10.34: Numerical Methods Applied to Chemical Engineering
Prof. K. Beers
Solutions to Problem Set 6: Numerical Optimization

Mark Styczynski, Ben Wang

1.(3 points) ***5.A.2*** *Compute the point* $\underline{x} \in \Re^2$ *that minimizes the cost function*

$$F(\underline{x}) = \underline{g} \cdot \underline{x} + \frac{1}{2}\underline{x} \cdot H\underline{x} \qquad \underline{g} = \begin{bmatrix} -2 \\ 1 \end{bmatrix} \qquad H = \begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix}$$

This is just a simple problem learning how to use fminunc(). All we need to do is code this function up and call fminunc with a somewhat appropriate initial guess… though your guess is as good as mine. The result of the code included at the end of this problem is:

```
x =

    1.0000
   -1.0000


F =

   -1.5000
```

Now, compute the constrained minimum subject to $x_1^2 + x_2^2 = 1$

This is the same problem, but you need to use fmincon(), write a nonlinear constraint function with one equality constraint, and provide the appropriate series of arguments to fmincon. The result of the code included at the end of this problem is:

```
x =

    0.7420
   -0.6704


F =

   -1.3766
```

As we'd expect, the constrained minimum is greater than our (putative) global minimum.

Then, compute the constrained minimum along the unit circle with the additional requirements that both x_1 *and* x_2 *be nonnegative.*

OK, same cost function, same nonlinear constraint, but also a bound limiting the solution to the first quadrant of two-dimensional space. All we need to do is add in the appropriate bounds; note that you can add in lower bounds with no upper bounds, and Matlab knows exactly what that means. The result of the code included at the end of this problem is:

```
x =

   1.0000
        0


F =

  -0.5000
```

Again, as we'd expect, this further constrained answer is even greater than either of the previous two.

Here's the relevant code:

```
function marksty_P5A2()

clear all; close all;

%PDL> Set an initial guess
x0 = [0; 1];

%PDL> Find unconstrained minimum...
[x, F] = fminunc(@unconFunc,x0)

%PDL> Find first constrained minimum subject to nonlinear constraint.
[x, F] = fmincon(@unconFunc,x0,[],[],[],[],[],[],@nonLinConFunc)

%PDL> Find final constrained minimum with additional bound constraints.
[x, F] = fmincon(@unconFunc,x0,[],[],[],[],[0;0],[],@nonLinConFunc)

end

function F = unconFunc(x)
% This is the basic function we are either finding the unconstrained
%  minimum of or will optimize subject to constraints.
g = [-2;1];
H = [3 1; 1 2];
F = dot(g,x) + dot(.5*x,H*x);
end

function [C, Ceq] = nonLinConFunc(x)
% This is the nonlinear constraint function reflecting equation
%  5.175... there and no nonlinear inequalities, though there is
%  one nonlinear equality.
```

```
% The format it solves for is C <= 0, Ceq = 0.
C = 0;
Ceq = x(1)^2 + x(2)^2 - 1;
end
```

Grading: We're mostly just looking for right answers here. If you got the right answers, you got credit, and each step was worth one point. Otherwise, partial credit was given for each part attempted.

2. (3 points) ***5.B.1*** *We wish to use the enzyme whose kinetics, described by (5.51), were studied earlier in this chapter, in an immobilized-enzyme packed bed reactor. Neglecting any internal mass transfer resistance (we assume the enzyme is immobilized in very small pellets), we compute the outlet substrate concentration by solving the ODE-IVP*

$$\frac{dc_s}{dW} = -\frac{1}{\alpha_c v}\left[\frac{V_m c_s}{K_m + c_s + K_{si}^{-1} c_s^2}\right] \qquad c_s\left(W = 0\right) = c_{s0}$$

C_s is the substrate concentration in M, and is constrained to lie in $[10^{-4},2]$. W is the mass of enzyme in the reactor in mg, and we integrate (5.176) to the total mass W_R = 1 g. v is the volumetric flow rate through the reactor in L/min. $\alpha_c = 10^6$ µmol/mol is a conversion factor, and the kinetic constants are V_m = 200 µmol/min/mg$_E$, K_m = 0.201 M, K_{si} = 0.5616 M. Plot the inlet substrate concentration c_{s0} that maximizes the outlet molar flow rate of product, as a function of v.

So, there are two approaches here… you can integrate the equation analytically and use that result, or you can just use the equation with Matlab's ODE integrator. Since the latter is less prone to typos and math errors, we'll stick with that one. So, we assume that we can integrate this equation easily to the final W of 1 g, or 1000 mg in terms of the units of Vm. That means we know what c_s(W = 1000 mg) is, and so we can "maximize the outlet molar flow rate of product". That's really the most important part of this. We don't want to minimize C_s*v, because that could be done by just using the smallest allowable inlet concentration. What we want to do is maximize ($C_{x0} - C_s$)*v, which will give us a representation of molar flow rate of product (no matter what the reaction stoichiometry is). Now, someone pointed out that you should in theory be able to just maximize ($C_{x0} - C_s$) at any given v, because the v won't be changed by the minimization routine. Indeed, this seems like it should be allowable in principle. In fact, though, it causes some sensitivity to initial guesses at higher flowrates. I'll get more to this sensitivity in a second, but the key is that we understand that in our minimization function, we first integrate the ODE and then return the function that is to be minimized with the resulting value of the final C_s.

With that, what we need to do is pretty straightforward: just maximize the molar flow rate of product (as defined above) subject to the constraints. Of course, there was some confusion about what the constraints should be, but we'll accept both possibilities: just constrain C_{s0} to be in the range in the problem statement, or define all C_s (including C_{s0}) to be in that range. The former is straightforward, since C_{s0} is what we are changing to optimize our function. The latter is a little more difficult, but it only entails integrating the same ODE as before again, this time in the nonlinear constraint function. It just

increases the time it takes to execute the problem and slightly changes the range of reliable answers that you can get. This can all be seen in the code included at the end of this problem.

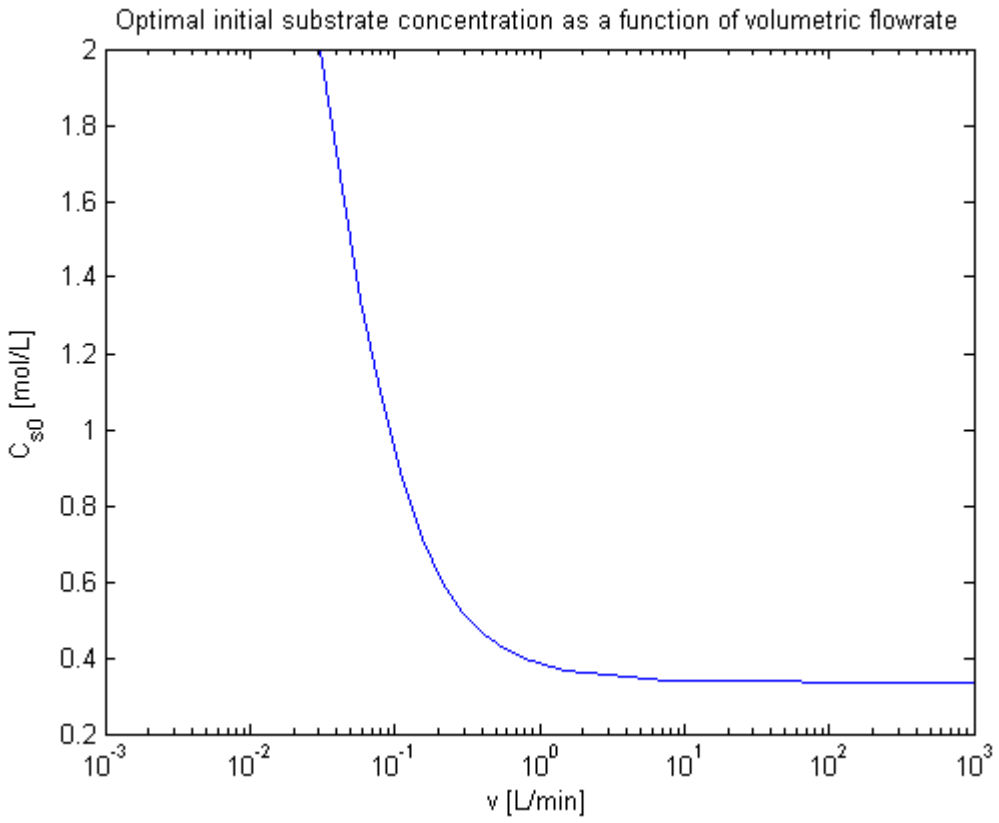
The final thing to consider is sensitivity to initial guesses and other unusual things… now, some of you may not have noticed this if your initial guess was two, but if you used $-(C_{x0} - C_s)*v$ as your cost function and did not constrain C_s, then somewhere around v=0.003, your result is sensitive to your initial guess. This is somewhat unexpected… at low enough v, you expect everything to react, so you'd think that 2 would be the ideal answer. This may be something that can be fixed by providing the Jacobian in your function, but I didn't bother and don't mind if you didn't either.

Next possibility: $-(C_{x0} - C_s)*v$ as your cost function and you constrained C_s. In this case, you will get wild values of C_{s0} below about v = 0.03 due to the fact that you're exceeding maximum function evaluations. Again, this may be something that can be solved with a Jacobian, but it's acceptable to just avoid the problem by looking at an appropriate range of v.

Final possibility we'll look at: $-(C_{x0} - C_s)$ as your cost function. You'll see similar issues on the low scale of v depending on whether or not you constrained C_{s0}, but in addition when you get to v of about 600, you'll see guess dependence… you might think that this is because there's just not that much reaction to occur, but the other cost function converges to the expected result.

So, yeah… those are the ranges of interesting/frustrating results. Jacobians may fix this stuff, but as I pointed out in the email, it's sufficient if you just provide a wide enough range of v to display different behavior but avoid the troublesome regions. If you do include troublesome regions and get troublesome answers, you'll need to have some sort of explanation for why you're getting unusual answers and what exactly is happening.

OK, enough writing… here's a representative plot (in this case, for unconstrained C_s and for the cost function of $-(C_{x0} - C_s)*v$), followed by sample code.




```
function marksty_P5B1()

clear all; close all;

%PDL> Set up range, initial guess.
% I'm using this initial guess just so that it's obvious
%  when I have sensitivity to the initial guess
guess = 1e-4;
% This is the region of consistent behavior for my combination
%  of cost function and constraints
vVec = logspace(-2.5,3,40);
options=optimset('LargeScale','off');
%PDL> Minimize for each v.
for i=1:length(vVec),
    v = vVec(i);
    % Note the commented-out part at the end... I've been
    %  switching back and forth frequently between the two
    %  different options.
    [x, F] = fmincon(@(x)minFunc(x,v),guess,[],[],[],[], ...
        10^(-4),2,[],options);%@(x)nonlinConFunc(x,v),options);
    storage(i) = x;
end
% PDL> Plot results.
% The plot looks much nicer on a semilogx since we're going
%  over such a wide range.
semilogx(vVec,storage)
xlabel('v [L/min]')
```

```
ylabel('C_s_0 [mol/L]')
title('Optimal initial substrate concentration as a function of
volumetric flowrate')
end

%PDL> implement constraints, function to integrate, and
%  cost function.
function F = minFunc(Cs0,v)
% This is the function we need to minimize...
options.disp = 0;
% We find the final Cs by integrating.
[t, y] = ode45(@integFunc,[0 1000],Cs0,options,v);
Cs = y(length(y));
% Then find the product molar flow rate.
F = -(Cs0 - Cs)*v;

end

function f = integFunc(t,Cs,v)
% This is the function we need to integrate, per the
%  problem statement.
W = 1000;
alphaC = 1e6;
Vm = 200;
Km = 0.201;
Ksi = 0.5616;
f = -Vm/(alphaC*v)*(Cs/(Km + Cs + Cs^2/Ksi));

end

function [C, Ceq] = nonlinConFunc(Cs0,v)
% If we were implementing constraints on all Cs, this is
%  the function we'd use.  No equality constraint, and
%  we require that Cs(final) (and thus all Cs, since the
%  function is monotonically decreasing) are above 1e-4.
C = zeros(2,1);
options.disp = 0;
[t, y] = ode45(@integFunc,[0 1000],Cs0,options,v);
Cs = y(length(y));
C(1) = 10^(-4) - Cs;
% We can also (needlessly) reinforce the maximum of 2, though
%  our constraints on Cs0 should take care of this.
C(2) = Cs - 2;
Ceq = 0;

end
```

Grading:
(-1 point): Plot range insufficient… you must show limiting cases at both high and low v, where the low v results should max out at C_{s0}=2 before you reach regions of instability.
(-1 point): Constraints or cost function coded incorrectly
(-1 point): Code doesn't run immediately due to bug
(-0.5 point): Small parts of write-up incomplete or missing

3. (4 points) ***5.B.3*** *We wish to determine the best path for a road connecting two points in hilly terrain. Let* $\underline{r} \in \Re^2$ *be the coordinates of a point in km and let the elevation at that point, also in km, be z(*$\underline{r}$*). We represent the measured ground elevation data as a sum of contributions from individual hills, each hill being represented by a Gaussian function,*

$$z(\underline{r}) = \sum_{k=1}^{N_h} z_{\max}^{[k]} \exp\left\{-\frac{1}{2}\left(\underline{r}-\underline{r}_c^{[k]}\right)\cdot\left(\Sigma^{[k]}\right)^{-1}\left(\underline{r}-\underline{r}_c^{[k]}\right)\right\}$$

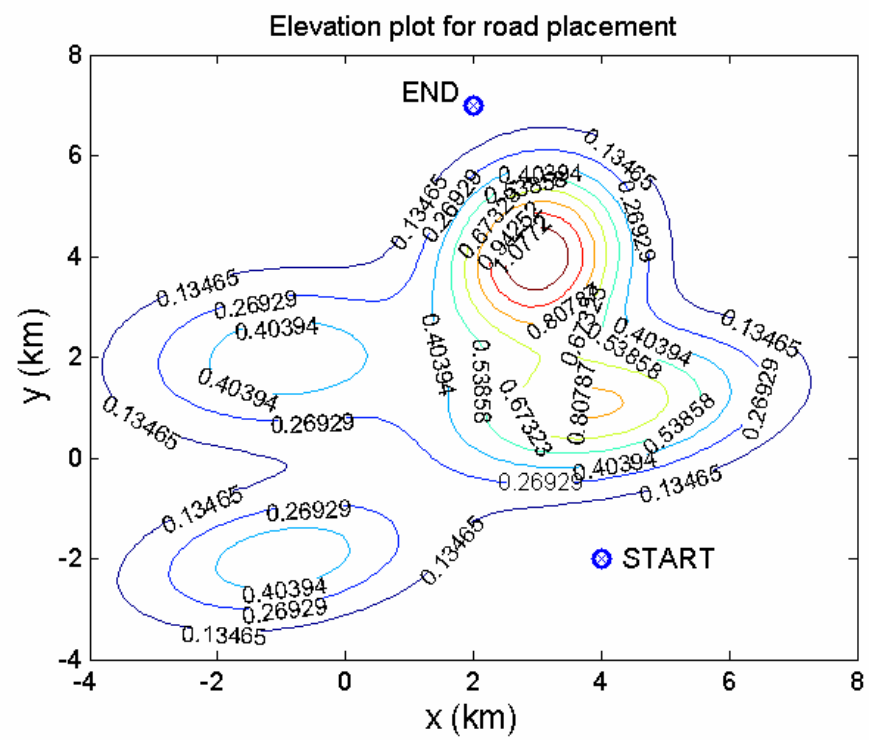
In the region of interest, we use a representation with four hills,

$$z_{\max}^{[1]} = 1.2 \qquad z_{\max}^{[2]} = 0.8 \qquad z_{\max}^{[3]} = 0.5 \qquad z_{\max}^{[4]} = 0.5$$

$$\underline{r}_c^{[1]} = \begin{bmatrix} 3 \\ 4 \end{bmatrix} \qquad \underline{r}_c^{[2]} = \begin{bmatrix} 4 \\ 1 \end{bmatrix} \qquad \underline{r}_c^{[3]} = \begin{bmatrix} -1 \\ -2 \end{bmatrix} \qquad \underline{r}_c^{[4]} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$$

$$\Sigma^{[1]} = \begin{bmatrix} 1.0 & 0.1 \\ 0.1 & 1.5 \end{bmatrix} \qquad \Sigma^{[2]} = \begin{bmatrix} 3.0 & 0.5 \\ 0.5 & 1.0 \end{bmatrix} \qquad \Sigma^{[3]} = \begin{bmatrix} 2.5 & 0.4 \\ 0.4 & 0.8 \end{bmatrix} \qquad \Sigma^{[4]} = \begin{bmatrix} 3.0 & 0.2 \\ 0.2 & 1.2 \end{bmatrix}$$

Figure 5.16 shows the elevation controus along with the start (4, -2) and end (2, 7) positions of the planned road.



All land is available to build upon. Our task is to find the shortest path between the two end points subject to the constraint that the grade cannot be greater than 8%, i.e. the slope cannot be large in magnitude than 0.08.

Let $0 \le s \le 1$ *be a contour variable and* $\underline{r}$*(s) be the path of the road, subject to*

$$\underline{r}(0) = \underline{r}_{start} = \begin{bmatrix} 4 \\ 2 \end{bmatrix} \qquad \textit{and} \qquad \underline{r}(1) = \underline{r}_{end} = \begin{bmatrix} 2 \\ 7 \end{bmatrix}$$

We discretize the path by setting N contour positions $s_k = k/(N+1)$ *and the coordinates* $\underline{r}^{[k]} = \underline{r}(s_k) = \begin{bmatrix} x^{[k]} \\ y^{[k]} \end{bmatrix}$. *We wish to minimize*

$$F_C = \left(\underline{r}^{[1]} - \underline{r}_{start}\right)^2 + \sum_{k=1}^{N-1}\left(\underline{r}^{[k+1]} - \underline{r}^{[k]}\right)^2 + \left(\underline{r}_{end} - \underline{r}^{[N]}\right)^2$$

Subject to the constraints that for each road sement,

$$\frac{\left|z\left(\underline{r}^{[k+1]}\right) - z\left(\underline{r}^{[k]}\right)\right|}{\sqrt{\left(x^{[k+1]} - x^{[k]}\right)^2 + \left(y^{[k+1]} - y^{[k]}\right)^2}} \le \Gamma_{\max} \qquad \Gamma_{\max} = 0.08$$

Using this approach, propose a path for the road to follow.

This is another relatively straightforward problem; the hardest parts here (beyond the usual fixing of assorted typos that keep programs from working) are designing reasonable initial guesses and plotting the hills and path.

It is extremely useful to realize that this program allows for significant functionalization. For instance, you will need to get the height of many different r values at many different times, and you don't necessarily want to confine yourself to a set grid of points, so it is easiest to make a function, say "marksty_getHeight", that will return the height of a given point.

From there, we realize that two additional functions will be useful: a cost function, as defined by equation (5.180) in the book, and a constraint function, as defined by equation (5.181). As was noted in an email to the class, the squaring of the vectors in the cost function is meant to imply dotting those vectors with themselves, giving a scalar output for the cost function (which is good, since we haven't covered multiobjective optimization). We put these in the function marksty_distFun. The constraints are a little bit different… though there is one constraint equation, it references each point on your path, so you really need a constraint for each of the steps you take… this means that you will need N+1 constraint functions. We put these inequality constraints in the function marksty_conFun.

In my code, I supplied the N interior points (excluding endpoints) as the variables to be optimized. This means I need no equality constraints. An equally valid method is to supply the cost function the N+2 total points (including endpoints) and have two equality constraints, one each for the values of the start and end points. Either way, you must be consistent between your cost and constraint functions and must define your cost function appropriately. It is also important to note that *either way* you should only have N+1 inequality constraints. It does not make sense to have N+3 inequality restraints; at best some would be redundant, and at worst some may be wrong.

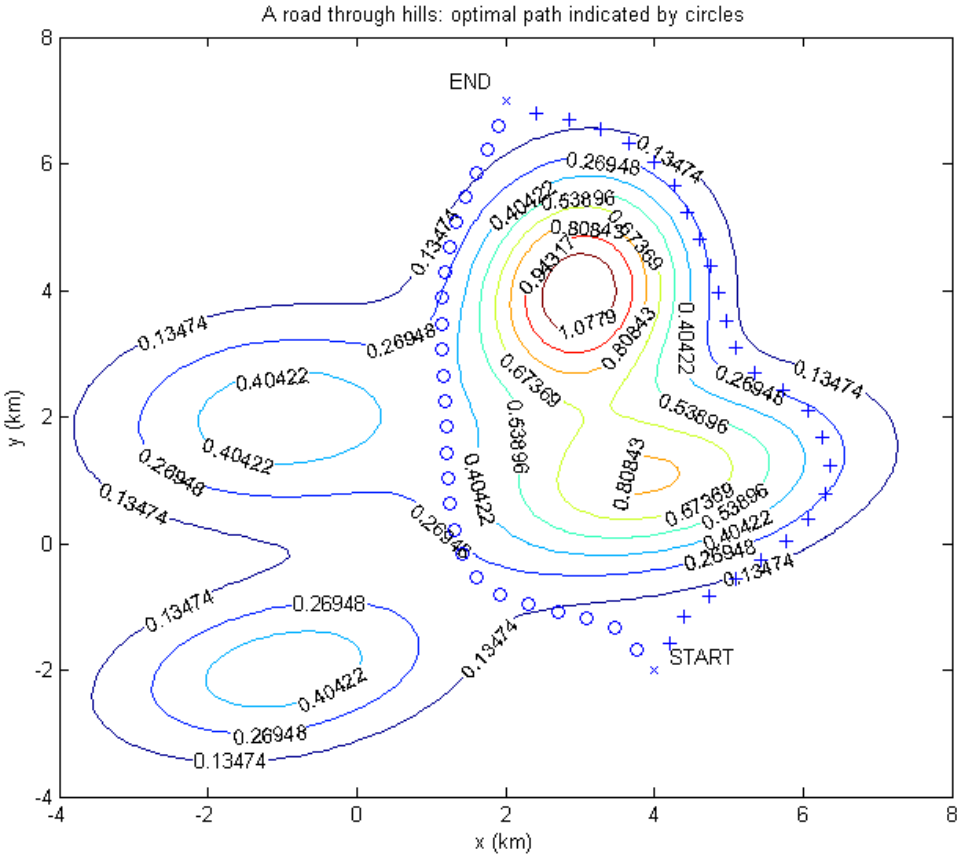
Now what about that contour variable? Is it necessary? Not strictly. It is a useful book-keeping tool, for sure. There were a couple of questions about whether each of the points had to be equally far apart from the others, but this is not intrinsic to the definition of a contour variable as far as I know. So really, you could completely ignore it for the purposes of solving the problem (which is what I did).

To plot the hills, it is probably easiest to make a 2-D grid and find the height for each point in the grid; note that this does not preclude us from getting the height at exact points along our path, it merely gives us a matrix with which we can plot hill contours and see how reasonable our final path is.

The final issue is setting initial guesses. As a practical matter, I found that my code behaved much better if my initial guess was close to satisfying constraints even if it was suboptimal in distance; if I gave it the best distance (a straight line) that obviously violated constraints, its behavior was somewhat unpredictable. Again, this is to be expected for any routine that solves for local extrema. This is why we always do our best to give a good (or at least reasonable) initial guess. Looking at the hills, we see two obvious ways to circumvent the hills that are most likely going to be our winners: either through the valley or completely around the hills on the right side of the figure. We'll set up initial guesses for those and see what Matlab thinks about them.

Another note: since we are using the better-behaved quadratic cost function (rather than the square root of each distance vector dotted with itself), there will be a significant dependence of the final "cost" upon the number of points in the interpolation. For instance, if we were going a distance of 4 units with one intervening point, our minimum cost function would be where the intervening point is the midpoint. This would yield a total cost of 8 ($2^2 + 2^2$). If we put two more points in there, the minimum cost would be 4 ($1^2 + 1^2 + 1^2 + 1^2$), even though we are traveling the same distance.

With all of that being said… we have all of the concepts down, and we can put it all into code. Below you'll find plots of the contours and the two paths I found… the path through the valley turns out to be the better one. For a grid of 25 points, I find the optimum cost to be 4.3482 km^2. The path can be seen on the plot below, delineated by circles going through the valley. For the morbidly curious, the (x, y) values are included after the code.




```
% Mark Styczynski
% 10.34
% HW6, Problem 5.B.3

clear all; close all;

% PDL> Choose number of grid points.
%  After some post hoc experimenting, we find that it is unlikely
%   that increasing the number of grid points will change the
%   character of our solution, so we keep it to N = 25.
N = 25;

% Let's just set the size of this for now.
rGuess = zeros(2,N);

% We can imagine that there may be local optima, so we
%  propose a couple of possible paths... one guaranteed
%  to go through the "valley", another guaranteed to just
%  circumvent all mountains.

%PDL> Set up initial guesses
% We choose the point (1, 6) and note that a line between
%  the beginning and that point will likely not violate
%  constraints and will lead us through the valley.  We then
```

```
%  interpolate to make N total points leading us there.
for k=1:N,
    rGuess(1,k) = 4 + (1 - 4)/(N+1)*k;
    rGuess(2,k) = -2 + (6 - (-2))/(N+1)*k;
end

% PDL> Find constrained optimum.
rVec = fmincon('marksty_distFun',rGuess,[],[],[],[],[],[], ...
    'marksty_conFunc')

optimum1 = marksty_distFun(rVec)

% PDL> Set up initial guesses
%  The point (6,6) will lead us around the hills instead.
%  Same concept as above.
for k=1:N,
    rGuess(1,k) = 4 + (6 - 4)/(N+1)*k;
    rGuess(2,k) = -2 + (6 - (-2))/(N+1)*k;
end
% PDL> Find constrained optimum.
rVec2 = fmincon('marksty_distFun',rGuess,[],[],[],[],[],[], ...
    'marksty_conFunc')

optimum2 = marksty_distFun(rVec2)

% This value is greater than the previous one, so we believe
%  that the route through the valley is the shortest one
%  that obeys the constraints.

% Some code for plotting borrowed from Dr. Beers, 2004.

% PDL> Set up a mesh to plot hill contours.
x = [-4:.1:8];
y = [-4:.1:8];
[XX,YY] = meshgrid(x,y);
ZZ = zeros(size(XX));
for ix=1:length(x)
    for iy=1:length(y)
    r = [XX(iy,ix); YY(iy,ix)];
    ZZ(iy,ix) = marksty_getHeight(r);
    end
end

% PDL> Make contour plot
figure;
[C,H] = contour(XX,YY,ZZ,8);
clabel(C,H);
xlabel('x (km)'); ylabel('y (km)');
zlabel('z (km)');
title('A road through hills: optimal path indicated by circles');
hold on;
% PDL> Plot paths on contour plot.
r_start = [4;-2]; r_end = [2; 7];
plot(r_start(1),r_start(2),'x');
text(r_start(1)+0.2,r_start(2)+0.2,'START');
```

```
plot(r_end(1),r_end(2),'x');
text(r_end(1)-0.75,r_end(2)+0.3,'END');
plot(rVec(1,:),rVec(2,:),'o');
plot(rVec2(1,:),rVec2(2,:),'+');
return;
```

```
function Fc = marksty_distFun(rVec)
% Function to return the cost/distance for a set of points.

% Assume rVec = [r1 r2 r3 ...]
% So its size is 2 x N

rVecSize = size(rVec);
N = rVecSize(2);
rStart = [4; -2];
rEnd = [2; 7];

%PDL> Find first step distance.
Fc = sum((rVec(:,1) - rStart).^2);

%PDL> Add interim step distance.
for k=1:N-1,
    Fc = Fc + sum((rVec(:,k+1) - rVec(:,k)).^2);
end

%PDL> Add final step distance.
Fc = Fc + sum((rEnd - rVec(:,N)).^2);
```

```
function [C, Ceq] = marksty_conFunc(rVec)
% A constraint function to enforce the maximum slope requirement.
% Assume rVec = [r1 r2 r3 ...]
% So its size is 2 x N

rVecSize = size(rVec);
N = rVecSize(2);
rStart = [4; -2];
rEnd = [2; 7];
maxSlope = 0.08;

% PDL> Constraint for the first step
C(1) = abs(marksty_getHeight(rVec(:,1))- marksty_getHeight(rStart))/...
    norm(rVec(:,1) - rStart) - maxSlope;

% PDL> Constraint for intermediate steps.
for k=1:N-1,
    C(k+1) = abs(marksty_getHeight(rVec(:,k+1))-
marksty_getHeight(rVec(:,k)))/...
        norm(rVec(:,k+1) - rVec(:,k)) - maxSlope;
end

% PDL> Constraint for final step.
C(N+1) = abs(marksty_getHeight(rEnd)- marksty_getHeight(rVec(:,N)))/...
    norm(rEnd - rVec(:,N)) - maxSlope;
```

```
% PDL> No equality constraints.
Ceq = 0;
```

```
function z = marksty_getHeight(r)

% Function to obtain the height z of a point r in hilly terrain.
% r is a 2-d vector

zMax = [1.2 0.8 0.5 0.5];
rc = [3 4 -1 -1; 4 1 -2 2];
sigmaInv(:,:,1) = inv([1 0.1; 0.1 1.5]);
sigmaInv(:,:,2) = inv([3 0.5; 0.5 1]);
sigmaInv(:,:,3) = inv([2.5 0.4; 0.4 0.8]);
sigmaInv(:,:,4) = inv([3 0.2; 0.2 1.2]);
z = 0;

% PDL> Add the component elevations together.
for k=1:4
    z = z + zMax(k)*exp(dot(-.5*(r - rc(:,k)), ...
        sigmaInv(:,:,k)*(r - rc(:,k))));
end
```

rVec =

3.7561	-1.6696
3.4778	-1.3325
3.0967	-1.1832
2.7041	-1.0683
2.3116	-0.9536
1.9322	-0.8003
1.6183	-0.5321
1.4272	-0.1696
1.3033	0.2199
1.2473	0.6252
1.2311	1.0315
1.2149	1.4382
1.1988	1.8446
1.1825	2.2514
1.1663	2.6579
1.1503	3.0645
1.1342	3.4713
1.1520	3.8776
1.1908	4.2825
1.2523	4.6847
1.3409	5.0817
1.4589	5.4714
1.6030	5.8516
1.7615	6.2255
1.9007	6.6076

Grading:
2 points: Concepts, approach, and understanding of the problem (writeup, etc)
1 point: Trying multiple initial guesses… partial credit was given depending on the amount you addressed global/local optimality
1 point: Getting the correct path and providing either a plot or coordinates for it