Lab 5: Particle Swarm Optimization

Overview

Particle Swarm Optimization (PSO) is a stochastic method for parameter estimation (contrast with deterministic methods like DIRECT, Mesh Adaptive Search, etc) that does not rely on the objective function being differentiable. Further advantages include speed, ease of parallelization, and very few parameters to set.

Unfortunately, PSO has some drawbacks, which we'll explore in this lab. The most important drawback is that PSO *does not guarantee convergence to a global optimum*. Contrast this with DIRECT, with does make such a guarantee. PSO can get "stuck" in local optima, depending on how the algorithm was initialized. The stochastic portion of the algorithm is intended to compensate for this shortcoming by perturbing the current best solution to see if there is a better one nearby.

Videos

Most similar to this lab: http://www.youtube.com/watch?v=tTNmx0xw2QY

Inspiration for PSO: http://www.youtube.com/watch?v=V71hz9wNsgs

Inspiration, simulated: http://www.youtube.com/watch?v=GUkjC-69vaw

Simulated crowds with PSO: http://www.youtube.com/watch?v=-jF5sAqBp4w

Pre-coding

Observing the movements of groups of insects, flocks of birds, and schools of fish were part of the inspiration for PSO. In searching for a global optimum across some parameter space, one can envision your current knowledge consisting of the observations of a bunch of birds flying across the objective surface and evaluating it at each point they interact with.

PSO is iterative, meaning at each step of the simulation, the parameters of the system are updated. The parameters of PSO, while few, bear mentioning:

- $\mathbf{x_i}$ The position of particle *i* at a given iteration
- $\mathbf{p_i}$ The best position of particle *i* at a given iteration (updated only when the position $\mathbf{x_i}$ is better than the last best position $\mathbf{p_i}$)
- $\mathbf{v_i}$ The velocity of particle *i* at a given iteration
- $\mathbf{g}$ The globally best position at a given iteration (best of all the $\mathbf{p_i}$'s)

- **c** The cognitive parameter. This controls how "self-aware" a particle is, or how much of its next position depends on its previous position. Higher value = better memory, lower value = more and more drunk(ard's walk)
- **s** The social parameter. This controls how much the movements of other particles influence this one. Higher value = more of a sheep, smaller value = less likely to jump off a cliff when everyone else does
- **w** The inertia parameter. This controls how much the particle's current velocity depends on its previous velocity. Higher value = no brakes, lower value = lead foot on the brake pedal (NOTE: at least in this lab, it's important to note that this value will decrease linearly throughout the duration of your simulation; this is generally good practice, as high initial inertia will give your particles good coverage of the parameter space, then as inertia falls will allow them to congregate around what is hopefully the global optimum without overshooting)

Coding!

In this example, you'll be using PSO to find the global optim[a|um] of an image, using pixel intensity as your objective function. To make things slightly interesting, I've encoded the image in a (not at all difficult to translate) binary format that Python can read but your standard image viewer cannot. As you evaluate the objective function for particles at each time step, those pixel values will be revealed in such a way that you will slowly enumerate the structure of the image.

Within the folder you downloaded from the website, you should have the following files:

- This document (hooray!)
- **mysterious.npy**: A NumPy-encoded image. If only we knew what it was…!
- **util.py**: A bunch of utility methods, fully implemented, that you (ideally) won't need to worry about.
- **lab5.py**: This contains the core of the PSO implementation. *This is the only file you will be modifying!*

You'll notice lab5.py is incomplete; specifically, the four update methods that form the heart of PSO are completely blank. The method stubs are there, so you can be assured you won't need any other parameters than what the methods provide. Furthermore, if you need a reference for how to interact with the Particle object, this is defined within the util.py file. Still, you should be able to look at how the methods are called within the simulation loop near the bottom of the file in order to figure out how the Particle objects work.

Here are the methods you'll need to implement (in order of use in the file):

- updateGlobalBest(): updates the entire simulation's best guess as to the global optimum
- updateVelocity(): updates a particle's velocity vector for the current iteration

- updatePosition(): updates a particle's position vector for the current iteration
- updateParticleBest(): updates a particle's personal best position vector

You do not need to modify anything else besides these four methods in order to get your PSO working!

You will need to make use of the **objective()** method in order to evaluate the fitness of a particle's current location. This method is already implemented; you just need to figure out when to use it.

Each method has different parameters available to it. Check the documentation string just below the definition of each method to see explanations of all the parameters you have access to for your calculations.

In order to test your code, you can see instructions on how to run it by typing the following on the command line:

```
python lab5.py –h
```

This will bring up the list of parameters available to you. You can run the script with no parameters, and the default values that are listed for all the parameters will be used. However, you may wish to tweak the values to see how they impact the algorithm's behavior. See the "pre-code" section for a full explanation on what each parameter controls.

In order to view the algorithm's output, have a look at the very bottom of the lab5.py file. Just above the line `if __name__ == '__main__'`, you'll see some comments. They explain how to uncomment some code that prints out graphs of your PSO's current state. The first one prints a graph for every iteration of the algorithm (obviously this can create a lot of graphs if you have a lot of iterations, but is very revealing as to your algorithm's behavior). The second one prints only a final graph, after all the iterations have run (makes the whole simulation run much faster).

I would suggest using the default number of iterations (20) as you tweak the other parameters of the algorithm and printing out every iteration's graph, then upping the number of iterations to 100 (or more!) and only printing out the final graph.

YOUR MISSION, THEN (SINCE YOU HAVE NO CHOICE BUT TO ACCEPT IT):

1. Check out the lecture notes "ParameterEstimation2"; toward the end is the summary on PSO.
2. Open up "lab5.py", scroll straight to the bottom where the simulation loop is, and make sure you understand what's going on.
3. **Start coding!** Implement the four methods in "lab5.py" that are screaming at you to "FIX ME!".

4. Before you test your code, uncomment one of the lines in the simulation loop for "plotIteration" (either the one that plots on every iteration, or just after all the iterations have finished). If you get overexcited and skip this step, or just don't like the number 4, you won't see any results. Booooooring.
5. Run your code! Simply calling `python lab5.py` from the command line will do the trick by using some sensible defaults.
6. Get creative; tweak the command line parameters. Remember the `–h` flag will bring up all the options you have to play with. See how much of the mystery image you can reveal, and where the global optim[a|um] are/is!

Other

Can you implement a better (aka: anything at all) terminating condition for this algorithm? This might show up on a homework assignment…

More comprehensive Python optimization library:
http://code.google.com/p/ecspy/