Homework 7: AlgCelerator

Due: Nov. 13, 2018 at 6:00 PM (early)
Nov. 16, 2018 at 6:00 PM (on time)
Nov. 18, 2018 at 6:00 PM (late)

This is a partner homework as usual: work on all problems together; you are responsible for everything you and your partner submit and it is an academic code violation to submit something that is not yours. Check for your partner in the Google doc that was sent out. **If you don't have a partner, please let us know.**

Each of the 4 problems in this assignment may be turned in separately, for a separate deadline. Run `cs157_handin hw7-p3` to copy everything from your current directory to our grading system as a submission for problem 3. (Change "3" to a number 1 to 4 for the other problems!)

Throughout this assignment you will be finding inputs that optimize (minimize) the output of certain functions. Each function can be found in `/course/cs157/pub/stencils/hw7`, along with stencil code and helper code. Each of these functions, in addition to outputting a number, will also graphically display its result (at most a few times per second, to give you some feedback without spending too much time rendering graphics). **Pay attention to the graphics! The functions you need to optimize automatically create great graphics for you. This is the only way to get intuition for what is going on in this assignment.** One good insight can save you hours of execution time! All the code for this assignment is at your fingertips, so also consider playing around with the starter code to get it to report different things, etc.

See the section on Matlab at the end of this document for reminders about helpful Matlab features. Matlab's graphics will occasionally crash; if this happens, press `Ctrl-C` to stop execution, and type `close all` to reset the graphics.

Make sure that you clearly indicate on each problem which track you're in (either **1-credit** or **full**). **1-credit track:** do problem 1 **first** and then **any 2** of the problems 2, 3, 4.

> Our last venture, **SmartUp**, was incredibly successful and produced dozens of new and promising startup ideas, each with VC funding. Unfortunately, our intern, Jasper, was tricked by a small garden gnome into selling all of the new companies on eBay. Saddened by this loss, we have decided to pivot to an incubator company and have invited the new owners of our previous startups to join us.
>
> Introducing **AlgCelerator**, our revolutionary new incubator that will disrupt the tech world, one idea at a time.

Problem 1

> One of our **AlgCelerator**-backed companies is **Speedarithms**, a startup whose mission is to be "the Uber of optimization". Its flagship algorithm is a high-tech version of local search.

The code that you create in this problem will be useful for the next three problems. If you understand the "punchlines" from this problem, you will need to write very little code for the rest of this assignment.

In this problem you will be writing a *local search* optimization routine, along with a few different routines for generating "local proposals". Local search, when given a function f to minimize, and a current input x, repeatedly tries to modify x so as to decrease $f(x)$. Each of these modifications is generated by a *local proposal* function, that, when given x, proposes a nearby x'. The simplest kind of local search accepts proposals that decrease f, and rejects proposals otherwise. As it turns out, it is often reasonable to allow "neutral" moves, that is, moves that do not affect the function value; further, it is even useful to allow *slightly harmful* moves, for example, for a parameter ϵ, accepting proposals that increase f by at most ϵ.

For this part, turn in code as described below. Document anything that needs clarification, though this part will not need much.

1. Fill in the code `localSearch`. In particular, we suggest you make use of the Matlab `now` function which returns the current date and time as a real number, in units of days (namely, comparing `now*60*60*24` at two different moments will will tell you how many seconds have passed in between); this will help you write a stopping criterion. This function will not be runnable until you write one of the proposal functions below.

   You will run this code with *function handles* as inputs, namely the symbol "@" before a function name. Many optimization problems (including problem 3 below) come with bounds on their inputs, and it is useful to make sure the proposals obey these bounds; thus your code will have parameters `lowerBoundOnX` and `upperBoundOnX` that will help ensure that proposals are valid.

   Note: `localSearch` can operate on a variety of variables as inputs, including multi-dimensional arrays. However, for the following assignment and the rest of the problems, you can assume that x will always be a one-dimensional vector. This will simplify some of the array manipulation.

   (**Hint:** You will find it very useful, later, for your `localSearch` function to output information about its partial progress. In particular, perhaps every time a proposal is accepted, report the value `func(x)` to the command window—by just putting this statement in the code, without a semicolon at the end.

2. Fill in the code `wideScaleRandomNoiseProposal`, which chooses a *single* radius from a wide range, and modifies each coordinate of the input by the (same) radius times a (different) randomly chosen positive or negative number (use `randn` instead of `rand` to get both positive and negative numbers). Make sure the radius is chosen so that it is in the range of at least $[0.0001, 100]$, and its *logarithm* is uniformly distributed. Set radius to be 10 to the power of a random number chosen uniformly, using `rand`. This kind of proposal function will usefully propose both small and large changes. (Can you intuitively see why?)

   Note: the Matlab command `size` (`http://www.mathworks.com/help/matlab/ref/size.html`) returns the size of each dimension of a Matlab variable, which may be useful for this code; alternatively, `numel` returns the number of elements in the variable, which is equivalent to the product of the dimensions.

3. Test your code from above. **Note:** Only part c needs to be turned in.

(a) Easy case: Try the 2-dimensional "horseshoe" function from class, optimized for 3 seconds (the parameter `inf` at the end means infinity, and should make your `localSearch` function cut off after exactly 3 seconds, according to the second to last parameter):

```
localSearch(@g5,@wideScaleRandomNoiseProposal,[0 0],0,-10,10,3,inf);
```

(b) Medium case: Try to optimize the (square of the) distance from the origin in 100-dimensional space, starting far from the origin. (Yes, we all know the optimum is at the origin, but does your proposal function let your local search routine find the origin efficiently?) Your routine should fairly quickly converge to value of around 10^{-5} (corresponding to distance 0.003 from the origin); if not, try to see why, intuitively, your local search routine is not making and accepting good proposals, and then go back to previous parts and make sure you are implementing the right thing.

```
x0=1000*randn(1,100);
localSearch(@(x)sum(x.^2),@wideScaleRandomNoiseProposal,x0,0,-inf,inf,10,inf);
```

(c) Hard case: Fill in the stencil `absMinimize` to minimize the sum of the absolute values function `f(x)=sum(abs(x))`, in 100 dimensions. While this is a convex function, and seems like it should be well-behaved, high dimensional geometry plays tricks on us. If, as above, you repeatedly search for local proposals that will improve the function value, you will very quickly find yourself in a deep narrow valley, where a small fraction of the 100 coordinates are large, the rest are near zero, and further progress downhill becomes vanishingly improbable. To see this, try setting `epsilons` to a vector with several zeros, as in `epsilons=[0 0 0 0]` in the stencil, and run it. To fix this, we need to try larger values of epsilon, which is counterintuitive: $\epsilon > 0$ means that local search will accept proposals that are *worse* than the current situation. Despite how strange this sounds, having larger values of epsilon emphasizes exploration instead of exploitation—when epsilon was 0, the search was getting stuck, so we need to make epsilon larger, to let local search get unstuck and explore more effectively. Your task here is to design a "schedule" for the epsilons, that starts with something high, and gradually decreases epsilon so that the function optimizes effectively. You know that epsilon is too small if the search freezes into a "spiky" configuration like you saw for $\epsilon = 0$; however, by the end of the schedule of epsilons, the last epsilon should be small enough for so **the objective function ends up less than 1**. See what intuition you can build about how the shape of this "cooling schedule" affects the optimization. In the problems below, see if clever choices of epsilon can help get you around obstacles.

4. In some cases modifying *all* the coordinates at once may be too drastic. Fill in the code `wideScaleRandomNoiseOneCoordinateProposal`, which functions exactly as above, except it only modifies a *single entry* of the input, chosen at random.

5. In problem 2 below, the proposal of the previous part will have the effect of modifying either the x or y coordinate of a random disk, though what you really want to do is modify *both* coordinates of a random disk. In this part, fill in the code `wideScaleRandomNoisePairProposal`, which chooses a random consecutive *pair* of coordinates, $(1, 2)$ or $(3, 4)$ or $(5, 6)$ or ... and modifies these entries randomly as above.

6. Sometimes many different types of proposals may all help. In this problem fill in the code `wideScaleRandomNoiseMix3Proposal` which picks a random one of the previous three functions and calls it to generate a proposal.

Problem 2

Another **AlgCelerator**-backed startup is **CircleSpace**, a circle-centric Chrome-extension venture revolutionizing the way shapes are incorporated into web design. In order to overcome users' Trypophobia[a], the creators wanted to inundate the senses with as many circles as possible as people explore the web. Thus, **CircleSpace** aims to fill all visited web pages with as many circles that can possibly fit within the rectangular browser. We need your help!

[a] https://en.wikipedia.org/wiki/Trypophobia

In this problem you will be minimizing the output of the function `arrangeCircles`, which attempts to arrange nonoverlapping disks of radii 1, 2, 3, ..., 10 so that they fit into the smallest square. Read the description written in the comments at the top of the file to figure out what the function does, and try running it with example inputs, such as `arrangeCircles(1:20)`. Your job is to minimize the output of this function.

You cannot expect to find the exact optimum of this function, but the methods of the previous part will get you quite far. Potential things to try include: play with the value of `epsilon` in the `localSearch` function; try restarting the search if it does not make progress (and perhaps modify `localSearch` to output debugging/partial progress information); try creating new proposal functions or reusing ones from other problems; if you have intuitions about which direction to modify an arrangement, you can try encoding your intuitions as hints to the local search routine via a modification of the `arrangeCircles` function—though keep in mind that your results will of course be evaluated on an unmodified `arrangeCircles` function.

Turn in:

1. The best arrangement of circles you have found, stored as a variable `x` in a file `bestcircles.mat`, via the Matlab command

   ```
   save bestcircles x
   ```

2. Runnable code that you used to generate `x` above, including a main routine `arrangeCirclesRunner` that calls (presumably) the `localSearch` function with a specially chosen set of parameters, along with anything else you found useful. **Important:** For this and subsequent problems, be sure to turn in `LocalSearch` along with any proposal functions it needs—even if you already turned them in for Problem 1. The course staff needs to be able to run your submission for this problem in order to grade it. Make sure you turn in runnable code!

3. The majority of your grade will come from the *explanation* found in the comments of `arrangeCirclesRunner`.

Your grade will be 40% from the quality of the solution `x` you found (in this problem, 41 is a good square size to aim for); 60% of your grade will come from the writeup, which should demonstrate an understanding of how you achieved the performance you did, explain any unexpected choices you made, and explain any innovations in your code. Unlike in previous assignments where everyone's code was supposed to be identical, this assignment encourages creativity and exploration; for us to be able to grade it, you need to document your code well. Points will be taken off if we cannot figure out how your code works. Note that because you will be using randomized algorithms on this assignment, running the same code twice will produce different results; nevertheless, you should aim to write code that is good and not just lucky, and your writeup should justify this. In principle you could run your code from now until the assignment is due and report the best result; but in such cases, we will not be able to re-run your code to verify your results, and you must rely on your writeup to convince us of your code's quality. Of course, code that accomplishes the same task, reliably, in less time, is preferred. However, please do not hard-code magic hand-picked initial conditions into your code, as this does not demonstrate general optimization principles, and you will lose points.

Problem 3

> Our venture capitalists have been so busy funding startups they have forgotten how to do basic human tasks like 2-legged walking. Instead, they have hired you to control their legs for them.

Optimization is amazing. You can write code to solve unusual, unexpected, complicated problems that you would not know how to solve yourself. In this problem you will try your hand at one of the legendary problems of robotics: moving around effectively on two legs. To give you a sense of how hard this problem is, try the Flash game at `http://www.foddy.net/Athletics.html` known as "QWOP".

You have a simulation of QWOP written in Matlab, called `VCwop`, that takes as input a sequence of 20 pairs of commands to our venture capitalist's legs, and you will need to write code to figure out how to get her as far forward as possible before hitting the ground or reaching the end of the 20 commands. (It turns out venture capitalists' legs are more springy than in the original QWOP game, so you might even witness a backflip if you are lucky!)

The input to `vcwop` is a sequence of 40 numbers, interpreted as 20 pairs of numbers, each between -1 and 1. The first number in each pair represents the force on the venture capitalist's thighs: positive brings one leg forward, negative brings the other leg forward. The second number in each pair represents the force on the venture capitalist's calves (the lower part of her legs): positive brings one foot forward, negative brings the other foot forward. Try a few example inputs:

```
vcwop(rand(2,20)*2-1)
```

Note that if `vcwop` is called from the command line, it will display an animation, but if it is called from within a function, it will display a "time–lapse" display of her trajectory. (Feel free to edit `vcwop` if you want to tweak this behavior.)

The return value of `vcwop` is negative the x-coordinate of the venture capitalist's hip when all 20 movements have been processed, or when her hip or head hits the ground, whichever happens first. (As usual, we are minimizing the result of this function, hence the negative sign; the venture capitalist still wants to get as far to the right as possible.)

This problem is rather open-ended, and there are many things you could try to improve the venture capitalist's performance. Intuitively, there is something sequential about this optimization problem: unless the venture capitalist starts out well, she will not get far. Bad starts for the runner include hitting her head early. Good starts might include flying upright at high speed to the right.

As explained in more detail for problem 2 above, turn in:

1. (40% of the points) The best parameters you have found, getting the runner as far to the right as possible, stored as a variable `x` in a file `bestvcwop.mat`, via the Matlab command

   ```
   save bestvcwop x
   ```

   Good performance is reaching location 8; getting past 9 is impressive.

2. Runnable code that you used to generate `x` above, including a main routine `vcwopRunner` that calls other optimization routines you wrote.

3. The majority of your grade will come from the *explanation* found in the comments of `vcwopRunner`.

The code for `vcwop` runs somewhat slow, since it is a physics simulation. So we are also providing C++ code for this function in `vcwopC.cpp`, runnable from Matlab as usual. However, you need to have a compatible C++ compiler installed; **compile the C++ code** by typing `mex vcwopC.cpp`, and follow instructions as needed (though this should work by default on department machines). You can then run your code by replacing any call like `vcwop(x)` with the 100+ times faster `vcwopC(x)`. The C++ code is new this year, so let us know if it causes problems (you might need to Google for how to install C++ compilers for Matlab on your system); but you can always just ignore this and use the regular Matlab code. Also, even after switching to C++ code, remember to plot or otherwise monitor partial progress of your optimization to develop intuition, for example, by calling the Matlab code `vcwop` when you have a trajectory worth displaying.

Problem 4

Introducing Amazon for Amazon (**A4A**). Our very own **AlgCelerator** has decided to back this promising startup due to the massive potential market. **A4A** cuts out the middle man (Amazon) and replaces it with a better middleman (**A4A**). First, **A4A** orders half of Amazon's warehouse stock, then lets the orders roll in from customers who have mistakenly navigated to the hundred most common misspellings of Amazon.com. However, **A4A** has one big problem, it seems that delivery is actually really hard! **A4A** has only one delivery truck (our very own intern Jasper) for the entirety of the United States. We need you to help plot out a more efficient route for Jasper to take.

In this problem you will write optimization code for the traveling salesman problem (NP-hard!). Locations of 100 American cities (latitude and longitude) are stored in the Matlab data file "cities.mat". (You can load this by typing `load cities`; you could play around with the data, or possibly even use it directly in your optimization code, though this is not necessary.)

The input to the function `travelingSalesman` is an order in which to visit the 100 cities, namely a *permutation of the numbers 1 to 100.* Try running this with the trivial ordering: `travelingSalesman(1:100)`.

The return value of this function is the total distance traveled: if your permutation starts 57, 12, 34, ..., then the total distance is the distance between cities 57 and 12, plus the distance between cities 12 and 34, etc. Your goal is to minimize this function.

Note that "wide scale random noise" is not an appropriate way to locally search the input space, since inputs must have a very particular form. Inputs must consist of exactly one copy of each of the numbers (cities) 1 through 100, in some order. Thus your challenge is to think of reasonable local proposals that transform valid permutations into other valid permutations. One example that was suggested in class (try this!) is to pick two random cities and swap them in the list. See how this works; see if it seems to be missing "obvious" things to try; come up with new proposals on your own; once you have some ideas for proposal functions, try a random mix of your best ideas!

As explained in more detail for problem 2 above, turn in:

1. (40% of the points) The best parameters you have found, describing the shortest tour of the 100 cities (expressed as a permutation of the numbers 1 through 100), stored as a variable `x` in a file `bestsalesman.mat`, via the Matlab command

   ```
   save bestsalesman x
   ```

   A good value to aim for is 350. The best we have seen is 327.

2. Runnable code that you used to generate `x` above, including a main routine `salesmanRunner` that calls other optimization routines you wrote.

3. The majority of your grade will come from the *explanation* found in the comments of `salesmanRunner`.

Matlab hints:

The Matlab command `dbstop if error` sets Matlab to go into debug mode any time there is an error, or you interrupt execution with `Ctrl-C`. The command prompt will change to "`K>>`", to indicate that Matlab is in the middle of code execution. You can also reach debug mode by putting breakpoints in your code. You can see the call stack by typing `dbstack`, and can move up and down the call stack by typing `dbup`, or `dbdown`. The easiest way to transfer variables to the base workspace is with the Matlab `save` and `load` commands, which saves variables to a specified `.mat` file on disk, and lets you load them later. To quit debugging mode, type `dbquit`; otherwise, you might get two or three levels deep in debugging mode, which can confuse you and Matlab.

To profile your code for speed, type `profview`, enter code to run in the code box, and click around.

As mentioned at the start, Matlab's graphics may occasionally freeze, in which case you should stop execution with `Ctrl-C`, reset the windows with `close all`, and, if the command prompt is `K>>`, type `dbquit`. Note that it is advisable to write your optimization code to output enough information about partial progress so that you can tell whether or not it is stuck.

You can download Matlab for free as a student. Here is the link to Brown's page about how to download Matlab: `http://www.brown.edu/information-technology/software/catalog/matlab`