COMPSCI 527 — Homework 5

Due on November 19, 2015

Work on this assignment either alone or in pairs. You may work with different partners on different assignments, but you can only have up to one partner for any one assignment. You may not talk about this assignment with others until all of you have handed in their work. See Mechanics→Homework on the class web page for details on the homework policy.

Hand in your work as explained in the instructions for homework 1 (of course, change `hw1` to `hw5`).

There is not very much to *do* for this assignment, but you will need to read and understand code. Also, some questions are somewhat open-ended. Be clear and detailed but also concise in your answers.

1. After downloading the code that comes with this assignment and making the `code` directory your MATLAB workspace directory, type

```
TrackingExperiments
```

to the MATLAB prompt. The program will load eight images, run three different versions of a feature tracker on the same six image feature points, and show three groups of three images each. The first group runs the Lucas-Kanade tracker, which is Newton's method with a particular way of approximating the Hessian of the SSD (Sum of Squared Differences) error function.

(a) Look at Figures 1, 2, and 3. Of the six points being tracked, some are tracked well, some are lost during tracking, and some are tracked but do not correspond to fixed points in the world. State which point is in which of these three categories. For points in the last two categories, explain briefly and clearly in what way and for what reason the results are not satisfactory. Assume that the goal of tracking is to follow points in the world for use in reconstruction.

(b) The cell array `sN` returned in file `TrackingExperiments` is available in the MATLAB workspace after running this script. This cell array has size 7×6, and has one row for every interval between consecutive frames (1-2, 2-3, ..., 7-8) and one column per feature point in the initial frame. Entry, say, `sN{1, 3}` shows some information about the result of tracking feature 3 between frames 1 and 2. Specifically, the field `iteration` shows the number of the last iteration. Field `current` is a `point` data structure that shows the state of search that was current when the search terminated for that frame. A `point` has four fields: `x` is the value of the displacement $\mathbf{d}$ in that iteration, `y` is the value of the SSD function at `x`, and the fields `g` and `H` are the gradient and Hessian at that point in time. The fields `first` and `previous` in `sN{1, 3}` are also `point` data structures, but refer to the initial and previous value, rather than the current one. So you can find the Hessian of the SSD function in the last iteration for feature 3 between frames 1 and 2 as `sN{1, 3}.current.H`.

Use the MATLAB function `cond` and `svd` or `eig` to find the condition number $\text{cond}(H)$ and smaller singular value $\sigma_{\min}(H)$ of the Hessian H for all features in the last frame in which all features are present. State what frame that is, and make an understandable table with the required information. Approximate to two decimals.

(c) Do the values of either $\text{cond}(H)$ or $\sigma_{\min}(H)$ tell you that something is wrong with some of the features? Which values, what do they tell you, and why? Explain.

(d) Report the number of function evaluations (evaluations of the function `ssd` in file `ssdCost.m`) used by each of the three minimization methods.

(e) Are there any differences in the results across the three methods? Also consider resolution in your answer.

(f) The scene did not move while the images were taken. Can you guess in what direction the camera moved, and why can you tell?

2. All three experiments in `TrackingExperiments` are run by the function `experiment`, which in turn calls `track`, which calls `minimize`. The only thing that changes between the three experiments, other than the number of the first figure in which to display results, is the first argument to `experiment`, which is respectively set to `Newton`, `descent`, or `grid`. Each of these three variables contains a `method` data structure. A `method` describes how the generic function in `minimize` is to behave. This function initializes the `state` of a search (we encountered `state` structures earlier) and iterates by making a `method.move` until convergence, or until a function `cost.OK` determines that the current state is unacceptable, or until a function `method.done` determines that the search has converged.

Each of the three functions `NewtonMethod`, `descentMethod`, or `gridMethod` returns a different `method`. Every `method` must have a `name` field (a printable string with the method's name); a `move` field that is a handle to a function that performs the basic move during search; an `order` field that states how many derivatives are needed for that method (0 if only the value is needed, 1 if the gradient is needed, 2 if the Hessian is needed); a `done` field that is a handle to a function that returns `true` iff convergence has occurred, a `maxIteration` field that contains the maximum number of iterations allowed, and a boolean `verbose` field that determines whether to print out messages during execution of the search for a minimum. In addition, a `method` structure can contain any field that is needed for specific methods.

We covered all three search methods in class, including the `gridMethod`, which just tries all possible values on a fixed grid of given pitch and returns the position and value of the smallest cost it finds. We covered `descentMethod` in the context of back-propagation, and the method provided with this assignment uses no momentum. We covered Newton's method for the Lucas-Kanade tracker.

The `ssdCost` function makes a `cost` object, which contains a handle `cost.f` to the `ssd` function itself; a function handle `cost.OK` to a function that determines if a value encountered during search is OK; plus any parameters that either `cost.f` or `cost.OK` need to run (images, position `pos` of the feature in image I, window size `winsize`). Study in particular the functions `ssd` and `OK` in file `ssdCost.m`. Note that `ssd` takes an `order` argument. If `order` is 0, then the `point` returned by `ssd` contains only `x` and `y`. If `order` is 1, then `ssd` also places the gradient `g` into the `point`, and if `order` is 2, then `ssd` also places the Hessian `H` into the `point`. Newton's method is oder 2, gradient descent is order 1, and grid search is order 0.

Read the functions `NewtonMethod`, `descentMethod`, `gridMethod`, and `ssdCost` to understand how these functions are implemented, and look at `minimize` to see how they are called. This way of programming separates relatively cleanly[1] the general structure of a minimization search (implemented in `minimize`) from the specifics of the `method` and `cost`, at the expense of a slight degree of inefficiency.

(a) Write expressions for the gradient $\mathbf{g}$ and Hessian H of the following "banana" function:

$$f(\mathbf{x}) = (a - x_1)^2 + b(x_2 - x_1^2)^2$$

where a and b are parameters.

(b) At which point $\mathbf{x}^*$ does the "banana" function reach its minimum value, and what is the value $f(\mathbf{x}^*)$ of the function at the minimum?

(c) We know where the minimum of the "banana" function is, but we pretend we do not, and look for it by numerical minimization. Mimic the `TrackingEperiments` to create a new script `BananaExperiments.m` that tries the three methods above—Newton's method, gradient descent, and grid search—to minimize the "banana" function. Your task is to complete the file `bananaCost.m`, which already contains some code to get you started, uncomment the code that is commented out in the script file `BananaExperiments.m`, and replace the `NaN` on line 31 of this script with the appropriate vector from your previous answer.

PROGRAMMING NOTE. The code for function `banana` needs to be partially vectorized, just as that for function `ssd` in `ssdCost.m` in problem 1 is. For `ssd`, partial vectorization means that when the input argument `order` is zero, the input argument `d` can be a $2 \times n_d$ array, rather than a single 2×1 column vector. In that case, the function `ssd` returns a `point` where field `x` is all of `d` and field `y` is a vector of length n_d with all the SSD costs at `d(:, 1)`, ..., `d(:,` n_d`)`. This feature is convenient when plotting the SSD cost for a set of points, in which case the function `ssd` can be called just once. This code is only *partially* vectorized, because when `order` is greater than 0 vectorization is not implemented (so gradient and Hessian are not vectorized).

Your code for `banana` needs to be partially vectorized because the function `gridResidual` relies on this feature. Try not to use explicit `for` loops in `banana`.

[Hint: There is no constraint on cost here, so the `OK` function is trivial.]

Hand in your code for `bananaCost`, as well as a printout of the figure resulting from running the complete `BananaExperiments`. Do not cut-and-paste code, but rather use the `VerbatimInput` command. For instance,

```
\VerbatimInput[fontsize=\small,fontshape=n,xleftmargin=15mm]{bananaCost.m}
```

(d) The number of function evaluations for each method is shown in the legend of the figure you produced in your previous answer. Did any of the methods fail to converge in the maximum number of iterations allowed? If so, which method, and how many iterations?

(e) Which method took fewest iterations? How many iterations?

(f) Are individual iterations for each method equally expensive? If not, which is most expensive, and which is least?

(g) Explain why the path taken by grid search looks the way it looks.

(h) What would you need to do for grid search to achieve an accuracy similar to that achieved by the better of the other two methods? How expensive would that change be? Does the increase in expense depend on the dimensionality of the search space (which is 2 in the example)?

[1]Object-Oriented Programming (OOP) would make the separation even cleaner, but is avoided here because few students know the MATLAB flavor of OOP.