Simulating the Physical World

How to Train Your Robot

Chapter 5

Brandon Rohrer

How to Train Your Robot

About This Project

How to Train Your Robot is a side project I've been working on for 20 years. It's a consistent source of satisfaction. A big part of the joy is sharing what I learn as I go. And who knows? Maybe someone will find it useful.

Brandon
Boston, USA
July 24, 2023

Simulating the Physical World

Chapter 5

In which we create alternate universes.

It’s time to invent physics!

Like some geeky twist on the Biblical creation story, on Day (Chapter) One we picked our goal: Human-Directed Reinforcement Learning. On Day Two we invented time. On Day Three we made inter-process communication. On Day Four we made animation. Now on Day Five, we are creating physics. Each step builds on the one before and we're about to make use of everything we've done so far.

Because robots are things of metal and plastic that bump around in the world, our primary focus will be on how physical things move and interact. If you don’t remember your last physics class or have never had the pleasure, not to worry. We will provide all the ingredients for explaining what we’re doing and why we’re doing it.

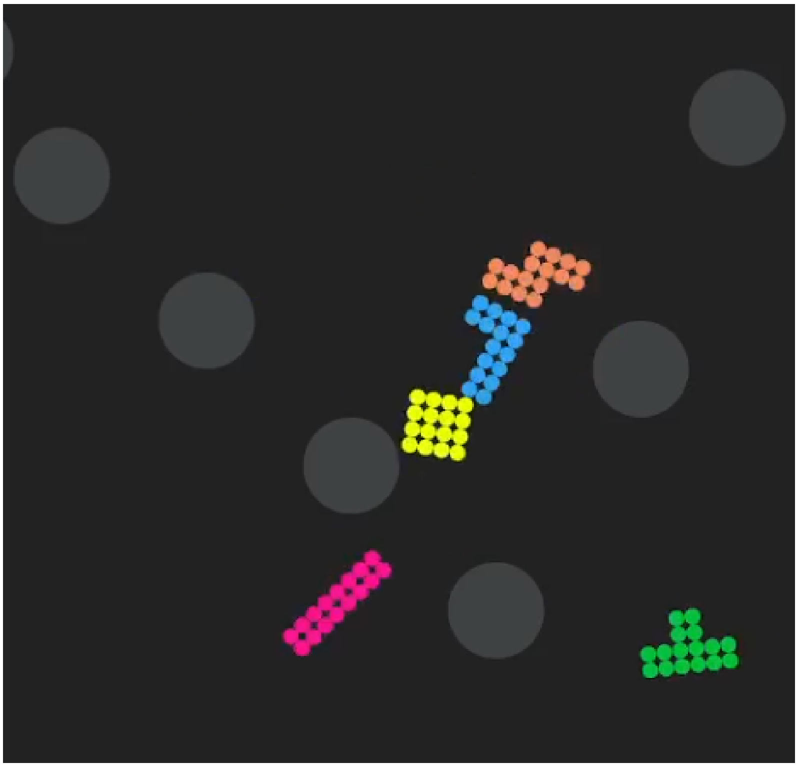
Simulation takes us a big step closer to what we ultimately want to do with robots. It helps us to understand how everything works together using virtual robots and helps us to debug our algorithms in a faster and cheaper way. And if we should find ourselves short on the money, time, and space it takes to raise a robot, simulation gives us a rich playground for working with our ideas.

We're going to do some other things while we're at it. In order to do all of this in a computer, we're going to need to talk about numerical integration. More than anything we've done so far, simulation will push the processing capabilities of our CPUs. This gives us the excuse to do some fun algorithmic work that helps us lighten the load and make our code run faster.

Best of all we get to watch things move. I get a little thrill every time I get to set up a simulation and watch it dance.

Tetris

To show how the principles and hacks of simulation work together, we'll refer to a working example showing Tetris-style blocks bouncing off each other and obstacles in their environment as gravity pulls them down to earth.



For the video, check out https://vimeo.com/842023685

When I set out to write this chapter, I approached it the same way I did the others–layering ideas on top of each other, one at a time, and showing a working code example at each step. For simulation this quickly ballooned out of control. There are so many tiny steps involved in getting an example running that it would be awkwardly long. Making it worse, a lot of the little pieces of code interact with each other making it difficult to illustrate them in isolation. It became clear that this approach would not work.

Instead we're going to work with the small-but-complete Tetris example, zooming in on snippets of code and operating concepts one at a time. This will let us observe them in action, surrounded by their supporting cast members, giving us a good idea of how they might be used and extended in other examples.

Let's start at the beginning.

Force equals mass times acceleration

A guy named Isaac Newton lived halfway in history between Charles Darwin and William Shakespeare. Like Shakespeare and Darwin, Newton spent a lot of time observing and writing about what he saw. But where Shakespeare observed people and Darwin observed animals, Newton watched objects–planets and apples and such–especially how they move about in space and bump into each other.

This quote from Newton captures the wide-eyed curiosity with which he approached the natural world.

> I do not know what I may appear to the world, but to myself I seem to have been only like a boy playing on the sea-shore, and diverting myself in now and then finding a smoother pebble or a

> prettier shell than ordinary, whilst the great ocean
> of truth lay all undiscovered before me. [1]

Thanks to his childlike curiosity, Newton made a lot of observations during his lifetime, but we are interested in one in particular–the relationship between how hard you push on something and how fast it moves.

Newton's insight was that the harder you push on a thing, the faster its velocity changes. More precisely, the rate of change of velocity, acceleration, is proportional to the force applied to an object. If you have two bowling balls and you push on them both for one second, but you push one twice as hard as the other, it will end up going twice as fast.

The equation showing this relationship is

$$acceleration \propto force$$

or

$$a \propto F$$

for short.

The other thing that Newton called out is that the mass of an object makes a difference. If I push both a bowling ball and an automobile as hard as I can, the bowling ball will end up with a decent velocity, where the car will barely be creeping along. The greater the mass, the less the acceleration. Put

[1] Memoirs of the Life, Writings, and Discoveries of Sir Isaac Newton (1855) by Sir David Brewster (Volume II. Ch. 27)

another way, the acceleration is inversely proportional to mass, meaning that doubling the mass results in halving the acceleration.

$$acceleration \propto \frac{1}{mass}$$

or

$$a \propto \frac{1}{m}$$

Combining the effects of force and mass on acceleration gives a complete formula for it.

$$a = \frac{F}{m}$$

Or in its more classic form,

$$F = ma$$

This will be the foundation of every simulation we're going to make from here out. All of our efforts will go into describing the force for different physical interactions, handling the subtleties of mass, and calculating acceleration in a stable and efficient way.

Numerical differentiation

Having the relationship between acceleration and force is helpful, but we still have to connect it to position. Position is something that can be measured with a ruler or a potentiometer or time-of-flight of a laser. It's the basis of most **kinematics**–the measurement and calculation of how things move.

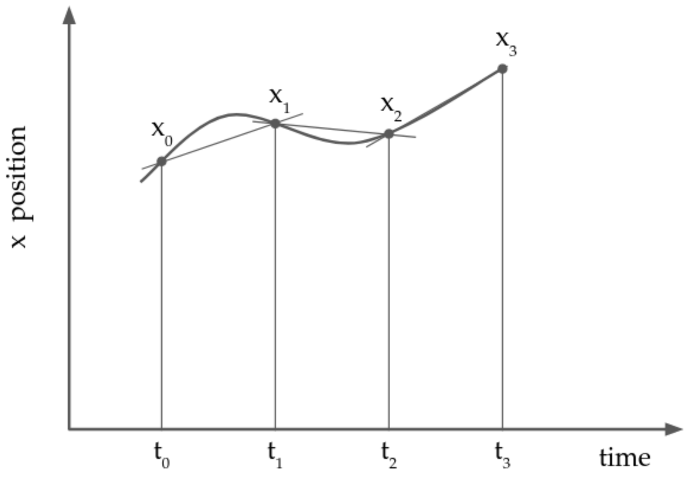
The challenge in going from position to velocity is that velocity is the rate of change of position. And calculating acceleration adds another layer of challenge because it is the rate of change of velocity. Rate of change is a theoretical quantity, something that can only be calculated precisely using the infinities and infinitesimals of calculus. The best we can do is approximate it. But don't worry. We're going to approximate the hell out of it.

Picture a ping pong ball on a flat table being blown back and forth by the air currents in the room. With an overhead camera and some clever image processing we can measure the position of the center of the ball accurately along the x-direction.

The actual x-position of the ping pong ball varies continuously. The ball moves gradually through space; it doesn't glitch or jump or transport itself. Our camera has a typical video frame rate of 30 frames per second. That means we get to measure the ping pong ball's position every 33 milliseconds. This is fairly fast compared with the expected motions of the ball. We will find that the position changes only the tiniest bit between measurements. At this speed, it's easy to pretend that our measurements are continuous.

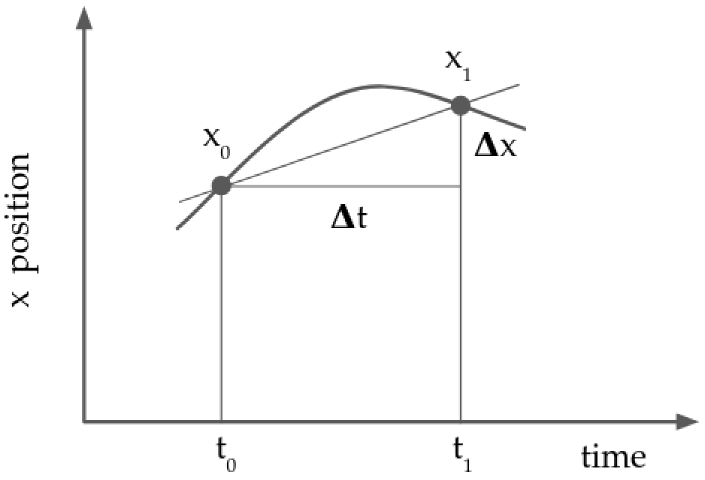
Compared to the movements we are measuring, they practically are.

Now imagine that we are more limited and that we have to send our video imagery to a distant cloud server in order to perform the image processing and position calculation, and that it can only complete this round trip once per second. A lot of things can happen in one second. The ball can speed up. It can slow down. It can reverse direction. It can even wobble, reversing direction several times. Now our measurements resemble this picture.



The behavior of the ball between measurements is more varied. When we were measuring 30 times per second, we could be more confident that drawing straight lines between our measurements would trace the ball's position accurately. Now the ball's actual position shows more variety.

We feel this limitation in our measurements even more acutely when we go to estimate velocity. Our measurements are discrete in time, that is, they only occur occasionally and we can only guess at what happens in between them. If the measurements were sufficiently far apart, the ping pong ball could go to the moon and back between them and we would never know. Because of this, we can't claim to know the actual velocity at any point. The best we can do is calculate the average velocity for each of the time intervals between our measurements.



For instance, between time t_0 and t_1 the average velocity is the change in position divided by the change in time.

$$v_{01} = \frac{x_1 - x_0}{t_1 - t_0}$$

Or rather

$$v_{01} = \frac{\Delta x}{\Delta t}$$

This approximation assumes constant velocity over the time period between t_0 and t_1. If that were actually the case, the ball would follow the straight line shown in the plot, instead of the curved one. Unfortunately we have no way of knowing that from our measurements. The only reason we know of the discrepancy is that this is an example pulled from our imagination, so we have perfect knowledge of what's going on behind the curtain.

This is the heart of numerical differentiation. Change in any quantity divided by change in time gives an approximation of the derivative of that quantity with respect to time. It can convert position to velocity, angle to angular velocity, even velocity to acceleration.

The **Δ**t term is the consistent throughout the simulation. It's the duration of one clock tick. Thanks to our careful work on building a pacemaker clock in Chapter 2, we know **Δ**t will be repeatable. It shows up in the code as the constant `CLOCK_PERIOD_SIM`. A Python statement of this would look like this.

```
v_01 = (x_1 - x_0) / CLOCK_PERIOD_SIM
```

There are other variations of how to calculate the derivative, but this particular formulation (called the forward difference) is particularly useful because it only uses past measurements rather than looking ahead to future ones. It helps when we want to get an answer now, instead of in two or three time steps.

Numerical integration

As you may have guessed, we can go the other way too. This is fortunate. In our simulation work we'll follow the pattern

1. Calculate the forces on a body.
2. Calculate the resulting acceleration.
3. Calculate the velocity.
4. Calculate the new position.

Differentiation gets us from position to velocity to acceleration, but we actually need to go the other direction.

We can approach this by rearranging the equations above to get

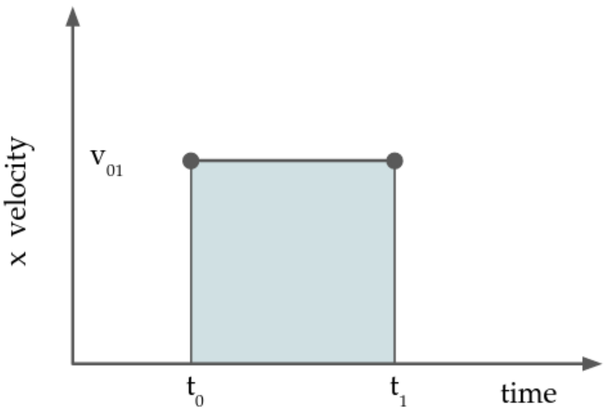
$$\Delta x = v_{01} \Delta t$$

or in slightly modified form

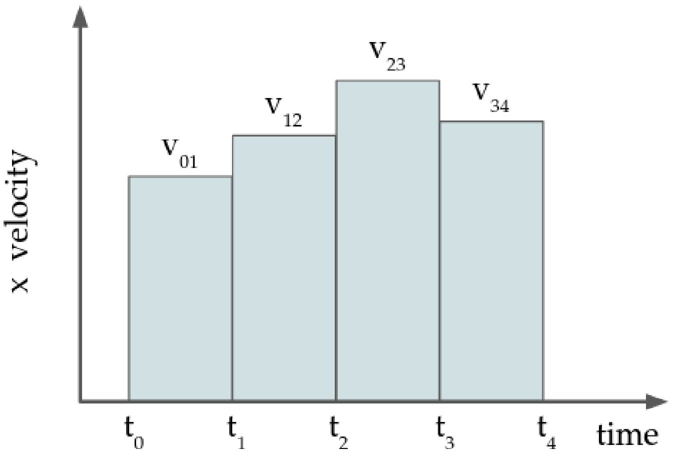
$$x_1 = x_0 + v_{01} \Delta t$$

This nugget gives us what we need to take the velocity, find the change in position it generates, and increment the old position to get the new position.

Visually, the change in position is the area under the velocity curve during this time step, the height v_{01} multiplied by the width, t_1 - t_0.



Repeating this operation for every time step into the future tallies up all the increments of motion. It integrates the velocity curve over time by slicing it thinly and adding the contribution of the velocity for one time step, and then the next, until the simulation runs to completion (or crashes).



Translated to Python, it looks like this for the first time step.

```
x_1 = x_0 + CLOCK_PERIOD_SIM * v_01
```

We can also represent all the future timesteps, reusing variables in this shorthand.

```
x += CLOCK_PERIOD_SIM * v
```

This is how numerical integration shows up in the simulation code. It is used to get position from velocity and angle from angular velocity. It's also used to get velocity from acceleration. The theory and the math are exactly the same.

The full process of going from force to acceleration to velocity to position is on display in the `update_positions_numba()` function in the `body.py` module.

```
a_x = f_x / m
a_y = f_y / m
a_rot = torque / rot_inertia

v_x += CLOCK_PERIOD_SIM * a_x
v_y += CLOCK_PERIOD_SIM * a_y
v_rot += CLOCK_PERIOD_SIM * a_rot

x += CLOCK_PERIOD_SIM * v_x
y += CLOCK_PERIOD_SIM * v_y
angle += CLOCK_PERIOD_SIM * v_rot
```

body.py at tyr.fyi/5files

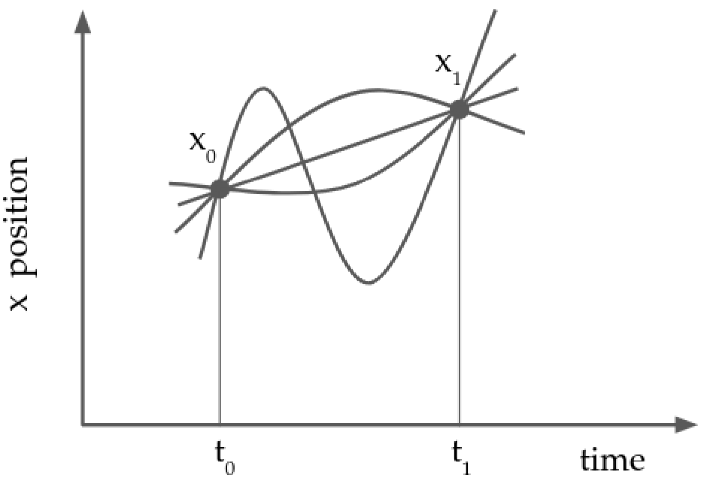
Lines occur in triplets with one line each to handle x-position, y-position, and angle.

There are other ways to implement numerical integration. Other simulation programs will use more sophisticated numerical integration techniques and variable time steps. There are clever methods for making several passes with different time steps and checking whether they agree, to check whether the integration is numerically stable. But at their core, all mechanics simulations (that is, simulations dealing primarily with moving pieces that can bump into each other) do something very similar to the code above.

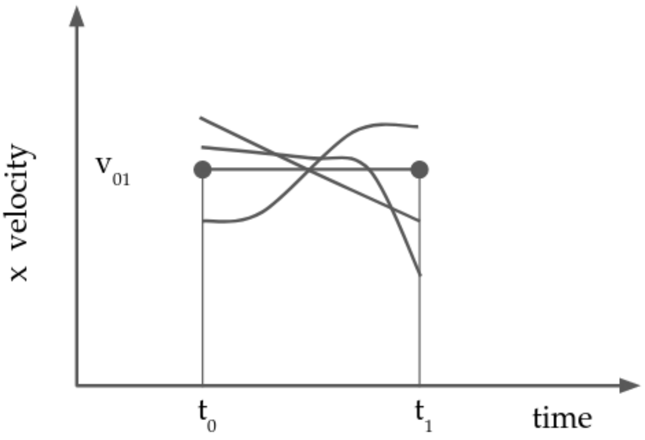
Time and Integration

If you'll humor me for a short tangent, another look at our one-step numerical integration helps us better understand the blind spots in this method.

After measuring x_0 at t_0 and x_1 at t_1 we can only guess at what time course x took in between. It could have looked like any of these.



Similarly, while we know the average velocity between these two time points, the actual time course of velocity could have looked like any of these.



The only thing we can be certain of is that they (by virtue of being continuous) cross the v_{01} line at least once. Another thing that we can be reasonably confident in is that on average, that crossing will tend to occur toward the middle of the time period, halfway through the time step, at $(t_1 + t_0) / 2$, especially if we are careful to use a fast simulation clock with a small enough Δt so that velocity changes slowly compared to the the clock ticks.

Another way to look at this is that the forward difference calculation is a decent estimate of the derivative one half time step ago, at $t_{½}$ if you can imagine such a thing. Each application of numerical differentiation introduces a delay of half a time step. Taking a second derivative to estimate acceleration would add yet another half time step delay, resulting in an acceleration estimate that is a full time step behind the current position measurement.

The reverse is true too. Our implementation of numerical integration makes the subtle assumption that the current acceleration was the acceleration for a half time step previously and will still be the acceleration one half time step from now. Using it to estimate velocity then ends up predicting the velocity one half time step in the future. And using that in turn to estimate position ends up predicting the position another half time step into the future. By the time we've gone from acceleration to position, we've also turned it into a prediction for one time step from now.

This works out beautifully for our method.

1. Calculate the forces, f_{t+0} on a body based on (among other things) the current position, x_{t+0}.
2. Calculate the resulting acceleration, a_{t+0}.
3. Calculate the velocity, $v_{t+½}$.
4. Calculate the new position, x_{t+1}.

By the time we've updated x, we've also advanced it forward in time. It's ready and on hand for the next iteration of the simulation. We don't have to do additional work to let the simulation play out for a full time step. It's built right into the calculations.

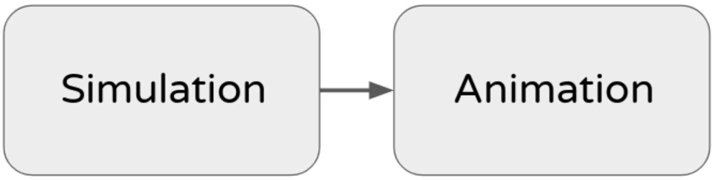
I expect this will either be blindingly obvious or somewhat confusing, but I was so delighted when I realized it that I wanted to make sure I included it in the chapter.

Just humor him. Smile and nod.

Separation of simulation and animation

In Chapter 3 we outlined how to spin up multiple processes and get them to talk with each other. This ability helps us out a lot when running a real-time simulation. Both animation and simulation can be computationally demanding. Forcing them to share clock cycles would severely limit the quality of both. But putting them into separate processes gives them both more runway and more oxygen.

Our process diagram is pretty simple. The simulation calculates positions for all the objects and passes them to the animation, like so.



The code reflects this separation too.

run.py is the top level script that kicks off the simulation and animation processes.

sim.py has the high level code for initializing and running the simulation.

viz.py contains all the code for running the animation.

There are a few other files that support these.

body.py has all the code for how rigid bodies move and interact with each other and with walls. It lives in the `Body` class.

walls.py has another bit of code for representing walls. They are kind of like rigid bodies, but they don't move and have a specific shape, so their code and the calculations we perform with them can be specialized.

config.py has all of the constants and configuration parameters gathered in one place for anyone who wants to tweak the simulation. It's a better user experience than making people comb through all the files to find the relevant line when they want to change the color of a body or slow down the simulation clock.

Both simulation and animation need a pacemaker clock to run well, but because they are different processes, there's no reason they need to be running at the same clock speed. The animation only needs to run at the right speed to make a good moving picture. 30 Hz works great. But for a real-time rigid body mechanics simulation that includes collisions, it's necessary to run much faster than this. 1000 Hz works well most of the time.

Due to this timing mismatch, the simulation doesn't need to pass the positions of its bodies to the animation at every simulation time step. In fact, preparing the message to be sent through the interprocess queue is so unexpectedly slow that we don't want to do it any more often than necessary. The message that gets passed between processes is the state of the simulation–the positions of every part of the rigid bodies. It has to be passed as a single, serialized object–something that is a nice orderly sequence of bytes, rather than scattered across many memory locations. A solid serialization tool we discussed in Chapter 3 is the **json** format, by way of Python's json package. However, for mysterious (to me) reasons, json serialization of the simulation state took a large fraction of the simulation time budget when I ran it on each time step.

To get around this speedbump, the simulation is aware of the animation clock speed and only adds a new version of the state to the queue when it knows the animation is going to need it. Another convenient side effect of keeping all our user-adjustable parameters in **config.py** is that it provides a centralized location for the rest of our code to pull those parameters from. It means that **sim.py** can learn about animation pacemaker clock speed without having to know anything about the **viz.py** or that animation code. It can just reach in to **config.py** and reference `CLOCK_FREQ_VIZ`.

Closing processes

Having multiple processes is a tremendous advantage, but doesn't come without costs. One of these costs is that shutting down the simulation means shutting down three different programs, the simulation, the visualization, and the runner that kicks them both off. When I list the processes running during the simulation I get these three. (Note that $ represents the bash command prompt here, and is not part of the command.)

```
$ ps -a

      PID TTY          TIME CMD
...
 454346 pts/1    00:00:00 python3
 454347 pts/1    00:00:23 python3
 454348 pts/1    00:00:17 python3
...
```

Based on their cumulative CPU run time, it's easy to find the runner with 00:00:00. It doesn't do anything other than kick off the other two. It takes closer inspection to figure out which of the others is the simulation and which is the animation. They both are CPU intensive–the visualization because of re-generating video frame objects in Matplotlib and the simulation because of doing a lot of math really fast. Here it turns out the process with 00:00:23 is the simulation.

When one of these processes crashes, runs to completion, or is manually terminated with a Control-C, the other two are still running. One of them at least will continue to demand an entire core. If you start a new simulation run, and only kill one of the processes, and repeat this enough times, you will eventually run out of cores and your computer will cease to be useful until you do some housekeeping.

A convenient bash command for this on my Linux machine is

```
$ pkill -9 python3
```

It kills all currently running Python 3 processes. If you have other Python processes running and don't wish to burn them all down, you can also kill processes one at a time by process ID with

```
$ kill -9 454347
```

No longer platform independent

This brings us to the second and more profound drawback of working in multiple processes–it's harder to make it work across all popular platforms. The commands for killing the remaining processes are specific to bash. Assuming that the processes are named python3 is specific to my own environment. I'm content to manually kill the remaining processes when terminating the simulation, but this isn't a good engineering solution, and it doesn't necessarily generalize to your system.

A good solution to this would involve some health monitoring among the processes. Yet another process would look to each for a heartbeat indicating they are still functioning. When one stopped for too long, it would be assumed dead and the others would be killed by the monitoring process, which would then shut itself down. I would need to do some more work to make sure this behaved well across Linux, MacOS, and Windows, and didn't make assumptions that could easily be violated.

This would take some effort, but it would certainly be tractable. And I might undertake it at some point. But it's important to remember that it is a tangent, a distraction from the core goal of demonstrating Human Directed Reinforcement Learning in a physical robot. Cross-platform compatibility is very nice to have, but it is negotiable, and if it delays us too much from reaching our goal we can put it in a big bin labeled "work that we tell ourselves we're going to do later, but if we're being honest with ourselves we're never going to get to it."

The platform independence problem is actually more than just an issue of me being lazy. Starting in Python 3.8 there are difficult-to-reconcile differences in how to create new processes in different operating systems.

We use Python's multiprocessing package to spin up our independent processes. The function `set_start_method()` lets us choose between two options for our case: `"fork"` or `"spawn"`. We've been using `"fork"` because it handles our multiprocessing Queue appropriately and lets our processes both access it. The trouble is that as of Python 3.8, using the `"fork"` method raises an error in MacOS. I don't understand the details of why, but it can result in unsafe behavior and fixing it doesn't seem to be a priority for Apple. And while it's possible to get away with using `"spawn"` for what we're doing, it seems to require a specialized re-implementation of Queues, which seems like a hassle and, worse, makes the code more complex and less readable.

The challenges to platform independence don't stop there. We can see others creeping over the horizon: optimization and hardware compatibility. Already to get the performance we need for real-time simulation, we've had to do some optimization. (More on this in the next chapter.) The nature of optimization is fine-tuning, filling in the corners, and shaving down the edges. It makes use of specific knowledge of the system and how its pieces interact. Critically, optimization is antithetical to platform independence. It exploits every bit of platform specific knowledge it can get. Cross platform portability is blind to platform differences. Optimization is hyper aware of them.

Hardware compatibility is a challenge to cross platform portability for a similar reason. Hardware interfaces demand a detailed knowledge of how the system works so they can talk to it. The majority of what a computer does internally has been successfully abstracted, so as long as our code does nothing but logic and read/writes to memory we don't have to think about what processor it's running on. But as soon as we move outside of that protected domain, particularly to anything that physically interacts with the wider world, that quickly changes.

If we want to write code that runs anything like microphones, cameras, motors, encoders, ultrasonic range finders, contact switches, or laser scanners, then suddenly the specifics of our platform matter very much. Flipping bits and interpreting bytes, finely sliced timing and idiosyncratic error messages all make regular occurrences in low level hardware code. To ensure compatibility at this level of granularity, drivers and control software have to be written for particular chips and operating systems. The drivers are often very specific about what hardware and software they support, even down to the version number of both.

A notable exception to this is monitors and displays. Most software just works on most displays. Take our visualization library matplotlib, for example. We can just import it and make a plot, regardless of whether we're on MacOS, Windows, or Linux. But this platform agnosticism can only occur because of the extremely dedicated matplotlib developer team who write and organize the platform specific backend renderers and account for differences in various display technologies. For the most part, all of this is automatically handled and you only have to think about it if you are feeling adventurous and want to go mucking around in that part of the code.

Platform-independent solutions are really just 3 (or 7 or 23) platform-specific solutions in a trenchcoat. When something Just Works™ everywhere, it means there is a small army behind the scenes working furiously to make a very disjointed world appear seamless.

With this in mind, I'm faced with a decision about how to handle the code I'm writing: A) Put in the work to make it platform independent or B) Focus on a single implementation. After some soul-searching and Scotch sipping, I've landed on B. I don't have the time to single-handedly do A and make any meaningful progress. And more importantly, it's not the part I'm good at. And most importantly of all, it's not the part I find most satisfying.

This opens up an opportunity. If you feel inspired to port any part of the How to Train Your Robot code to another platform, know that I fully support you. I will cheer for you, answer your emails, and proudly amplify your work should you choose to share it.

If this makes you sad, here is a picture of a puppy to make you feel better.

Development platform

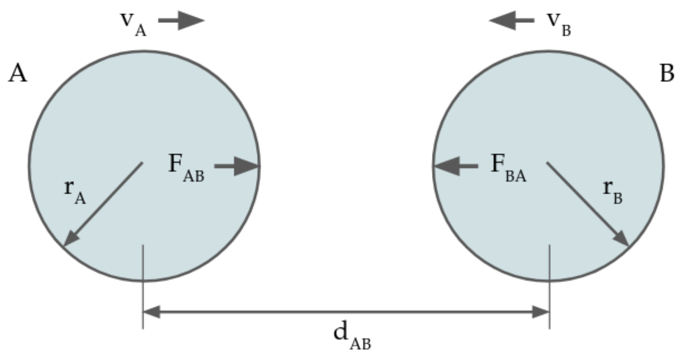
So now to be very clear about the setup I'm working with:

Hardware	Lenovo Thinkpad T480
Memory	16 GiB
Processor	Intel Core i7-8650U@1.90GHz x 8
Graphics	Mesa Intel UHD Graphics 620 (KBL GT2)
Disk Capacity	512 GB
OS	Ubuntu 22.04.2 LTS, 64-bit
GNOME version	42.5
Windowing system	Wayland
Python version	3.10.6
Numpy version	1.21.5
Matplotlib version	3.6.3
Numba version	0.56.4

In the spirit of supporting portability, I try to avoid doing anything aggressively exclusionary. I avoid unnecessary library imports that might inadvertently break some platforms. I avoid the latest bleeding-edge versions and brand new features that put unnecessary strain on dependency management and version maintenance. But it is with no small relief that I declare bankruptcy on guaranteeing platform independence. The deeper we get into controlling actual robots, the more the specifics of the platform will become un-ignorable.

Modeling atom-atom contact

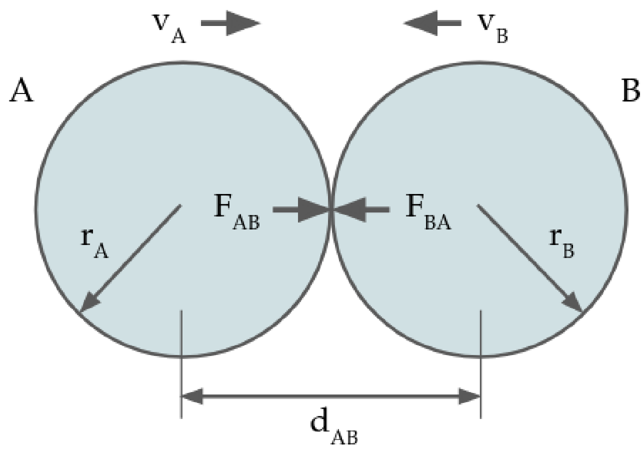
Now on to the physics! The code is going to make the most sense if we start with a couple simplified examples of what it's trying to accomplish. For the first pass, imagine there are two billiard balls that have a direct, head-on collision.



Ball A and ball B have radii r_A and r_B. The distance between them is d_{AB}. They are moving at velocities v_A and v_B. And importantly, F_{AB} is the force that ball A is exerting on ball B and F_{BA} is the force that ball B is exerting on ball A. Thanks to Newton's third law of motion, we know that F_{AB} and F_{BA} will always be equal in magnitude and have opposite signs. Until the two balls come into contact, both of these forces are zero.

$$F_{AB} = -F_{BA} = 0$$

At the last instant right before the two balls collide they are moving toward each other but still not touching. The distance between them is the sum of the two radii, but there is still no contact force.

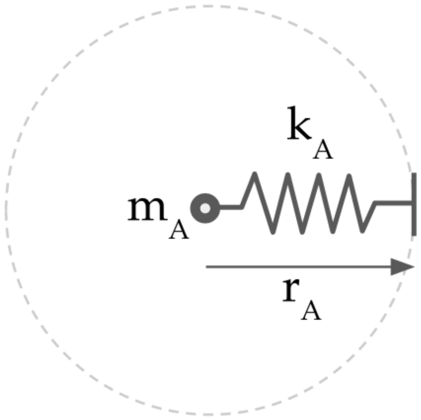


$$d_{AB} = r_A + r_B$$

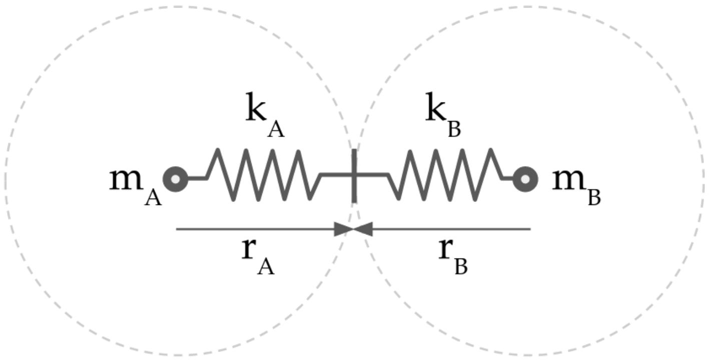
This is where things begin to get interesting. Both balls have momentum. They are heading toward each other and can't change that fact instantaneously. They have a kinetic energy that has to go somewhere. They bump into each other. For a short time they try to occupy the same space. The way they resolve this contention is that they both compromise a little. They flex. They each compress ever so slightly, temporarily deviating from their perfectly round profiles.

To picture what's going on, it's helpful to imagine balls with more give, like a tennis ball or racquetball. When you squeeze two of these together with the force of your hands, you can see how they flatten to accommodate each other. Billiards do the same, but because they are so stiff the deformation is too small for us to perceive.

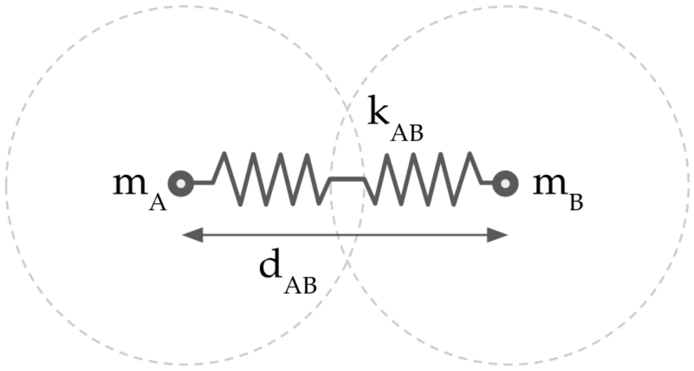
Now that we know what's going on, we can write a set of equations to describe it. For modeling the types of collisions we expect to see, it's reasonably accurate to describe our billiard ball as a point mass at its center attached to a spring, with an uncompressed length of r. The stiffness of that spring, k, represents how rigid the ball material is. It tells us exactly how much force it takes to deform the ball by a certain amount, expressed in units of force per distance.



When two balls first collide, their two centers of mass m_A and m_B are a distance $r_A + r_B$ apart. They push on each other through two springs stacked end to end, one with stiffness k_A and one with k_B.



This picture can be simplified a little bit by pretending the two springs are one.



The resting length r_{AB} of the combined springs becomes the sum of their individual unstretched lengths.

$$r_{AB} = r_A + r_B$$

The amount of compression the spring experiences is the difference between its unstretched length and the actual distance between the two point masses.

$$\text{compression}_{AB} = r_{AB} - d_{AB}$$

This expression is set up so that compression will always be non-negative. Whenever $d_{AB} > r_{AB}$ the two balls are not touching, the spring sits at its full resting length, and the contact model no longer applies. When this is the case, compression is set to 0 regardless of d_{AB}.

The stiffness of two springs in series is the reciprocal of the sum of the reciprocals of their individual stiffnesses.

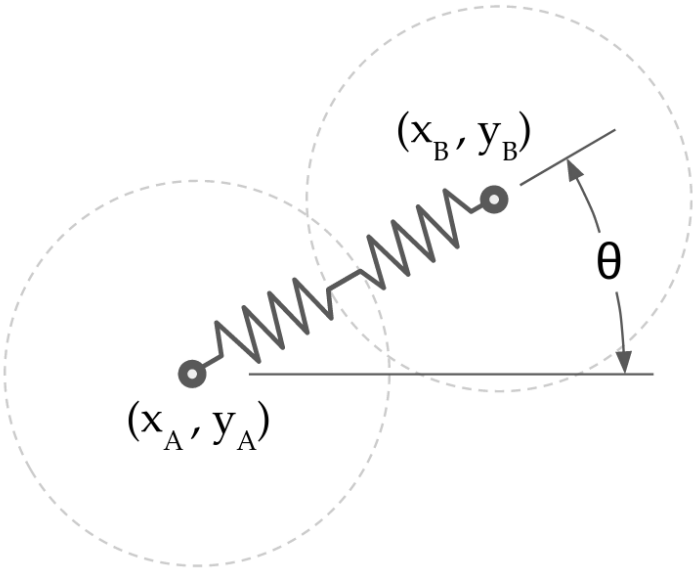
$$k_{AB} = \frac{1}{\frac{1}{k_A} + \frac{1}{k_B}}$$

Finally we are at a place where we can calculate the spring force.

$$F_{AB} = -F_{BA} = \text{compression}_{AB}\, k_{AB}$$

The force acts along the line connecting the two balls' centers to push them apart. In our example, the force of ball B on ball A, F_{BA} pushes ball A to the left and the force of A on B, F_{AB}, pushes ball B to the right.

When extending this model to two dimensions, we have to do some trigonometry to keep the forces pointed in the right direction. Consider the case where the line of contact between the balls is no longer parallel to the x-axis, but is rotated counterclockwise from it at some angle, θ, and the centers of balls A and B are at positions (x_A, y_A) and (x_B, y_B) respectively.



The distance between centers becomes the square root of the squared difference in each dimension.

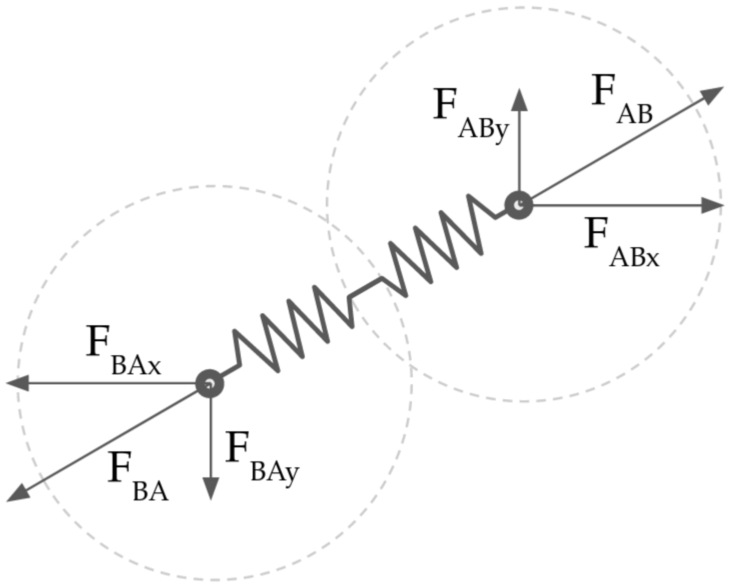
$$d_{AB} = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}$$

The interaction force is still computed in the same way, but afterward we have to break the interaction forces into their x- and y- components.

$$F_{ABx} = F_{AB}\cos(\theta)$$
$$F_{ABy} = F_{AB}\sin(\theta)$$

which can also be written as

$$F_{ABx} = F_{AB}\,(x_B - x_A) \,/\, d_{AB}$$
$$F_{ABy} = F_{AB}\,(y_B - y_A) \,/\, d_{AB}$$



This formulation is particularly nice because it avoids the intermediate computation required to find the angle. The corresponding forces for F_{BA} are, as always, the same magnitude as the F_{AB} components, but of opposite sign.

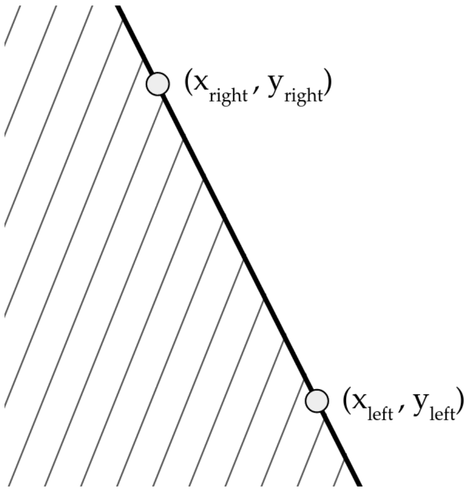
While it might seem like a small victory, describing how two circular objects interact physically actually gets us pretty far. In the same way that tiny atoms compose all matter, it turns out you can approximate nearly everything as a combination of circular pieces. We'll come back to demonstrate this process later. To capitalize on this concept moving forward, I'll refer to these circular building blocks as atoms.

While we stopped at two dimensions, you might be able to picture how to extend atom-atom interaction to three dimensions, and calculate the forces between elastic spheres running into each other. Everything else we'll do going forward can be extended to three dimensions too, but we're not going to take it that far. If you're feeling inclined to pick up where this chapter leaves off, please do! Creating your own 3D simulations would be an incredible way to develop your intuition for the principles involved while building something pretty dang cool. As always, I'll be happy to reshare your successes.

Modeling atom-wall contact

One trick that will be handy to add to our bag is to develop some math and code to represent atoms bouncing off of walls. While we could describe a wall as a lot of atoms, all tightly arrayed in a line, it turns out to be way more efficient to describe it as a single straight line. And while we could describe it as having some stiffness of its own, it works out well to model it as a hard constraint, having infinite stiffness.

Two points define a line. The input arguments in the `add_wall()` method of **walls.py** include the x- and y-coordinates of two points for doing this. In our two-dimensional simulation they tell exactly where the boundary of the wall sits along its entire length.



The only other detail we need to describe our wall is to specify which side of the line is full of bricks and stones and which is open air. This we can infer by convention. For an observer staring at the wall, one of the points will fall on the left and one on the right.

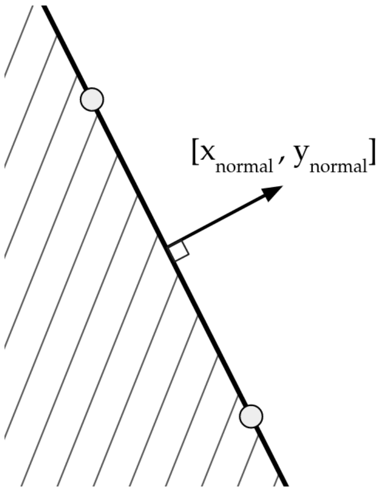
Armed with this, we know everything we need to fully define a wall. An important characteristic of any half-plane, like a wall, is a normal vector–a vector of length 1 that is perpendicular to the wall. It can be written as $[x_{normal}, y_{normal}]$ or $x_n\hat{i} + y_n\hat{j}$. An arrow from (0, 0) to this pair of x- and y-values points directly away from the wall.

The distance between the left and right points, d_{LR}, is given by:

$$d_{LR} = \sqrt{(x_{left} - x_{right})^2 + (y_{left} - y_{right})^2}$$

The components of the normal vector are given by:

$$x_n = \frac{y_{right} - y_{left}}{d_{LR}} \quad , \quad y_n = \frac{x_{left} - x_{right}}{d_{LR}}$$



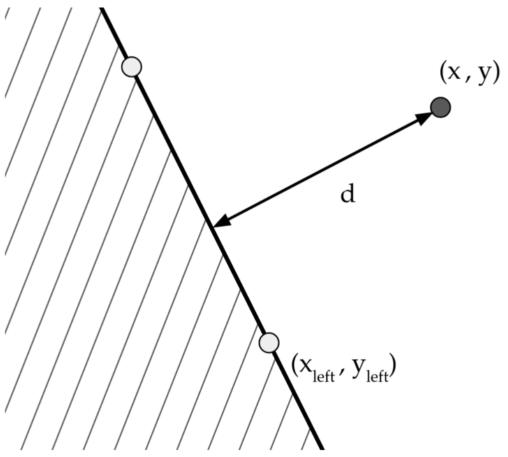
These are attributes of the wall that never change so they're calculated when the wall is first created in the function `calculate_wall_normal()`. The Python for this looks similar to the equations above.

```
dist_lr = (
        (x_left - x_right) ** 2 +
        (y_left - y_right) ** 2
) ** 0.5
x_normal = (y_right - y_left) / dist_lr
y_normal = (x_left - x_right) / dist_lr
```

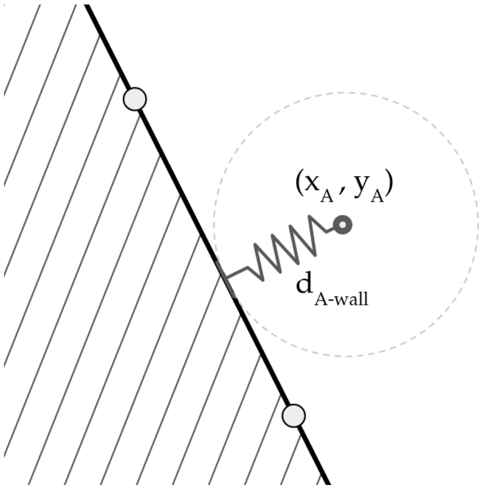
body.py at tyr.fyi/5files

Having the normal gets us a long way toward being able to calculate the distance from the wall to the center of a ball. Starting from the Wikipedia reference on Distance from a point to a line, we can derive a distance calculation consisting of two subtractions, two multiplications, and an addition.

$$d = (x - x_{left})x_n + (y - y_{left})y_n$$



Although (x_{left}, y_{left}) is called out in the equation, you can substitute in any point on the line and get the same answer. I think that's pretty cool.



Being able to find the distance from the wall to the center of a ball lets us then calculate the amount of compression of the virtual spring representing the ball's stiffness, and from that the interaction forces in the x- and y-directions similarly to how we did it with two balls colliding. Having the components of the normal vector precomputed ends up being convenient.

$$F_{wall\text{-}A} = (r_A - d_{A\text{-}wall})\, k_A$$
$$F_{wall\text{-}Ax} = F_{wall\text{-}A}\, x_n$$
$$F_{wall\text{-}Ay} = F_{wall\text{-}A}\, y_n$$

With this step complete, we are now able to tally up all the forces on an atom due to walls and those due to other atoms.

This total force is exactly what gets used to calculate the atom's acceleration, velocity, and position for a very short time into the future via numerical integration. We've come a long way! This is a good place to pause and enjoy the view.

The view really is spectacular.

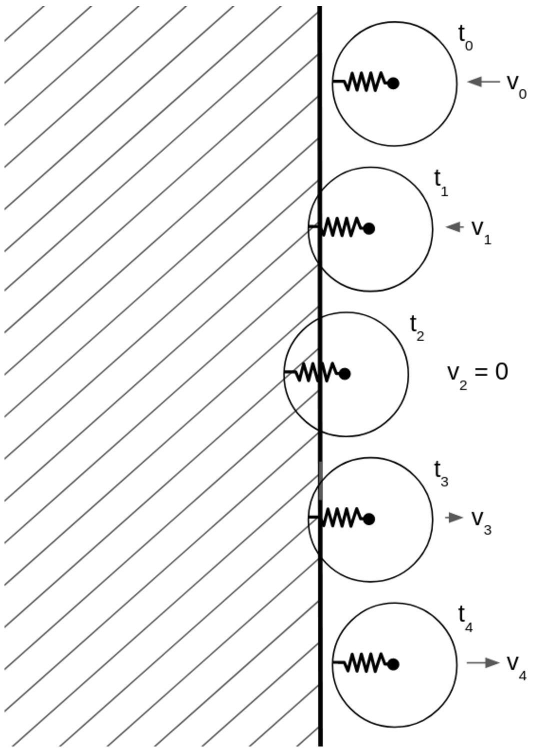
The perils of slicing time

But don't get too comfortable. We still have some hills to climb before we summit.

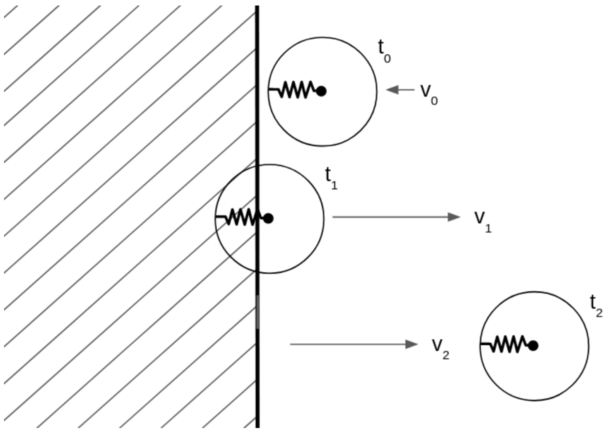
As Yogi Berra observed, "It's tough to make predictions, especially about the future." As we illustrated earlier in painful detail, numerical integration makes predictions about the future. Not very far into the future–just one time step–but it opens the door just a crack, enough to let some weirdness creep in. It makes the implicit assumption that velocity will remain constant for the duration of the next time step. Intermittent physical contact (objects bumping into each other) break this assumption.

An instant before an atom hits a wall it is happily sailing through space. No forces are acting on it. It has no reason to believe that will ever change. The simulation cycles through another time step. The atom continues on its current trajectory and hops forward. Suddenly, on the following time step it discovers that it has hopped part way into a wall. This is mostly OK. Our model prepared for this. The atom acts like a spring and a reaction force is generated, pushing the atom back away from the wall.

When time is sliced very thinly, this process happens smoothly. The atom may stay in contact with the wall for several time steps while its kinetic energy is converted to potential energy, compressing its internal spring, and then converted back into kinetic energy, bouncing the atom back away from the wall.



But when time is sliced too thick–when time steps are long compared to the dynamics of the model–this process gets choppier. Having time steps that are too long allows atoms to coast too far into the future without experiencing the consequences of their actions. It can let an atom sail halfway into a wall before it experiences the repulsive force that will send it the other way. As a result, that force can end up being much greater than it would have been, had it kicked in at the appropriate time.



Teleporting through solids might be normal in the quantum realm, but that's not where we're operating. A good mechanics simulation will represent time at a high enough resolution to avoid this.

When watching a simulation, this effect becomes noticeable as a jumpiness in objects' motion. Atoms will bounce off a wall faster than they approached it, as if the wall were a hot skillet or the atoms were over-caffeinated. Under a gentle gravity, atoms will fail to come to rest on the ground, but instead will jitter, especially if there are a number of them piled on top of each other. This is aesthetically unappealing and exhibits low fidelity to the physics we hope to model.

Correcting for the glitchiness of discrete time

There are a few ways to address this that I know of. The simplest is to increase the frequency of the simulation clock, proportionally decreasing the duration of the time step. The thinner slicing of time that results can smooth out the interaction, if you get the time steps short enough.

How fast the simulation clock needs to run depends entirely on the accelerations involved. Accelerations get larger when masses are smaller and stiffnesses are higher, but it all comes down to accelerations–the rate of change of velocity. The less the velocity changes from time step to time step, the less error is introduced by the simulation's predictions of constant velocity throughout the timestep. And if velocity is changing too fast for this assumption to be valid, chopping it in half is likely to help the situation.

I can only offer a rough rule of thumb. It's based on the Nyquist-Shannon Sampling Theorem, which says that the sample rate needs to be at least twice the highest frequency of interest in a signal. For our simulation, it means that the time slice needs to be no more than half the period of oscillation of the fastest resonance we want to represent. For our mass-spring models, the period of resonance is

$$T = 2\pi\sqrt{\frac{m}{k}}$$

According to this guideline, the period of the simulation clock should then be

$$T = \pi\sqrt{\frac{m}{k}}$$

for the smallest mass m and largest stiffness k in our simulation. However, this guidance applies to coupled systems, capable of vibration. For intermittent contact where objects are colliding with each other, the transition from free motion to contact is instantaneous. I don't know for sure how that affects the requirements on sampling rate, but my intuition is that it may require even a bit faster sampling to cover it well.

For an empirical point of reference, in the example simulation I mentioned at the beginning of the chapter, we are working with an atom mass of 0.01 kg and a stiffness of 1,000 N/m. Substituting these into the equation above gives a sampling period of 0.01 seconds or 10 milliseconds, corresponding to a simulation frequency of 100 Hz. But when I try to run the simulation at 100 Hz it is quite jittery and fails to settle down at all. At 1000 Hz (ten times the

Nyquist frequency) it behaves much better, but still with noticeable jitter.

Side note: Sampling rates, numerical integration, and simulations are HUGE topics and a lot of people have spent a lot more time than me thinking about them and trying things out. Please keep in mind that this chapter is not an attempt to show the best way to do all of this. In some cases it may not even be a good way. It is intended only to highlight the issues and concepts involved and to show some examples of how they can be navigated. If you want to go deeper, you absolutely should! There are other resources, tools, and methods out there that are far more sophisticated that what we'll build here.

Increasing the rate of the simulation clock, and thereby decreasing the duration of the time steps, can only take us so far. Pretty soon we run out of processor clock cycles. I'm working with a mind blowing 1.9 GHz CPU clock. (I remember it being a big media event when CPU clocks broke 100 MHz, so anything over 1 GHz still feels unreal.) That's 1.9 billion CPU cycles. With a simulation rate of 1 kHz, that leaves 1.9 million CPU cycles per simulation cycle. If each arithmetic operation takes a dozen or so CPU cycles and each atom interaction takes a dozen or so arithmetic operations, then that brings the ballpark estimate for the total number of atom-atom interactions that can be modeled down to around 13,200. That sounds like a lot until you consider that, at least in the naive implementation, we have to consider every atom interacting with every other atom in every time step. 13,200 total interactions only covers 115 atoms interacting with each other. ($13{,}200 \cong 115^2$) The scope of what we can simulate in real-time got real small real quick.

If we weren't constrained to keep up with the wall clock there are a lot of other approaches we could use. It's typical to raise the simulation rate as high as necessary to get good results. It's not uncommon for some of the largest supercomputers in the world to spend weeks simulating just a few microseconds of some critical phenomenon. Most physics simulations ignore the wall clock entirely. By holding ourselves to real-time, we are playing in hard mode.

Another way to level up simulations to industrial grade is to use more sophisticated numerical integration. If you allow yourself to look forward in time just a little bit, there are methods for getting more accurate integration. Or if you assume mostly smooth changes in acceleration, you can calculate estimates of higher derivatives and use those to get more accurate integration. Or if you can take the CPU cycles to perform integration at multiple timescales, compare the results, and adapt your simulation clock rate accordingly as in the popular Runge-Kutta methods, you can get more accurate integration. Here is an overview of a handful of other methods[2] known to give better results than simple forward integration. Most of these would require a bit of deviation from real-time simulation, or at least introduce a delay, but it's helpful to know that there are many ways up this particular mountain.

[2] Integration, Steve Rotenberg, CSE291: Physics Simulation, UCSD, Spring 2019

Since keeping the simulation running in sync with the wall clock is important to us, and because I don't want to go too far down the methodology rabbit hole, I focused on cheap tricks for keeping the simulation well behaved. A reliable one is to decrease the stiffness and increase the mass of the atoms. This worked OK, but not great. It succeeded in preventing the atoms from shooting across the screen like bullets after a poorly timed collision, but they still never really came to rest. The jitter remained. Even worse, they lost their clean bouncing behavior and moved visibly into each other, like squishy foam balls. It's fine to program a simulation for that behavior, but if that's all it can ever offer you, that's really unsatisfying.

Another hack that worked better was instituting an absolute velocity cap. Imagine if the speed of light were one meter per second. It totally eliminated atom launches, and even quieted most of the jitter in atoms at rest. But this solution was ultimately unsatisfying too. The jitter never entirely went away. And atoms often bounced off walls with noticeably more speed than they had approached with, even though it was limited to 1 m/s. And most of all, it felt like a particularly ugly hack. It felt like giving up on the goal of making a plausible mechanics simulator and declaring failure.

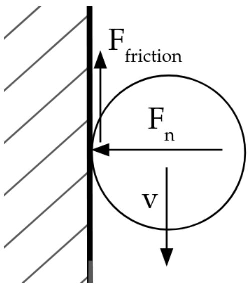
One good thing did come out of the hard velocity limit hack, though. The notion that energy could be sucked from the atoms during their interactions with each other and walls felt like the right track. That's where the problems were originating. What's more, losing energy during interactions is directly motivated by well studied physical phenomena–sliding friction and inelastic collisions.

Friction and inelasticity

The most common model for sliding friction is refreshingly simple. When two objects are in contact and one is sliding along the other, the friction force is proportional to the contact force between them. The greater the interaction force, the greater the sliding friction

$$F_{friction} = \mu_f F_n$$

The coefficient of proportionality is the friction coefficient, μ_f. It controls how high the friction force is for any given contact force. Very slippery interactions, like a melting ice cube sliding across a tile floor, have a small μ_f, like .02 or even lower. Very grippy interactions, like rubber on concrete or unlubricated aluminum on aluminum have a μ_f of 1.0 or higher.



The friction force always acts in the direction opposite the relative velocity of the two objects. If the atom is moving downward against a vertical wall, friction will resist that motion with a counteracting force pushing upward. That resistance will serve to counteract the atom's downward momentum somewhat. It will slow the downward motion. If the friction is great enough it can even bring the sliding of the atom to a complete stop.

To represent this contrarian behavior more precisely in our equations, we can include a *sign*() function to get the direction of the velocity. If velocity is positive, *sign* = 1; if velocity is negative, sign = -1; and if velocity is 0, so is *sign*.

$$F_f = - \mathit{sign}(v)\ \mu_f F_n$$

And since we are working two dimensions

$$F_{fx} = - \mathit{sign}(v_x)\ \mu_f\ |F_{ny}|$$
$$F_{fy} = - \mathit{sign}(v_y)\ \mu_f\ |F_{nx}|$$

In this form it is readily recognizable in the Python code, in the `wall_forces_numba()` function of **body.py** that handles atoms interacting with walls.

```
f_x_walls_sliding = (
        -sliding_friction *
        np.abs(f_y_walls_contact) *
        np.sign(v_x_atoms))
f_y_walls_sliding = (
        -sliding_friction *
        np.abs(f_x_walls_contact) *
        np.sign(v_y_atoms))
```

body.py at tyr.fyi/5files

The pieces of the equation are all there, but in a different order. There is a similar pair of expressions in **body.py**'s `body_interactions_numba()` that handles friction between colliding atoms.

The `sliding_friction` variable represents the coefficient of sliding friction, μ_f, between the atom-and-wall or atom-and-atom. Coefficients of friction are the result of multiple physical phenomena, operating at different length scales, and depend on the combination of materials involved. This table isn't exhaustive, but gives a good taste of how they vary with different pairings.

In the simulation code I took a shortcut and, rather than listing a separate coefficient of friction for every possible pairing of atoms with each other and with walls, I called out a coefficient of friction for each atom and each wall separately. Then for every interaction, I took the average of the two and used it as μ_f between the two objects, allowing the properties of both materials to contribute to their sliding friction behavior.

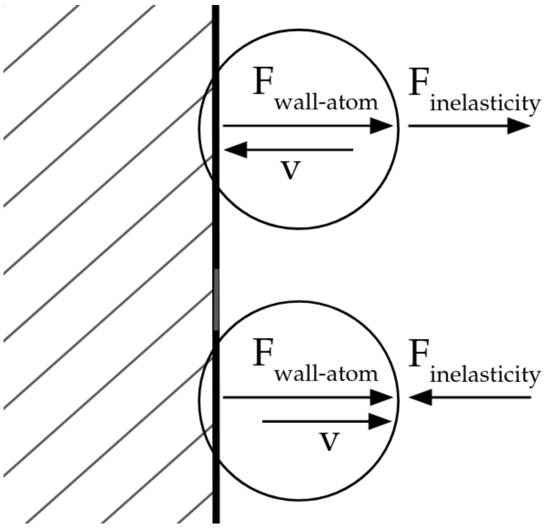
The final result of this is a mechanism for bleeding off energy from the simulation that preferentially targets fast moving objects. It tends to slow objects down so they are less likely to glitch into another object, and when they occasionally do, that's OK because there is a way to tame that excess velocity and bring it back into a plausible realm. This all combines to help objects move in a way that is acceptable to the eye. And the best part is that it's motivated by actual physical phenomena, so we avoid aesthetically offensive hacks like hard velocity limits.

In the physical world, friction diverts the kinetic energy of a moving object into other places and forms, like heating up the surfaces where they are in contact. Tiny particles get broken off where the local contact pressure is too great. Microscopic nubs grab, flex, and release. Atoms briefly bond, altering the waveforms of their electrons, only to be pulled apart again and snap to their original configuration. These phenomena all turn some of that object's kinetic energy into heat.

Another valve through which energy gets vented is collisions. A fully elastic collision is one where no kinetic energy is lost, but there are also inelastic collisions (which are all of them if we're being honest) in which some of that energy gets turned into heat. Picture the difference between throwing a rubber bouncy ball at the wall (nearly elastic) and throwing a lump of wet modeling clay (inelastic). That's a useful range of behaviors to be able to represent in a physical simulation.

From my brief search, there's no universal way to represent collision elasticity in computational models of collisions. Luckily, thanks to our work with friction, we're three-quarters of the way there. We can appropriate the mathematical structure of our friction model and, with a couple tweaks, get an elasticity model out of it.

$$F_e = -\,\mathit{sign}(v)\ \mu_e F_n$$



The differences between the elasticity model and friction are primarily changes in direction. The velocity of interest here is the velocity in the direction of the normal force, rather than the velocity perpendicular to it. The coefficient of inelasticity, μ_e, varies the effect between fully elastic ($\mu_e = 0$) and completely inelastic ($\mu_e = 1$). And importantly, there is no longer an absolute value operator giving us only the magnitude of the normal force. When calculating elasticity, the direction of F_n acting on the object matters as much as its magnitude.

These two equations spell out the inelasticity along each axis.

$$F_{ex} = - \mathit{sign}(v_x)\ \mu_e F_{nx}$$
$$F_{ey} = - \mathit{sign}(v_y)\ \mu_e F_{ny}$$

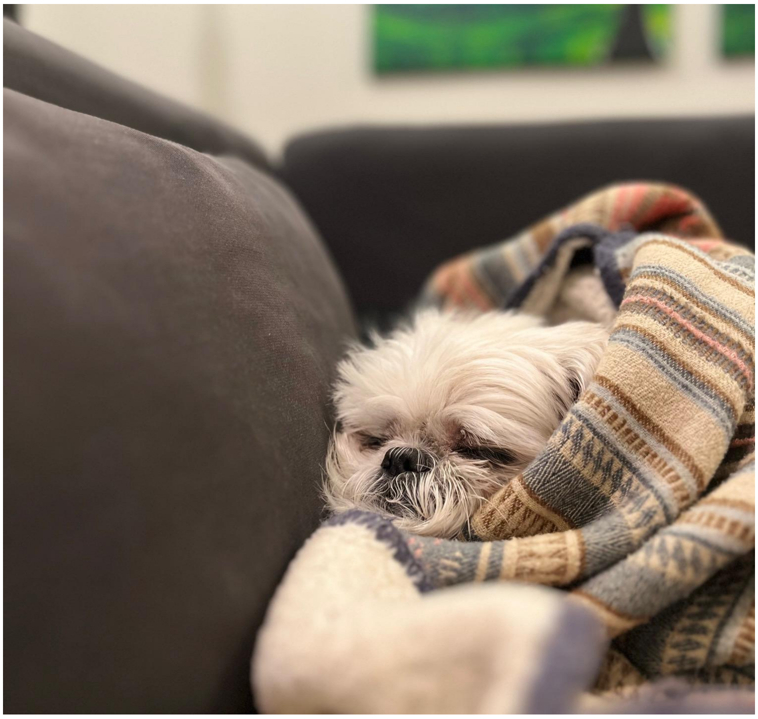
In this form it's clearer how $\mu_e = 1$ can lead in an inelastic collision. After an atom has pushed as far as it can go up against a way and reverses direction, v_x and F_{nx} will have the same sign, and simplifying the equation yields $F_{ex} = -F_{nx}$. After removing the inelasticity effect, there will be no force left to propel the atom back away from the wall. It will hit the wall and remain, like a spitball.

The Python code for simulating this is readily recognizable from the equations. For two atoms acting on each other:

```
f_x_ab_inelastic = (
        -inelasticity * f_x_ab_contact * np.sign(d_v_x))
f_y_ab_inelastic = (
        -inelasticity * f_y_ab_contact * np.sign(d_v_y))
```
body.py at tyr.fyi/5files

Where `f_x_ab_contact` is the contact force of atom A on atom B in the x-direction, `d_v_x` is the difference in velocities of atom A and atom B in the x-direction, `inelasticity` is the coefficient of elasticity between the two atoms, and `f_x_ab_inelastic` is the resulting adjustment to the contact force due to inelasticity effects. The equations for inelasticity in atom-wall collisions look similar, but make use of the walls' precomputed normal vectors to shortcut calculation of the direction of the normal force. As with friction coefficients, each atom and wall has a coefficient of elasticity assigned to it in **config.py** and the inelasticity of the interaction is determined by their average value.

The symmetry between inelasticity and friction calculations has a nice feel. After all, they are both mechanisms for physical imperfections and micro-scale phenomena to convert kinetic energy to heat energy. Why wouldn't they be the same?! I don't know if this is actually a physically justifiable way to model collision inelasticity, but it works within our computation budget, is straightforward to explain, and produces the desired effect in objects' motion and stability. I'm going to call it a win.



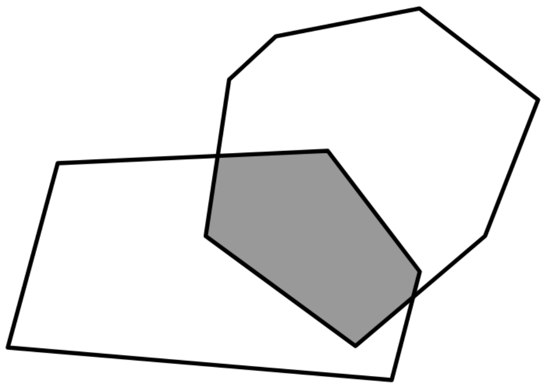
Is the math part over yet?

Non-circular objects

Circles are great, but it turns out there are lots of things in the world that aren't circles. Most of them in fact. (Unless you are a physicist, in which case even cows are spherical.) This fact raises the question of how to extend the work on circular atoms to objects that are not circles.

Before proceeding, it's helpful to understand that there is no one right way. There are several approaches, and they all come with trade-offs. When choosing a method, it's not helpful to ask "Which one is best?" Instead ask "Which one is best for what I want to accomplish?"

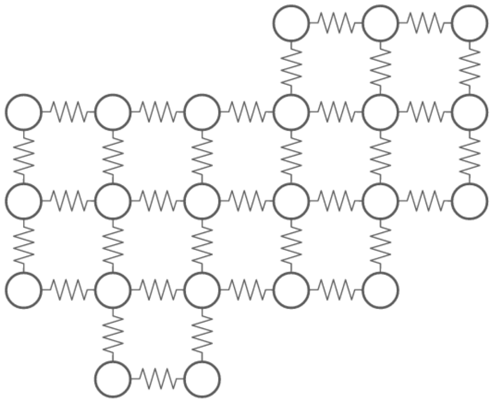
One popular method for two dimensional mechanics simulations like ours is to create all objects as polygons. There is an efficient way to calculate whether two polygons are touching with the impressive name of Hyperplane Separation Theorem (also known as the Separating Axis Theorem).



Unfortunately this approach requires us to work with convex polygons only (no divots, depressions, or cavities in the perimeter) and doesn't tell us how much the two polygons overlap (which we would need to calculate the contact forces between them). There are other methods that can also handle concave polygons and can calculate the overlap (but these come at the expense of more computation). There are simplifications and approximations that streamline these computations (but these come at the expense of additional conditions and assumptions). And so on, down the rabbit hole. In short, this is a common approach, and it can work well for many applications, but simplicity and generality are not its strengths.

For our work here we are going to take a different path and sidestep all of polygons' fascinating algorithmic complexity. Since we already know how to work with circular atoms, we can use the old mathematician's trick of reducing a new problem to a previously solved problem: we are going to build everything out of circular atoms.

The most consistent way to implement this is by creating something like a crystal—a collection of atoms, where each one has a well defined preferred position, but is connected to all of the others by springs, so that the whole assemblage can flex and compress and vibrate. This makes use of all the math we’ve already done. We calculate the distances between atom centers the same way. The compression of the springs connecting them and the resulting forces are computed the same way too, with the exception that now we allow compression to be negative–the springs can stretch as well as compress–which can result in negative normal forces. In the code this is as straightforward as removing the line with `np.maximum(0, ...)` that limits the compression to be positive.



This approach is appealing in theory because it makes use of the work we've done already with minimal changes. It has a generality, an aesthetic sense that there is one reusable solution to all the problems. Unfortunately, it comes with a considerable computational cost. Now that objects can be composed of many atoms, and the number of atoms that we can feasibly work with is limited, our ability to simulate complex environments has been limited even further. Previously, we made a ballpark estimate that I could simulate about 100 atoms on my current hardware. If I start constructing objects out of a dozen or so atoms each, that means I'm working with a single digit number of objects. And the problem only gets worse the more complex those objects become.

Another gap in which the lovely theory doesn't quite reduce to workable practice is that the internal stiffness between atoms in the crystals has to be quite high to get nearly rigid objects. A small amount of flexion and vibration is reasonable, but when a simulated robot arm begins flopping around like a noodle, that's a problem. Increasing the stiffness to what would be necessary for rigid body behavior makes the simulation unstable. The critical frequencies involved scale up as the square root of the stiffness and soon exceed the capacity of our simulation at the update frequencies we can support. We are left choosing between non-rigid objects or unstable simulations. Not an appealing choice.

In what has become a recurring theme in simulation, we can work around this by making some more simplifying assumptions. Instead of a crystal, where atoms within an object are allowed to push and pull on each other, we can assume that the object is perfectly rigid, that no matter what happens to it, its atoms will always remain in precisely the same configuration. The object as a whole can tumble and bounce, shake and spin, but its shape will never change in the slightest.

This necessitates several changes to how the simulation works. All of the atoms in a rigid body are still allowed to interact with walls and with all the atoms in other rigid bodies in the same way as before. All the interaction forces of collision and inelasticity and friction are calculated for each atom in the body. The difference comes in what happens next. The forces in the x- and y-directions on all atoms are summed to get the total x- and y-forces on the body. (Code snippets to follow are taken directly from **body.py**'s `update_positions_numba()` function.)

```
f_x = np.sum(f_x_atoms)
f_y = np.sum(f_y_atoms)
```

And the torques resulting from all those forces are summed to get the total torque acting on the body. (The torque from a force is given by $\tau = Fr$, where r is the moment arm of that force, the shortest distance from its line of action to the body's center mass.)

```
torque = (
    np.sum(f_y_atoms * (x_atoms - x)) -
    np.sum(f_x_atoms * (y_atoms - y)))
```

Then the position updates occur at the body level. Its x- and y- accelerations are calculated from net forces on the body's mass.

```
a_x = f_x / m
a_y = f_y / m
```

And its angular acceleration is calculated from the net torque on the body's moment of inertia.

```
a_rot = torque / rot_inertia
```

From there, the x-, y-, and rotational-velocities are calculated using our integration skills.

```
v_x += CLOCK_PERIOD_SIM * a_x
v_y += CLOCK_PERIOD_SIM * a_y
v_rot += CLOCK_PERIOD_SIM * a_rot
```

Likewise with x-, y-, and angular-position.

```
x += CLOCK_PERIOD_SIM * v_x
y += CLOCK_PERIOD_SIM * v_y
angle += CLOCK_PERIOD_SIM * v_rot
```

Then the positions and velocities of every individual atom are backed out, relying on their local positions within the body, their x- and y- distance from the center of mass when the angle is zero, `x_atoms_local` and `y_atoms_local`. First a rotation transformation is applied to account for the body being at some non-zero `angle`, resulting in the relative x- and y-positions `x_atoms_rel` and `y_atoms_rel`.

```
x_atoms_rel = (
    x_atoms_local * np.cos(angle) -
    y_atoms_local * np.sin(angle))
y_atoms_rel = (
    y_atoms_local * np.cos(angle) +
    x_atoms_local * np.sin(angle))
```

Then a translation transformation is applied to account for the position of the body's center of mass at `(x, y)`.

```
x_atoms = x + x_atoms_rel
y_atoms = y + y_atoms_rel
```

To get each atom's velocity, first the contribution of the body's rotational velocity is taken into account, multiplied by the moment arms of each atom's relative x- and y-position.

```
v_x_atoms_rel = -v_rot * y_atoms_rel
v_y_atoms_rel = v_rot * x_atoms_rel
```

Then that relative velocity is summed with the translational velocity of the body's center of mass, `v_x` and `v_y`.

```
v_x_atoms = v_x + v_x_atoms_rel
v_y_atoms = v_y + v_y_atoms_rel
```

If you've been keeping track, we've introduced a few more constants that we need to keep track of: the total mass of the body `m`, the rotational inertia of the body `rot_inertia`, and the positions of the atoms within the body relative to its center of mass, `x_atoms_local` and `y_atoms_local`. These constants can be calculated once during the initialization of the `Body` object, so they don't weigh down the computations at each iteration.

The mass of the body is the sum of the masses of all its atoms.

```
self.m = np.sum(self.m_atoms)
```

Initially the x- and y-positions of the body's atoms can be given with respect to an arbitrary point, like a lower-left atom, for instance. To create a set of positions with respect to the center of mass, first the center of mass is calculated, then those initially specified positions are offset so that they use the center of mass as an origin.

```
x_c = np.sum(self.x_atoms * self.m_atoms) / self.m
y_c = np.sum(self.y_atoms * self.m_atoms) / self.m
self.x_atoms_local = self.x_atoms - x_c
self.y_atoms_local = self.y_atoms - y_c
```

Finally, those local x- and y-coordinates are used to get the distance of each atom from the center of mass and are combined to find the total rotational inertia.

```
self.rot_inertia = np.sum(self.m_atoms * (
    self.x_atoms_local**2 + self.y_atoms_local**2))
```

This approach saves computation on two fronts, one obvious and one more subtle. Moving rigid bodies introduces a fair bit of complexity (and physics!) to our computations, but they end up reducing the total number of operations we have to do, at least once the objects grow to have more than a few atoms. The computations above scale linearly with the number of atoms. Back when we were considering the full crystal model where every atom could interact with every other atom, the computations scaled as the square of the number of atoms. The more atoms in a single body, the more computation we save by not allowing them to push and pull on each other.

The other source of computational streamlining comes from a shortcut we can now employ. At this point we only have to consider atoms from one body interacting with another. When those two bodies are very close, we have to consider all possibilities–every atom from one body potentially interacting with every other atom from the other. But when those two bodies are far enough away from each other, we can skip all that and just say that we know for certain that none of their atoms are touching each other. It's like putting your squabbling children in separate rooms. You know they're not going to be poking each other or invading each other's imagined envelope of personal space.

In the simulation this is done with a bounding box, an imaginary box that completely encompasses the body. It has four sides, one each for representing the lowest and highest x- and y- extent of the body. This is the full function that gets called once per simulation clock cycle to keep it up to date with the body's latest movements.

```
def update_bounding_box(self):
    self.bounding_box["x_min"] = np.min(
        self.x_atoms - self.r_atoms)
    self.bounding_box["x_max"] = np.max(
        self.x_atoms + self.r_atoms)
    self.bounding_box["y_min"] = np.min(
        self.y_atoms - self.r_atoms)
    self.bounding_box["y_max"] = np.max(
        self.y_atoms + self.r_atoms)
```

body.py at tyr.fyi/5files

Then during the interaction calculations we can first check whether two bodies are even in the same neighborhood, whether the bounding boxes `bb_a` and `bb_b` overlap at all. If there is no overlap, there is no need to do any further computation on the two bodies' interactions, which, needless to say, is marvelously efficient. The fastest code is the code that never runs.

The method for testing the overlap of two bounding boxes is itself pleasingly concise. It starts by assuming overlap, then checking for any one of four conditions that would invalidate it–whether one box's smallest x extent is greater than the other's largest and vice versa, and then doing the same for the y-direction.

```
overlaps = True
if (
    bb_a["x_min"] > bb_b["x_max"]
    or bb_b["x_min"] > bb_a["x_max"]
    or bb_a["y_min"] > bb_b["y_max"]
    or bb_b["y_min"] > bb_a["y_max"]
):
    overlaps = False
```

body.py at tyr.fyi/5files

Using this proximity checking trick helps us skip a truckload of unnecessary computation.

I love skipping work!

Stationary features

We haven't yet talked about how to make the pieces of a simulation that just sit there. We have figured out how to build boundary walls, but fixed ledges and towers and obstacles and free-floating sculptures are a separate problem. Luckily we have a solution that lets us simultaneously show off the flexibility of our body objects and another optimization trick.

In the Tetris-inspired simulation created for this chapter, there are six circles placed in a V-formation, dominating the space. These are atoms, and together they are a single body, just like the other rigid bodies bouncing around and off of them. They have different radii and are not contiguous, but they are still only a body. The one thing that differentiates them from the bodies that move around is that they have an attribute called `free` that is set to `False`. The atoms of this stationary body can exert forces on the atoms of other bodies, and forces can be exerted back on them, but they are not allowed to move.

The way they are locked in place is particularly satisfying because of its efficiency. The entire function content of `update_positions()` in **body.py** is wrapped in an `if self.free:` check. If any body is not free, its position update is skipped entirely and not a single CPU clock cycle is wasted on it. It's a small win, but a palpable win nonetheless.

This approach also illustrates how the atoms in a body need not be touching each other. They can be separated by small gaps or large. They can also be overlapping. When this ability to arbitrarily position them is combined with the ability to vary the radius of each atom, it becomes possible to sculpt bodies of any shape. The flexibility of form is limited only by the number of atoms your computation budget allows you and your patience for decomposing the object down into its component circles.

Initialization

Most of the other important aspects of the code we have already discussed in previous chapters, like maintaining the timing and sustaining the quality of the animation. One last method that's worth calling out is how we initialize something complex, like a body. In order to build a body you have to know a lot of things:

- Initial x- and y-position of the center of mass of the body. (Initial angle is assumed to be zero.)
- Initial x-, y-, and angular-velocity of the body.
- Sliding friction and inelasticity coefficients for all atoms in the body.
- Initial x- and y-positions of each atom in the body, relative to some datum.
- Radius, mass, and stiffness of each atom.
- Whether the body is free to move.

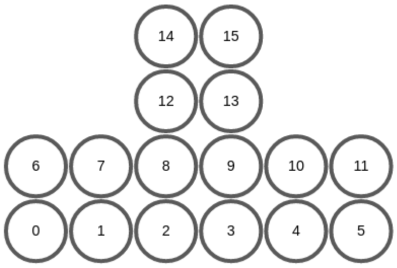
With one body to initialize this is awkward. With six bodies, this information becomes more than an inconvenience. It is unwieldy enough that it is hard to manage and understand. Some tricks to initialize bodies in an orderly manner will pay off.

There are several common ways to handle complex initializations, and they all have their benefits and drawbacks. The method that seemed to make the most sense here was to pass a dictionary containing all this information in the form of separate key-value pairs for a single body. For example, here is the initialization dictionary for the T-shaped body.

```
n_atoms = 16
x_spacing = 0.1
y_spacing = 0.1
T_BODY = {
        "id": "T",
        "x": 4,
        "y": 5,
        "v_x": np.random.sample() - 0.3,
        "v_y": np.random.sample() - 0.1,
        "v_rot": np.random.sample(),
        "x_atoms": (np.array(
            [0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5, 2, 3, 2, 3]
        ) * x_spacing),
        "y_atoms": (np.array(
            [0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 3, 3]
        ) * y_spacing),
        "r_atoms": np.ones(n_atoms) * x_spacing / 2,
        "m_atoms": np.ones(n_atoms) * 0.01,
        "stiffness_atoms": np.ones(n_atoms) * 1000,
        "sliding_friction": 0.25,
        "inelasticity": 0.3,
}
```

config.py at tyr.fyi/5files

To get the array of numbers for `x_atoms` and `y_atoms` it's helpful to have a reference diagram with each atom numbered.



The bottom is row zero and the left is column zero. From there the `x_atoms` and `y_atoms` values are the column and row number of each atom, multiplied by the spacing between rows and columns, `y_spacing` and `x_spacing` respectively.

The velocities are initialized with some random variation so that each simulation run plays out a bit differently.

By the time we're done here, the `T_body` dictionary has everything it needs to construct a new body. This all gets passed to the `Body()` class which reads it in and uses it to initialize and kick off the T-shaped object in the simulation.

The initialization dictionaries all reside in a list called `bodies` in **config.py**, since they are configuration parameters that a user could conceivably want to tweak to their liking. When **sim.py** starts up it walks through the list of bodies' initialization parameters and creates the body associated with each dictionary, one at a time.

This approach has the added benefit of separating out our simulation-specific setup information from the underlying physics engine that brings it to life. Ideally, a new user could make an entirely different simulation that bears no similarity to the bouncing Tetris shapes by changing only **config.py**. The rest of the simulation code wouldn't need to be modified. Everything for specifying the simulation would be neatly encapsulated in **config.py**.

This is a great ideal to shoot for, but in practice it's hard to achieve. Throughout the process of developing our simulation code, we've made design decisions and engineering trade-offs based on assumptions about what we would be simulating. They are baked in pretty deep.

That's OK. We have been clear from the beginning that our goal isn't a general purpose mechanics simulator, but rather a special-purpose robot simulator. That has let us focus on a particular regime of physical mechanics and hopefully represent it with enough fidelity to let us build robots. There are some pieces we haven't built yet, like how to connect two bodies to each other with a joint and apply torques to them with a virtual motor, but that will come later.

Coda

Take a second look at the video of the Tetris simulation. Knowing what you know now, it's impossible to un-see all those virtual springs between the atoms as the blocks bounce off of each other. It's a little mind bending to think about how, even though the picture is only updating 30 times per second, the positions of each atom are being updated 1000 times per second. We've created a virtual world. It's a small one, and a simple one, but it operates in a self-consistent way entirely by its own set of rules. There's a peculiar joy that comes from making something so whole.

Despite all of my good reasons for doing so, I still have reservations about not making this platform-independent. It means that you probably won't be able to run it yourself, unless you happen to have a Linux machine or are willing to do a bit of legwork to port it over to your Mac or Windows box. Even without being able to run it, my hope is that this example and the chapter behind it has struck a useful path through the conceptual countryside that is numerical simulation.

If by some chance you end up getting this to run, congratulations! It can be a good jumping off point for you. By making small changes in **config.py** you can experiment with different parameters, variations of gravity and stiffness, elasticity and mass. You can play with more fundamental settings like clock speed and explore how they affect the simulation quality. You can create new rigid bodies in whatever shape you like and leave them to interact. If you run any of these experiments, let me know! I would love to hear about them.

What's next?

This chapter has most of the secrets and tricks that make a passable physics simulation possible. There's one area that we didn't give the attention it deserves–**optimization**. As I was writing, optimization started out as a subsection of this chapter, but quickly grew too big to contain. It's a rich topic, and one that comes in handy far outside of simulations. How to make your code run faster will be the sole focus of Chapter 6.

Having a physics simulation immediately cries out for several other innovations. So far our rigid bodies are completely unconstrained. In order to make complex mechanisms, we need a way for them to be able to be attached, to be fixed to each other with a rotational joint or a sliding interface.

It would also be useful to introduce force-generating elements, like a motor or a thruster or a linear actuator. For simulation purposes, these add an extra term into our force equations that seems to come from thin air. Mathematically, it’s not hard to introduce, the subtlety comes from how to spread that force out over time, and how it changes, depending on the movement of the bodies involved.

Once our simulation has the capability to generate forces, the next logical step is to allow a human to initiate those forces. By the time we get to this point, we have created the video game. This opens up the door to interactive simulations, which is exactly what we will need if we are to simulate the full spectrum of human robot interaction.

And finally, this stage that all of this is building up to: allowing an autonomous learning algorithm to initiate forces. It is at this point that we can really tuck in to the business of developing human directed reinforcement learning.

All of this is to come in future chapters. It is exciting to see how the foundation we’ve been laying sets us up to build something grand.

Now I desperately need a nap.

Recap

Numerical integration, specifically the forward difference, makes it possible to calculate positions from velocities and velocities from accelerations. F = ma makes it possible to calculate accelerations from forces.

The **forces** are determined by the physics we want to simulate. It's worth thinking carefully about what these are and how to calculate them efficiently.

Simulation is **computationally demanding**. To support it, it's helpful to run simulation in its own process and avoid any unnecessary computational steps.

Optimization and hardware integration are **platform-specific**. I'm no longer trying very hard to make platform-independent robotics code.

Building a world out of circular **atoms** is one way to architect and physics engine. There are others. The best way to do it depends on what you want to do with it when you're done.

Forward differentiation can be **unstable** when the simulation clock is too slow compared to simulation dynamics. Even when stable, it can add energy to the system. **Friction** and **inelastic collisions** remove energy and have stabilizing effects.

Python dictionaries are a handy way to **initialize complex classes**.

About the Author

Robots made their way into Brandon's imagination as a child while he watched The Empire Strikes Back on the big screen, one buttery hand lying forgotten in a tub of popcorn. He went on to study robots and their ways at MIT and has been puzzling over them ever since. His lifetime goal is to make a robot as smart as his pup. The pup is skeptical.

To see more of his work, visit brandonrohrer.com.