analysis

Arun Nanduri

July 20, 2022

Contents

1 Analysis

The key pieces of any simulation are:

- how the state is represented
- how the derivatives, or instantaneous changes in the system, are calculated
- how these derivatives are integrated to produce a new state representing the evolved system

These questions will be discussed first from the point of view of the underlying physics of the system, and then from the perspective of implementing them as well-architected code.

1.1 Modeling and Physics

To start with, I have chosen to model the pen as an infinitely thin rod of finite length and non-uniform density. This model is useful because it removes the need to consider rotation of the pen about its longitudinal axis, and because it dispenses with tracking any internal state of the pen (inner pieces, ink, or moving pencil graphite in the case of a pencil). It also means we do not have to specify the shape/profile of the pen, which is also useful as that is basically irrelevant for the physics we are concerned with here. The non-uniform density assumption means that the pen's center of mass will be located somewhere along its length, and not necessarily in the middle of the pen.

To wit: When a pen is thrown into the air with a flicking motion, it will typically be spinning end-over-end. It will also then be in free fall, at least until it hits a table or the floor. Therefore, the physics we will consider are the effect of gravity on a body thrown in the air, and rotation of a freely spinning body, when no torques act upon it. Extending the scope of the physics we incorporate into the simulation will be discussed in another section below, including how they may be implemented.

1.1.1 Gravity

The effect of a uniform gravitation field on a freely falling body is to provide a constant acceleration to the center of mass of the body. Mathematically,

$$\ddot{\vec{r}} = -g\hat{z}$$

where $\vec{r}$ is a three-dimensional vector giving the position of the pen's center of mass (COM) with respect to some arbitrary coordinate frame, g is a scalar giving the magnitude of the gravitational acceleration (it can be negative), and $\hat{z}$ is a unit vector in the z direction. We are assuming a constant gravitational field in the negative z direction here. Integrating this equation twice, we obtain the equations of motion for the position of the COM of the pen:

$$\vec{r}(t) = -\frac{1}{2}gt^2\hat{z} + t\vec{v_0} + \vec{r_0}$$

where $\vec{v_0}$ is the initial velocity of the pen's COM when it is thrown into the air, and $\vec{r_0}$ is the initial position of the pen.

1.1.2 Rotation

A useful aspect of the model we have so far is that the rotational component of the pen's motion, about its COM, is decoupled from the overall motion of the COM itself. Since the only force acting upon the pen is gravity, and gravity acts upon the COM of the pen and therefore exerts no torque, there are no torques acting on the pen. This means that the axis of rotation of the pen, if one exists, will not change, and greatly simplifies the representation of the pen's rotation in three dimensions. Mathematically,

$$\ddot{\theta}\hat{b} = 0$$

where $\hat{b}$ is a unit vector pointing in the direction of the pen's axis of rotation, and θ is a scalar giving the angular displacement of the pen along that axis of rotation. Integrating this equation twice, we obtain the rotational equations of motion for the pen about the COM:

$$\theta(t) = t\omega_0 + \theta_0$$

where ω_0 is the initial angular velocity of the pen, and θ_0 is the initial angular position of the pen.

1.2 State representation

Given the above analysis, the variables we need to track as a part of the pen's state are its

- linear displacement (position)
- linear velocity
- axis of rotation
- angular displacement
- angular velocity

where the first three are vector-valued and the last two are scalar-valued. These variables are grouped in a `State` struct in the code.

The pen itself is modeled as a struct that contains a `State` struct and a few other fields that describe the pen, namely its mass, length, and moment of inertia (rotating about an axis perpendicular to its length through the COM). These variables are specified once when initializing the pen, and do not change (unlike the `State`). They are kept so that if the simuation is extended to incorporate other physics, this information, which will be needed, is on hand.

1.3 Calculating derivatives

Although we could simply use the closed form equations of motion (EOM) found above for the pen's motion, we will resort to numerical integration to simulate the pen's motion. Using the closed form EOM would be maximally accurate from a numerical point of view, but from a physics/modeling perspective, they may be accurate for a modeled world that may not be realistic enough. Computing derivatives at each time step and integrating them numerically allows us to architect the simulation in a more general manner that can handle models for which there are no closed form EOM (which is all models of interest).

To this end, a `Dynamics` class is introduced that holds ambient information about the world needed to find the system derivatives at any point in time (right now just the gravitational acceleration). Conceptually, we want this class to provide us with the ability to go from specifying the state of the system to a representation of the change in the system over some small time step.

The "change in the system" is represented by a `StateDelta` struct. This struct holds four members that are all analogs of variables in the `State` struct, except for the axis, since that is unchanging. The variables in the `StateDelta` struct can be added directly onto the corresponding variables in the `State` struct, so the `StateDelta` represents the actual change in the system over a small time period.

What the `Dynamics` class's member function, `get_derivative`, gives us however is not a `StateDelta`, but a lambda expression taking in a time interval and returning a `StateDelta`. This is because, properly speaking, the system dynamics are instantaneous rates of change or time derivatives (ds/dt) of the state, and such quantities cannot be added directly to the state. They need to be multipled by a time value (dt) to be turned into a incremental change in the state (ds) that can be added onto the state.

We want to use the type system to encode this relationship. That is why the object used to represent the derivatives of the system, and what the dynamics class supplies, is a lambda expression; only once a dt is provided can a `StateDelta` be obtained and added onto the `State`.

Another way to think about this is that the dynamics is a mapping from both the current state of the system and the length of a small time interval to a change in the state of the system; what the `get_derivatives` function returns is this mapping curried with respect to the first argument (the state).

1.4 State integration

Integration is performed by an integrator class that provides an interface so that different concrete integration methods can be used. The interface provides a public integrate method that resolves a time interval into small step sizes; a non-public step method which can be customized is used to perform each step.

Two derived classes that represent two different ways of doing each step are provided. They are a class for the forward Euler method, and a class for the fourth-order Runge-Kutta (RK4) method.

The forward Euler method is useful because it is the simplest integration method to code, and therefore is useful when prototyping and for checking that the rest of the simulation is working properly. If fine-grained accuracy is not required, it also can be the least computationally expensive integration method.

The RK4 method is more complex, but gives good accuracy for numerical integration, and is a very commonly used method in high-fidelity simulations. This is the integration method I would choose to use for the simulation.

Quantitatively, the accuracy of these integrator methods can be characterized by their accumulated truncation error. This error is a function of the step size dt used for the integration. Since smaller step sizes always give more accurate results, but result in increased computational expense, the idea is to find the rate at which accuracy increases with smaller step sizes (or equivalently, decreases as we increase the step size).

The forward Euler method is first-order, meaning that the accumulated truncation error as an integration is carried out is proportional to the step size dt. That means that if we wish to obtain results that have half the error, we will need to double the step size and therefore double the computational expense.

On the other hand, the RK4 method is fourth-order, meaning that the accumulated truncation error is proportional to dt^4. This means that our doubling of the step size would result in approximately 1/16 of the error that we had before.

The stability of the RK4 method is also greater than the forward Euler method; that is, there is a class of systems for which the forward Euler method will give qualitatively incorrect results (that exponentially increase) if the step size is too big. This can still happen with the RK4 method, but for a given step size the region of stability (in some parameter space characterizing the dynamics) will be larger.

Finally, it is worth mentioning that both of these methods are so-called explicit integration methods, which calculate the state of the system as a fold (in the sense of functional programming), or in other words, use multiplication and addition and the value of the system state at earlier times to calculate the state at later times. There are also implicit methods, which simultaneously solve for the system state at different times and replace the arithmetic of explicit methods with solving non-linear equations. They are thus more computationally expensive, but are needed when integrating stiff systems, a hallmark of which is high-frequency oscillations in the system dynamics. Explicit methods may need a prohibitively small step size to avoid instability on stiff systems, but implicit methods are able to deal with stiffness with reasonable computational expense. The simple system here with gravity acting on the pen is not stiff, and so the explicit methods provided are sufficient.

1.5 Inaccuracies and improvements

1.5.1 Air resistance

One step to improve the verisimilitude of the simulation could be to incorporate air resistance. This may not be expected to have a huge impact on the pen's motion, and was therefore neglected initially, but it's not too complicated to model, and the effect can be gradually turned on by varying the coefficient of resistance, which will help in verifying the simulation's correctness. A way to model air resistance would be to incorporate a drag term in the dynamics equations. At the high Reynolds numbers we would expect for the typical scenario of a pen falling through air at atmospheric conditions, the drag term would be proportional to velocity, so the dynamics equations for the motion of the COM would look like

$$m\ddot{\vec{r}} = -mg\hat{z} - C_l v^2 \hat{v}$$

where m is the mass of the pen, C_l is some coefficient representing the strength of the air resistance, and v is the magnitude of the velocity of the pen's COM. The coefficient's value is determined by the density of air and the area of the pen. The force of the air resistance directly opposes the objects motion, so it points in the opposite direction of the object's velocity. Since our relatively simple model assumes an infinitely thin rod, it is somewhat at odds with adding this resistance term, but it could still be useful in a phenomenological sense, if we did some experiments to determine a reasonable value fo C_r (perhaps by measuring the terminal velocity of the pen).

Similarly, the air resistance would affect the rotational motion of the pen. It wouldn't change the axis of rotation, but it would oppose the rotational motion in a similar way to how the COM's motion is affected. The dynamical equation for the rotational motion of the pen could perhaps be modified to be

$$I\ddot{\theta}\hat{b} = -C_r\omega^2$$

here C_r is another coefficient of resistance representing how it affects the rotational motion of the pen, which could also be empirically determined.

To code this new physics into our simulation, our `Dynamics::get_derivative` function would be modified. Inside the lambda, we would add on the new terms representing the contribution of air resistance. For these terms, we'd have to make use of the pen's mass and moment of inertia, which are stored in the `Pen` struct, so those quantities would probably have to be stored in a dynamics object customized for the pen itself.

1.5.2 Impulses

Another way to improve the simulation would be to model collisions of the pen with hard surfaces–this would simulate the flipped pen eventually hitting a table or the floor. This would be a more involved change, as the effect of such a collision could change the axis of rotation of the pen.

More specifically, if we restrict ourselves to elastic collisions such as those we might approximately expect to happen upon hitting a hard table or non-carpeted floor, we can model the effect on the pen of a collision as follows: Because the kinetic energy of the pen will be conserved and no other object is assumed to gain kinetic energy, the velocity of the pen at the point of impact will be exactly reversed by the collision. And, if we assume that the collision occurs when one end of the pen makes contact with the surface, we can model the effect of the collision by an impulse imparted at that end of the pen, normal to the surface.

Together, these assumptions would allow us to figure out the linear and angular impulse imparted to the pen, which we could then incorporate into the `Dynamics::get_derivative` function. Again, we'd need to use the pen's mass and moment of inertia, but that information would allow us to return a `StateDelta` when the collision time is reached. The `StateDelta` struct itself would also have to be modified to hold a change in the pen's axis of rotation, in case the pen hits the surface at an angle.

1.5.3 Templating the simulation code

A way to improve the architecture of the simulation itself could be to template the integrator code. Right now, it is specialized to take in the `State` type that corresponds to the state of the pen. Since the pen is the only object being simulated right now, this is currently acceptable, but if we wanted to generalize the code to work on different types of objects or states, allowing the `State` and `Dynamics` objects to be template parameters makes sense. The code in the integrator files has been left in header files anticipating this purpose.

1.6 Building and running the code

Building the code requires a C++ compiler supporting C++17 or later, and Bazel installed on a Linux system.

The code can be built on Linux by running

```
bazel build spinning_pen
```

To run the code in main(), use

```
bazel run spinning_pen
```

The tests can be run with

```
bazel test --test_output=all //:spinning_pen_tests
```