset if the base of the post ever is observed to through the boundaries of an obstacle (pogo has been tripped up by not clearing the obstacle), OR if the top end of the pogo hits the ground.
  + If neither of those conditions occur but the base of the pole leaves the graphed xmin,xmax range, the second LED shall continue flashing with a 1Hz frequency until the game is reset.
* Graphics to show the pendulum’s post, the ground, and indication of rotation/translation. You must use the bottom of your display as the “ground”, but don’t need to draw a line there. The picture should update as often as possible and appropriate, with the goal being as short a lag time between a physical change and its update in the display. (As with the physics update, you’ll be finding out how far you can push your solution in later weeks.) Figure out the pixel geometry of your SDK’s LCD screen, and make the xmin and xmax (see below) relate to the outer edges of the leftmost and rightmost pixels of your screen, and scale the y axis assuming that the pixels are physically square.
* Configurable data to drive the game, with clear and appropriate units noted in comments for each 32b quantity shown below. (Your data will have space for each of these, even if you don’t implement anything for that datum.)
  + Data Structure Version (Defined as 2 for this table)
  + Gravity [mm/s^2]
  + Mass of Ball at end of post [g]
  + Length of Post (dpost in picture above) [mm]
  + Graphing Limits
    - (signed) Xmin [mm]
    - (signed) Xmax [mm]
  + Variant selected [enumeration, per Variant list]
  + MaxForce (for Variants 1,2) [mN]
  + MaxImpulse (for Variants 2-4) [g-m/s]
  + Non-pole Mass (for Variants 3,4)
    - Mass [g]
    - Center-of-mass distance from bottom of pole [mm]
  + Maximum distance from pole for the additional mass to move (Variant 3) [mm]
  + Maximum Angular Momentum (Variant 4) [g-mm^2/s]
    - Note: the current angular momentum is “instantly adjustable” via your slider, as if the flywheel has infinite angular acceleration for an infinitesimal time.
  + Number of Obstacles
  + Obstacle[x Number of Obstacles]:
    - (signed) xmin [mm]
    - (signed) xmax [mm]
    - height [mm]

**The Physics:**

While you can go look on the web to find out how to solve the math behind this (one student reports that solving the 2nd order ODE via Runge-Kutta seems to work well with double data types), I want to share some simple-minded modeling of the physics with you here, to help you ground your work if this still seems daunting in the 3rd week of the project.

If we were modelling planets and gravity, we would simultaneously update all of the bodies’ accelerations based on all of the effects of all of the other bodies.

Based on Fg=, over all the other bodies (i), a mass (m) at position (p) will accelerate at that vector sum divided by the mass, with projections in the x and y directions. Once the acceleration has been computed, the velocity can be computed, and then the position, as (again, these are vectors):

* vnew = vold + a·Δt
* pnew = pold +vnew· Δt

So at a high level, if we can keep track of current position and velocity of a ball (e.g. the one at the end of the post), and compute the forces upon it, we’ll be golden to update the physics every Δt.

**Inverted Pendulum:**

It is my (unproven) belief that you can, with a sufficiently-short Δt, and a single modeling hack (to be discussed below), avoid complex (and potentially time-consuming) calculations on our little computationally-challenged SDK, and retain a playable game.

**Balancing forces:**



The force on the ball due to gravity when the stick is straight up is counteracted by the support by the stick, even though the slightest perturbation in Theta will lead to the massless&frictionless base scooting out of the way and the ball falling. One simplification to the model might be to omit the Kronecker Delta, arguing that the stick is never EXACTLY vertical—an assumption that if utilized would need to be accounted for in unit testing because the stable point of an unmoving vertical post would no longer exist.

One could certainly model this from the base’s perspective instead, but then the massless cart would infinitely accelerate if a Force is applied when Theta=0. This is one of those places where the modeling I’m proposing appears simpler than modeling more smoothly (RK4)—but you’ll see a downside next.

Since this solving technique is essentially using the Euler method for rectangular integration, any weirdness at a single update point propagates until the next time of update. This nastiness centers on Theta=0.  
When we apply a Force that might cause the sign or zero-ness of Theta to flip, the boundary conditions are a bit obnoxious, in that:

* When we update through (or to) this point in a Δt, our updated position might be more than dpost above the ground. The simple modeling hack for this is that if the y position of the ball as you finish updating it is >dpost, set it to dpost.
* When we update through (or from) this point in a Δt, the sign of Theta needs to flip—but since this update scheme actually computes Theta as a result of |cos-1(y/dpost)|, the sign of the result is unknown. So you have to hack in some sign management. For example, if y got sufficiently-close to dpost, and the y-velocity just went from positive to negative, then assume that the cart flipped to the other side. Note that this essentially assumes that the base has a little momentum—which was the same fundamental adjustment to the “ideal system” that was required to get the RK4 to solve correctly.

So, the summary of the solution is as follows. Given the current value of F and Theta:

1. Compute the ball’s acceleration in both directions
2. Update the ball’s velocity in both directions (if the sign just went negative and the old y-position was “close” to dpost, then note to self that the sign on Theta must change)
3. Update the ball’s position in both directions (clamping the y value)
4. Compute new Theta based on the old sign, indication of sign flip, and |cos-1(y/dpost)|.