COMPSCI 527 — Homework 6
Due on November 17, 2016

Please refer to homework 1 and the class mechanics web page for homework policies and formatting/submission instructions. The submission checklist for this assignment is as follows:

`hw6.pdf`, `lineSearch.m`, `nGradient.m`, `nHessian.m`, `camelCost.m`, `eigenvalues2x2.m`.

Keep in mind that your code (or snippets thereof, as requested) must also show up in `hw6.pdf` .

Function Optimization

We explored several function optimization methods in class, including the following:

- Gradient descent with a step proportional to the gradient magnitude
- Gradient descent with line search
- Newton's method

We discussed the first method in the context of deep nets (that discussion also included a momentum term, which we set to zero in this assignment). We saw Newton's method both in the context of general function optimization and in relation to the Lucas-Kanade tracker.

These methods need first and sometimes second derivatives of the function

$$f(\mathbf{x}) \; : \; \mathbb{R}^n \to \mathbb{R}$$

to be minimized. In other words, the implementation of f must also be able to return the gradient $\mathbf{g}$ and Hessian H of f at the current point $\mathbf{x}$ upon request. We consider three options for differentiation in this assignment:

- *Exact differentiation* is possible when f is known in analytic form, and standard calculus can be used to compute its derivatives.
- *Numerical differentiation* approximates a derivative with a central finite difference:

  $$\frac{\partial f}{\partial x_i} \approx \frac{f(\mathbf{x} + \delta \mathbf{e}_i) - f(\mathbf{x} - \delta \mathbf{e}_i)}{2\delta}$$

  where $\mathbf{e}_i$ is the i-th column of the $n \times n$ identity matrix.
- When f is the Sum of Squared Differences (SSD) residual that the Lucas-Kanade method minimizes to track point features between two images, derivatives of f can be approximated by linearizing one of the images as a function of feature displacement, and then differentiating the resulting residual function analytically. A further approximation is involved when computing the Hessian of f, as explained in the notes on point correspondence. Let us call the resulting method *Lucas-Kanade differentiation*, even if the general idea is due to Newton and Raphson.

All these methods conform to the general framework implemented in the provided function `minimize.m`: The function f to be minimized and some of its derivatives (depending on the optimization method used) are computed at the initial point $\mathbf{x} = \mathbf{x}_0$, and then the optimizer makes repeated moves (whose nature depends on the optimization method as well) until either the move fails, or an undesired point $\mathbf{x}$ is reached (as determined by the caller), or one of several possible convergence criteria has been reached.

Programming Optimization Methods

For generality and flexibility, we use a mild form of object-oriented programming to implement the optimization methods for this assignment. With this approach, all optimization methods can be implemented with a single function `minimize` that takes as inputs information about the cost function and the optimization method to use, in addition to the starting point $\mathbf{x}_0$.

Specifically, the function `minimize` takes a `cost`, a `method`, and a starting point `x0`. It returns the final point `x` and a `state`:

```
function [x, state] = minimize(cost, method, x0)
```

Input and output variables are described next. While these descriptions are lengthy, files are provided with this assignment that also show how all these variables and functions are used.

`x0` and `x`: The initial point `x0` and final point `x` are $n \times 1$ vectors of reals. Note that `minimize` computes the point where the minimum is achieved, not the value of f at the minimum.

`cost`: The `cost` variable is not a function, but rather a data structure that has both mandatory and optional fields. The mandatory fields are described next. You may want to read this description while perusing the provided file `ssdCost.m`. This file implements a `cost` data structure for sum-of-squared-differences.

`cost.f` is a handle to the user-provided function to be minimized. The corresponding function has header

```
function point = f(x, cost, order)
```

The first argument `x` is the point or array of points on which the function is to be computed. Specifically, the first dimension of `x` is equal to n, the dimension of the domain of f. The simplest case occurs when `x` is $n \times 1$ (a column vector). This means that the caller wants to compute f at a single point, and the output from `cost.f` is a scalar. When `x` has more dimensions, the caller wants to compute f at several points. For instance, `x` could be a $3 \times 4 \times 8$ array. In that case, $n = 3$ and the user is asking for the values that f achieves at the $4 \times 8 = 32$ points in $\mathbb{R}^3$ specified in `x`. The output from `cost.f` in this case is a 4×8 array. This is an example of a *vectorized* function, which is particularly convenient when the values of f are needed on an entire grid of points. During optimization, $\mathbf{x}$ is always a single vector. Because of this, it is typically not necessary to also vectorize the computation of gradients and Hessians. An exception to this occurs in problem 5, where you are asked to vectorize gradient and Hessian as well.

The second argument to `cost.f` is the data structure `cost` itself. Since `cost` can have any number of optional arguments, this mechanism allows the user to pass any arguments the function needs to do its job, while keeping the header of the function fixed.

The argument `order` is equal to 0 if only the value $y = f(\mathbf{x})$ is needed (possibly vectorized); to 1 if both y and the gradient $\mathbf{g} = \nabla f(\mathbf{x})$ of f at $\mathbf{x}$ are needed; and to 2 if y, $\mathbf{g}$, and the $n \times n$, symmetric Hessian matrix

$$H = \begin{bmatrix} \frac{\partial f^2}{\partial x_1^2} & \cdots & \frac{\partial f^2}{\partial x_1 \partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f^2}{\partial x_n \partial x_1} & \cdots & \frac{\partial f^2}{\partial x_n^2} \end{bmatrix}$$

of f at $\mathbf{x}$ are all needed. Since the number of outputs from `cost.f` is variable but we need a fixed header (because `minimize` must be able to work with functions and methods of all kinds), the only output argument from `f` is `point`, a data structure with mandatory field `point.y` and optional fields `point.g` and `point.H` (when `H` is present, so must be `g`). So in its fullest form `point` is a rich description of a point of f, equivalent to its second-order Taylor expansion.

`cost.diffMethod` is one of the three strings `'exact'`, `'numerical'`, or `'Lucas-Kanade'` (each spelled exactly like this) that specifies the desired differentiation method. Not all methods are valid for all cost functions. For instance, exact differentiation is not available for the sum-of-squared differences cost function.

`cost.OK` is a handle to a user-provided function with header

```
function good = OK(cost, method, state)
```

that examines anything it wants from its inputs and returns `true` if everything is OK and optimization can proceed (provided that the minimization method does not itself determine that it is done). The function `OK` returns `false` otherwise, in which case optimization is aborted. The role of this function is to enforce any criteria the caller may want to impose on a solution. For instance, in point-feature tracking, it is advisable to abort optimization when the current point is too far from `x0`. For problems where no such constraints exist, the body of this function would be the single line

```
good = true;
```

It is up to `OK` to print out any diagnostic messages.

method: This variable specifies the optimization method to use, and has several mandatory fields, in addition to any method-specific optional fields. Files `descentMethod.m` and `NewtonMethod.m` are provided, and you may want to refer to them as you read the description below. The mandatory fields of `method` are as follows:

method.name: A string that specifies the name of the method, to be used in plots and figure titles.

method.move: A handle to a function with header

```
function state = move(cost, method, state)
```

that specifies how the method moves from the current state to the next state. The input and output `state` is a state variable that keeps track of the status and history of optimization and includes a field `state.x` that specifies the current value of $\mathbf{x}$. The `state` structure is described in greater detail later below.

method.order: The order of the method (0, 1, 2) as described under `cost.f` above.

method.done: A handle to a function that determines if convergence has been achieved. This function is provided for you in file `converged.m`.

method.delta, method.epsilon, method.maxMotion, method.maxIteration: Values used by `converged`. Please see this function for their meaning. In particular, `delta` and `maxMotion` specify a minimum and maximum amount of change $\|\mathbf{x}_k - \mathbf{x}_{k-1}\|$ in the value of $\mathbf{x}$ that is needed to continue optimization. Similarly, `epsilon` specifies a minimum amount of change $|f(\mathbf{x}_k) - f(\mathbf{x}_{k-1})|$ required to keep going, and `maxIteration` specifies the maximum number of iterations allowed.

method.verbose: `true` if diagnostic messages are to be printed, `false` otherwise.

In addition, `method` may contain whatever additional fields a particular optimization method requires to do its job.

state: The `state` variable is a data structure that keeps track of both the status and history of optimization. It always has the following fields:

state.first, state.current, state.previous: These `point` data structures (`point` was described under `cost.f` above) keep track of the initial, current, and previous $\mathbf{x}$ and, if applicable, gradient and Hessian.

state.iteration: An integer that specifies the number of the current iteration, starting at 1.

In addition, if the user calls `minimize` with two output arguments, the `state` structure also keeps track of optimization history with the following fields:

state.history: A vector of all `point` structures traversed during optimization. *You will make extensive use of this field in this assignment.*

state.method: A copy of `method.name`, useful in plots.

state.diffMethod: A copy of `cost.diffMethod`, useful in plots.

state.version: This field is optional, and is a copy of `method.version` if the latter field exists. This field is defined in `descentMethod`, and is used to keep track of which variant of steepest descent (`'alpha step'` or `'line search'`) is being used. This field is not defined in `NewtonMethod`.

Line Search

1. The provided file `lineSearch.m` initializes the bracketing triple for line search. Complete the code in that file as described in the class notes on optimization. You do *not* need to optimize by golden search, just implement the basic method on page 5 of the notes. The optional parameter `delta` specifies the width $c - a$ of the bracketing triple (a, b, c) below which iterations will stop, and defaults to 10^{-5}.

Hand in your code (both in the PDF file and as a separate MATLAB file). *In your PDF file, include only the part of code you wrote, not the entire function.* Then show the results from the following calls:

```
lineSearch(@(x) sin(x), 0, 1)
lineSearch(@(x) -cos(x), pi/2, sin(pi/2))
lineSearch(@(x) x'*x, [1; 2], [2; 4])
lineSearch(@(x) x'*x, [0; 0], [0; 0])
```

Make sure the results are correct, or else you'll be in trouble for later problems.

Differentiation

2. Use calculus to compute formulas for the gradient $\mathbf{g}(\mathbf{x})$ and Hessian $H(\mathbf{x})$ of the function

$$f(x_1, x_2) = 2x_1^2 - 1.05x_1^4 + \frac{x_1^6}{6} + x_1x_2 + x_2^2$$

where $\mathbf{x} = [x_1, x_2]^T$.

3. Write a function with header

```
function g = nGradient(f, x, delta, vars)
```

that uses central finite differences (defined in the introduction to this assignment) to compute `g`, a column vector that approximates the gradient of `f` at `x`. The optional parameter δ was defined in the introduction, and is not to be confused with other deltas in this assignment. The optional parameter `vars` is a vector of indices that specifies which variables to include in the gradient. If `vars` is not specified, it defaults to `1:length(x)`, and the full gradient is then computed. Otherwise, if `vars` is a vector $[v_1, \ldots, v_k]$ with $k \leq n$, the output approximates

$$\mathrm{g} = \begin{bmatrix} \frac{\partial f}{\partial x_{v_1}} \\ \vdots \\ \frac{\partial f}{\partial x_{v_k}} \end{bmatrix}.$$

The flexibility afforded by `vars` will be useful later. A template file `nGradient.m` is provided. It is OK to loop for i between 1 and the length of `vars`.

Hand in your code (both in the PDF file and as a separate MATLAB file) and show the result of the call

```
nGradient(@(x) x'*x, [1;2;3])
```

Make sure your result is correct, or else you'll be in trouble for later problems.

4. Write a function with header

```
function H = nHessian(f, x, delta)
```

that uses `nGradient` to compute the Hessian matrix of `f` at `x` (a single column vector). The meaning of the parameters is the same as the correspondingly named parameters in `nGradient`. Use the minimal number of function evaluations needed, keeping in mind that the Hessian is a symmetric matrix. It is OK to loop for i between 1 and the length of `x`. A double loop is not necessary, but will be tolerated. A template `nHessian.m` is provided. [Hints: The Hessian is the gradient of the gradient. The MATLAB function `triu` might be useful as well.] Hand in your code (both in the PDF file and as a separate MATLAB file) and show the result of the call

```
nHessian(@(x) x'*x, [1;2;3])
```

Make sure your result is correct, or else you'll be in trouble for later problems.

5. The provided file `camelCost.m` contains a partial implementation of a function with header

```
function cost = camelCost(diffMethod)
```

that returns the `cost` data structure for the function f in problem 2. Complete the implementation of `camelCost` by replacing the comments that start with

```
% Your code here
```

with appropriate code.

You should compute gradient and Hessian only as dictated by the `order` argument. That is, compute these quantities only if requested. You should do so with exact or numerical differentiation depending on the value of `diffMethod` (no `Lucas-Kanade` option, of course).

Vectorization: The computations of `point.y`, `point.g`, and `point.H` must be vectorized for exact differentiation. Specifically, if the input `x` to `cost.f` has size `[2 sz]`, then the outputs satisfy the following size predicates:

```
all(size(point.y) == sz)
all(size(point.g) == [2 sz])
all(size(point.H) == [2 2 sz])
```

For numerical differentiation, only the computation of `point.y` needs to be vectorized.

Hand in your code (both in the PDF file and as a separate MATLAB file). *In your PDF file, include only the part of code you wrote, not the entire function. Specify what part (see comments in `camelCost.m` for part numbers) each snippet of code refers to.* Then show a `mesh` plot of the array `ePoint.y` resulting from the calls

```
camelExact = camelCost('exact');
ePoint = camelExact.f(x, camelExact, 2);
```

(the `order` parameter is 2 because you will need the Hessian in a later problem). In the snippet above, `x` is a $2 \times 101 \times 101$ array of points on a 101×101 grid of uniformly spaced samples in the square

$$-2 \leq x_1 \leq 2 \quad , \quad -2 \leq x_1 \leq 2 \, .$$

The provided script `problem5.m` makes `x` for you. It also makes vectors with `x1` and `x2`, which are useful for plotting. Label the axes appropriately.

6. Show a plot of the magnitude of the gradient obtained by exact differentiation of the function in problem 2 along the line with equation $x_2 = 0$ and for x_1 between -2 and 2. [Hint: You already have the samples for this plot from the previous problem.] Also show a separate plot of the difference between the magnitude of the gradient computed on this line by exact and by numerical differentiation. This difference will be very small but nonzero. Label the axes appropriately. No code needed here.

7. Write a function with header

```
function lambda = eigenvalues2x2(H)
```

that takes an arbitrary array `H` of `double` numbers with size `sz = [2, 2, ...]`) and computes an array `lambda` of size `sz(2:end)` that contains the eigenvalues of the 2×2 matrices in `H`. To avoid using for loops over the entries of `H`, you cannot use the MATLAB function `eig` to compute eigenvalues. Instead, your code should use the analytic formula for computing the eigenvalues of a 2×2 matrix. [Partial credit if you use `eig` and loops.]

Hand in your code (both in the PDF file and as a separate MATLAB file). If you did the math in problem 2 correctly, the Hessian of the function f in problem 2 does not depend on x_2. Give a single diagram with two plots showing the maximum and minimum eigenvalue of that Hessian as a function of x_1. Include a legend that identifies which plot is which.

Optimization

8. Use `descentMethod`, `NewtonMethod`, and `camelCost` to find a minimum of the function f in problem 2 by starting at $\mathbf{x}_0 = [0, 1.5]^T$ and then at $\mathbf{x}_0 = [1, 1.5]^T$. For each of these starting points, use

- Gradient descent with alpha step
- Gradient descent with line search (you implemented line search in an earlier problem)
- Newton's method

Use exact differentiation in all cases. You may use values defined in the provided file `problem9.m` (for the next problem) as parameter values for the various methods. Set $\alpha = 0.1$ for the first method.

Do not hand in any code. Instead, make a contour plot of $f(\mathbf{x})$ for x_1 and x_2 ranging between -2 and 2 (101 samples in each direction) with the command

```
contour(x1, x2, ePoint.y, 20, 'k');
```

where `ePoint` is what you computed in problem 5. Then `hold on` to the plot and superimpose plots of the six paths corresponding to the optimizations described above. Make the six paths distinguishable by using suitable markers (not just color), and add a legend that describes each path unambiguously and legibly. In the legend or in separate text, also note the number of steps each optimization took. Make your figure as wide as a whole page, so it can be read easily.

[Hints: if `s` is the `state` variable returned as second argument from `minimize`, then you can extract the path with the command

```
path = [s.history(:).x];
```

If `s.history` has length k, then $k - 1$ steps were taken.]

Tracking

The provided file `ssdCost.m` implements the cost function for sum-of-squared-differences tracking. For completeness, it implements both numerical differentiation and the Lucas-Kanade method. However, you will only need the Lucas-Kanade version for the problems in this section.

You can make an SSD cost object by calling

```
cost = ssdCost(I, J, p0, hWin);
```

which invokes the Lucas-Kanade method by default. The argument `p0` gives the point in `I` to be tracked, a 2×1 vector with the row and column coordinates (in this order). The argument `hWin` is half the window size used by the tracker.

The two images `I` and `J` provided with this assignment (do `load data` to retrieve them) are of type `single` and are slightly blurred to prevent trouble when differentiating them. These two images are related by an artificial distortion. Specifically, pixel `I(r, c)` moves to pixel

```
J(r + D(r, c, 1), c + D(r, c, 2))
```

where the array `D` is provided as well. Thus, `D` contains the true displacements from `I` to `J`. You can use this array to determine how well the Lucas-Kanade tracker performs.

9. The file `problem9` provided with this assignment sets all the necessary parameters and tracks a single point, so you see how to use the relevant functions. Without changing any of the tracking parameters, track each of the 49 points in the provided array `P0` (found in `data.mat`). The code in `problem9` shows those points superimposed on image `I`. You will need to loop explicitly over the points to track each of them. To track a different point you need to either create a different `cost` object, or set `cost.pos` to the new point coordinates.

Give no code for this problem, but provide three figures instead.

- The first figure shows image `J`. Superimposed on it are the the true tracking results (computed from `P0` and `D`) drawn as small circles (draw these in a visible color; red works well). Each circle is connected by a line segment to an asterisk that shows the corresponding tracking result. See the figure made by `problem9` for an example. Note, however, that `problem9` superimposes the points on `I` rather than `J`.

  Most points are tracked to within a pixel or two from the correct results, so their circles and asterisks overlap, but some points are not tracked as well. [Hint: All the drawing for this figure can be done with the `plot` command. Use the `'o'` marker for circles and the `'*'` marker for asterisks.]

- The second figure is a histogram of the magnitudes of the errors you displayed in the first image. Do this by calling

  ```
  hist(n, 1:16);
  ```

  where `n` is a vector with the 49 error magnitudes (the error magnitude is the Euclidean norm of the difference between true and computed displacement.)

- The third figure shows no image, but only displays the optimization paths traversed by the tracker for the 49 points. The path for point `k` should start from `P0(:, k)`, so you will need to compute these paths from `P0` and the histories returned by `minimize`. Show each path as a black line starting at a cross (`'x'` marker at `P0(:, k)` for point `k`) and ending at an asterisk. Draw a 400×400 square around your plot to represent the image boundaries.

10. Pick one of the points from the previous problem for which the tracker worked well, and one for which it did not (your choice). State the row, column coordinates of the two points (make sure you say which coordinates are for the bad point, which for the good). Give two mesh plots of the SSD residual surface for the two cases, and explain in what way the two plots indicate that the problem for the good point is more likely to have a reliable solution than the one for the bad point. Feel free to add more pictures or other data if they help make your point. [Hints: the SSD residual surface is the function $e(\mathbf{d}) = \epsilon(\mathbf{x}_I, \mathbf{d})$ in equation (1) of the class notes on point correspondence. Pick a range of values for $\mathbf{d}$ that supports your point.]