EE 194 / BIO 196: Modeling Biological Systems
Homework 5: The Manduca Crawl

Intro and recap from class

Manduca sexta is a tobacco hornworm (which, in all accuracy, is a caterpillar and not a worm!). Manduca cannot do much. Unlike Jumbo, it cannot go to Tufts. It cannot hold a job. But it can crawl – and crawling is surprisingly complex. We're going to use a computer to try and control Manduca's legs and muscles so that it crawls really fast.

Like any self-respecting hornworm, Manduca has 5 pairs of legs (which again, in all accuracy, are prolegs). We're going to simplify things and assume that each leg of a pair acts in synchrony with its partner. In other words, we need only control 5 "legs" (each of which is really a leg pair) rather than 10. So for our purposes, we're going to ignore the fact that Manduca has both a left side and a right side. We'll just assume 5 legs going from front to back, right down the middle of Manduca's belly (see Figure 1 in the Appendix).

Legs are great for crawling, but they're not enough. Neither Manduca (nor any of us) can walk on ice. Legs are pretty useless without friction. Manduca does not get to wear grippy shoes, nor even to have grippy toes. He does, however, have one thing we do not – a "crochet" at the end of each leg. This crochet is like a little pincher. When Manduca pinches with the crochet at the end of a leg, that leg is stuck to the ground. We'll use the terminology "locked," saying that a leg is locked if its crochet is pinching.

Even 5 sticky, pinch-ey little legs are not enough. Like us, Manduca needs muscles to move around. While we humans have hundreds of muscles, our Manduca has only four; one muscle between each pair of legs (see Figure 1 again).

Your job will be to get Manduca to move. Specifically, you have to decide which legs to lock, which muscles to activate, and the timing of both of these. If you're unlucky, Manduca might stay still or even crawl backwards. If you do well, it will crawl forwards quickly. Either way, you will hopefully learn a lot about optimization ☺.

Just how hard is your job? Well, there are 5 legs (which can be locked or not) and 4 muscles (which can be on or off). So there are 9 things that can each be in two states, giving us 2^9 or 512 choices.

But that's not all; we still have to talk about timing. We're going to break down Manduca's crawling time into 10 short time periods. In each time period, you will get to decide which legs are locked and which muscles are on. That gives you a grand total of 9*10=90 decisions to make, and so there are 2^{90} (or just over 10^{27}) choices. If you try 100 choices every second, it will take over 40 trillion years to try them all. Clearly, you will have to be more clever than that!

Before we talk about strategy, though, we should talk about just how you can "try" a solution, anyway. No, you are not going to do micro-surgery on a real worm, cutting and re-attaching its neurons to implement your plan. That is probably beyond the scope of medical technology at this point, and certainly beyond what you can do in a few hours. Instead, you will use a simulated Manduca. The simulated Manduca is a Python class called Manduca, which (among other things) can take your 90 decisions as input,

simulate their effects using a biomechanical model, and return a value for how far Manduca crawled.

The good news: We will give you the *Manduca* class… and it will even be (mostly) written for you! The code can be accessed from the class web page.

We now have a model of Manduca. If we tell the model which legs are locked and which muscles are on over time, the model will solve the differential equations of motion and in turn tell us how far and fast Manduca crawls. The next task is to define exactly how we tell the model when we want legs to lock and muscles to contract.

We represent the leg-locking schedule with a 10x5 array *legs*. Each row represents one time interval, and each column represents one leg, so that a "1" in *legs*[1,2] would mean that leg #2 is locked during time interval #1 (which is from 10 to 20 seconds).

Similarly, we use a 10x4 array *muscles* to control Manduca's muscles. Again, each row represents one time interval; now each column represents one muscle. The values in the *muscles* array must be either 0 or 100. Zero means that the muscle is off. 100 means that the muscle is on, and is contracting with a force of 100 Newtons (a Newton is the standard unit of force in physics). Thus, a value of 100 in *muscles*[1,3] would mean that muscle #3 exerts a force of 100 Newtons during time interval #1 (from 10-20 seconds).

In this homework, you will use a genetic algorithm to try to make *Manduca* cover more ground (i.e., a larger distance) in a given amount of time. We will assume this time period is 100 time units.

In a genetic algorithm, remember that we are dealing with a population of individuals. In our case, each individual in the population is one solution; i.e., one gait pattern. Some patterns will result in Manduca moving faster than other patterns. The goal, of course, is to find the individual (i.e., the gait pattern) that is most evolutionarily fit – i.e., the one that makes Manduca crawl farthest.

The Homework

To keep the assignment reasonable, we give you a mostly-written class:

- class *Manduca*. This class object stores arrays for legs and muscles, as well as computing its own fitness value. The class has the following capabilities:
  - You can create a new Manduca with *Manduca*(*legs, muscles*). *Legs* should be a 10x5 array of 1s (to indicate a leg locked) and 0s (unlocked); *muscles* should be a 10x4 array of 100 (for an active muscle) or 0 (inactive). We incorporate these into an object and return it to you.
  - *Manduca.mutate* (*self*). You must write this function (see below).
  - *Manduca.mate* (*self, parent2*). You must write this function (see below).
  - *Manduca.fitness*(). This function runs a simulation to see how far the given Manduca crawls and returns that value. In fact, to make things run faster, it also saves the value away so that the next time you call *Manduca.fitness*() for the same object, it just grabs the old result without needing to run an entire new simulation.
  - *Manduca.copy*(). This function returns a copy of itself. The nice thing about a copy is that you can modify the *legs* and *muscles* arrays of the copy without changing the original.

As noted, you must write the two functions that generate new children.

- *Manduca.mate* (*self, parent2*). The *self* Manduca must mate with *parent2* and return a child. You should use genetic material from both parents. Perhaps the easiest thing is to preserve rows of legs and muscles; i.e., take an entire child's row of legs and muscles from one parent or from the other, unchanged; but randomly picking which parent to take from. So, you might wind up having legs[0,:] and muscles[0,:] coming from parent #2; legs[1,:] and muscles[1,:] from self, legs[2,:] and muscles[2,:] also from self, etc. Remember that when you assemble a child, you must not change the parents at all.
- *Manduca.mutate* (*self*). It takes genetic material from *self* and performs several mutations. It does not return any values.

What is a mutation exactly? To mutate a single individual, *mutate*() should first pick how many mutations to make in that individual. Then, for each mutation, pick what type of mutation to make: change a leg between locked↔unlocked (i.e., 1↔0) or change a muscle between on↔off (i.e., 100↔0). If you flip a leg, then there are 5 legs and 10 time intervals: so 50 choices on what to flip. Similarly, there are 40 choices on which muscle to flip.

If you like, you can also implement a secret sauce for mutation. In this unusual genetic mutation, you take one or more rows and replicate them. So, e.g., you might take rows #3 and 4 and copy them forwards to rows #7 and 8.

Here is a bit more information about the rest of the code (you do not need to modify this code, but might be interested in how it works). The top-level function is *manducaEv*(). It mostly just instantiates a random initial population, and then calls *run_generation* (*population*, *n_matings*, *n_mutations*) repeatedly. That function runs one generation of simulated evolution; it creates children (by matings and mutations), and selects the fittest

individuals to survive to the next generation. It also randomly leaves some less fit solutions, to promote genetic diversity.

Detailed description:

- Take the existing files manducaEv.py and manducaFitness.py. You should not touch manducaFitness.py; all of your code will go into manducaEv.py.
- Fill in the functions *Manduca.mate*() and *Manduca.mutate*(), in manducaEv.py.
- Run the entire evolutionary process three times with different initial random seeds. Each run should have a population of 20 solutions, and create 10 matings and 10 mutations in each generation. You should run each for 70 generations.
- Run once more with 210 generations.

***Debugging your code*:** Code that has randomness built in can be difficult to debug. A few tricks that may help:

- Always seed the random-number generator yourself, rather than letting Python do it for you. This way, your runs will be repeatable. In fact, we've already done this for you with a parameter to *manducaEv*().
- Print out the first few matings and mutations fully, and hand-check that they look reasonable.

Discussion questions: You should turn in a file HW5.pdf (any file format is fine, as long as it is readable). It should include the following:

1. Graph the best distance vs. generation number for each of the three runs.
2. How do the three runs compare? Why aren't they identical?
3. Think about what constitutes a good initial solution. Does this solution have to make Manduca move forward?
4. Instead of doing three runs of 70 generations each (with different initial seeds) and keeping the best, you might consider one large run of 210 generations (in fact, you tried both). Which worked better for you? Would you expect one approach to *definitely* be better than the other?

Animation

- If you would like to see an animation of your results, you can use manducaGraph.py. This requires two steps.
- First, create a file with the particular strategy you would like to animate. This is a file with one line per time segment, and the leg and muscle controls for that segment all on the same line (separated by a vertical bar). ManducaEv.py prints this out for you every 20 generations.
- Next call the function *read_from_file_and_graph* (*filename*). You can find this function in manducaEv.py. Just change the top-level call in manducaEv.py from calling *manducaEv*() to call *read_from_file_and_graph*() instead.

Extra credit

- Run once more with 1000 generations. This time, instead of dividing the 100-second time interval into 10 segments of 10 seconds each, divide it into 100 segments of 1 second each. This is easy to do; in your function manducaEv(), change the call to

random_manduca() to instead call *random_manduca*(100). This will make all of the initial population have 100 time segments; i.e., the *legs*[] and *muscles*[] arrays will each have 100 rows rather than 10. From there on, the code will/should simply adapt to the new sizes. Note that this means your mate() and mutate() functions must be smart enough to note how many time segments there are and adapt; i.e., not to have "10" hard-coded in.

- What was your final result? Was it qualitatively different from your other results? If so, why do you think that was? Using the animation software may help you answer this question; or you could just look at the tabular output from manducaEv.py.

Logistics:

- Use any lab PC system to write your code. You may use your own laptop if you prefer.
- The due date for this assignment is on the class calendar
- Submit your project at https://www.ece.tufts.edu/ee/194MSO/provide.cgi, which is also accessible from the course web page. You should turn in manducaEv.py, containing your code, as well as manduca.pdf (or similar extension), with the graphs and answers to the discussion questions

Appendix: Biomechanics of the Manduca model

We will use a simplified Manduca model that consists of a symmetrical structure, with 5 legs and 4 muscles. The legs may be locked or unlocked. The legs are drawn as brown circles, and the muscles as green lines.

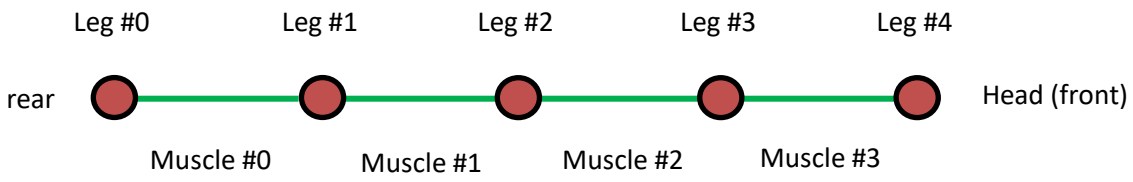

Figure 1

The body segments will be modeled as springs. Like the soft tissue in your own body, a Manduca body segment pushes back when you try to compress it.

Figure 2(a) shows a simple spring. It is anchored on one end, and free to move on the other. A basic spring has a resting length L_0, a mass m, and a spring constant k. Δx is the displacement of the spring's end from its equilibrium condition, or $x - L_0$.

The restoring force exerted by the spring is $F_s = -k\,\Delta x$

A perfect spring has no friction and never loses any energy. Real-life soft tissue returns to its original shape slowly, losing energy as heat in the process. A more realistic spring model therefore involves "damping", with a damping coefficient c. The bigger the damping coefficient, the more energy is lost as heat, and the slower the spring will return to its resting state. The damping force is expressed as $F_d = -c\,dx/dt$.

Applying Newton's second law to the mass m, we have $F = ma = m\,d^2x/dt^2$, where F is the total force on the mass.

The only forces acting on m are the spring's restoring force and viscous forces:
$m\,d^2x/dt^2 = -kx - c\,dx/dt$.

This can be re-arranged into:
$m\,d^2x/dt^2 + c\,dx/dt + k\,(x - L_0) = 0$

Manduca's muscles are controlled using an electrical pulse that makes them contract. This is called an actuation force, A. We will need to account for it!

Also, Manduca's muscles may not be anchored by its locked leg positions.

So we need to explore what a spring does when both ends are free! See figure 2(b). This situation is very similar to the case in Figure 2(a), but we need to additionally describe the forces on the other end of the spring. Two differential equations are required as follows:

$$m\,x_b'' = -k\,(x_b - x_a - L_0) - c\,(x_b' - x_a') - A \qquad (1)$$

$$m\,x_a'' = -k\,(x_a - x_b + L_0) - c\,(x_a' - x_b') + A \qquad (2)$$

Clearly, we cannot model Manduca as a single spring, so we can try to do it with two springs as follows:

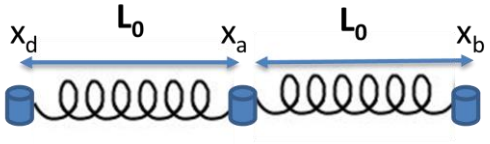

Figure 3. A three-legged Manduca.

The cylinders are legs – we will assume them unlocked. We can write the following equations for each of the points.

$$m\,x_b'' = -k\,(x_b - x_a - L_0) - c\,(x_b' - x_a') - A_{ab} \qquad (3)$$

$$m\,x_a'' = -k\,(x_a - x_b + L_0) - c\,(x_a' - x_b') + A_{ab} + \\ -k\,(x_a - x_d - L_0) - c\,(x_a' - x_d') - A_{da} \qquad (4)$$

$$m\,x_d'' = -k\,(x_d - x_a + L_0) - c\,(x_d' - x_a') + A_{da} \qquad (5)$$

This can be easily extended to capture the Manduca model in Figure 1. We can also extend it to model what happens when we lock one or more of Manduca's legs; i.e., that leg's position remains constant, while its velocity and acceleration become zero.