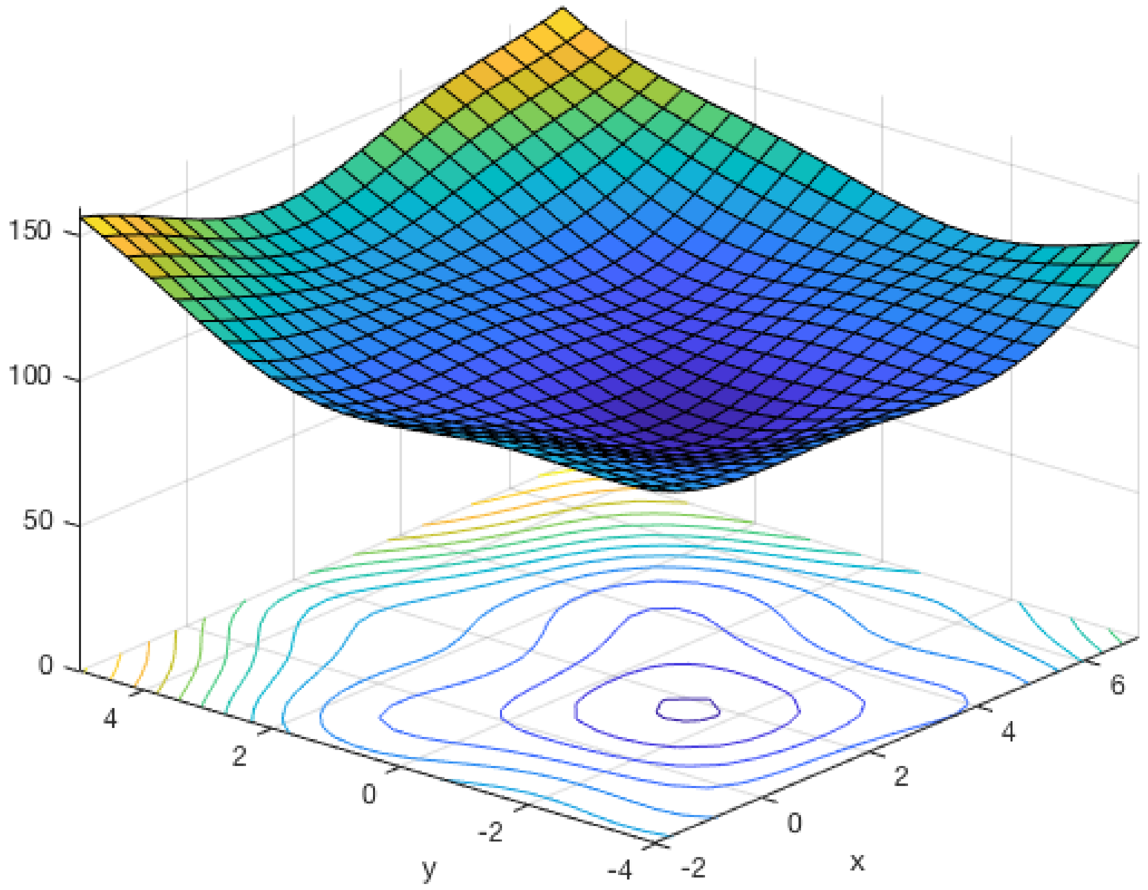
AMATH 301 - Spring 2020

4/29 Lecture: Gradient Descent

In Activity 5, we performed gradient descent on the function

$$f(x, y) = (x-2)^2 + (y+1)^2 + 5\sin(x)\sin(y) + 100$$

Here is the graph of the function.



We saw that you can define this function as an anonymous function with two variables.

```
f = @(x,y) (x-2)^2 + (y+1)^2 + 5*sin(x)*sin(y) + 100;
```

But for gradient descent, it is convenient to define the function as a function of one variable which is a vector with two components.

```
f = @ (p) (p(1)-2).^2 + (p(2)+1).^2 + 5*sin(p(1)).*sin(p(2)) + 100;
```

The vector formula is a little bit harder to type and harder to check for correctness. One trick is to define the function of two variables and then use an adapter function that takes in a vector, splits it apart, and feeds it into the first function.

```
fxy = @ (x,y) (x-2).^2 + (y+1).^2 + 5*sin(x).*sin(y) + 100;
f = @ (p) fxy(p(1),p(2));
```

We also defined a function for the gradient.

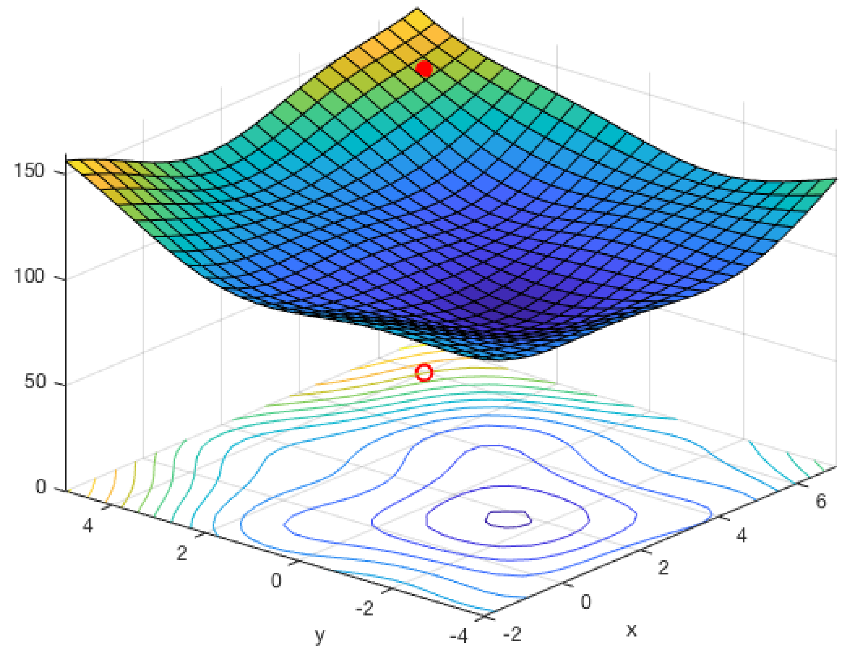
```
fgrad = @ (p) [2*(p(1)-2) + 5*cos(p(1))*sin(p(2))
               2*(p(2)+1) + 5*sin(p(1))*cos(p(2))];
```

You can also simplify the process for defining this function.

```
f_x = @ (x,y) 2*(x-2) + 5*cos(x)*sin(y);
f_y = @ (x,y) 2*(y+1) + 5*sin(x)*cos(y);
fgrad = @ (p) [f_x(p(1),p(2)); f_y(p(1),p(2))];
```

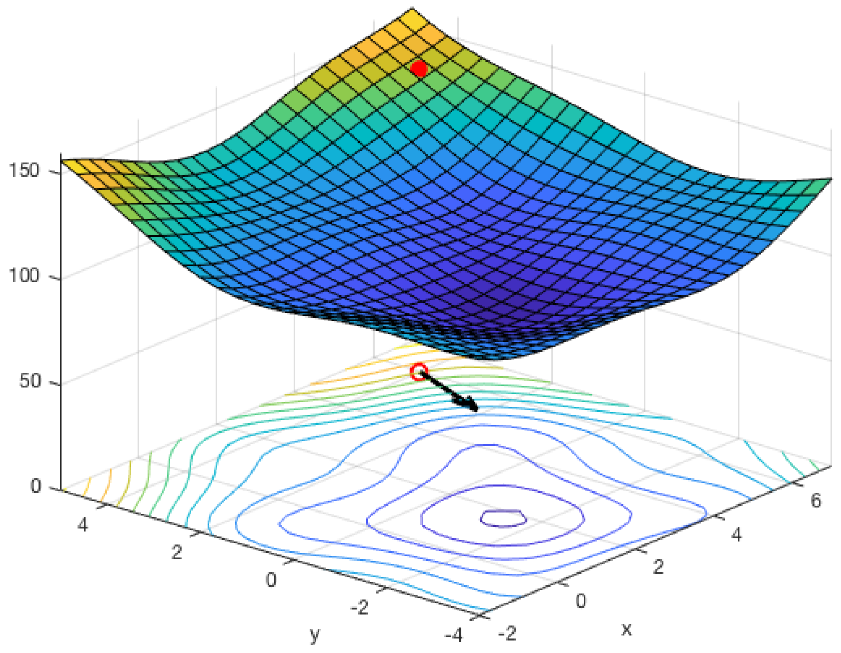
The steps of gradient descent are as follows. First, you choose an initial guess.

```
p0 = [6;4];
```



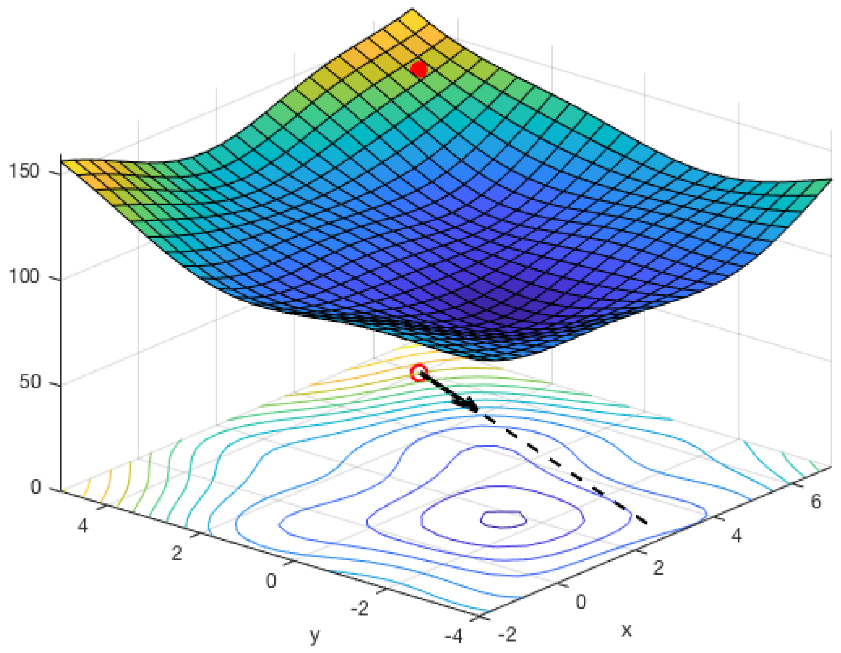
The you calculate the gradient to determine which direction to move for the next guess (you go in the direction of the negative gradient).

```
grad = fgrad(p0);
```



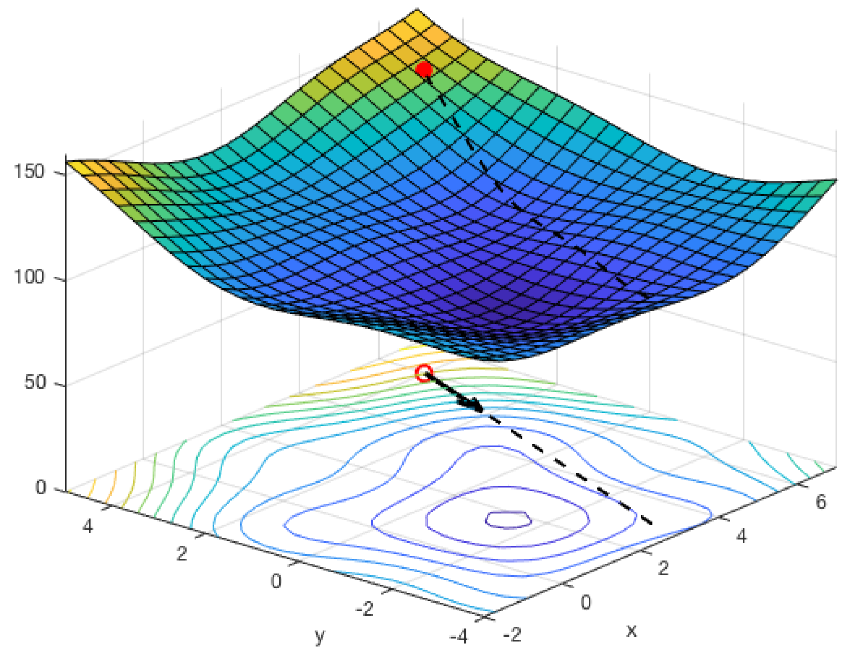
Then we create a function which defines the path that we will travel along to reach the next guess.

```
phi = @ (t) p0 - t*grad;
```



Then we create a function of the heights on the surface of all the points on the path.

```
f_of_phi = @ (t) f(phi(t));
```

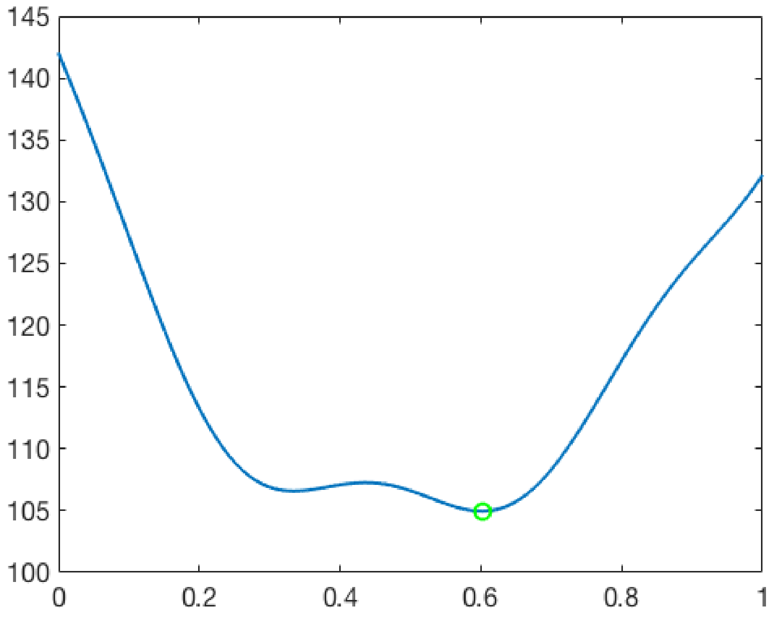


Then we find the value of t which corresponds to the minimum height along the path.

```
tmin = fminbnd(f_of_phi,0,1);
```

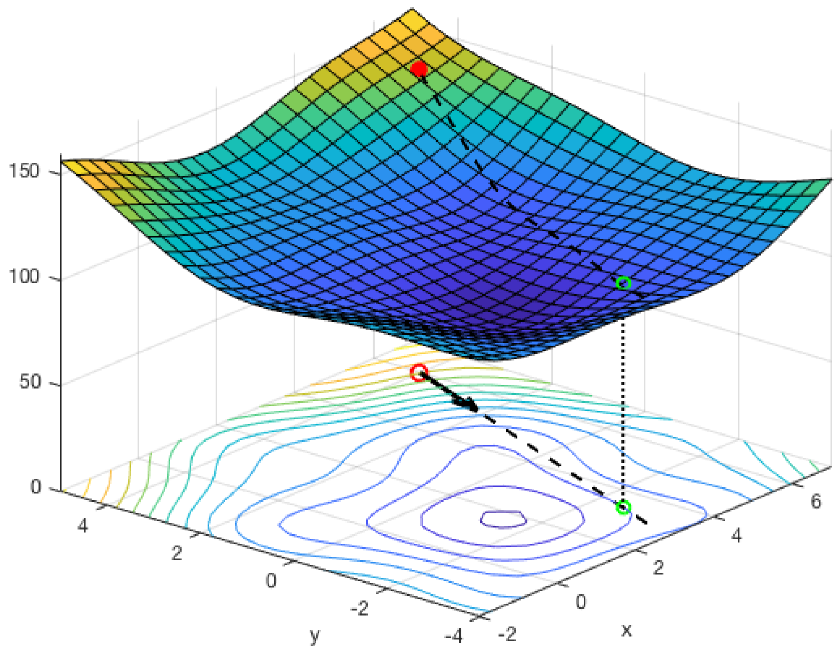
To get a better sense of what the function $f(\phi(t))$ actually is and what `fminbnd` is minimizing, we can plot the function. We will make a vector of t values and plug each one into `f_of_phi`.

```
tplot = linspace(0,1,1000);
for k = 1:length(tplot)
    fplot(k) = f_of_phi(tplot(k));
end
figure(2)
plot(tplot,fplot,'Linewidth',2)
hold on
plot(tmin,f_of_phi(tmin),'og','Markersize',10,'Linewidth',2)
set(gca,'Fontsize',16)
```



To find the next guess (i.e. the point along the path), we plug that value of t back into the function ϕ.

```
p1 = phi(tmin);
```



If you collect the lines of code that we used so far, we have

```
fxy = @ (x,y) (x-2).^2 + (y+1).^2 + 5*sin(x).*sin(y) + 100;
f = @ (p) fxy(p(1),p(2));
f_x = @ (x,y) 2*(x-2) + 5*cos(x)*sin(y);
f_y = @ (x,y) 2*(y+1) + 5*sin(x)*cos(y);
fgrad = @ (p) [f_x(p(1),p(2)); f_y(p(1),p(2))];

p0 = [6; 4]; % Choose an initial guess
grad = fgrad(p0); % Find which direction to go
phi = @ (t) p0 - t*grad; % Define the "path"
f_of_phi = @ (t) f(phi(t)); % Create a function of "heights along path"
tmin = fminbnd(f_of_phi,0,1); % Find time it takes to reach min height
p1 = phi(tmin); % Find the point on the path and update your guess
```

Notice that the function definitions at the top do not change for each iteration of gradient descent, but the lines below all do. If we want them in a form that is more ready to put into a loop, we can just use a variable p to keep track of the "current guess".

```
p = [6; 4]; % The current guess
grad = fgrad(p); % Find which direction to go
phi = @ (t) p - t*grad; % Define the "path"
f_of_phi = @ (t) f(phi(t)); % Create a function of "heights along path"
tmin = fminbnd(f_of_phi,0,1); % Find time it takes to reach min height
p = phi(tmin); % Find the point on the path and update your guess
```

With this example code, you are well-prepared to put it into a loop for your homework. The only other issue to figure out is the stopping criteria - when should you stop the loop? One possibility is to do something like Jacobi and Gauss-Seidel: stop when the current guess and the next guess are close together.

```
norm(pnew-p) < tol % Using the 2-norm
norm(pnew-p,Inf) < tol % Using the infinity norm
```

Another possibility is to check when the gradient is close to the zero vector because at the minimum, the gradient is exactly the zero vector. We can check if the gradient is close to the zero vector by seeing if its norm is close to zero.

```
norm(grad) < tol % Using the 2-norm
norm(grad,Inf) < tol % Using the infinity norm
```

Any of these choices are reasonable options.

A built-in MATLAB function which finds the minimum of a function of multiple variables is `fminsearch`. The function needs to be a function with a single input (i.e. `f = @ (p)` and **not** `f = @ (x,y)`). It outputs the point that minimizes the function.

```
pmin = fminsearch(f,[6;4])
```

```
pmin = 2x1
    1.6949
   -1.4063
```

You can also have `fminsearch` tell you the function value by asking for two outputs.

```
[pmin, fval] = fminsearch(f,[6;4])
```

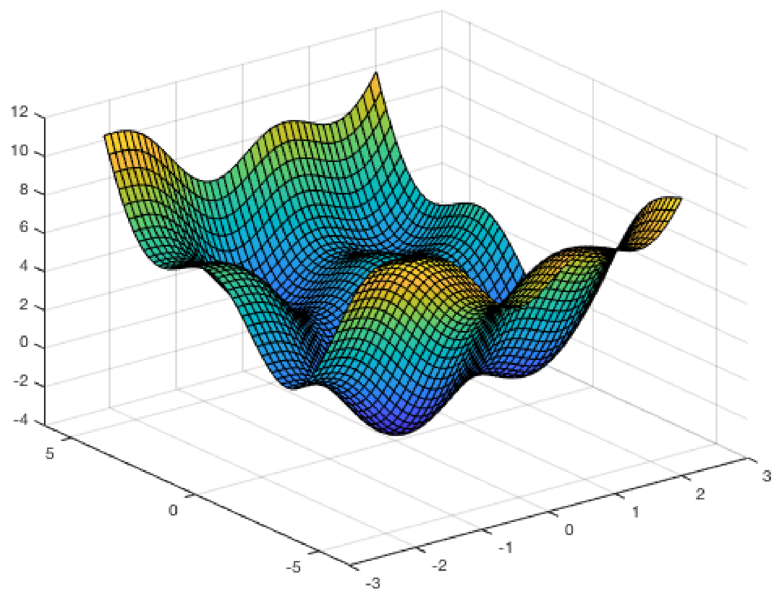
```
pmin = 2x1
     1.6949
    -1.4063

fval = 95.3636
```

Recall that for golden section search, we needed a unimodal function - meaning it only has one local minimum. If a function has more than one local minimum then all of the optimization methods that we learned might only find a local minimum instead of the global minimum (or true minimum). In general, the types of functions for which most optimization procedures work are **convex functions**. For functions of one variable, convex just means "concave up" (i.e. the second derivative is positive). Another way to define convex is that if you connect two points on the curve with a straight line, the line will be above the curve. The same idea holds for functions of multiple variables. Here is a function of two variables which is not convex. Notice that is has several local minima.

```
f_plot = @ (x,y) sin(3*x) + 2*sin(y) + .7*x.^2 + .2*y.^2;

x = -2*pi/3:.1:2*pi/3;
y = -2*pi:.2:2*pi;
[X,Y] = meshgrid(x,y);
surf(X,Y,f_plot(X,Y))
```



If we do gradient descent on this function, the intial guess will determine whether we end up at a local minimum or the true minimum.

Another strategy for gradient descent is to not find the value `tmin` by using `fminbnd`. Instead, at each step of the algorithm we can choose some predetermined constant `tstep` that determines how far we move in the direction of the negative of the gradient. To implement this, you can replace the code

```
phi = @ (t) p - t*grad; % Define the "path"
f_of_phi = @ (t) f(phi(t)); % Create a function of "heights along path"
tmin = fminbnd(f_of_phi,0,1); % Find time it takes to reach min height
p = phi(tmin); % Find the point on the path and update your guess
```

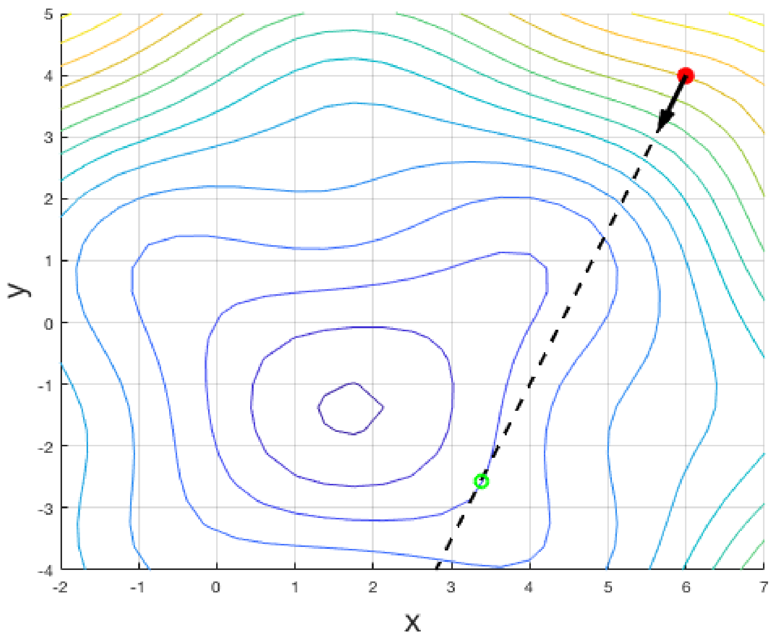
with the code

```
p =  p - tstep*grad;
```

This is obviously much simpler to program. Let's see some of the practical differences.

First, let explore what happens when we do several steps of gradient descent using `fminbnd` at every step. (The code is hidden so you can't use it on your homework). Let's start by just looking at the first step.

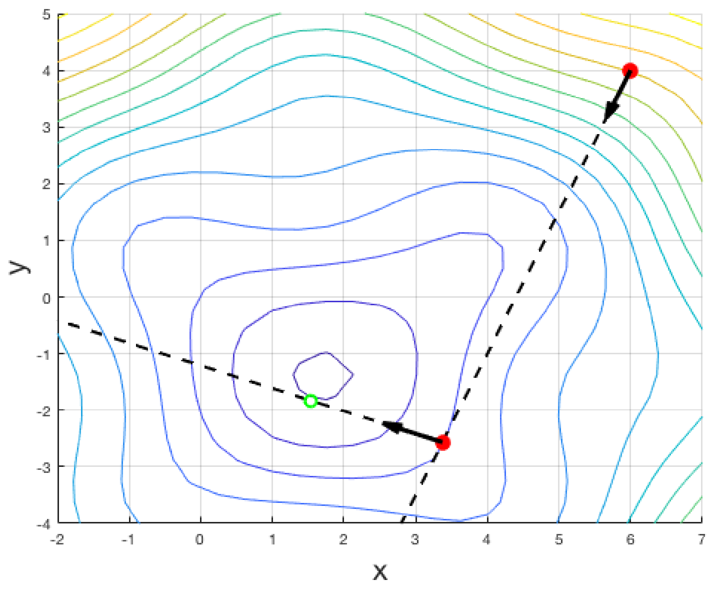
```
n_iter = 1; % Number of iterations
```



Notice that the direction that we travel is perpendicular to the contour line at the initial point. This will always be the case at each step of gradient descent - we move perpendicular to contour lines. Note also that you can tell from the contour lines when you should stop. Each "ring" as you move out from the minimum is a higher elevation. From the red dot to the green dot, you are traveling toward rings that are closer to the minimum. But after the green dot you would be traveling toward more outer rings. That is graphically how you can tell where the next point will be.

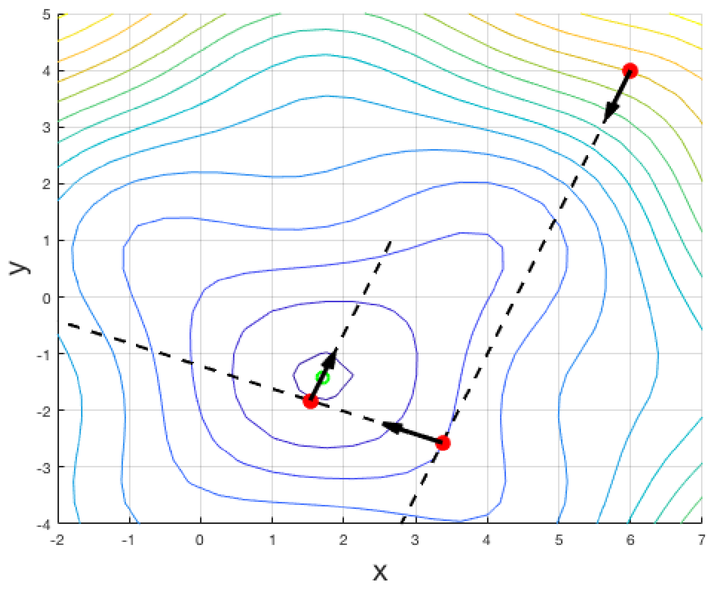
The next step of gradient descent will start from the green dot. The path should be perpendicular to the contour line on which the green dot is located.

```
n_iter = 2; % Number of iterations
```



Now we can do a third iteration.

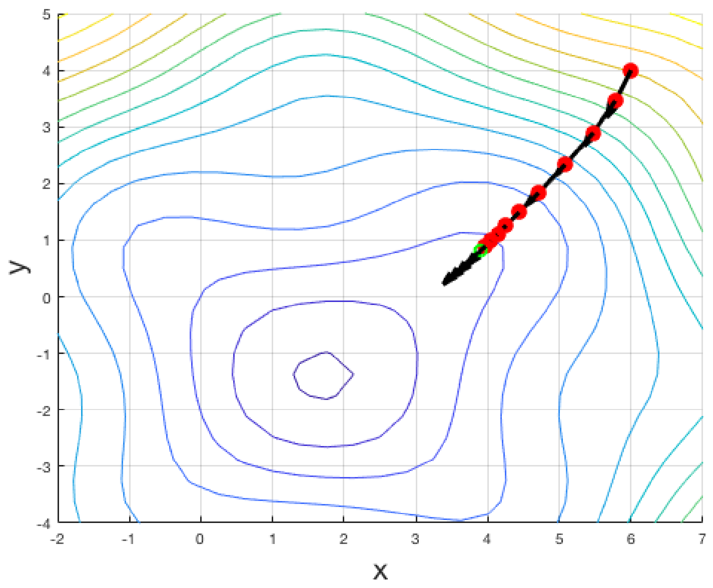
```
n_iter = 3; % Number of iterations
```



We essentially reach the minimum in 3 iterations.

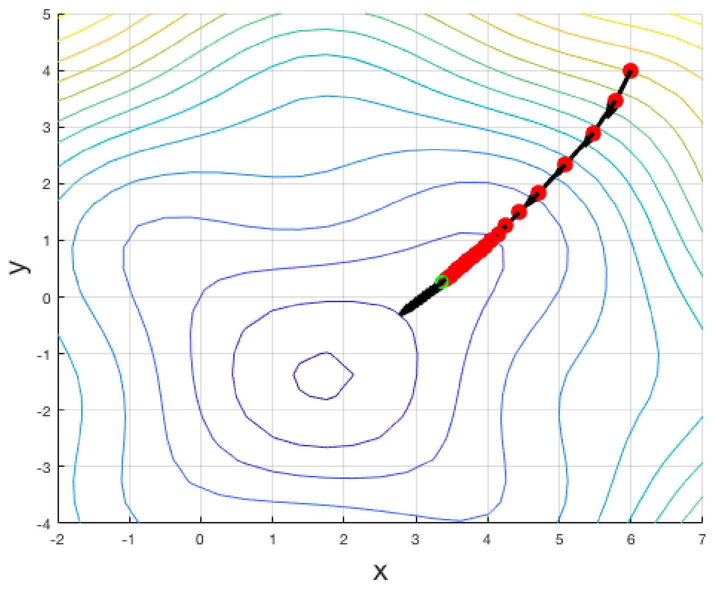
Now let's try using a fixed step size `tstep`. We will try 10 iterations with `tstep = 0.05`.

```
n_iter = 10; % Number of iterations
tstep = .05;
```



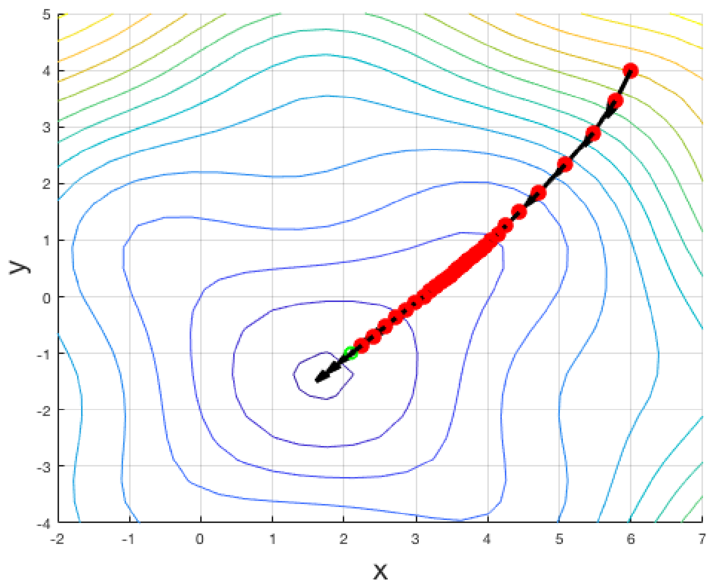
Notice that it is going to take a lot more iterations to reach the minimum than the `fminbnd` method.
However, the path looks to be a much smoother path toward the minimum.
Let's use more steps.

```
n_iter = 20; % Number of iterations
tstep = .05;
```



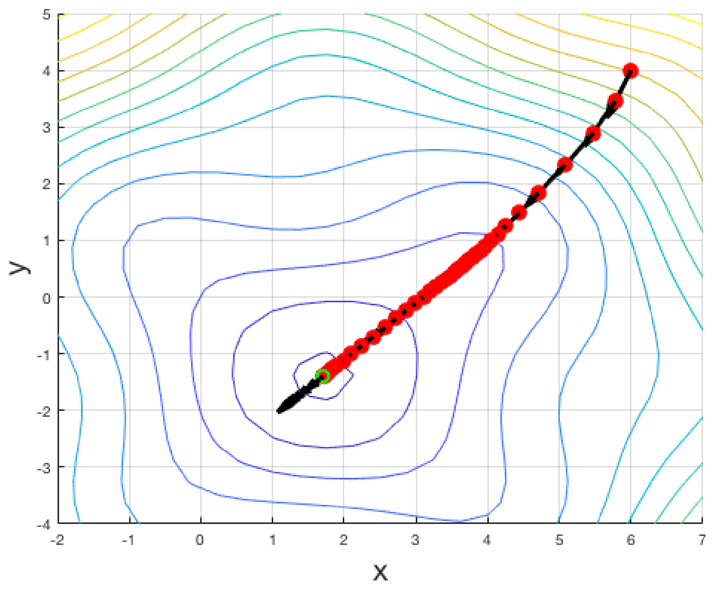
We are still not at the minimum after 20 iterations. Notice that the points are getting closer together. That is because even though `tstep` is fixed (we move along the path for the same amount of time each iteration), the gradient is different at different points and the gradient represents the velocity that we are moving. So when the gradient is small, the points are close together. Let's try more steps.

```
n_iter = 30; % Number of iterations
tstep = .05;
```



Even 30 iterations isn't enough to get to the minimum.

```
n_iter = 40; % Number of iterations
tstep = .05;
```



The minimum is reached somewhere between 30 and 40 iterations.

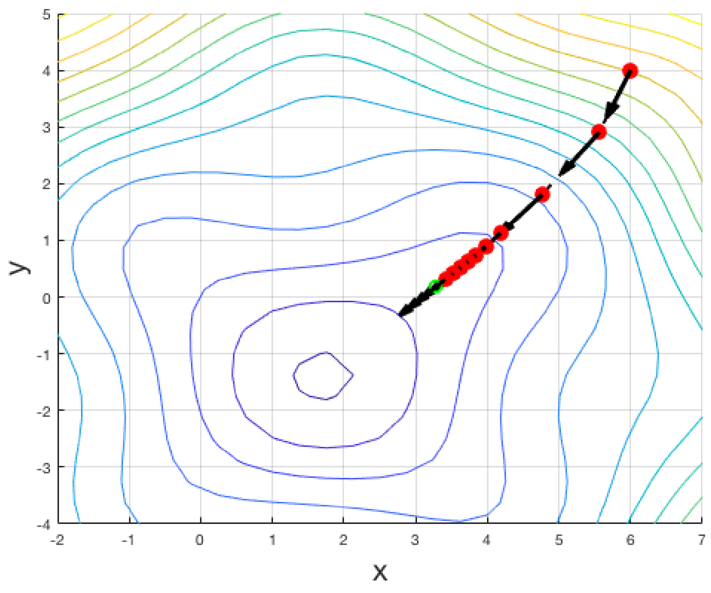
So this method took many more iterations than using `fminbnd` at each iteration. So it must be slower, right? Not necessarily. Even though it took more iterations, each iteration was relatively fast because the formula

```
p = p - tstep*grad
```

is simple. For the other method, we need to solve an optimization problem (using `fminbnd`) for EVERY iteration. That could potentially be very slow. So which method is faster depends on the problem. Sometimes it is faster to do less iterations (`fminbnd`) and sometimes it is faster to do avoid solving an optimization problem at each step (fixed step size). Often times, the most "expensive" part (meaning most operations) of the fixed step size algorithm is calculating the gradient at every step. So the faster method depends on how slow it is to calculate gradients vs how slow it is to do an optimization using `fminbnd`.

The method can converge in fewer iterations with a better choice of `tstep`. Let's try doubling it.

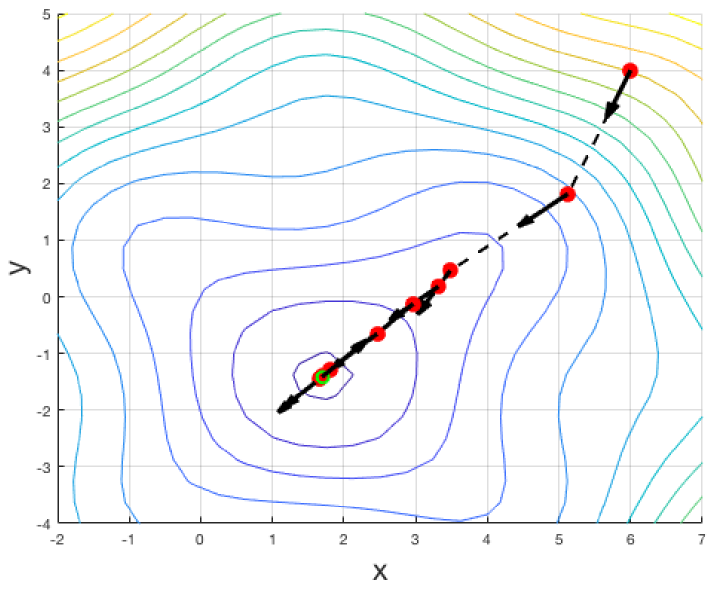
```
n_iter = 10; % Number of iterations
tstep = .1;
```



Notice that we are closer to the minimum after 10 iterations than we were with `tstep = 0.05`.

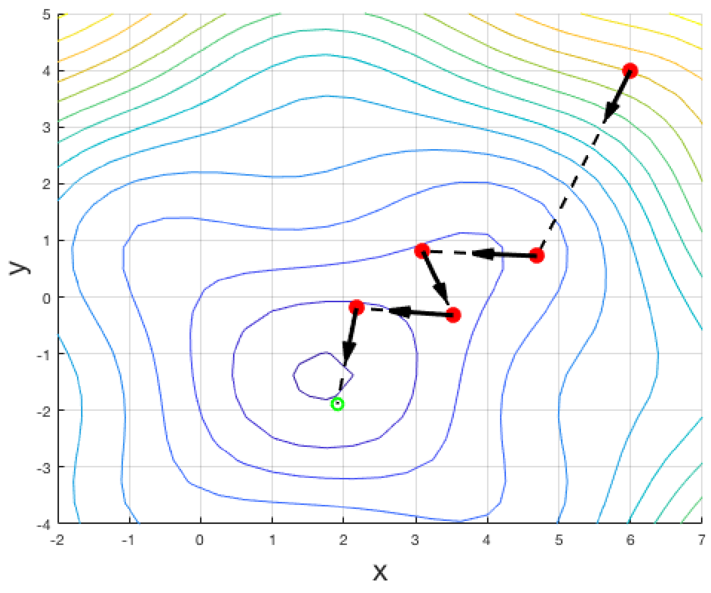
We can do even better with a larger step size.

```
n_iter = 10; % Number of iterations
tstep = .2;
```



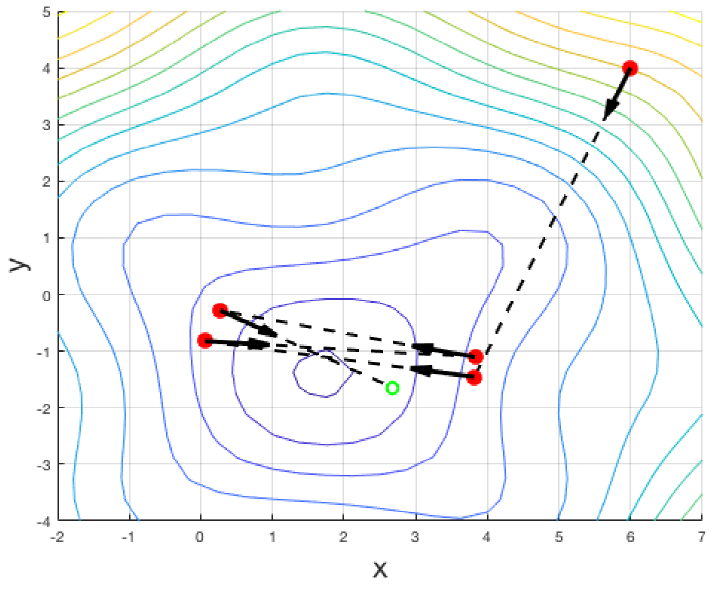
Notice that we reached the minimum in under 10 steps! However, the path is less smooth than before. The path becomes even more jagged if we increase the value of tstep.

```
n_iter = 5; % Number of iterations
tstep = .3;
```



We still get close to the minimum in 5 iterations, but the path to getting there is far from direct. Things get even worse if we keep increasing tstep.

```
n_iter = 5; % Number of iterations
tstep = .5;
```



The first step takes us very close to the minimum, but then we start jumping back and forth across the minimum. This is typical of choosing a step size (or learning rate when doing machine learning) that is too large. You either hop back and forth across the minimum or you can miss it completely. So for this problem, something like `tstep = 0.2` seems like a happy medium. The problem is that in practice, it is impossible to know what value of `tstep` will be best except by guessing and checking. If things are taking too long, you may have chosen too small of a value. If your guesses are erratic and jumping all over, you may have chosen too large.