Computer Lab 6 — Optimization by Hill-Climbing — Prelab

(c) Philip D Loewen, UBC Vancouver, February 2012

Summary. We explore two ways to approximate the maximum value of a given function. First is exhaustive search, which is crude and slow, but easy to implement. 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:

```
Xnodes = a : 0.01 : b;          % Many x-values, separated by 0.01
Yvals  = f(Xnodes);             % Calculate many y-values (f is "vectorized")
[ymax,k] = max(Yvals);          % Built-in function max works on vectors.  See Idea 2.4.
xmax   = Xnodes(k);             % "k" selects the location of the largest entry in Yvals
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 an inline definition for the function above:

```
f = @(x,y) (2*y - y.^2) .* sin(x);
```
(2)

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. Matlab's `meshgrid` command builds a rectangular grid of evaluation points from given nodes on the coordinate axes. This should be familiar by now:

```
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
```

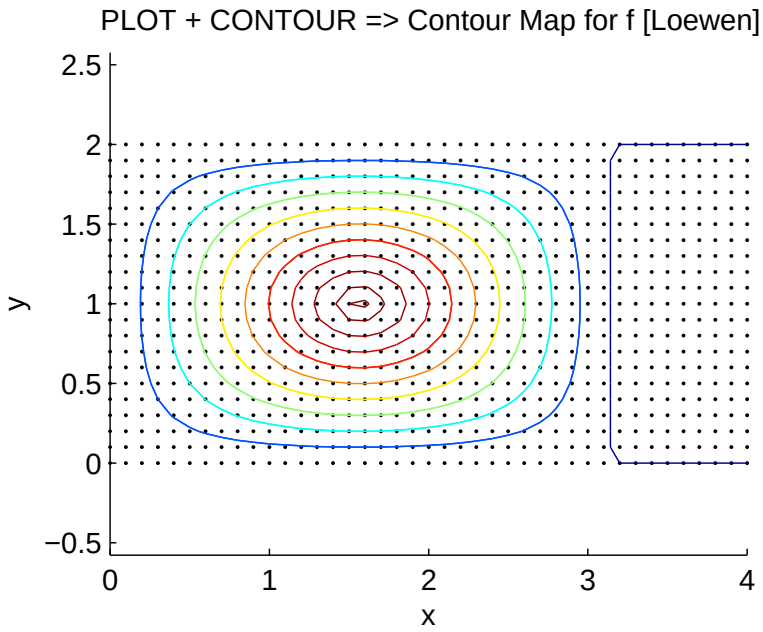

Fig. 1: Contour Map for f

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);
```

See Figure 1. 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 is a good reason for this: close to any critical point, the function's quadratic approximation is very accurate, and a nondegenerate quadratic function has contours that are either ellipses or hyperbolas.)

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);
```

See Figure 2. 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.

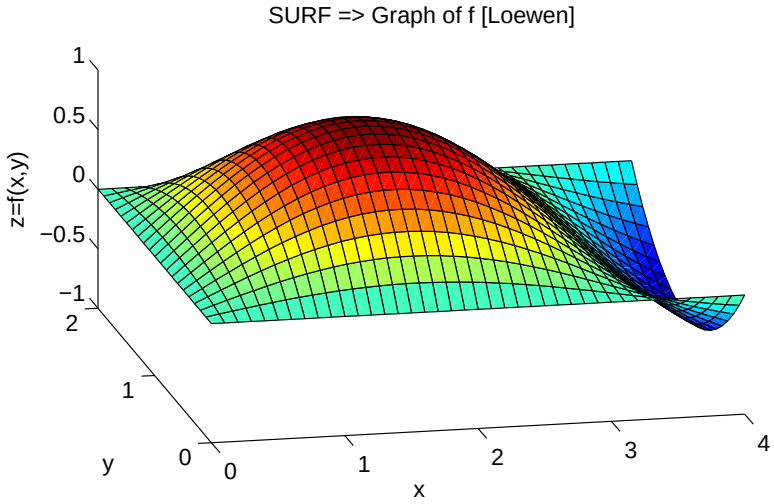

Fig. 2: The surface $z = f(x, y)$

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: *Numerical Derivatives*

3.1. First Derivatives for 1-variable functions. For a smooth function $\phi = \phi(t)$ and point of interest t_0, the centred difference formula says

$$\phi'(t_0) = \frac{\phi(t_0 + h) - \phi(t_0 - h)}{2h} + O(h^2) \qquad (h \to 0^+).$$

Choosing a specific small $h > 0$ and ignoring the error term above provides a usable approximation to $\phi'(t_0)$. 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 $|h/t_0|$ to be around 10^{-6}. Since our inputs are around 1, we can choose $h = 10^{-6}$.

3.2. Matlab Implementation. One way to translate Idea 3.1 into Matlab is to build a function function[1] named `D1` that takes ϕ and t as inputs and returns an approximation for $\phi'(t)$. To pass a function like ϕ as a parameter, we use the function handle, as discussed in Computer Prelab 5, Idea 7. When `D1` is working properly, we want the command `D1(@sin,t)` to return an answer very close to `cos(t)`. Here's an inline definition that achieves all this:

```
h  = 1.E-6;
D1 = @(phi,t) (phi(t+h)-phi(t-h))/(2*h);
```

To approximately differentiate $\phi(t) = t^2 \sin(3t)$ without building a separate `m`-file to hold its definition, we would say

```
phi  = @(t)  t.^2 .* sin(3*t);    % This makes phi a "function handle"
T    = linspace(0,1,13);          % Consider a row-vector of t-values
dphi = D1(phi,T);                 % Here is a row-vector of phi'(t)-values!
```

The top line makes `phi` a *function handle*, not a *function*: this explains why the bottom line can refer directly to the variable `phi` instead of requiring the input `@phi`. Notice the vectorized operators `.^` and `.*` in the definition of `phi`. Our approximate differentiator `D1` operates correctly with vector or matrix inputs, provided the input functions do too.

3.3. Partial Derivatives and the Gradient (Two Independent Variables). Now suppose $f = f(x, y)$ is given, together with a point of interest (x_0, y_0). Both $f_x(x_0, y_0)$ and $f_y(x_0, y_0)$ are just derivatives of one-variable functions, so by analogy with the approximation above we have the second-order approximations

$$f_x(x_0, y_0) \approx \frac{f(x_0 + h, y_0) - f(x_0 - h, y_0)}{2h},$$
$$f_y(x_0, y_0) \approx \frac{f(x_0, y_0 + h) - f(x_0, y_0 - h)}{2h}.$$

[1] A "function function" is a function for which one of the inputs is itself another Matlab function. The most familiar example is `ode45`.

The gradient of f at (x_0, y_0) is the two-element vector

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

3.4. Matlab Implementation. To extend the one-variable setup in Idea 3.3 to $f = f(x, y)$, we could build functions named `D2x` and `D2y` that return one partial derivative each. To process a 3-variable function $G = G(x, y, z)$, we could build functions `D3x`, `D3y`, and `D3z`: now you can see how the number n in `Dnx` encodes the number of input variables expected by the function whose x-partial derivative we are to calculate. Turning these brief hints into code will be one of your tasks in the lab.

IDEA 4: *Gradient Geometry (Two Independent Variables)*

Let's continue with $f = f(x, y)$. At each point (x_0, y_0), the vector $\nabla f(x_0, y_0)$ is perpendicular to the level curve of f that contains (x_0, y_0). To illustrate this with a picture like Figure 3 below, we 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 drawn with its tail at mesh point $(X(i, j), Y(i, j))$ will be $\langle U(i, j), V(i, j) \rangle$. To set up for the `quiver` command we must calculate appropriate matrices U and V. A nice way to do this would be to have a vectorized Matlab counterpart for each function f_x and f_y, so we could say something like

$$U = f_x(X, Y); \quad V = f_y(X, Y);$$

to get the partial derivatives at each node with just two powerful commands. Idea 3.4 above will help with this.

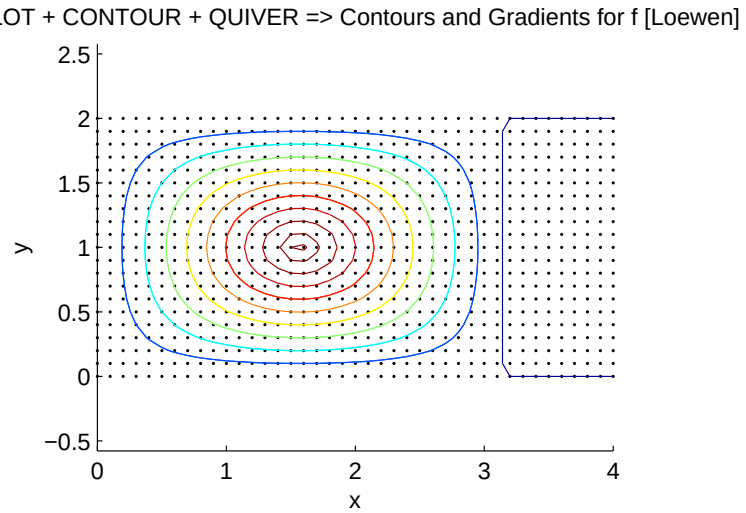

Fig. 3: Contours and Gradient Arrows for f

In this context the plotting command `axis equal` is particularly valuable: stretching one axis more than the other distorts the angles in the picture, which seems to spoil the orthogonality property mentioned above.

Here is a fine point about using `quiver`: it automatically scales the arrows you specify to make the picture look good according to some criteria set by The Mathworks. Multiplying `U` and `V` by a stretch-factor before saying `quiver(X,Y,U,V)` therefore achieves nothing: the

autoscaling obliterates your stretch-factor and gives you the original plot again. If you want to apply a stretch-factor `s`, you must supply it to `quiver` separately, as in

```
quiver(X,Y,U,V,s);
```

Say `help quiver` for more information.

IDEA 5: *Hill-Climbing*

Inspiration. 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

$$\left\langle \frac{dx}{dt}, \frac{dy}{dt} \right\rangle = \nabla f(x(t), y(t)) = \left\langle f_x(x(t), y(t)), f_y(x(t), y(t)) \right\rangle. \tag{3}$$

Any point where f has a local maximum will satisfy $\nabla f = \mathbf{0}$. So this is a point where the trajectories of equation (3) 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.

Matlab Setup. Matlab's `ode45` command deals nicely with systems like (3). Its model equation, shown here using the traditional position-vector notation $\mathbf{r} = (x, y)$, is

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

Interpretation: Some vector-valued function of time and location $\mathbf{V} = \mathbf{V}(t, \mathbf{r})$ is given, and we seek the trajectory $\mathbf{r} = \mathbf{r}(t)$ for which the true velocity $\mathbf{v}(t) = \dot{\mathbf{r}}(t)$ agrees at every instant with the specification expressed by $\mathbf{V}(t, \mathbf{r}(t))$. Matlab's approximate solution process is triggered by commands like these:

```
interval = [0,20];          % A large final time, say 20 -- experiment with this!
startpt  = [0;0];           % Launch hill-climber from (x,y)=(0,0)
options = odeset('RelTol',1e-6,'AbsTol',[1e-6 1e-6]);
[T,XYtraj] = ode45(V,interval,startpt,options);
```
(4)

The tricky part is producing the input argument `V` for `ode45`. This should be a function handle, and the function it identifies needs to deal with location information packed into a single column vector. Let's call this variable `R`, so that `R(1)` holds x and `R(2)` holds y. A suitable definition for `V` is outlined below, in a mixture of mathematical and computer-friendly notation:

```
V = @(t,R) [ f_x, f_y ];
```
(5)

(The scalar input `t` for this function is ignored on the right side, but it must be present to match the structural expectations of `ode45`.) It will be up to you to replace the entries f_x and f_y above with Matlab expressions that return suitable values. Your functions `D2x` and `D2y` will be useful here.

The values `T` and `XYtraj` returned by `ode45` describe a parametric curve in the (x, y)-plane. The parameter 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);
```

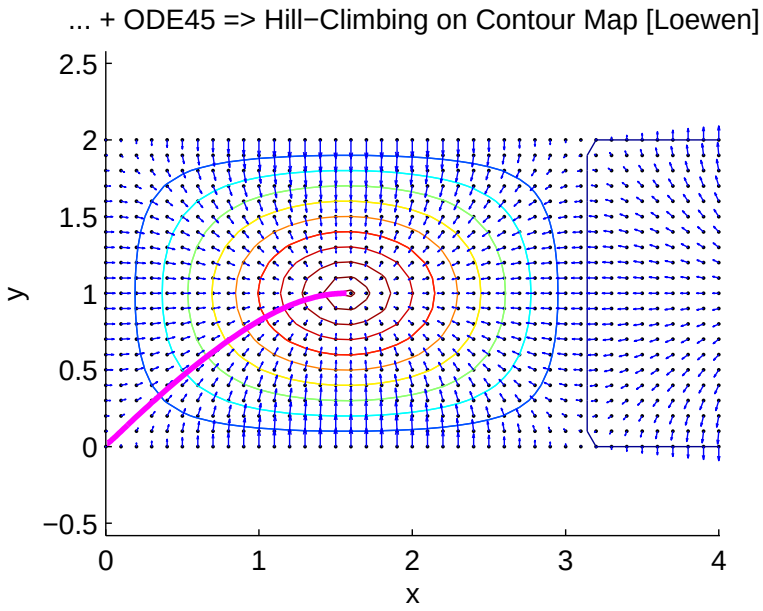

Fig. 4: An uphill trajectory on the contour map

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

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

See Figure 4.

Limitations. This approach provides a nice illustration of the gradient and what it means, and it works well on small problems of the sort one meets in school. However, it is not a competitive professional method for general optimization. Some of the reasons for this will help you avoid problems in the lab:

1. If the time duration specified for `ode45` is too short for the trajectory to climb all the way to a critical point, or if the tolerances `ode45` is given are too loose to enforce high accuracy, the method might stop before reaching a true maximizer. The interval and tolerance specifications in line (4) above are supposed to address this, but you may need to stretch the interval or shrink the tolerances for best results.
2. This method completely ignores any preassigned set S in which we required to search for a maximizer (compare line (1) of Idea 2). If f has only a *local* maximum in S and larger values elsewhere, nothing prevents our hill-climbing point from leaving S and taking 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. Be ready to press Control-C a lot if the trajectory-finder gets lost on a wild-goose chase and becomes unresponsive.
3. Much more efficient algorithms are available. We're looking for just one point, after all, so most of the effort invested in calculating details of a trajectory to get there is actually wasted. A highly efficient alternative called Newton's Method uses repeated quadratic approximations to the objective function f to jump from point to point, getting closer to the maximizer each time. Mech 2 students know all they need to implement this, too, but not today.

Reporting Results. The vectors `T`, `Xtraj`, and `Ytraj` above 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)`. It's nice to be sure that the gradient is small at the stopping point, so we should include it in our reporting scheme, like this:

```
N     = length(T);
xopt  = XYtraj(N,1);
yopt  = XYtraj(N,2);
fopt  = f(xopt,yopt);      % Assuming you named your function "f"
gopt  = V(0,[xopt;yopt]);  % Cheap way to get grad f(xopt,yopt)
disp(sprintf(['Max value is %5.3f, found at (x,y)=(%5.3f,%5.3f); ',...
              'gradient norm is %9.2e.'],[fopt,xopt,yopt,norm(gopt)]));
```
(6)

See Idea 6 below for some more details on the `sprintf` command.

More plots are always welcome. Figure 4 above shows 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 (3): 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 Fig. 4 now appears in 3D: see Figure 5.

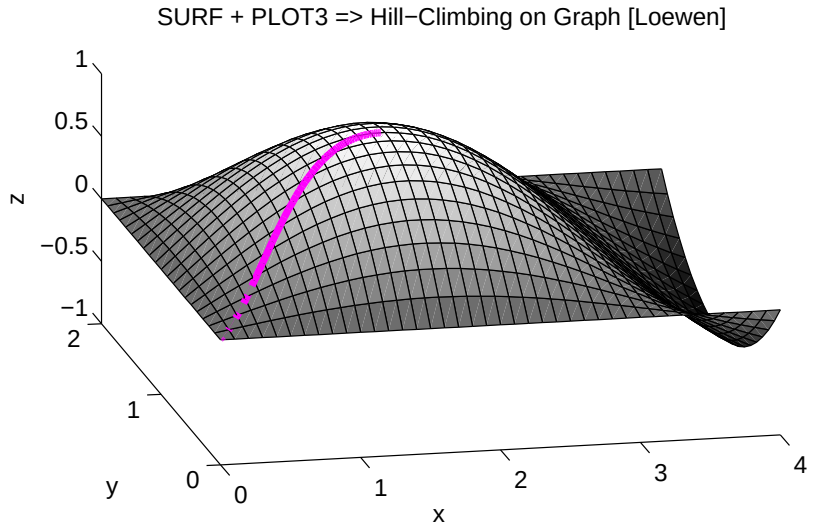

Fig. 5: An uphill trajectory on the graph surface

IDEA 6: *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]);
```

PRE-LAB ASSIGNMENT:

Read both the Prelab and Lab Activities.

PL1: Carry out an efficiency analysis on the exhaustive-search method in Lab Activity 2. Using the set S and function f for your assigned lab day, find (and justify) approximate values for ...

(i) the number of times the computer must evaluate f in parts (d) and (e) of Activity 2,

(ii) the number of function evaluations that would be needed to get a componentwise absolute error not larger than 1×10^{-4},

(iii) the number of bytes of storage required for each of matrices `X`, `Y`, and `Z` in scenario (ii) above, assuming each matrix entry uses 8 bytes; and

(iv) the number of function evaluations and bytes of storage that would be required by the obvious generalization of this scheme to maximize a function $g(w, x, y, z)$ with four input variables, each chosen from the interval $[0, 1]$, if we insist on a componentwise absolute error of at most 1×10^{-4}.

PL2: Recall Idea 3 above. The following inline definition for a central-difference approximation will give the wrong answer almost every time:

```
D1 = @(phi,t) (phi(t+h)-phi(t-h))/2*h;
```

Explain what's wrong with this definition, and change exactly one character to repair it.

PL3: Write out, in Matlab notation, inline definitions for the function-functions `D2x`, `D2y`, `D3x`, `D3y`, and `D3z` described in Idea 3.4.

PL4: Calculate by hand the exact gradient of the function $f(x, y) = (2y - y^2)\sin(x)$ used to produce many of the illustrations above.