Making Your Python Code Run Faster

How to Train Your Robot

Chapter 6

Brandon Rohrer

Copyright © 2023 Brandon Rohrer
All canine photos courtesy Diane Rohrer
All rights reserved

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
August 12, 2023

Making Your Python Code Run Faster

Chapter 6

In which we really get things moving

Whenever you start talking about optimization, some crusty engineer will pipe up from the back of the room quoting Donald Knuth, and declaim "Optimization is the root of all evil!" It's OK to ignore them. They're working through past trauma.

The thing is they aren't totally wrong, but they left out an important part of the quote: "*Premature* optimization is the root of all evil." The trouble with optimization is not that it's a bad thing. The trouble is that it's so much fun. It's addictive. It can quite easily become an obsession that eclipses all other concerns and takes up all the oxygen in the room.

To illustrate this, consider automobiles. There's an engine that produces torque, turns the wheels, and makes a people container move forward. The speed at which it moves can be measured, and as soon as you put a number on a thing engineers everywhere will feel the irresistible compulsion to make that number go up or down as far as it can go. While

there is nothing morally wrong with making a car go faster, when the pursuit of that goal neglects all other considerations, including the physical safety of drivers and pedestrians, it can definitely become an evil.

Speed isn't the only thing that can be measured and optimized. There is also torque, power, and acceleration. Importantly, there is also a whole collection of other things we can measure and optimize that have nothing to do with going faster —passenger carrying capacity, cost of ownership, performance on safety tests, fuel efficiency. The first step before jumping into optimization is deciding what to optimize. All of these quantities are arguably useful or beneficial in some way, but it's impossible to optimize for all of them at once. Inevitably, making one number get better will make another one get worse. It's hard to increase horsepower without decreasing fuel efficiency. Making a car perform better in safety tests usually involves more material which leads to more mass, lower acceleration, and higher construction costs. There is no free lunch. When choosing to optimize one thing, we choose to ignore or penalize everything else. This is dangerous when not done thoughtfully.

Of course, Donald Knuth was well aware of all of this. The larger context for the quote provides this nuance.

> Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.[1]

So the next time a curmudgeon blithely dismisses your optimization efforts, feel free to "well, actually" them with this.

[1] Knuth, "Structured Programming with Goto Statements". Computing Surveys 6:4 (December 1974), pp. 261–301, §1. doi:10.1145/356635.356640

It's all a game

The reason optimization is so damn fun is that it's structured like a video game. The thing you can measure, that's your score. Whether you're aiming high, like in Pac-Man, or low, like in golf, you always have a way to assess how well you're doing and whether your latest run set a new record. And because it's a software system, you can put on a serious face and pretend like you're doing Important Work, disguising the glee of your inner 13-year-old.

Code execution speed is a particularly addictive game to play, because it's often easy to iterate fast. Make a change, time of your code, make a change, time your code. Depending on your project, you can go through several iterations per minute, plenty to keep your thrill levels up.

The trick then is to find a problem where code execution speed is actually what's limiting you, where it really is the thing that needs to be optimized more than everything else. Lucky for us, physics simulations provide just such a playground.

This chapter was originally supposed to be just a section of the chapter on physics simulations, but there is so much to say on the topic, and it will have application in so many other parts of the algorithmic work that we do in the future, that it grew into a chapter of its own. To get the full context of the computations we will be optimizing, I recommend a skim through Chapter 5, but it won't be strictly necessary.

Real time physics simulation requires fast computation because we've set ourselves the high bar of keeping up with

the wall clock. There is a lot going on in even the simplest physical environment, and it's all happening extremely fast. The best any simulation can hope to do is roughly approximate it. Even 100 particles interacting with each other in real time creates a hefty computational load. Physics simulations are always thirsty for more compute capacity, and real time simulations even moreso.

This is our sweet spot! Our optimization is no longer premature, and Donald Knuth would not judge us for spending time making our code run faster. We know exactly what we're doing and why, and we can justify the game we're playing with a straight face.

Optimizing for adorableness also has its hazards.

Profiling

The best recipe I've seen for writing fast programs is this:

Chrisshy Keygen on Mastodon. Used with permission.

or restated:

1. Write slow programs
2. Find out where they are slow
3. Figure out how to speed up the slowest part
4. Go back to step 2

Step 2 is also called profiling and the tools that let you watch programs to find the slow parts are called profilers. There are several good ones to choose from, including a suite of profiling tools built right into Python called cProfile. My personal favorite is one called py-spy. Unlike cProfile, py-spy runs in a separate process so it won't trip up your Python process and introduce overhead. It can trace its lineage back to the one and only Julia Evans' work on rbspy.

This is the recipe I followed while developing the simulation code. There is a big temptation to skip step 2 and jump right

to writing fast code. The difficulty with this is that it's actually pretty tough to know what parts of your code are the slowest until you have A LOT of experience, and even then you can be surprised. You risk making your code overly complicated without fixing the slow parts. You might even make it slower by accident. Don't make me repeat the Donald Knuth quote.

Another thing to keep in mind is that your code might already be fast enough. If it does the job that needs doing without getting in your way, that's good enough. Faster code is not an inherent virtue. Fast *enough* code is all we need. In the case of our simulation, we want it to be fast enough to complete its simulation calculations within one cycle of the simulation clock. Once we get to that point we can sit back and have a cold drink.

Profiling with py-spy

When I use py-spy at the command line, one of the arguments I need to provide is the process ID (or pid), a number that identifies which process I want to spy on. To see what's currently running and get the pid I need, I use Linux's **ps** command. It returns a list that includes three Python processes.

```
$ ps -a

...
71974 pts/1    00:00:00 python3
71975 pts/1    00:00:27 python3
71976 pts/1    00:00:24 python3
...
```

Helpfully, it also shows the total CPU time taken by each. This helps to identify which is the runner process and which are the simulation and visualization processes. The runner has almost zero CPU time. The other two are comparable, so finding which one is the simulation can require trial and error.

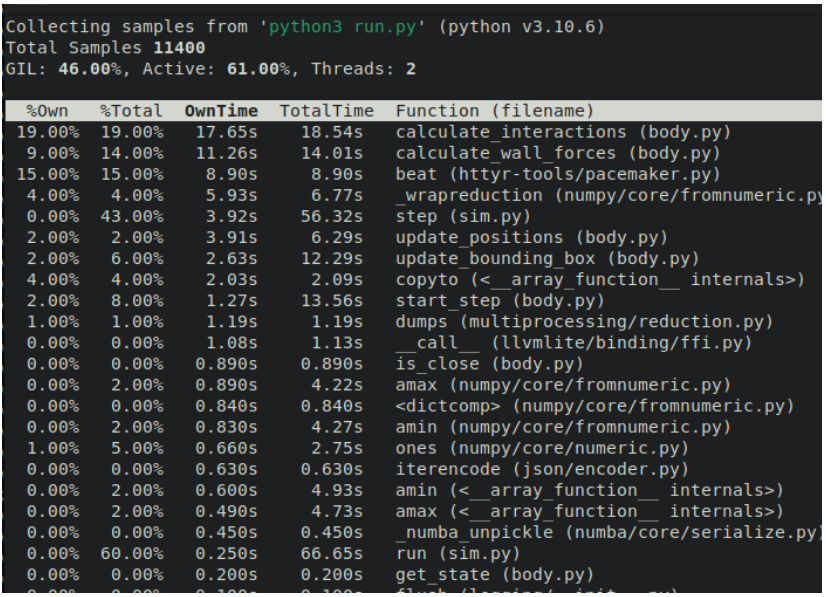
Here is the command to get py-spy to generate the "top" view of process 71975. **top** is the Linux command for surveying the complete set of processes running on the machine and the resources they are using. It is another way to get the pids of any active Linux processes.

```
$ py-spy top --pid 71975
```

Linux informs me that I have to have root permissions to do this and prompts me to modify the command to this.

```
$ sudo env "PATH=$PATH" py-spy top --pid 71975
```

This command launches a top-like view of a single process, where each row is a separate function.



```
Collecting samples from 'python3 run.py' (python v3.10.6)
Total Samples 11400
GIL: 46.00%, Active: 61.00%, Threads: 2

  %Own   %Total  OwnTime  TotalTime  Function (filename)
 19.00%  19.00%   17.65s    18.54s   calculate_interactions (body.py)
  9.00%  14.00%   11.26s    14.01s   calculate_wall_forces (body.py)
 15.00%  15.00%    8.90s     8.90s   beat (httyr-tools/pacemaker.py)
  4.00%   4.00%    5.93s     6.77s   _wrapreduction (numpy/core/fromnumeric.py
  0.00%  43.00%    3.92s    56.32s   step (sim.py)
  2.00%   2.00%    3.91s     6.29s   update_positions (body.py)
  2.00%   6.00%    2.63s    12.29s   update_bounding_box (body.py)
  4.00%   4.00%    2.03s     2.09s   copyto (<__array_function__ internals>)
  2.00%   8.00%    1.27s    13.56s   start_step (body.py)
  1.00%   1.00%    1.19s     1.19s   dumps (multiprocessing/reduction.py)
  0.00%   0.00%    1.08s     1.13s   __call__ (llvmlite/binding/ffi.py)
  0.00%   0.00%   0.890s    0.890s   is_close (body.py)
  0.00%   2.00%   0.890s     4.22s   amax (numpy/core/fromnumeric.py)
  0.00%   0.00%   0.840s    0.840s   <dictcomp> (numpy/core/fromnumeric.py)
  0.00%   2.00%   0.830s     4.27s   amin (numpy/core/fromnumeric.py)
  1.00%   5.00%   0.660s     2.75s   ones (numpy/core/numeric.py)
  0.00%   0.00%   0.630s    0.630s   iterencode (json/encoder.py)
  0.00%   2.00%   0.600s     4.93s   amin (<__array_function__ internals>)
  0.00%   2.00%   0.490s     4.73s   amax (<__array_function__ internals>)
  0.00%   0.00%   0.450s    0.450s   _numba_unpickle (numba/core/serialize.py)
  0.00%  60.00%   0.250s    66.65s   run (sim.py)
  0.00%   0.00%   0.200s    0.200s   get_state (body.py)
```

This unprepossessing snapshot is a treasure trove. In one view, we can instantly see where our program is spending its time. The third column is what we are most interested in. If we want to speed it up, we focus our attention on the top few lines. It shows the total CPU time that has been spent on any given function.

Here, py-spy calls out **body.py**'s `calculate_interactions()` function as taking up 17.65 seconds of CPU time, more than any other function. (And this is after I have already done a lot of optimizing to speed it up.) The interactions between atoms were by far the most time-consuming part of the simulation, and the interactions between atoms and walls in `calculate_wall_forces()` weren't far behind. The fact that

they are still the longest poles in the tent gives a hint as to just how much of a bottleneck they are. Before optimization they were many times slower.

The appearance of **pacemaker.py**'s `beat()` in line 3 is actually really good news. This is the function that sits and twiddles its thumbs when there is time to kill after simulation finishes all its calculations for the time step. The fact that it occupies such a sizable fraction of the time shows that the simulation has some computational elbow room.

The next line, `_wrapreduction`, comes from the Numpy core library. It's safe to assume for the first five years of your programming career that anything in Numpy core is as fast as it can possibly get. The only way to speed it up is to re-examine your approach and find a way not to call it in the first place. But if your slowest functions are in Numpy core, that's a good sign that your code is already running at high efficiency.

The other wildly valuable piece of information we can get from this is what we *don't* need to spend time speeding up. By the time we get past the first few items on the list, there is a rapidly diminishing return. The rest are background noise. Even if we know an easy trick to speed them up 100 times, it may not be worth our effort to implement it. In that area we can keep the code simple, naïve, hopefully easier to read, and focus on speeding things up at the top of the list.

Vectorization

There's probably a lot of smart stuff to be said about vectorization, but it boils down to this: whenever you can take your math and put it into Numpy arrays, do it. It's so much faster. Python is good at a great many things, but quickly iterating over for-loops is not one of them. The folks who do the under the hood optimizations for Numpy arrays are amazing, and I take advantage of their fine work wherever I can.

By way of demonstration, we can do a head-to-head comparison of two cases. The first is adding two lists, element by element. (The actual code I used here is in **sum_list.py** in the code repository for this chapter.)

```
N =  # a big number
A = list(range(N))
B = list(range(N))

# Time this snippet
for i in range(N):
    C[i] = A[i] + B[i]
```

The second case is adding the same numbers, but in the form of Numpy arrays. (Actual code in **sum_array.py** of the chapter repo.)

```
A = np.arange(N)
B = np.arange(N)

# Time this snippet
C = A + B
```

There's a bit more to the code to make sure that the compiler doesn't cheat and avoid doing the work, but the snippets above are the important part. (For a refresher on timing code in a way that measures what we think we're measuring, revisit Chapter 2.) For N = 10 million, the for-loop case takes 1300 milliseconds and the Numpy array case takes 18 milliseconds on my machine.

The exact numbers here aren't the important part. If you run these scripts, your results may be very different. It is the nature of optimization to be very specific, and your results will vary with different data types, array sizes, operating system, Numpy version, and processor type. So the important takeaway here is not that Numpy is 72 times faster, but that it can speed things up a lot, where "a lot" will vary by context.

Vectorization can take some work

It's not always easy to figure out how to take a calculation from a for-loop and make it into an array operation. Sometimes you have to do some mental gymnastics. This is illustrated well by our atom-to-atom distance calculations.

In two dimensions, the distance between any two points is the difference between their x-coordinates squared, plus the distance between their y-coordinates squared, all under the square root. Logically, if we want to find the distance between two groups of points, group A and group B, where group A has M points, and group B has N points, this will give a total of M x N different distances. A logically straightforward approach is to start with one point from Group A, find the distance from it to each of the N points in

Group B, then repeat for each of the M points in Group A. This is naturally expressed as one for-loop nested within another. The code flows like this.

```
for point_a in group_A:
    for point_b in group_B:
        distance_ab = (
            (x_a - x_b) ** 2 + (y_a - y_b) ** 2 ) **
.5
```

Sadly, Python does not execute for-loops quickly. After you do your profiling and find a function that needs to be sped up, and within that code you find a nested for-loop, you automatically know this will be a great place to start. This is the case for us here.

Let's start by just focusing on the x- difference. We can put all of the x- coordinates of our M points from group A into one array and all N of our x- coordinates from group B into another array. For M = 4 and N = 3:

$$\begin{bmatrix} x_{A0}, & x_{A1}, & x_{A2}, & x_{A3} \end{bmatrix}$$

$$\begin{bmatrix} x_{B0}, & x_{B1}, & x_{B2} \end{bmatrix}$$

Now the trick is to manipulate them so that we end up with a two dimensional M x N array of differences that looks like this.

$x_{A0} - x_{B0}$	$x_{A0} - x_{B1}$	$x_{A0} - x_{B2}$
$x_{A1} - x_{B0}$	$x_{A1} - x_{B1}$	$x_{A1} - x_{B2}$
$x_{A2} - x_{B0}$	$x_{A2} - x_{B1}$	$x_{A2} - x_{B2}$
$x_{A3} - x_{B0}$	$x_{A3} - x_{B1}$	$x_{A3} - x_{B2}$

Maybe you can already see a path to get there.

The trick is to take our one dimensional arrays of x-coordinates and expand them to be two dimensional in just the right way. In both cases, the expanded array will be M x N in shape.

For our group A x-coordinates, we want them to end up tiled across columns, copied so that each column has an identical set of them.

x_{A0}	x_{A0}	x_{A0}
x_{A1}	x_{A1}	x_{A1}
x_{A2}	x_{A2}	x_{A2}
x_{A3}	x_{A3}	x_{A3}

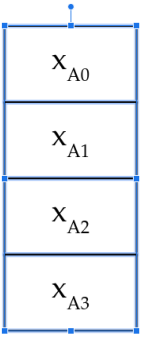
And we want our x-coordinates from group B to be tiled across rows. Each row of the resulting array will have an identical copy of the x-coordinates from group B.

x_{B0}	x_{B1}	x_{B2}
x_{B0}	x_{B1}	x_{B2}
x_{B0}	x_{B1}	x_{B2}
x_{B0}	x_{B1}	x_{B2}

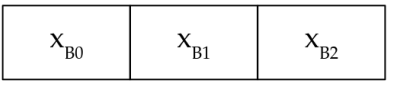
That's the trickiest part. With the tiled coordinates from both groups we can get the differences by doing an element-wise subtraction of one array from the other. We can also do element-wise squaring, add it to a similarly constructed set of y-coordinate differences, then take the square root of the

whole thing. The result is an M x N array, where the element (i, j) at row i and column j is the distance between point i in group A and point j in group B.

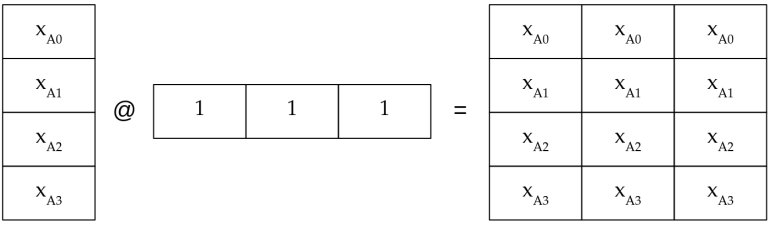
The mechanics of tiling a one dimensional array to get a two dimensional array are the last thing left to unpack here. The first step is to take our one dimensional array of coordinates, and turn it into a two dimensional array having only a single row or column. For group A we want it to be a two dimensional array of shape M x 1, M rows by one column.



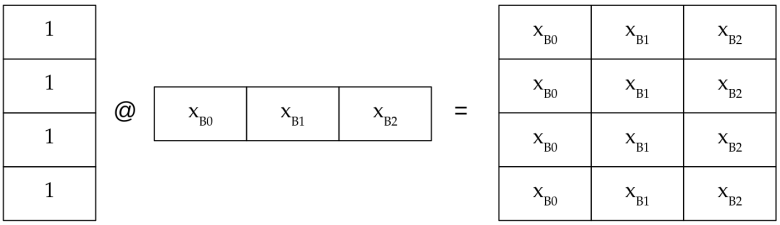
For group B we want it to be a two dimensional array of shape 1 x N, 1 row by N columns.



We can tile these out to their full sizes by doing matrix multiplication with arrays of all ones of the appropriate shape. For group A we can multiply it by an array of ones with one row and N columns. (In Numpy, the @ symbol is shorthand for matrix multiplication, as opposed to scalar or element-wise multiplication. For a refresher on matrix multiplication, check out this walkthrough.)



For group B we can multiply it by an array of ones with M rows and one column.



There is also a Numpy function `tile()` that will do this for you, but I find it helpful to keep a close mental eye on my rows and columns so they don't get out of hand and bite me.

We did the steps out of order, but we eventually got to the bottom of how to vectorize our many-to-many distance calculations. In Python, the whole sequence ends up looking something like this.

```
# Create single-row and single-column 2D arrays
x_a_row = x_a[np.newaxis, :]
y_a_row = y_a[np.newaxis, :]
x_b_column = x_b[:, np.newaxis]
y_b_column = y_b[:, np.newaxis]

# Create the all-ones tiling arrays
row_tile_a = np.ones((N, 1))
column_tile_b = np.ones((1, M))

# Calculate the arrays of x- and y-differences
d_x = (
    (row_tile_a @ x_a_row) -
    (x_b_column @ col_tile_b))
d_y = (
    (row_tile_a @ y_a_row) -
    (y_b_column @ col_tile_b))

# Calculate the points from all A points
# to all B points.
distance = (d_x**2 + d_y**2) ** 0.5
```

Not all vectorization will be this tricky, but sometimes it will be (and sometimes it will be even worse). However, if you proceed methodically and take the time to get them working, you can speed up for-loop calculations to run faster than a puppy at dinner time.

Did someone say dinner?

Speeding things up with Numba

After vectorization, the next line of defense for accelerating bottleneck code is Numba. You won't be too far wrong if you think of Numba as a video game power-up that supercharges your code so that it blasts down the race track in a blur. But to really get the most out of Numba, we'll have to lift the hood and look at the gears. This will be a bit of a tangent, but hang with me here. It pays off in the end.

Python is a scripting language, which means that when you tell Python to run, you are actually telling a specific program, a Python interpreter, to read a text file, your Python script, and step through it line by line. The interpreter takes each line and translates it into concrete, low level instructions for your CPU to follow.

This is opposed to working with a compiled language, where a program called a compiler reads your entire text file full of Java or Rust code, start to finish, and translates the whole thing into machine instructions at once. Only after it has successfully compiled can it be run.

(This is an oversimplification, but it's a useful mental model for my purposes here.)

The benefits of working with a scripting language, like Python, Ruby, or JavaScript, have to do with flexibility. I can type Python one line at a time in an interactive window and execute each one as I go. I couldn't do that if I were working with FORTRAN or C++. It also means that I can run less than perfect Python scripts. A buggy script may run part way before crashing or, if the flawed code is never reached

during execution, the script may run to completion just fine. This can hide flaws and make the programmer's life a little easier when they are hacking together quick solutions.

The benefits of a compiled language have to do with speed. When a compiler gets to see a program in its entirety, it can do a lot of clever things to make the program run more efficiently and more quickly. These compiler optimizations are a deep and fascinating rabbit hole of their own, but for the most part we get to treat them as a magical black box and reap the benefits without having to worry about how they do what they do.

Numba lets us have the best of both worlds. With Numba we can flag a function to be pre-compiled as a whole. Then when we call the Python interpreter, it will walk its way through the script as usual, except that when it reaches the flagged function it will pause to read that function in its entirety and compile it to low level machine instructions. Numba will use a toolbox full of compiler optimizations to make that code run faster than a line-by-line interpreted Python script ever could. It can end up being 10 times faster or more.

Pre-compiled Python is a hell of a drug. Once you get a taste, you're going to want to Numba-fy all your code. But before you do that, a word of caution. Numba is more finicky than other Python and harder to debug.

One of the flexibility benefits Python brings is duck typing. That means that when you start using a variable in your script, you don't have to tell Python whether it's an integer or a Boolean, a float or a string. Your Python interpreter will

make that judgment on the fly based on how you're using it and what you're asking it to do. This constant re-evaluation is both what makes Python so flexible and sometimes so slow. Numba doesn't tolerate this. For it to pre-compile a function, it needs to know the type of every variable coming in and the type of every variable it's expected to return.

To see Numba in action, we create a function to add our arrays together, element by element. Adding the `@njit` decorator at the beginning flags it for just-in-time compiling, that is, compiling it the first time it is needed. And when the interpreter reaches this, it briefly hands control of things to Numba so that it can read through the function and compile it down to optimized machine code.

```
from numba import njit

@njit
def add(A, B, C):
    for i in range(n_sum):
        C[i] = A[i] + B[i]
    return C
```

From sum_elements_numba.py at tyr.fyi/6files

On my machine this code takes an average of just 14 ms to add two Numpy arrays of 10 million elements together. This is even faster than the 18 ms I saw using Numpy's array addition operator, which is already heavily optimized. There's no doubt about it, Numba is a power tool.

You can see one of Numba's quirks in the way we built the function. It takes A, B, and C as input arguments, both the operands and the result. By handing it the return variable as

an input argument, we are giving it a running start. Before it even begins, we are telling Numba what the results variable's type should be, how many elements are in the results array, and where in memory to find the space pre-allocated to hold it. If we had to allocate this memory space from scratch each time we ran the function, that would add some overhead (just a few milliseconds, but still, it's not nothing).

The most confusing and frustrating debugging experiences I've had with Numba have dealt with types. Either Numba is uncertain about what return type to provide, or there is some subtle mismatch between the type it returns and the type I expected. Providing pre-allocated output arguments as inputs avoids this confusion. It gives Numba well specified templates to work from. It makes the function calls more verbose, sometimes with a very long list of arguments, but it's worth it. This is an area where Numba has made big improvements in the last few years, but still it's the largest point of friction I've come across.

Tricks and quirks like this come up more frequently when working with Numba. Having to think about things like types and memory allocation becomes harder to avoid. Traditionally, Python does a great job of hiding all that from us. When we set off on a quest to optimize our code, we give up that protection.

I want to emphasize that passing the return argument as one of the inputs is not strictly necessary. Someone with a better understanding of Numba than mine would probably not need to do this at all. But it is a technique that I have had good luck with, so I'm sharing it with you to use as you

wish. As we see more often when diving into optimization, Try It and Test It, your mileage may vary.

Another Numba quirk is that the first time through the code is always slow. When I run the simulation from the previous chapter it takes several seconds of staring at a black screen before the blocks render and start bouncing around. When I turn off Numba, this startup lag goes away entirely (although my code then runs too slow). Compiling these functions takes a little bit of time, small though they are.

This one-time delay during the first iteration is simple enough to handle by manually running the simulation through one time step before initializing the Pacemaker

```
print("Warming up simulation")
sim.step()
```

From `run()` in sim.py at tyr.fyi/6files

This gets the pre-compilation done and out of the way before the simulation is committed to keeping up with the wall clock and saves you from getting a lot of angry warnings from the Pacemaker.

Now that we have two rules of thumb (suggestions of thumb?) it's a good time to start a list.

1. **Pass variables for return arguments in as inputs.**
2. **Run through Numba-jitted functions once before kicking off the program in earnest.**

For-loops. Seriously.

Execution speed is hard to guess beforehand. When we were experimenting with Numpy arrays at first, we saw a huge speed up going from a naked for-loop to Numpy powered element-wise addition of whole arrays. If we modify our Numba function to do whole array operations instead of a for-loop, it actually slows it down considerably, to 42 ms.

```
@njit
def add(A, B, C):
    C = A + B
    return C
```

From sum_array_numba.py at tyr.fyi/6files

The details of why this would be are inscrutable to me, buried deeper in the black box than I've had an excuse to dig. I can only speculate that the assumptions and optimizations performed by the Numpy library and the Numba library end up fighting each other in a counterproductive way.

This suggests another helpful rule of thumb when trying to speed up your code with Numba:

3. **Try for-loops first.**

Numba loves for-loops. Working in Numba can cause a little mental whiplash. In regular Python, for-loops are a red flag. Or at least a pink one. But they are Numba's happy place. They are a long straightaway of racetrack that let Numba really goose the pedal and see what it can do.

This also reminds us of the cardinal rule of optimization: **Try It and Test It**. (I hope this is sounding familiar by now.) There's no way of knowing for sure beforehand what's going to be fast. Evolving Numpy and Numba libraries, coupled with variations in processors, mean that you and I are likely to get different results, and your own results might change between upgrades. There are no guarantees.

If you fail, fail hard

The decorator for just-in-time Numba compiling is `@jit` . It has a built-in behavior where, if it hits a difficulty and it can't compile, it reverts back to a more standard line by line Python-style interpretation. This is helpful if you want to make sure your code won't crash. One way or the other, Python will soldier through and find a way to execute your code.

The downside is that you don't have a good way to know whether your Numba optimization was successful. In that case, every compiling failure is a silent failure, the bane of reliability engineers everywhere. It's like termite damage to the foundation pylons. It gives no sign that anything is wrong until something very dramatic happens, and it's too late to do anything about it.

Thankfully we can avoid these silent failures by specifying the option

```
@jit(nopython=True)
```

This changes the behavior of the compiler so that if it fails, it crashes the program and spits out an error. This is a good

thing! It means we can fix whatever is wrong before it surprises us in an unpleasant way down the line.

This option is so popular that it has its own abbreviation, `@njit`. We will always use Numba this way. If we're serious about speeding up some code, we want to fully commit to it and throw a noisy error if it fails.

4. **Use `@njit` rather than `@jit`.**

While writing and debugging Numba-fied functions it's often helpful to comment out the decorator, turning off Numba, until you get the Python code behaving the way you want. Then you can uncomment it and start debugging all over again.

Avoid using Numba

Numba, like all potent magic, comes at a cost. The Python interpreter provides informative exceptions and error messages. The Numba compiler is a separate beast and sadly not quite as mature.

Compounding that, it's performing considerably more complex optimizations. They may span many lines of code and logic steps. When something breaks, it may not be immediately clear to the compiler what the cause is. It may only know that 1) something went horribly wrong and 2) it first noticed the problem when it was thinking about line 273. The actual bug may be on another line and you may have to do some trial and error to even figure out what the nature of the problem is. This leads us to our next rule of thumb.

5. **Only use Numba where it really makes a difference.**

It's worth it to isolate the lines where your program is spending most of its time and only include those in the Numba-jitted function.

Numba doesn't come for free, but it's also not the plague. Debugging it can be a hassle to deal with, but it's a big improvement over earlier generations of Numba which just reported that there was an error somewhere in the compilation function. Happy hunting! I want to give a huge shoutout to the people working on Numba that are putting in the work to make such an important tool work better and are doing a really good job at it.

Watch your types

Here's an example of an error I got while upgrading the Numba sections of the simulation code.

```
numba.core.errors.TypingError: Failed in nopython mode
pipeline (step: nopython frontend)
No implementation of function Function(<built-in function
matmul>) found for signature:

 >>> matmul(array(float64, 2d, C), array(int64, 2d, C))
```

I was trying to `@njit` a function containing Numpy's `matmul()` (**mat**rix **mul**tiply) which accepts two arguments. This error message told me that there is no such function that accepts a 2D array of floats and a 2D array of integers. When I commented out the `@njit` decorator, the code ran just

fine. Sadly, this is a common error when working with Numba.

The cause, as you may have already guessed, is that I got sloppy when creating my Numpy arrays, and I created arrays of different types. Numpy can perform matrix multiplications on a pair of arrays of the same type, be they integers or floats, but it can't mix types. However, we can happily sail through life never knowing this, because it also does us the convenience of checking the arrays' types, and if there is a mismatch, it casts the integer array up to a float array before performing the multiplication. This is a very Python thing to do.

This automatic checking behavior is not at all a Numba thing to do. Type checking takes time. Casting takes time. They break the fast-as-possible flow that the compiler optimizations are trying to establish. So rather than hide that complexity from the user, Numba surfaces it with an error that says, in effect "Get your house in order and try again."

The cleanest way I have found to do this is to explicitly declare the types for each of the Numpy arrays in my input arguments. When creating a new array, commands like `numpy.zeros()` and `numpy.ones()` have a `dtype` argument that lets you choose your type. For most Numpy functions, it defaults to `numpy.long`, but for example `numpy.random.randint()` defaults to `numpy.int_`. Even better, `numpy.arange()` chooses its type based on its other input arguments.
To save confusion later, you can force these functions to create arrays to your type specifications like so.

```
np.ones((n_rows, n_cols), dtype=np.long)
```

If you inherit a Numpy array from another piece of code and don't have control over how it gets created, you can also take care of the casting operation yourself. For an inherited array A of unknown type, we can ensure it is of type long by using the array's `astype()` method.

```
A_cast = A.astype(np.long)
```

This approach leaves nothing about our arguments' types to chance and leads us to the next Numba rule of thumb.

6. **Declare array arguments with a `dtype` (usually `numpy.long`).**

Here's a nice reference overview of Numpy's types, and here's a more encyclopedic resource, although in practice the only ones I ever touch are `numpy.long` (a.k.a. `numpy.float64`; on my setup this is also Python's `float`.) and `numpy.int_` (on my setup this is the same as `numpy.int64`. Python's `int` type is flexible. Its size in memory depends on how large of a number it is trying to represent.)

Don't make new arrays

On the theme of leaving nothing to chance, I've found it to be simpler to avoid creating any new arrays within the jitted Numba function. Numba makes a valiant effort to guess what type the new array should have, but somehow I still manage to confuse it more often than not. Creating new arrays can be the source of a lot of confusing type-related bugs.

7. **Avoid creating intermediate Numpy arrays.**

It's also not great for performance. Creating a new array means negotiating for space in memory with the computer's processes that supervise that sort of thing. It takes a little bit of time and if the code is trying to do it repeatedly, it can really slow things down.

If it simplifies the code a great deal to have an intermediate array to hold results, an alternative approach is to create that array outside the Numba-jitted function and pass it in as another argument. It's a good solution because it means that space in memory is set aside and recycled each time the function is called. There is no need to create a brand-new array from scratch each time.

Thinking about where arrays sit in memory and how many bytes each element occupies are usually things that Python (and Numpy) hide from us. For our convenience and sanity these details are taken care of out of eyeline. But when we get serious about optimization we can no longer afford the luxury of ignorance. The harder we push performance, the more leaky the abstractions become, and the more we are

forced to acknowledge the meshing of the gears that make our computational engine turn.

Don't use Numpy functions

Taking a couple of the previous suggestions together–3) try for-loops first, and 7) don't create intermediate Numpy arrays–it makes sense to avoid using Numpy functions to operate on arrays entirely. For example, adding two arrays with an addition symbol, `+`, aliases to Numpy's `add()` function. This violates both rules by skirting for-loops, and by creating a new Numpy array to hold the result.[2] To be extra cautious, we can extend this rule.

8. **Avoid operating directly on Numpy arrays or using Numpy functions.**

There are a lot of convenient functions in the Numpy library, like trigonometric functions for example. But most of these are also available in Python's native `math` library, and most of those that aren't can be reproduced in a few steps and some determination. We've already seen that Numpy and Numba have somewhat different goals and approaches and can contend with each other. Restricting Numba functions to Python native math operators keeps it on its home turf. These are the functions that it's better at optimizing.

[2] Yes, it's true that you can pass an empty output array to Numpy using the `out` argument, and it will be populated with the result. This ameliorates the performance hit a little bit. Now stop interrupting me with facts while I'm trying to make a point.

Avoiding Numpy functions within Numba functions is also a good way to avoid quirky interactions and unanticipated behaviors. Introducing one library adds complexity enough. Bringing in two of them and having them interact occasionally leads to some deeply frustrating experiences.

Take baby steps

Another development trick I have found invaluable in writing Numba-fied functions is building them up incrementally. I create a stub of the function, add just one or two lines, and check to make sure it runs, even though I'm certain it won't be giving correct answers yet. Because Numba error messages can be cryptic and hard to localize, it helps to have less territory to cover. It's a lot easier to spot a bug in two lines of code than in twenty.

Then after the first two lines of code compile and execute, I add two more. This way I know that any new errors that occur are probably due to the most recently added code.

9. **Build Numba-jitted functions incrementally, testing often.**

I confess that sometimes I get cocky and try to write a long Numba-jitted function in one go. Sometimes I get lucky, but usually that just ends up reverting back to the incremental strategy, with me commenting out all but my first two lines, and then gradually building the function up from there. Incremental development of this sort is a useful approach in general for any kind of code, but with more finicky languages and cryptic error codes it becomes invaluable.

Matrix multiplication

The one apparent exception to all of the above rules is matrix multiplication, the repeated application of the dot product across the rows and columns of a couple of two dimensional arrays. This particular operation comes up so often in computationally intense applications that it has become the standard by which all numerical computation packages are measured. It's the backbone of deep learning and modern machine learning methods. It is the problem for which an entire class of silicon hardware, graphical processing units (GPUs), have been optimized to solve. Because it has gotten so much attention at every level, there are a lot of tricks available to Numpy for speeding up matrix multiplications, and because that is an important measure of success it uses all of them.

There is a straightforward three-level for-loop approach to computing matrix multiplications, but there have been theoretical and algorithmic discoveries of more efficient ways to do this. On top of this, Numpy cheats and splits the computation across multiple threads if the arrays are large enough to benefit from it.

The bottom line is that, even if we don't understand how or why, it is enough to know that vanilla Numba will never beat Numpy for straight up matrix multiplication.

But wait! The Numba team has helped us out even here. There is another argument we can pass with the jitting decorator.

```
@njit(parallel=True)
```

By using this and swapping in Numba's `prange` function for range, Numba also can split its computation across multiple threads. Not only does it keep pace with Numpy, but on my machine it's almost twice as fast. On a comparison test in **matmul_comparison.py** I get

```
Vanilla Numba matrix multiplication :  579 ms
Numpy matmul(): 1270 ms
Numba matrix multiplication with parallel=True: 298 ms
```

Results will depend strongly on the specifics of the computation and the platform, so Try It and Test It. As always. With everything.

This rounds out our list of rules. They are definitely not firm or important or inviolable enough to be considered laws or commandments, so I present:

The Ten Suggestions

for working with Numba

1. Consider passing in pre-allocated return variables as input arguments.
2. Maybe run through Numba-jitted functions once before kicking off the program in earnest.
3. Try for-loops first, if it's not too much trouble.
4. Think about using `@njit` rather than `@jit`.
5. If you're OK with it, only use Numba where it really helps.
6. Possibly declare array arguments with a `dtype` (usually `numpy.float`).
7. Don't create intermediate Numpy arrays, if you can manage it
8. Avoid operating directly on Numpy arrays or using Numpy functions.
9. Weigh the benefits of building Numba-jitted functions incrementally, testing often.
10. If you Numba-fy matrix multiplication, you could use parallel=True. Or just leave it to Numpy.

I expect that every one of these suggestions will have exceptions and corner cases where they don't apply. There will be platforms, perhaps, where they don't hold true and particular computations where it makes sense to break them. But my hope is to provide a reasonable starting place if you are new to Numba and want to hit the ground running. As always, in optimization the one cardinal foundational law remains:

Try It And Test It

As an endorsement for the Try It and Test It methodology, I have to confess that I didn't know about half of these suggestions before I started writing this chapter. I would write a paragraph, create some sample code to verify what I just wrote, be surprised by the result, and then do some more investigation. This led to several new and important insights, particularly about how Numba and Numpy fight each other.

As a result, I revamped the simulation code. My rough estimate is that the code as a whole is now 50% faster. The Numba-decorated functions themselves are at least twice as fast.

Here's a sample of what the new code looks like.

```
@njit
def body_interactions_numba(
    f_x_a, f_x_b, f_y_a, f_y_b,
    k_a, k_b, r_a, r_b,
    x_a, x_b, y_a, y_b,
    v_x_a, v_x_b, v_y_a, v_y_b,
    sliding_friction, inelasticity,
):
    epsilon = 1e-12

    for i_row in range(x_a.size):
        for j_col in range(x_b.size):
            d_x = x_a[i_row] - x_b[j_col]
            d_y = y_a[i_row] - y_b[j_col]

            d_v_x = v_x_a[i_row] - v_x_b[j_col]
            d_v_y = v_y_a[i_row] - v_y_b[j_col]

            r_ab = r_a[i_row] + r_b[j_col]
            k_ab = 1 / (
                (1 / (k_a[i_row] + epsilon)) +
                (1 / (k_b[j_col] + epsilon))
            )
            distance = (d_x**2 + d_y**2) ** 0.5 + epsilon
            compression = r_ab - distance
            compression = max(0, compression)
            f_ab_contact = k_ab * compression

            f_norm = f_ab_contact / distance
            f_x_ab_contact = f_norm * d_x
            f_y_ab_contact = f_norm * d_y

            f_x_a[i_row] += f_x_ab_contact
            f_x_b[j_col] -= f_x_ab_contact
            f_y_a[i_row] += f_y_ab_contact
            f_y_b[j_col] -= f_y_ab_contact
```

From body.py (comments removed) at tyr.fyi/6files

Notice the complete lack of any Numpy functions within. It's just for-loops and element wise operations. It has a lot of intermediate variables, but they are all floats and integers, no numpy arrays. The only arrays in play are the ones passed in when the function is called. It's slightly less readable, but to my eye it's not bad.

I have mixed feelings about this. On one hand, I'm thrilled about the increase in efficiency. This expands the capabilities of the simulation quite a bit! And it makes more efficient use of my resources! And by some measures, it's more transparent, and easier to explain! It's a huge engineering win.

On the other hand, I sat down to write this thinking that I had some experience I could share that would be helpful to folks, only to discover that I didn't know nearly as much as I thought I did. This is uncomfortably humbling, but I've made my peace with it. Consider it an extra fervent endorsement of the Try It and Test It method. There is no magic here, only trial and error. Take this as a license to question and test anything an expert has ever told you about optimization. Run the numbers yourself. See what results you get. And if you want to extend this to a more general life lesson, that is entirely up to you, but I won't try and talk you out of it.

Are we done yet?
I feel like we should be done by now.

Other paths to optimization

This is about as far as it makes sense to take our optimization efforts. The optimization dilettante (such as we are) can invest enough time to vectorize code and, where necessary, Numba-fy it. After that things get much more expensive and time consuming. But it's worth at least being aware of the next levels of seriousness.

If you are prepared to dedicate an engineering team to speeding up your calculations, it means that you have a few million dollars to burn and are probably making the same type of calculation many times over. If it is a one-of-a-kind calculation, say, representing some novel physics in a

simulation, that will probably lead you down the path of hyperspecificity. Common strategies include writing low level C code for very specific high-performance computers. Specialized simulations for large scale weather and high energy physical phenomena fall into this category.

You can also invest time in theoretical and mathematical work to simplify the problem. It's often possible to create approximations that are close enough to be useful, but save a lot of math steps. One of my former coworkers had spent time as a physics postdoc and used this approach to model molecules using the Schrodinger Wave Equation. The simulation got unwieldy in raw form after just a few atoms were added to the molecule. He was able to extend it to much larger molecules by artfully simplifying the representation of the equation, keeping only the most essential aspects.

If your problem reduces to a common calculation, such as a matrix multiplication, and your goal is to perform it many times as quickly as possible, then you can go a different route and push for brute force scaling. The first line of attack in scaling is parallelization. CPU clock speeds have topped out in the last few years due to the limitations of basic physics, but the number of processors available has steadily grown. If you can find a way to chop your computation up into smaller, bite-size jobs and farm them out to separate processors then you can scale them up almost arbitrarily. You are only limited by the overhead it takes to parcel the task into subtasks and communicate those subtasks to each of the participating processors.

We are already doing a lightweight version of parallelization in our code by having the simulation run on one processor, while the visualization runs on another. This is a strategy we'll extend in future chapters when we integrate other components that require heavy pre-processing and other algorithmic building blocks that can be computationally demanding in their own right.

Monitoring

Our cardinal rule of Try It and Test It isn't limited to when we first write the code. Interesting programs change over time as they run. In the case of our simulation, some configurations are more computationally demanding than others, when bodies are in close proximity for example. Any code that makes use of a stream of data will be encountering new and unforeseen states on a regular basis. Any programs that accept human input or feedback signals, like human directed reinforcement learning to choose an example completely at random, have the additional complexity of dealing with an entirely unpredictable, and sometimes mischievous or adversarial, human being on the other end. A program that behaves well under development and testing conditions may run into difficulties later when running for real, also known as running in the wild or **in production**.

Once a program or system is running in production, the problem of tracking gets renamed **monitoring**. It starts to look much less like a single measurement, and more like a continuous observation process. Our simulation has an example of this already. At each iteration we have been checking whether the simulation iterations have taken more

than their allotted time. When they do, we have been reporting this with a text message in the console. This is a primitive form of monitoring. It suraces an important condition that we need to be aware of, but it's not something we worry about so much that we feel the need to shut down the program.

It's easy to lose one's direction when setting up monitoring. There is just so much information and so many different ways to show it. It's useful to have a point of focus so as to avoid losing your bearings. For me, this comes in the form of a question I'm trying to provide the answer to. For the physics simulation this question is clear: *Is the simulation code taking too long to compute?* In order for our simulation engine to claim real time performance, it has to simulate the physics that occurs during one clock cycle in less than one clock cycle worth of time. If it can't manage this, it falls behind.

This is a good question because it's something we can measure. It is surprisingly easy to pose questions that are actually unmeasurable. If instead we had asked a question like *Is the simulation running smoothly?* or *What is the health of the physics engine?* we would've had to take additional steps to figure out what we meant by those terms and how to reduce them to numbers.

What number to report?

So far, we can answer the question of whether the simulation competitions are running fast enough with a yes or no. Or, to be more precise, with a lot of yeses and nos. Every simulation clock cycle we will have another chance to measure whether the computation finished in the allotted

time and so we wind up with 1000 yeses and nos per second. That's more than we can easily digest, and it raises the question of how to communicate that volume of information to a human with limited reaction time and attention span.

It's helpful to play the *What if?* game when deciding what to show and how to show it. What if a single clock cycle ran too long? Would I try to speed up my code? Modify the simulation? Or would I write it off as an anomaly? There are a lot of things that could cause a single cycle to go over time, and most of these have little bearing on the performance of the simulation, so the right answer is to do nothing. A single time step's violation is inconsequential, and we wouldn't want to do anything in response if it occurs.

The implication of this for monitoring is that we don't need to worry about reflecting every single overtime violation. But it raises the follow up question of How many times does the simulation have to go overtime before we worry? 1% of cycles? 10%? 50%?

This is a tough question to answer, and the very fact that it is a tough question to answer is a useful signal. If you find yourself in the position of picking arbitrary thresholds out of a hat, it usually means that you haven't quite found the right thing to measure yet.

Let's take a step back and consider the situation more carefully. The thing that we care about is how far the simulation has deviated from real time performance. A very small deviation suggests a different course of action from a large deviation. We don't just care about whether a computation went over time, we also care about the amount

of the overtime. And we don't just care about the amount of overtime in a single clock cycle, we care about the total amount, added up over time. Thankfully this is a thing that we can directly to a number on. It's captured in the number of seconds of deviation from the wall clock.

The process of asking *If we had the answer, what would we do with it?* is immensely helpful in focusing our monitoring efforts. We can refine this by playing the *What if?* game again. What if we see this simulation has fallen behind? What will we do? Maybe close a web browser that's competing for resources? Or maybe revisit the code and do some more optimization? Or maybe ignore it and let it run?

Which of these options we choose will likely be influenced by whether the deviations are growing or shrinking, and how fast, whether they are an isolated blip, or a sustained deficit. To get at this, we will need to know not just the wall clock deviation at the current time, but also a bit of history.

Now we're really getting somewhere. The next question to answer is how far back in time do we care about? We are focused on correcting any ongoing deviations, rather than summarizing the long-term history. For this it's probably enough to look at the most recent data, say, one minute's worth.

One value to represent them all

That's still 60,000 data points if we are working with a 1,000 Hz simulation clock. We can afford to do some aggregation and show the deviation once per second. That will be frequently enough to let a human observer see anything hinky taking place and address it. But this raises the question of how to aggregate the 1,000 timing measurements gathered during a second into a single number that represents them all.

There are several options open to us, and they each tell a different story. The most popular aggregation method is averaging. Averages are efficient to compute and have a nice intuitive interpretation. It's hard to go too far wrong with an average. However, for the question we are trying to answer it's worth thinking through the case where, if the overtime deviation were high for a quarter of a second but zero for the other three-quarters, how would we want to represent that? If we represent it with the average, then it will appear that the deviation was consistently low for the entire second. We're missing out on some of the information that we care about. A brief period of high overtime deviation is of interest. Its appearance suggests that perhaps there's a problem that needs addressing.

Another alternative for aggregation is the median. It is excellent for when you want to ignore outliers and focus on typical values. Unfortunately, in our hypothetical case where the deviation is high for a quarter of a second, the median would be zero, completely masking the phenomenon we're hoping to capture. The median is clearly not the aggregation method we want to use.

The other extreme is to be deliberately sensitive to outliers and take the maximum value. Here we put the absolute worst case available to us front and center and let it be the representative measurement for the entire second. This approach is not wrong, but it's definitely harsh. Coming back to the specifics of our use case and the question that we are trying to answer, this might be overly sensitive. There are lots of things that can cause a momentary glitch for a few milliseconds that is quickly resolved. This is nothing that is likely to be even perceptible to a human and would certainly not degrade their experience. There's no need to be quite this brutal.

A good middle ground between the median and the maximum is to choose a percentile, say the 90th, as a representative value. This is a good way to answer the question *When things are rough, how rough do they get?* It also allows for very short term blips and outliers to slip by without disproportionately focusing on them.

What we're bumping up against here is that the thing we want to measure is not a fixed value. It's something that is constantly changing, and we have so many measurements of it that it doesn't make sense to consider them individually anymore. Instead, they are much easier to interpret as a distribution, a spread of values between some high and low and everything in between.

The challenge is that no single summary statistic tells you what the shape of the distribution is. The median only tells you the point that lies between the bottom half and the top half of your measurements. It says nothing about how those

are grouped. Similarly, the 90th percentile only tells you the point above which sit 10% of your measurements. To get a richer sense of the distribution, it would be useful to look at a whole set of statistics. One common way is to look at the 10th percentile, the 20th, etc. up to the 90th, that is, to look at each decile. This is a helpful way to get a sense of the overall shape of the distribution.

When we are more concerned about one end of the distribution, as is the case with overtime deviations, then we can take a different approach. Here, we want to focus on the higher end, just how far out of bounds it gets when things go off the rails. The 90th percentile is a great place to start because as we mentioned, it's a good way to represent the largest 10% of the measurements. It's used often enough that it even has its own abbreviation, p90, for labeling it on plots and tables. It's also abbreviated as a decimal, 0.9. If a simulation has no deviation up to the 90^{th} percentile, then we can say it's 90% reliable, or alternatively, it has one nine of reliability. It's also fairly common to look at system reliability at the 99^{th} percentile (p99 or two nines) and the 99.9^{th} percentile (three nines). Sometimes you can catch system reliability engineers (SRE's) bragging about five nines and six nines. After a while it stops being very meaningful. Seven nines means your system was down or out of specifications for three seconds a year. We're not sending anyone to the moon or protecting nuclear launch codes, so tracking our simulation's reliability at one nine is adequate.

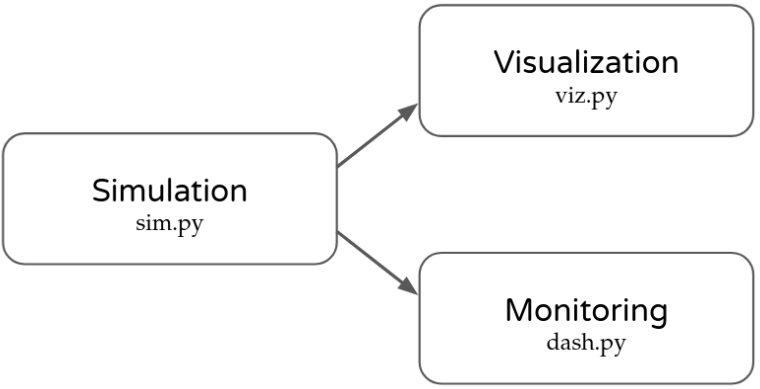
Making it into a movie

Now that we've decided what numbers we need to represent, the next step is to turn that into an animated plot.

It takes a little bit of doing to make this happen, but luckily, thanks to the investment we've already made, we have the tools to do it. In Chapter 3 we worked out how to create new processes. In Chapter 4 we worked out how to create animations. What remains is to tweak these capabilities for our new use case.

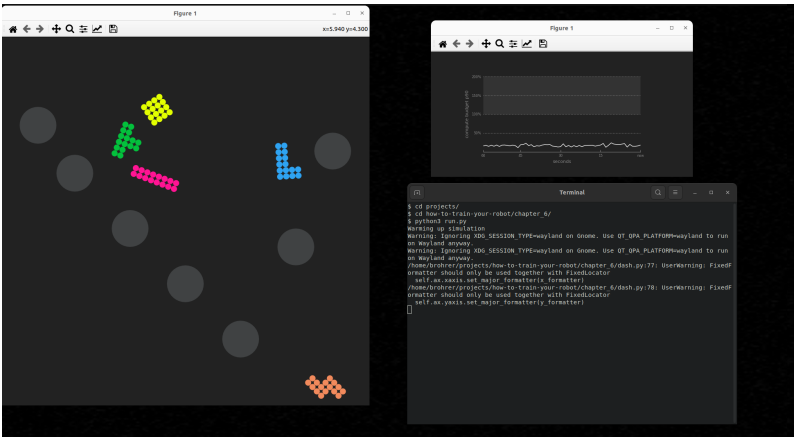
The first step is to create a new process to handle the monitoring. Our animation process is already going flat out, and we want to keep our simulation process as lean as possible. Creating an independent monitoring process is a good way to do this. Most of this implementation can be seen in **dash.py**. It follows the same pattern we set up for creating the animation, using a `Frame` object that gets regular updates, courtesy of a `run()` function that gets called to start the process. **run.py** also gets updated to set up the monitoring process and the communication Queue between it and the simulation. **sim.py** incorporates the extra Queue and sends the timing measurements over it to **dash.py**. And **config.py** gets a whole slew of new parameters to customize the appearance.

The new process diagram shows three processes connected by two Queues: one that connects the simulation to the visualization and one that connects the simulation to the monitor.



There's not much in **dash.py** that is conceptually new, but there are echoes of existing themes that bear repeating. One of these themes is extracting hard coded parameters to a configuration file. There are a lot of tweaks and customizations and hand-picked numbers and colors that can go into making a plot. I have tried to pull all of these out in order to keep the code as cleanly organized as possible. It saves a lot of hassle later when I want to go in and change one little setting. It's nice to have separate files for what the program does and others for the specifications of the fine details of how it does it. It's not always possible to separate these out cleanly, but I've found it's worth the effort to try.

Another theme we're revisiting here is that platform independence is hard. Now that we have two Matplotlib animations, running in two separate windows, it's nice to be able to place them so they're not on top of each other.



The function to control the placement of the window within the screen looks like this

```
mngr = plt.get_current_fig_manager()
mngr.window.setGeometry(x_left, y_top, width, height)
```

where `x_left` the distance in pixels from the left edge of the screen to the left edge of the window, `y_top` is the distance in pixels from the top of the screen to the top edge of the window, `width` is the horizontal extent of the window in pixels, and `height` is the vertical extent of the window is pixels.

It works great, but this solution is specific to the QtAgg backend for generating windows and user interfaces. There are other ways to do this for other backends (like impympl,

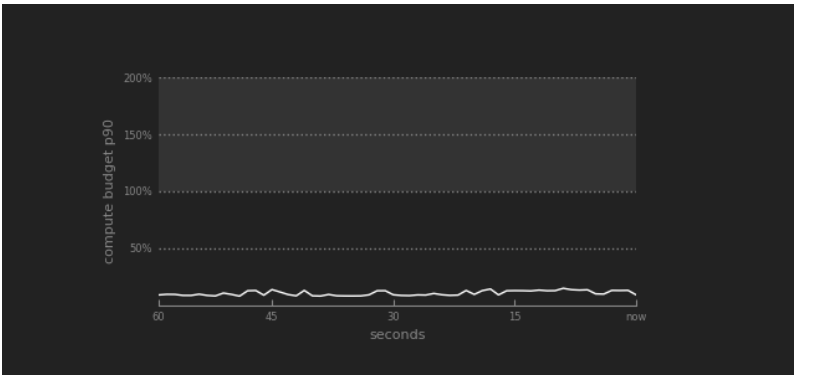
TkAgg, and wxAgg) but writing and testing all of those variants was not where I wanted to spend my time. While I warmly invite anyone who wants to extend the code to include these other backends, for now this will be yet another instance of platform specific development.

With the window size and location hard coded, we can focus on its contents. All we really need here is a single line showing the recent history of overtime p90 values. Plotting a line is the most basic of tasks in Matplotlib. It could be done in just one line of code, so the fact that I chose to do it in 55 lines deserves some explanation.

Matplotlib has a lot of reasonable defaults for how it displays and labels plots. I enjoy and take pleasure in adjusting the fine details until it looks just right to my eye. As a result, the plot generating code in **dash.py** is more verbose and contains more configuration parameters than it absolutely needs to. This is an aesthetic choice, a personal preference. Please don't feel like it's important to do it this way.

That said, I don't want to diminish the importance of good graphical design. A carefully constructed visualization will answer a question for the viewer, almost without them having to think about it. There are some small touches I've made here to eliminate as much cognitive burden as possible. One of these is to convert the y-axis into a percentage. The threshold that we are concerned about crossing is 100%. I removed the need for the reader to convert from milliseconds to interpret the result. Another small touch is to add a shaded area for the region above 100%. Intuitively everything below this is "in the clear".

Crossing into the shaded zone, represents a transgression in the literal sense, and naturally draws the eye. I've also tried to emphasize the information we care most about. I made supporting lines and explanatory text smaller and dimmer to help them fade into the background and to call attention to the line showing performance. Wherever possible, I have removed lines and text altogether. I've also tried to keep it simple and monochrome. This plot has one job, to answer the question of whether our simulation computations are taking too long. Anything that doesn't directly contribute to that job is a detraction.



Customizing Matplotlib is not particularly intuitive. There are a whole collection of special codes and incantations involved. I've created a set of posts that I reference often when creating plots, especially when it comes to axis lines, ticks and tick labels, text, axis layouts, and drawing polygons.

The Dashboard

A regularly updated performance plot of this nature is a common tool for monitoring. It can be a lot more intuitive than printing numbers in the terminal and looks better in a PowerPoint presentation. It's typical to see a handful of these plots in one window, in which case it's referred to as a dashboard, evoking the collection of dials and status lights on the dashboard of a car.

This is a good time to pronounce a Surgeon General's style warning about dashboards. They are addictive. A little bit of visibility into something that was a mysterious black box is a taste of power. Once you get a small hit, it's natural to want more. At every step along the way of this analysis, we decided a list of things that we weren't going to show. What if we made plots for all those as well? What if we included lines for p50 and p99? What if we plotted the life history of these values, rather than just the most recent 60 seconds? What if we broke out the computation time into the time spent on Numba-jitted functions and the time spent on everything else?

All of these are entirely feasible to do. We have the tools and the computation budget to pull it off. So why wouldn't we? There are some hidden costs here that are easy to ignore until too late. The most fundamental is that not all information is equally useful. We adhered to one guiding principle when designing this plot–clearly answering a question. What do we need to know in order to decide when to take corrective action? How would we know when computation was becoming a big enough bottleneck that we needed to do something about it? The focus on action, on

decision, on doing is the important part. We used the *What if?* test to gauge this. *What action would we take if we could measure this?* If the answer is "we would not do anything", then arguably that's not a piece of information we actually care about. The information that prompts action is what's most important.

Measuring things that are curiosities brings a superficial kind of satisfaction. It happens often enough that these measurements have their own name: vanity metrics. One can measure them, watch them go up or down, feel good about any improvements, and explain away any degradations. However they evolve, they are not directly tied to any decisions or substantive evaluations we might make.

A real danger of these vanity metrics is that they can obscure the important things we want to monitor. Human attention is limited. If we see one plot on a page, it will get all our attention. If we see a hundred plots on our page, our attention will be split a hundred ways. Every plot that gets added draws attention away from all the others. If it doesn't add substance, it squanders our very limited attention budget. Sticking to carefully designed metrics that clearly contribute to decision making ensures that our attention gets spent on the right things and that important issues don't slip through under the radar.

An even bigger cost can be analysis rabbit holes. Our curiosity can be our undoing. If a line on a plot suddenly jumps up or down, it's natural to wonder why, to form a hypothesis, and to feel an itch deep in our brains until we've tested it.

Just to be clear, this process of internal question and answer when looking at plots is a beautiful thing. It's just something that's supposed to happen a lot earlier in the process. During data exploration, when first building a system and experimenting with it or when investigating a new collection of data, there is no better way than to plot the data in different combinations and wonder to oneself about the blips and trends that emerge.

There is absolutely nothing wrong with the exploration process itself, but that's not what monitoring is for. Monitoring is for the single purpose of raising awareness when a problem arises, and anything that distracts from this is defeating the purpose and degrading the quality of the dashboard.

Dashboard-driven exploration becomes an even bigger time suck once other people get involved. At work we have a lot of intelligent and curious colleagues, and it's natural for them to look at a plot and ask questions. And if the person asking the question happens to be your boss, or their boss, or their boss, it becomes very hard to say no when they ask you to investigate it. All the more reason to not open the lid to Pandora's box any more than a crack.

Another way that dashboards can get out of control is when you try to extend them not just to identify when problems occur, but to give you all the information you would need to diagnose a problem when it does–to learn not just when action is necessary, but to help you decide precisely what that action should be. The difficulty is that it's impossible to know what you will need to diagnose a problem until you actually see the problem itself.

Consider our overtime monitor. Depending on whether it is consistently high, or it gradually climbs, or it makes a sudden jump into overtime territory, the follow up questions would be very different. We could in theory try to foresee all the ways in which it might fail, then create dashboard plots to tell us exactly what we would need to know to correct each of those.

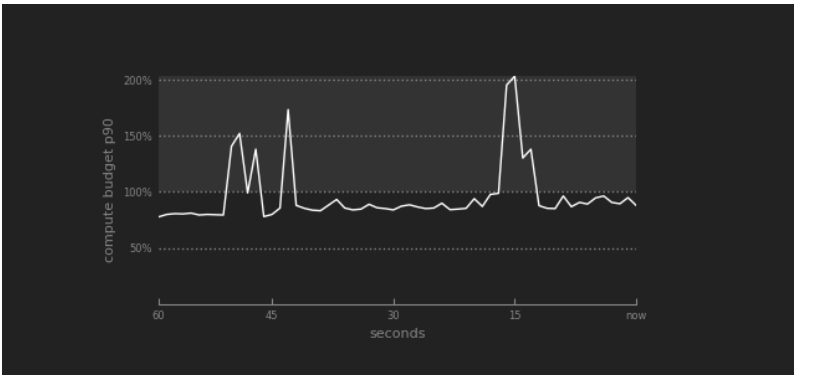
This prognostic diagnostic approach is appealing, because it feels like we're doing good engineering. We're investing time now to save time in the future. But it never works out quite that cleanly. It's hard to foresee every eventuality, and we almost always end up having to do additional ad hoc investigation to pinpoint the source of a failure or degradation. In addition, there are cases that may never occur, so by pre-computing them and plotting them before they are needed, we are doing unnecessary work. Then, as mentioned at length a few pages ago, we would be carrying around the additional cost of dashboard clutter, obscuring the really useful plots that notify us when something goes wrong. The most useful thing a dashboard can do is help us avoid silent failures, to alert us to the fact that something needs attention. The act of actually fixing whatever went wrong is a separate piece of work that we will not be able to dodge by adding plots.

Interpreting the dashboard

To show how we might use this in practice, we can simulate some concerning behavior. Thanks to our aggressive optimization earlier the simulation has no trouble completing all of its computations and keeping up with a

1,000 Hz simulation clock. To push its limits I have to ratchet up the simulation clock to 10 kHz. And that means that each simulation clock cycle only lasts for 100 µs. (In case you can't tell, I'm feeling very pleased with myself.)

With this more stringent timing constraint, we can see that the simulation does in fact cross 100% line. It tends to hover around the 80 to 90% range, but occasionally spikes to 150 and even 200%.



So what does this mean for us? This plot is supposed to directly inform actions and decisions. How do we translate this to an action?

It's still not a cut-and-dried if-then situation. If it were an absolutely clear decision like that, we could hard code it and have the program throw an error every time the line crossed 100% and shut everything down. But that's not the case. When the line crosses 100%, it is supposed to draw our attention rather than take some automatic action. The best thing to do at that point is to take a careful look and start asking questions. Is it crossing the hundred percent line by a

little or by a lot? Are the crossings brief or sustained? Is there any noticeable degradation in the simulation? Is it glitchy or sluggish? Do the incursions seem to be getting more extensive over time?

These are subjective judgments, best made by a human eye and mind. They require some focus, some thought and consideration. Making these evaluations is cognitively expensive. Human attention, both ours, our teammates, our customers and users, is one of the scarcest and most precious resources we will manage. Asking someone to keep an eye on the dashboard and think carefully about any anomalies is not a request we make lightly. We show our respect for the individuals involved by the careful thought we put into choosing what to plot, and even moreso, what not to plot.

Did someone call for a watchdog?

Shut it all down

One last thing: Coordinated process shutdown. In Chapter 5 I mentioned that when one process dies, I have to manually shut down all the other Python processes with a pkill statement at the command line. During the course of writing this chapter, I finally got tired of that and wrote a cleaner fix.

It turns out that gracefully coordinating the shut down of multiple processes is trickier than I had expected. It's not too difficult to set up the top level runner **run.py** to watch all of the child processes and, if there's a problem with one, close

them all, then shut itself down. But that doesn't handle the case where the runner itself runs into trouble and gets shut down or crashes first. I even tried adding another process whose sole function was to watch the runner, and if it had difficulties, close the runner down, then all of the child processes, then itself. That kind of worked, but there were still odd corner cases where it didn't catch everything. Also, I was unsatisfied with the extra overhead and system complexity I needed to add.

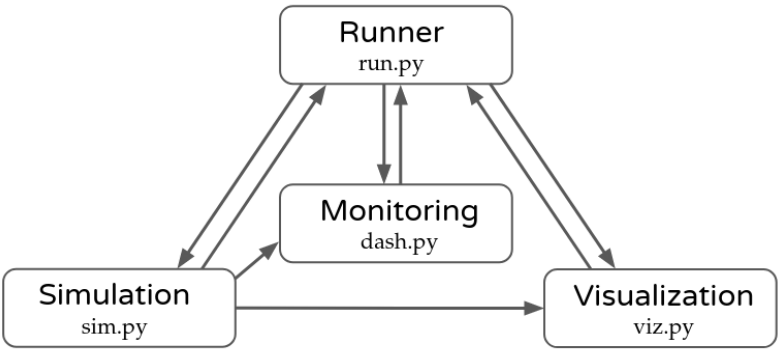
Part of the challenge comes from the fact that handling Queues is complicated. I'm still fuzzy on a lot of the details, but it appears that if a process is killed while it is holding a lock on a Queue, it can enter a zombie-like state where it is waiting around for closure that will never come.

There are different levels of demandingness that you can use when terminating a Linux process. There is SIGINT, a signal to interrupt, which asks the process to stop and gives it plenty of opportunity to finish up whatever it was working on. It's like a teacher requesting that students taking an exam wrap up the paragraph they are writing. There is also SIGTERM, a signal to terminate, which is a little more insistent. It is the classroom equivalent of "put your pencils down now, and close your test booklet." It lets processes follow whatever instructions they have for gracefully shutting down. Then there is a SIGKILL, a signal to kill. This is like yanking the test booklet away from the student mid-sentence. It's not graceful, but it definitely stops everything that's going on.

My best understanding is that when a process receives a SIGTERM while communicating via a Queue, that

connection makes it impossible for it to finish what it is doing. So while the process is technically following instructions, it never shuts down entirely. It just sits there, pencil in hand, staring at the page, racking its brain for a next word that will never come. In our case, we are not too worried about graceful shutdowns. We just want everything to turn itself off. In our child processes, we can do this with `os._exit()`. In the runner process we can take advantage of the `sys.exit()` function call. It builds in the termination of all the child processes. It is able to handle the Queues gracefully, I assume because it has access to everything at the top level and can terminate the processes on both ends of the Queue.

The design that ended up being the most effective, and in my opinion the most elegant, is a multi-way deadman switch pattern that resembles a suicide pact. Every child sends a regular heartbeat to the runner, and the runner in turn sends a heartbeat signal to each child. If any of those heartbeat signals stop coming, the recipient assumes that the process on the other side has died, and terminates itself. If a child should crash, the runner will self-terminate, which then precipitates terminating the rest of the children. If the runner should crash, each child will notice the interrupted heartbeat and terminate themselves.



This pattern proved itself to be quite effective. It is an interesting case study, where the goal is to make the system fragile in just the right way. This runs contrary to the notion of graceful degradation, where if one part of the system fails, the rest soldiers on in some way. Here the goal is catastrophe, where if one part of the system fails the rest follows quickly and the whole structure crumbles to the ground.

The net result of all of this is that when something crashes, I don't have to manually kill the rest of the Python processes. It's a small win. I think it's worth it. But it's not without cost. **run.py** is now considerably longer and more complex. Rather than there being two inter-process Queues to create and manage, now there are eight. There is also an additional block of code in what used to be a very streamlined `run()` function in each child. This is necessary for sending heartbeats to the parent runner and monitor its heartbeat to make sure it is still running. On the upside, it doesn't require the creation of any new files or tools, and it is relatively easy to explain: the parent holds a deadman switch for each child, and each child holds a deadman switch on the parent.

This is also an illustration of how engineering needs can evolve over the course of a project. Earlier I said that it was premature to engineer a coordinated shutdown procedure. But with the addition of a little project maturity, anticipating the addition of more processes in the next few chapters, losing patience with the extra manual step, coupled with a festering curiosity, now feels like the time to build a solution. All of these factors are subjective and are certainly debatable, but here I get to exercise my privilege as sole developer and go with my gut.

One thing I haven't done here yet is to include some user-initiated keystroke to shut things down, like hitting the "q" key. That's because I am saving that for the next chapter. Stick around. Things are about to get good.

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.