Briefing Notes—Optimization by Hill-Climbing
Supporting material for MECH 222 Computer Lab 6

Summary. We explore two ways to approximate the maximum value of a given function. First is exhaustive search, which is crude and slow, but effective in low dimensions. Next is gradient-ascent, which is more efficient, more accurate, and extends easily to higher dimensions.

IDEA 1: *Simple Search in One Dimension.*

Imagine a function $f = f(x)$ of just one variable. Our job is to find the maximum value of f for inputs in some given interval $[a, b] = \{x : a \le x \le b\}$. If a vectorized Matlab representation of f is available as function `f`, we can locate the maximizer with an absolute error not larger than 0.01 by evaluating f at many closely-spaced points and selecting the input that gives the largest result. This takes just a few lines:

```
Xmesh = a : 0.01 : b;          % Many x-values, separated by 0.01
Ymesh = f(Xmesh);              % Calculate many y-values (f is "vectorized")
[ymax,k] = max(Ymesh);         % Built-in function max works on vectors.  See Idea 2.4.
xmax = Xmesh(k);               % "max" reports the location of the largest entry in Y
disp(sprintf('Max value is %5.3f, found at x=%5.3f',[ymax,xmax]));
```

Please read the comments in the code above. The built-in function `max` makes everything work, because it returns not just the largest component in a given vector, but also the location of that largest component. The corresponding component in the vector of input-values contains the desired x coordinate.

Take a moment to think about the following questions.

- What pictures could you draw to illustrate the process above?
- How accurate and efficient is this method?
- Could calculus be used to improve the results? How?

The last line in the block of code above provides a prototype for nicely formatted numerical output. There is a brief discussion of how this works in Idea 4, below.

IDEA 2: *Simple Search in Two Dimensions*

Now consider a scalar function $f = f(x, y)$ with two input variables, defined on a solid rectangle in (x, y)-space:

$$S = [a, b] \times [c, d] = \{(x, y) : a \le x \le b,\ c \le y \le d\}. \tag{1}$$

This writeup uses $f(x, y) = (2y - y^2)\sin(x)$ and $S = [0, 4] \times [0, 2]$, but the methods apply to a whole rainbow of alternatives.

2.1. Computer evaluation. A mathematical function of two variables translates easily into a Matlab function with two input arguments. Here is a three-line file named `f.m` that encodes our sample function:

```
% F - Evaluate the given function of two variables
function z = f(x,y)
z = (2*y - y.^2) .* sin(x);
```

Notice the vectorized operators ".^" and ".*". These allow us to evaluate many f-values with a single command: when `X` and `Y` are Matlab vectors or matrices with identical shapes, the command `Z = f(X,Y)` produces a new vector or matrix of the same shape. Each element of `Z` contains the f-value of the inputs in the corresponding elements of `X` and `Y`.

2.2. Meshes and Contours. The Matlab command `meshgrid` builds a rectangular grid of evaluation points from prescribed nodes on the coordinate axes. Here's how:

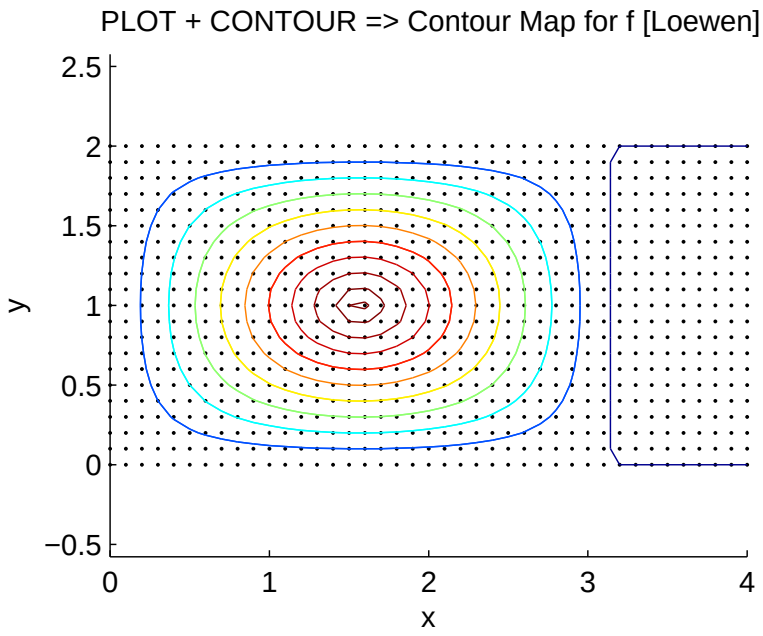
```
x = 0 : 0.1 : 2;                % 20 subintervals; 21 nodes
y = 0 : 0.1 : 4;                % 40 subintervals; 41 nodes
[Xmesh,Ymesh] = meshgrid(x,y);    % Create two matrices of shape 41-by-21
```

To display the mesh of evaluation points in the plane, say

```
figure('Name','My Mesh Points')         % Open a new window for plotting
plot(Xmesh,Ymesh,'k.','MarkerSize',8);  % Use mesh matrices Xmesh, Ymesh found above
```

Matlab automatically scales both axes independently to produce a figure that nearly fills the plot window. This often makes a perfectly square grid like ours look like the cells are rectangular. To insist on proportional scaling for both axes, say

```
axis equal;                % Enforce length ratio of 1:1 on both axes
hold on;                   % Optional:  allow new plots to overlay this one
```



The Matlab command `contour` adds a contour map of f to the sketch:

```
Zmesh = f(Xmesh,Ymesh);                % Calculate many z-values with a single command
contour(Xmesh,Ymesh,Zmesh);
```

Matlab finds the contours by doing linear interpolation between the function-values given at the node points—the same method students still use to predict steam-table entries for situations that fall in the gaps between situations given in the table. You can ask for 25 equally-spaced contours by inserting an optional argument:

```
contour(Xmesh,Ymesh,Zmesh,25);
```

Contour-counts other than 25 work the same say. Alternatively, you can replace the positive integer 25 with a vector listing the function values whose contours you want:

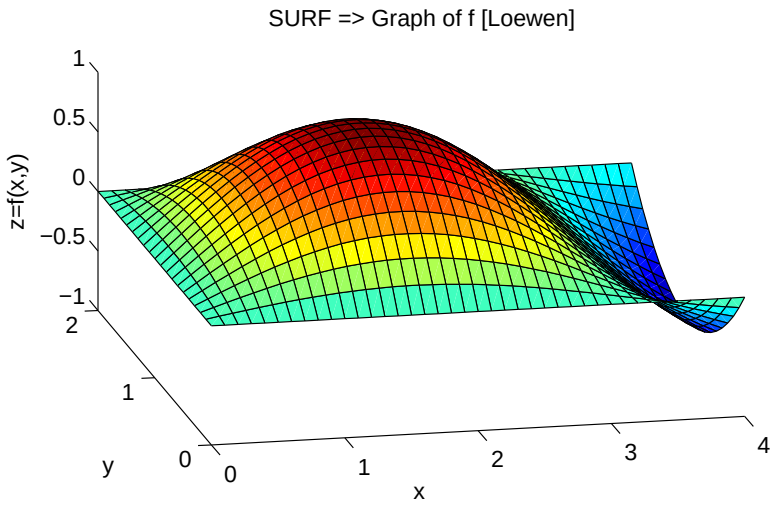
```
z_vals = Zmesh(16,:);          % Typical f-values including one near the max
contour(Xmesh,Ymesh,Zmesh,z_vals);
```

Reasonably close to the maximizing point, the contours are approximately elliptical. There are good reasons for this: see Idea 4.3, below.

2.3. Meshes and Surfaces. The graph of f is a surface in (x, y, z)-space, defined as the set of Cartesian triples satisfying $z = f(x, y)$. We can sketch it using

```
figure('Name','Graph of f') % Open a new window for plotting
surf(Xmesh,Ymesh,Zmesh);
```

Saying `surfc` instead of `surf` will include a few contours, but there is no provision for specifying which contours you want. Saying `mesh` instead of `surf` gives a wireframe picture of the surface.



2.4. Naïve Maximization. Matlab's built-in function `max` works on matrices as well as on vectors. According to `doc max`, "If `A` is a matrix, `max(A)` treats the columns of `A` as vectors, returning a row vector containing the maximum element from each column. ... `[C,I] = max(...)` finds the indices of the maximum values of `A`, and returns them in output vector `I`." So for the matrix `Z` of values above, the command

```
[C,I] = max(Zmesh);
```

will produce a row-vector `C` containing the largest entry in each column and another row-vector `I` that reports, for each column, which row contains the maximizer. We can find the overall maximum value `zmax` by selecting the largest of the column-maxima; that number and the index of the winning column `j` are given by

```
[zmax,j] = max(C);
```

The winning row will be `i = I(j)`. All in one block,

```
x = 0 : 0.01 : 4;                % 400 subintervals; 401 nodes
y = 0 : 0.01 : 2;                % 200 subintervals; 201 nodes
[Xmesh,Ymesh] = meshgrid(x,y);   % Create two matrices of shape 201-by-401
Zmesh = f(Xmesh,Ymesh);          % Many evaluations of f packed into one line
[C,I] = max(Zmesh);              % Find max in each column
[zmax,j] = max(C);               % Find largest among column-maxima
i = I(j);                        % Column j is best, and that selects row i
xmax = Xmesh(i,j);               % x-coordinate of optimal input
ymax = Ymesh(i,j);               % y-coordinate of optimal input
disp(['Maximum value of f(x,y) is ',num2str(zmax)])
disp(['found when x = ',num2str(xmax),', y = ',num2str(ymax)])
```

IDEA 3: *Gradients and Hill-Climbing*

3.1. Computing Partial Derivatives. Given a function $f = f(x, y)$ and a point of interest (x_0, y_0), the gradient $\nabla f(x_0, y_0)$ is the following two-component vector:

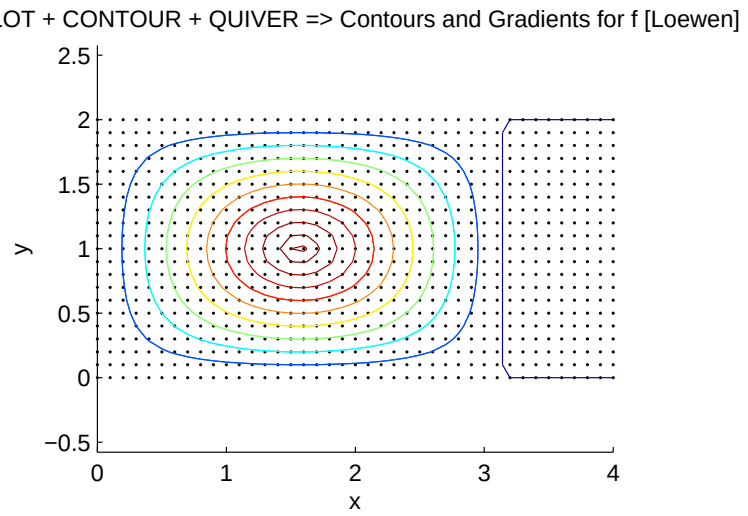
$$\nabla f(x_0, y_0) = \langle f_x(x_0, y_0),\ f_y(x_0, y_0) \rangle\,.$$

The partial derivatives have definitions discussed in Math class. When f is well-behaved near the point of interest, they enjoy the following additional properties:

$$f_x(x_0, y_0) = \frac{f(x_0 + h, y_0) - f(x_0 - h, y_0)}{2h} + O(h^2) \qquad \text{as } h \to 0^+,$$
$$f_y(x_0, y_0) = \frac{f(x_0, y_0 + k) - f(x_0, y_0 - k)}{2k} + O(k^2) \qquad \text{as } k \to 0^+.$$

Choosing specific small values of $h, k > 0$ and ignoring the error terms above provides a usable approximation to the gradient. How small should these perturbations be? It depends on the machine precision and the coordinates of the point of interest. For general use, we want $|h/x_0|$ and $|k/y_0|$ to be around 10^{-6}. Since our inputs are around 1, we can choose $h = k = 10^{-6}$.

3.2. Gradient Geometry. At each point (x_0, y_0), the vector $\nabla f(x_0, y_0)$ is perpendicular to the contour of f through (x_0, y_0). To illustrate this with a picture like the one below, give the single Matlab command `quiver(X,Y,U,V)`. Here the matrices `X` and `Y` provide the mesh-points where certain vector arrows are to be drawn. The vector arrow at point $(X(i,j), Y(i,j))$ will be $\langle U(i,j), V(i,j) \rangle$, so we must set up for the `quiver` command by calculating appropriate matrices U and V. (Here the plotting command `axis equal` is particularly important: stretching one axis more than the other modifies the angles in the picture.) There is no particularly elegant way to do this: a simple approach is to write two nested for-loops that generate all (i, j)-pairs of interest and to work out $U = f_x$ and $V = f_y$ at each point using the formulas suggested in idea 3.1. Note, too, that the vector arrows may be too long or too short to look good in your plot: after you build the matrices `U` and `V`, you may need to overwrite them with `m*U` and `m*V` before plotting, using some positive scalar `m` that you choose to make the diagram especially clear.



3.3. Hill Climbing Trajectories. At every point (x, y) where f is differentiable, the vector $\nabla f(x, y)$ points in the direction of steepest increase for f. So one way to make steady progress toward larger f-values is to imagine a point that moves in the (x, y)-plane in such a way that its velocity is parallel to ∇f no matter where it is. This idea is captured by the system of differential equations

$$\left\langle \frac{dx}{dt}, \frac{dy}{dt} \right\rangle = \nabla f(x(t), y(t)) = \left\langle f_x(x(t), y(t)), f_y(x(t), y(t)) \right\rangle. \qquad (*)$$

Matlab's `ode45` command deals nicely with systems like this. Its model equation, shown here using the traditional position-vector notation $\mathbf{r} = (x, y)$, is

$$\dot{\mathbf{r}}(t) = \mathbf{G}(t, \mathbf{r}(t)), \qquad \mathbf{r}(t_0) = \mathbf{r_0}.$$

To align this with $(*)$, we will need to create a file `G.m` that defines a function `G(t,R)`. The vector input `R` will provide the 2-element evaluation point for ∇f. The return value of `G(t,R)` must be a *column vector* containing the components of $\nabla f(x, y)$ where `R(1)` $= x$ and `R(2)` $= y$. The scalar input `t` for `G` remains unused when the function is evaluated, but it must be present to match the structural expectations of `ode45`.

Any point where f has a local maximum will satisfy $\nabla f = \mathbf{0}$. So this is a point where the trajectories of equation $(*)$ stop moving. We can expect the hill-climbing point particle designed above to slow down as it approaches the desired maximizing point. If we run the differential equation for long enough, we can take its final state as a good approximation to the maximizer. Here are some commands to do this, assuming a suitable function `G` is available:

```
interval = [0,20];          % Choose final time 20 (experiment)
startpt  = [0,0];           % Launch hill-climber from (x,y)=(0,0)
options = odeset('RelTol',1e-6,'AbsTol',[1e-6 1e-6]);
[T,XYtraj] = ode45(@gradf,interval,startpt,options);
```

The quantities returned by `ode45` describe a list of points from a curve in the (x, y)-plane parametrized by t. The t-values are listed in the column vector `T`, while the coordinates are packed into the tall skinny matrix `XYtraj`. Individual coordinate evolutions can be extracted using

```
Xtraj = XYtraj(:,1);
Ytraj = XYtraj(:,2);
```

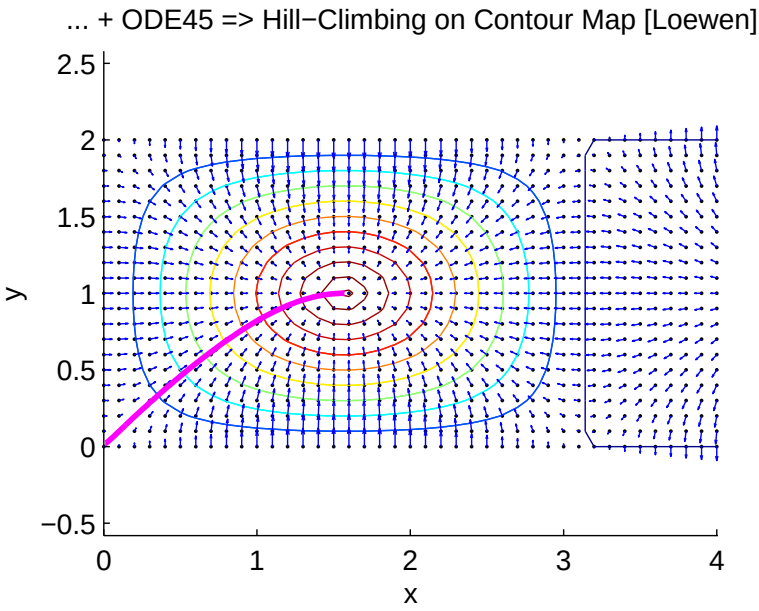
To draw a map of the hill-climber's path with a fat line in marvellous magenta, say

```
plot(Xtraj,Ytraj,'m','LineWidth',1);
```

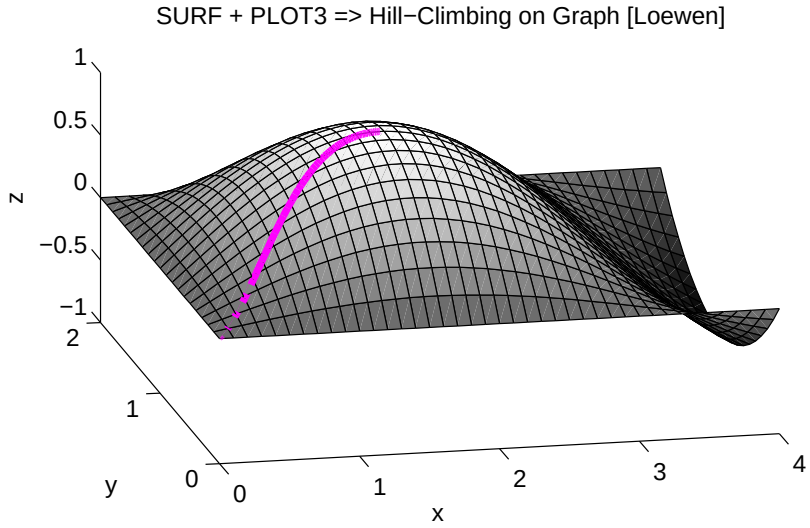
A sample figure appears below.

Caution: This method completely ignores the set S specified in line (1) of Idea 2. If f has only a *local* maximum in S and larger values elsewhere, our hill-climbing point can leave S and take off on a long journey to higher values far outside the region of interest. Avoid this by being aware of the possibility, and choosing initial points for which it doesn't happen. (Every method has its shortcomings, and this level of intervention is one of the undesirable features here.)

3.4. Locating the Peak. The vectors `T`, `Xtraj`, and `Ytraj` all have the same number of elements, which is determined at run-time by `ode45`. Saying `N = length(T)` will store that number in the variable `N` so you can refer to it. If the hill-climbing idea succeeds, the coordinates of the approximate maximizer will be $x =$`Xtraj(N)` and $y =$`Ytraj(N)`. (Failure is indeed possible, especially if the time interval passed to `ode45` is too short for the trajectory to make its way to a critical point.)



3.5. Another View of the Process. Paragraph 3.3 aboveshows a trajectory in the (x, y)-plane that visits points with increasing function-values. Since f has just two input variables, we can draw a different picture to see both the moving input point and the rising function value. The key is to draw the graph $z = f(x, y)$ as a surface in 3-space, and then overlay it with the space curve generated by solving the differential equation $(*)$: this is the parametric curve where $x = x(t)$ and $y = y(t)$ are returned by `ode45` and $z(t) = f(x(t), y(t))$ can be calculated by our known function f. Matlab's function `plot3` draws curves in space. In this alternative approach, the trajectory shown on the contour map in paragraph 3.3 produces the following graphic:



IDEA 4: *Beautifying Numerical Output*

To translate calculated values into character strings, displaying the numbers with precise formatting control, Matlab provides the function `sprintf`. It takes two inputs: a string mixing plain text and formatting codes, and a vector of numbers to be plugged into the

formatting slots. The return value of `sprintf` is a Matlab string; we use `disp` to display it. The last line in the code block of Idea 1 above illustrates this:

```
disp(sprintf('Max value is %5.3f, found at x=%5.3f',[ymax,xmax]));
```

The percent signs in the string show where numbers are to appear. Each one introduces a string formatting code. Each number in the second input argument—here the vector `[ymax,xmax]`—gets plugged into the format string in order. It's best if the length of the vector matches the number of format codes in the string.

The format code "`%`m.n`f`" reserves space in the output string for a floating-point representation having n digits after the decimal point and occupying a total of m character positions (or more, if it's too big to fit). Other format codes are available. For example `%3d` will give a 3-digit integer; `%9.2e` will give scientific notation with 2 decimal digits. Say `doc sprintf` to see some details.

A related Matlab command is `fprintf`, which combines the operations of converting numbers to strings and writing them to a file. If you omit the optional argument specifying which file to write into, Matlab will write to the screen instead. The only extra thing to know is that when you use `fprintf` instead of `disp` to write on the screen, you lose the end-of-line character supplied automatically by `disp`. Instead, you have to include it in the format string you give to `fprintf`, using the character code `\n`. A reasonable replacement for the line above is this:

```
fprintf('Max value is %5.3f, found at x=%5.3f\n',[ymax,xmax]);
```