Homework 7

Due: Apr 6, 2013 2:00 PM (early)
Apr 8, 2013 11:59 PM (on time)
Apr 10, 2013 11:59 PM (late)

This is a pair assignment, with all of the same rules as usual.

Each of the 5 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 5 for the other problems!)

Throughout this assignment you will be finding inputs to 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, can also graphically display its result, which is very useful for building intuition about how to improve your optimization code. One good insight can save you hours of execution time! When you call one of the functions with one input argument, `func(x)`, it evaluates the function at `x` without graphics; to graphically display the result, call the function with two arguments, as in `func(x,0)`—the value of the second argument is ignored. Keep in mind that displaying graphics takes time, so doing this too frequently will slow down your optimization.

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

Problem 1

Local search:

This problem will get you started with some code that you can use for the next three problems.

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); this will help you write a stopping criterion. This function will not be runnable until you write one of the proposal functions below. As in the lab on optimization, you will run this code with function handles as inputs, namely the symbol "@" before a function name.

Many optimization problems (including problems 3 and 4 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.

(**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. Even better, use `func(x,0)` to graphically display the output. However, since graphics is slow, you may want to only display graphics if, say, at least 0.1 seconds has passed since the last time graphics were displayed.)

2. Fill in the code `wideScaleRandomNoiseProposal`, which chooses a radius from a wide range, at least $[0.0001, 100]$, and modifies each coordinate of the input by the radius times a randomly chosen positive or negative number (use `randn` instead of `rand` to get negative numbers). Make sure the radius is chosen so that its *logarithm* is uniformly distributed (that is, choose a random number uniformly, with `rand`, and set radius to be 10 to the power of this random number). Note: the Matlab command `size` 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. 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.

4. 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.

5. 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.

You can test your code for this section before moving onto the next section with the function `g5` from the optimization lab (the input format of the function has changed from the lab, so use the version of `g5.m` distributed with this assignment, not the lab one). For example, run

```
localSearch(@g5,@wideScaleRandomNoiseProposal,[0 0],0,-10,10,10,0.01);
```

Note that because `g5` is a two-dimensional function, `wideScaleRandomNoiseProposal` and `wideScaleRandomNoisePairProposal` should have identical effect.

Problem 2

Circles:

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 graphically with example inputs, such as `arrangeCircles(1:20,0)` (remember, the extra "0" at the end asks the function to display its results graphically). 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; if you have intuitions about which direction to modify an arrangement, possibly try encoding your intuitions 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.

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. You have a week for this assignment, so in principle you could run your code for a week to produce an optimum; 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.

Problem 3

Bridges:

In this problem you will be designing a bridge that reaches as far across a width 100 chasm as possible before breaking. Because of the magic of optimization, you do not need to know anything about bridge design to do this. To be consistent with the rest of this assignment, you want to *minimize* the output of the function `chasm` which returns *negative* the distance the bridge reaches

across the chasm. Thus a return value of 0 means an awful bridge, and a return value of -100 means a perfect bridge that stretches across the entire 100 width of the chasm. The input to `chasm` is a 100-element vector describing the thickness of the bridge at each location; the thickness is allowed to be anywhere between 0.2 and 20.

Try running the code graphically with, say, a bridge shaped like a sine function:

```
chasm(10+5*sin((1:100)/2),0)
```

The colors on the bridge represent how much strain each segment is under, with yellow meaning low strain, and dark red meaning enough strain to break the bridge. The bridge is displayed as far out over the chasm as it can be without breaking, meaning that you "win" if you can get the bridge to stretch all the way to the other side. (Do not worry about the color of bridge segments that are resting "on land", as the bridge failed before these segments came into play.)

As in the previous problem, try to use the `localSearch` function to optimize the bridge. However, as discussed in class, you will need more than this to find a really good bridge in reasonable time.

Intuitively, you want a bridge that tapers, where the thickness starts large (to support the huge weight of the rest of the bridge as it is extended out into the chasm) and ends up very thin (so as to apply as little strain as possible to the parts of the bridge to the left). However, suppose you have a bridge where, say, at location 80 it is really thick, and fairly thin elsewhere. When the bridge is extended 20 units over the chasm, exposing segments 80 through 100, then the thickness at position 80 seems like a good thing; however, maybe by the time the bridge is extended 30 units over the chasm, when the really heavy segment 80 is now 10 units over empty air, this huge weight might cause the bridge to snap, yielding a return value of $(-)30$. The issue is that, while "intuitively" the problem is at position 80, the function we are optimizing only notices a "problem" at position 70. There are a variety of solutions to this, which include editing the return value of the function `chasm` itself so that we are optimizing a different function, or creating a proposal function that is somehow more appropriate for designing bridges. If you edit the `chasm` function, keep in mind that your answer will be tested against the original `chasm` function. Potential ideas for editing `chasm` include giving mild hints that "all else being equal, thinner is better", or trying to set up some kind of iterative optimization where the rightmost parts of the bridge are optimized first.

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

1. (40% of the points) The best bridge you have found, stored as a variable `x` in a file `bestbridge.mat`, via the Matlab command

   ```
   save bestbridge x
   ```

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

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

Problem 4

"Bilbwop":

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". In this problem, you will be playing this game via optimization code: you have a simulation of this game written in Matlab, called `bilbwop`, that takes as input a sequence of 20 pairs of commands to Bilbo's legs, and you will need to write code to figure out how to get Bilbo as far forward as possible before hitting the ground or reaching the end of the 20 commands.

The input to `bilbwop` 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 Bilbo's thighs: positive brings one leg forward, negative brings the other leg forward. The second number in each pair represents the force on Bilbo's calves (the lower part of his legs): positive brings one foot forward, negative brings the other foot forward. Try a few example inputs:

```
bilbwop(rand(20,2)*2-1,0)
```

The return value of `bilbwop` is negative the x-coordinate of Bilbo's hip when all 20 movements have been processed, or when his hip or head hits the ground, whichever happens first. (As usual, we are minimizing the result of this function, hence the negative sign; Bilbo 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 Bilbo's performance. As with the bridge problem above, there is something sequential about this optimization problem: unless Bilbo starts out well, he will not get far. Bad starts for Bilbo include hitting his 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 Bilbo as far to the right as possible, stored as a variable `x` in a file `bestbilbwop.mat`, via the Matlab command

   ```
   save bestbilbwop x
   ```

   (In lecture, we saw Bilbo getting to location 7.7, which is something to aim for.)

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

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

Problem 5

Image Denoising:

This problem is very different from the others on this assignment. While you will be writing code that optimizes a function, here you must *not* use local search.

Since cheap digital cameras are everywhere, an important problem with many applications and generalizations is "image denoising". Namely, given an image where the value at each pixel has

been corrupted by the addition of a random number, is there any way to recover the original image? (This corruption occurs at the hardware level of cameras, as cheap electronics cannot accurately read out the intensity of light at a given place on the camera sensor; the "original image" means the intensity of light at each gridpoint on the sensor, and it only ever reaches your computer in corrupted form.)

The language of optimization provides a very powerful framework to express different approaches to denoising. The general intuition behind denoising is that that, given an input image `inp`, we want to output an image `x` that balances two things: 1) `x` is "less noisy" than `inp`, and 2) `x` is "close to" `inp`. The reason for the second component is that otherwise optimization would return an all-gray image, which is certainly less noisy, but has nothing to do with the original image. Any way we choose to define the "noisiness" of `x`, and the "closeness" of two images `x` and `inp` will give us a different notion of denoising!

In this problem we will consider *quadratic* (L2) denoising. In lecture we will see L1 denoising, which is often much better, but much harder to implement. The function you will optimize is defined as the sum of the following two terms:

1. The sum over each pair of horizontally or vertically adjacent pixels of the *square* of the difference between their intensities in the image `x`, multiplied by a parameter k.

   For reasons which will become clear when you solve this problem, the notion of *adjacent* pixels here should *also* include: each pixel in the top row is adjacent to the corresponding pixel in the bottom row, and each pixel in the leftmost column is adjacent to the corresponding pixel in the rightmost column.

2. The sum over each pixel of the *square* of the difference between its intensity in `x` and its intensity in the input image `inp`.

That is, you should write a function `denoiseQuadratic` that takes in an input image `inp` (a 2-dimensional array; no color information), and a positive parameter `k`, and outputs an image `x` that minimizes the above function as computed on `x`, `inp`, and `k`.

However: You should *not* write optimization code. This problem is solvable with the magic of Fourier transforms in time $O(N \log N)$ for an image with N pixels. For other notions of denoising, this is impossible, but for quadratic denoising, this happens to work out, and your challenge is to figure out how.

One crucial fact about Fourier transforms: the sum of the squares of the magnitudes of x equals the sum of the squares of the magnitudes of its (1 or 2-dimensional) Fourier transform (possibly times a scaling factor, dependent on how the Fourier transform is defined, which does not matter). This is known as Parseval's theorem (in particular, for discrete Fourier transforms, see the last equation on the Wikipedia page.) We emphasize "magnitude" here because the entries of a Fourier transform may be complex numbers, and taking their magnitude gives us nice real numbers.

The final result of your code will look like a blurred version of the original image, though the question is *which* blurring. Non-quadratic notions of denoising can yield much more subtle results than just blurring, but they take a lot more work to compute.

(**Hints:** This problem makes crucial use of convolution. One of the last steps in solving this problem involves, for each pixel in the Fourier transform of the image, minimizing a quadratic function with

pencil+paper. Bear in mind that, due to rounding errors, some of the results may have unexpected imaginary components, so you might need to use the Matlab function *real* on your answers.)

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 so that you can tell whether or not it is stuck.