Christopher Lum
lum@uw.edu

Lecture 10d
Numerical Algorithms for Unconstrained Optimization: Gradient Descent

The YouTube video entitled 'Numerical Optimization Algorithms: Gradient Descent' that covers this lecture is located at https://youtu.be/qcFdpBi5i38.

Outline

-Introduction
-Numerical Optimization Algorithms
-Choosing a Descent Direction
    -A. Gradient Descent Method

Introduction

As we saw previously, in unconstrained optimization problems, it may be difficult or impossible to solve analytically for stationary points. Therefore, we result to numerical techniques to try and find these stationary points.

The general roadmap of what we will cover over the next several discussions is shown below

-Numerical Optimization Algorithms
-Choosing a Descent Direction
    -Gradient Descent ← topic of this lecture
    -Newton's Method
-Choosing a Step Size
    -Constant Step Size
    -Line Minimization
    -Armijo Rule

Numerical Optimization Algorithms

Numerical optimization algorithms attempt to find local minima (or maxima) by iteratively finding points until a stationary point is reached. Basically, we start at some point x^0 (an initial guess) and successively generate vectors/points x^1, x^2, ..., x^n until we reach a stationary point.

A simple numerical algorithm to generate these points is

$$x^{k+1} = x^k + \alpha^k d^k \qquad k = 0, 1, \ldots \qquad \textbf{(Eq.1.5)}$$

where α^k = step size (positive scalar)
d^k = direction (vector)

In Eq.1.5, the direction d^k can be chosen in any direction. Typically, this is done such that the cost function value decreases along this direction. Choosing the step size α^k is also a problem which should be examined later.

In many (but not all) cases, we desire that f is decreased at each iteration, that is

$$f\left(x^{k+1}\right) < f\left(x^k\right) \qquad k = 0, 1, \ldots$$

In doing so, we successively improve our current solution estimate and we hope to decrease f all the way to its minimum. These types of algorithms are called **descent algorithms** because the function value is always descending.

In the case of descent algorithms, the direction d^k is sometimes referred to as a **descent direction** as it typically leads to the cost function value being decreased if we move in this direction.

Show movie of a baby playing hide and seek to illustrate search algorithm structure (ie walk in a direction, reassessing the situation, walk in new direction, repeat...)

By examining Eq.1.5, we see that there are two main choices when designing an algorithm.

1. How do we choose the descent direction, d^k
2. How do we choose the step size, α^k

We investigate several choices for each selection in the next several sections

Once we choose a descent direction and step size, the basic framework of the numerical optimization algorithm is as follows.

1. Start at a point x^k (can be initialized with a best guess)
2. Choose descent direction as d^k
   a. Gradient descent
   b. Newton
3. Choose step size α^k
   a. Constant step size
   b. Line minimization
   c. Armijo rule
   d. Others (Goldstein rule, diminishing step size, etc.)
4. Compute new point as $x^{k+1} = x^k + \alpha^k d^k$
5. Repeat from step 1 until termination criteria is met
   a. Reach a stationary point
   b. $|| \nabla f(x^k) ||$ is less than a specified tolerance
   c. Exceed maximum number of steps

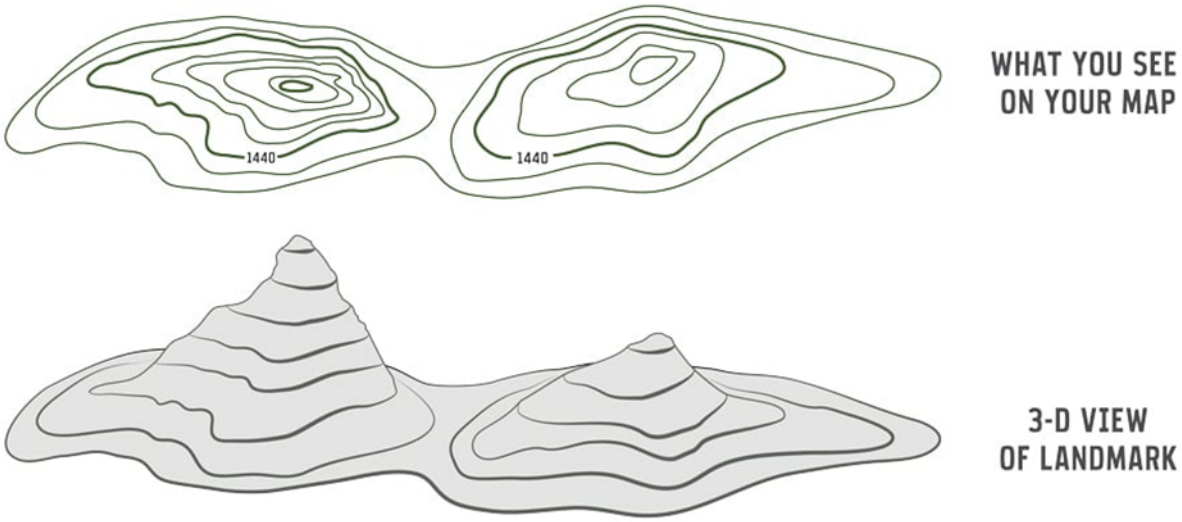
Choosing a Descent Direction

We would like to choose a direction d^k such that the cost function value decreases if we move in this direction.

A. Gradient Descent Method

Perhaps the most famous is the **method of steepest descent** or **gradient descent method**.

The basic idea behind gradient descent methods are to choose successive points that "go downhill" with respect to the cost function. Typically, the direction that is "downhill" is chosen as the negative gradient, which gives the name of this family of algorithms. This is most conveniently visualized when the decision vector is in $\Re^2$ and the cost function is therefore a 3D plot.



With this choice, we have

$$d^k = -\nabla f\left(x^k\right)$$

In many situations, we would like to normalize this to be a unit vector.

$$d^k = -\nabla f\left(x^k\right) / \| \nabla f\left(x^k\right) \|$$

We can easily verify that this is a descent direction by calculating the directional derivative of the function in the direction d^k. Recall from our discussion on vector differential calculus that if $\overline{b}$ is a unit vector, we can express the directional derivative of f in the direction of $\overline{b}$ as

$$D_{\overline{b}} f = \overline{b} \cdot \nabla f \qquad (\text{if } | \overline{b} | = 1)$$

$$= d^k \cdot \nabla f\left(x^k\right) \qquad \text{note: } d^k = -\nabla f\left(x^k\right) / \| \nabla f\left(x^k\right) \|$$

$$= \left(-\nabla f\left(x^k\right) / \| \nabla f\left(x^k\right) \|\right) \cdot \nabla f\left(x^k\right)$$

$$= -\frac{1}{\|\nabla f\left(x^k\right)\|} \left\{\nabla f\left(x^k\right) \cdot \nabla f\left(x^k\right)\right\}$$

$$= -\frac{1}{\|\nabla f\left(x^k\right)\|} \| \nabla f\left(x^k\right) \|^2$$

$$D_{d^k} f = -\| \nabla f\left(x^k\right) \|$$

We see that the left side of the expression $-\| \nabla f\left(x^k\right) \|$ is less than or equal to 0 in all situations. The only time it is equal to zero is if $\nabla f\left(x^k\right) = 0$ which is the definition of a stationary point.

Note that we can only guarantee that the cost function decreases in the direction of

$d^k = -\nabla f(x^k) / \| \nabla f(x^k) \|$ at the point x^k. If we move any distance away from x^k, the gradient may change as we will see shortly.

So for example, if feasible set is a subset of $\Re^2$ and we have a quadratic cost function

Example: 2D Quadratic Cost Function

Consider a cost function of the form

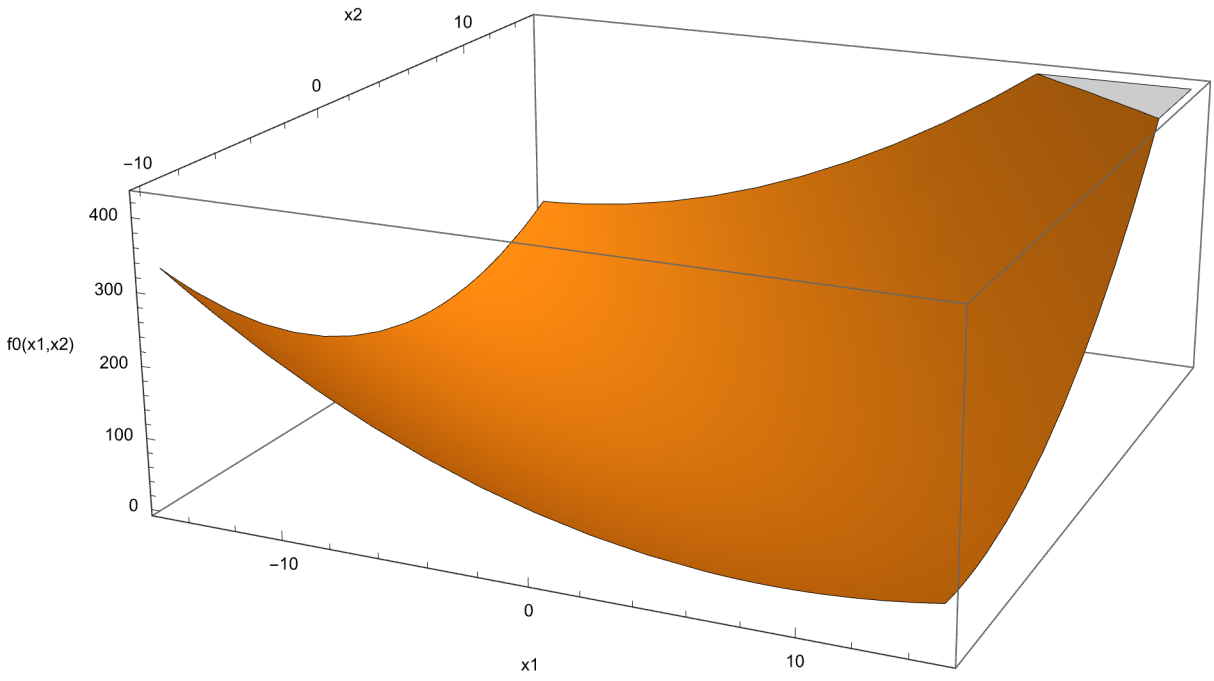
$$J = f(x) = \frac{1}{2} x^T H x + g^T x + r$$

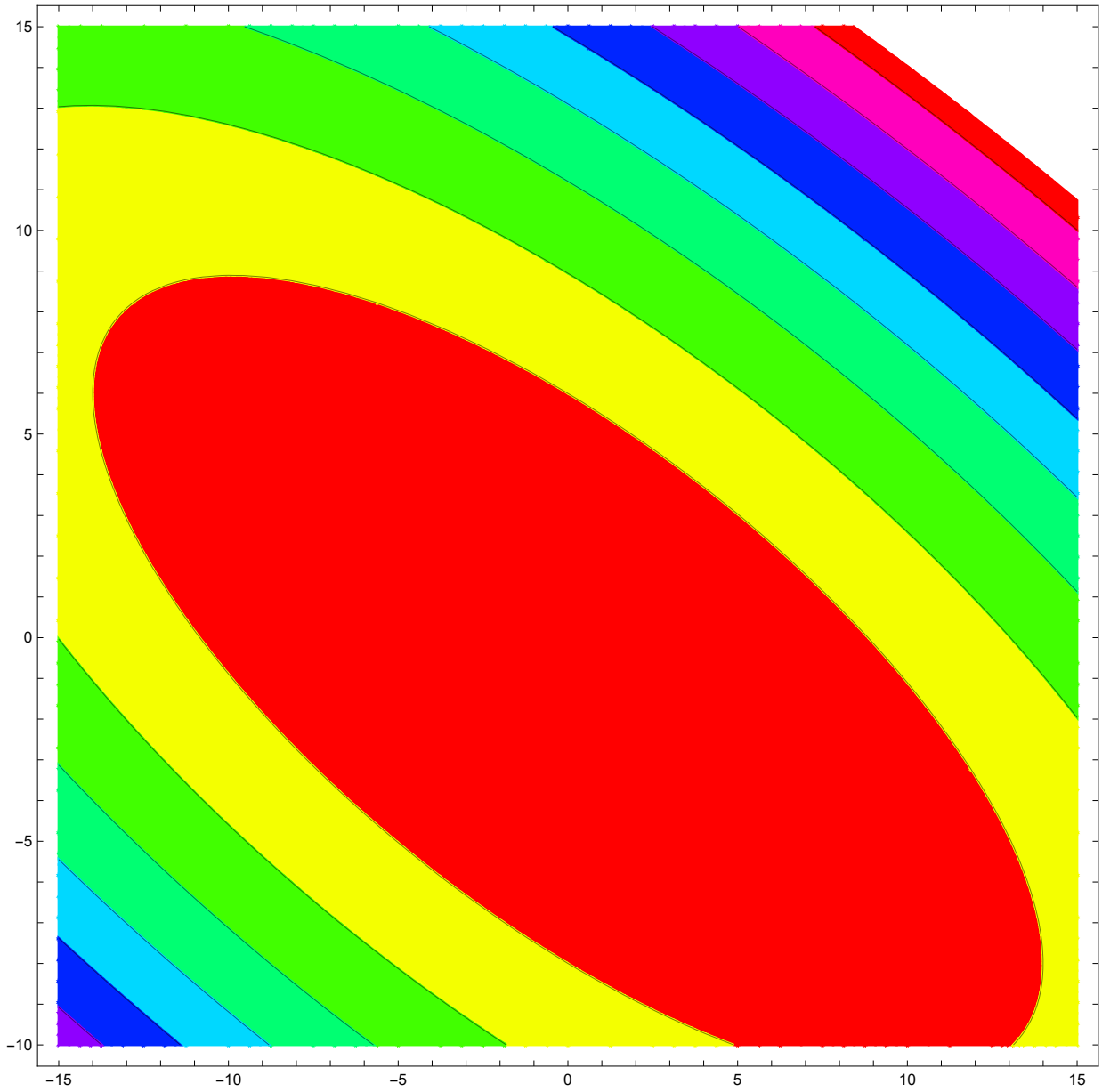
where $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ $H = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}$ $g = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$ $r = 2$

```
x = {{x1}, {x2}}; H = {{1, 1}, {1, 2}}; g = {{1}, {2}}; r = 2;
temp = 1/2 Transpose[x].H.x + Transpose[g].x + r;
f[x1_, x2_] = temp[[1, 1]] // Expand
```

$$2 + x1 + \frac{x1^2}{2} + 2\,x2 + x1\,x2 + x2^2$$

```
(*visualize the function*)
x1min = -15; x1max = 15; x2min = -10; x2max = 15;
fighSurface = Plot3D[f[x1, x2], {x1, x1min, x1max},
  {x2, x2min, x2max}, AxesLabel → {"x1", "x2", "f0(x1,x2)"}, Mesh → False]
fighContour = ContourPlot[f[x1, x2], {x1, x1min, x1max},
  {x2, x2min, x2max}, PlotPoints → 25, ColorFunction → Hue]
```


x2
10
0
−10
400
300
200
100
0
f0(x1,x2)
−10
0
10
x1



Depending on the initial condition, the algorithm would walk down the contour in successive steps.

For example, consider an initial condition of

$$x^0 = \begin{pmatrix} -5 \\ 12 \end{pmatrix}$$

```
x0 = {{-5}, {12}};
```

We can easily compute the gradient at this location

```
gradF[x1_, x2_] = ( D[f[x1, x2], x1] ) // Simplify;
                  ( D[f[x1, x2], x2] )
gradF[x1, x2] // MatrixForm
```

```
gradFx0 = gradF[x0[[1, 1]], x0[[2, 1]]];
gradFx0 // MatrixForm
```

$$\begin{pmatrix} 1 + x1 + x2 \\ 2 + x1 + 2\,x2 \end{pmatrix}$$

$$\begin{pmatrix} 8 \\ 21 \end{pmatrix}$$

We can compute the descent direction

```
normgradFx0 = (gradFx0[[1, 1]]^2 + gradFx0[[2, 1]]^2)^(1/2)
d0 = - gradFx0 / normgradFx0;
d0 // MatrixForm
d0 // MatrixForm // N
```

$$\sqrt{505}$$

$$\begin{pmatrix} -\frac{8}{\sqrt{505}} \\ -\frac{21}{\sqrt{505}} \end{pmatrix}$$

$$\begin{pmatrix} -0.355995 \\ -0.934488 \end{pmatrix}$$

We can also compute the cost function value at x^0

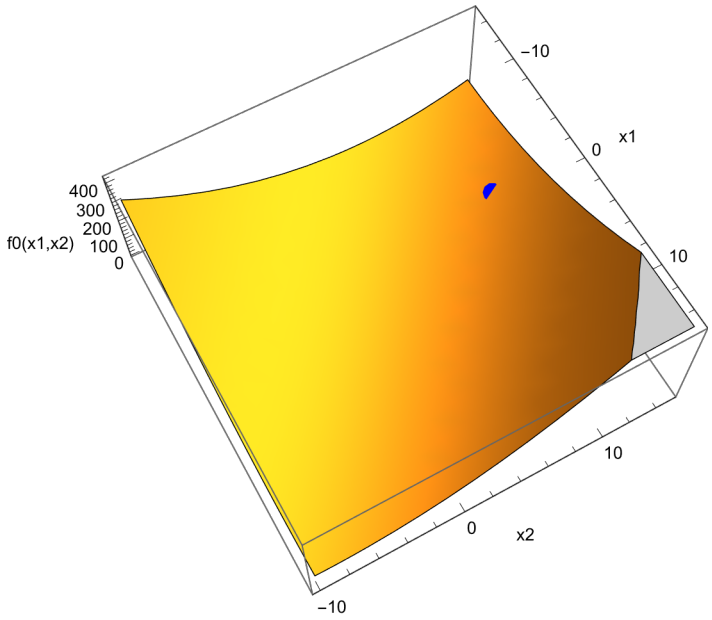
```
fx0 = f[x0[[1, 1]], x0[[2, 1]]]
```

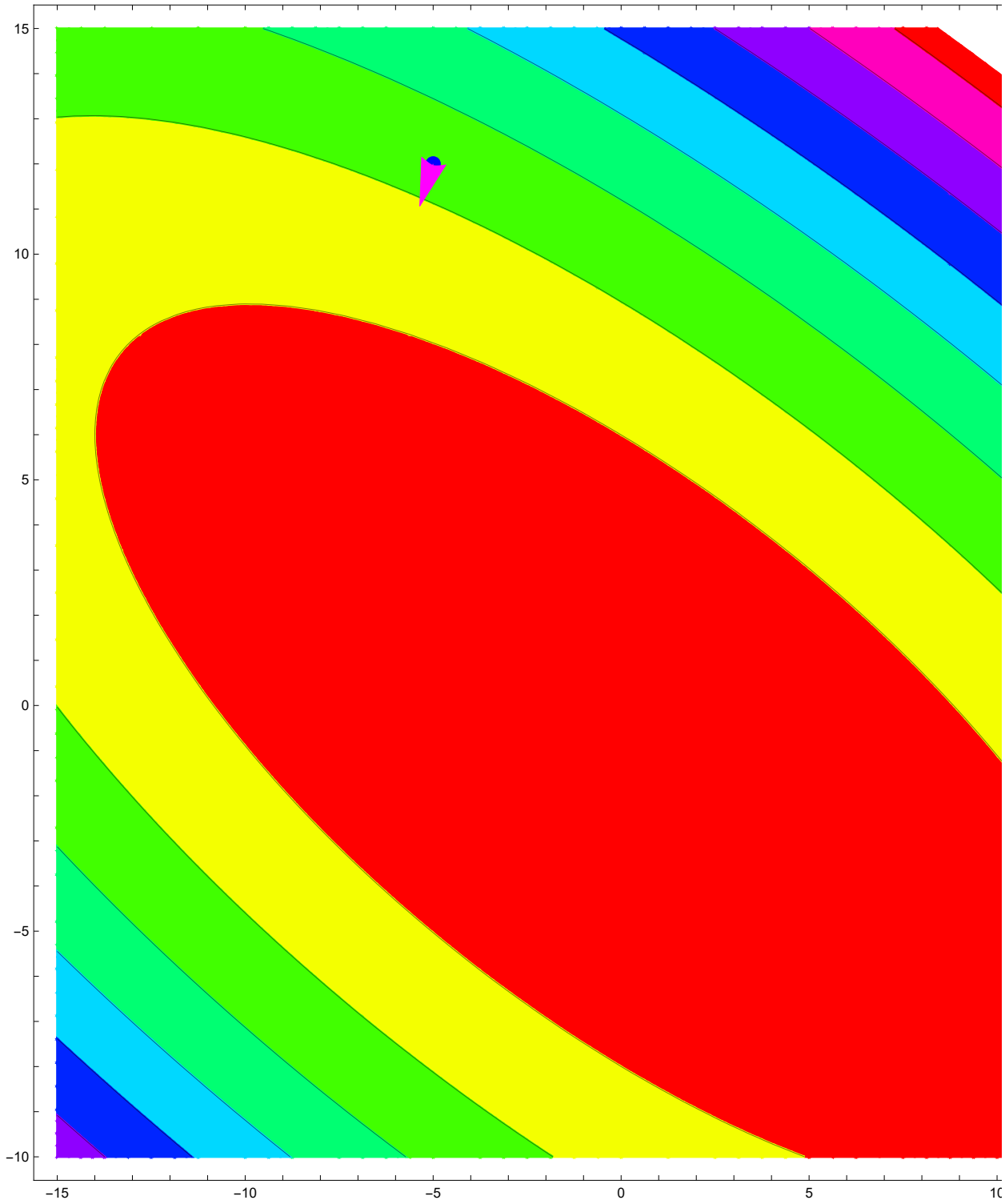
$$\frac{235}{2}$$

We can plot the descent direction at this location

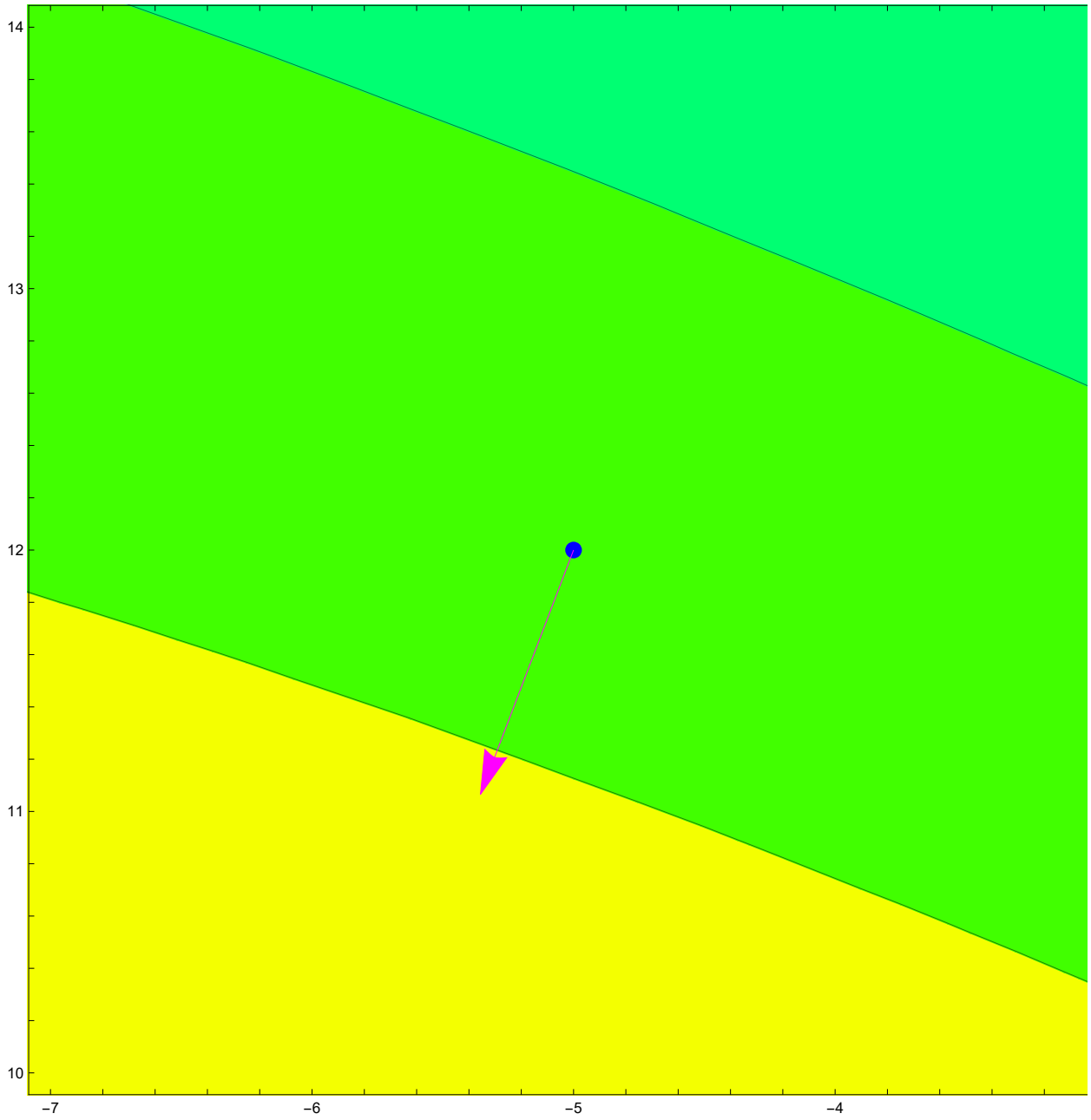
```
arrowStart3D = {x0[[1, 1]], x0[[2, 1]], fx0};
arrowEnd3D = arrowStart3D + {d0[[1, 1]], d0[[2, 1]], 0};

arrowStart2D = {x0[[1, 1]], x0[[2, 1]]};
arrowEnd2D = arrowStart2D + {d0[[1, 1]], d0[[2, 1]]};

(*Plot the point*)
fighpt2D = Graphics[
   {
    AbsolutePointSize[10], Blue, Point[{x0[[1, 1]], x0[[2, 1]]}]
   }
  ];

fighpt3D = Graphics3D[
```

```
    {
     AbsolutePointSize[10], Blue, Point[{x0[[1, 1]], x0[[2, 1]], fx0}]
    }
   ];

(*Plot the direction*)
fighArrow3D = Graphics3D[
    {
     Magenta,
     Arrow[{arrowStart3D, arrowEnd3D}]
    }
   ];

fighArrow2D = Graphics[
    {
     Magenta,
     Arrow[{arrowStart2D, arrowEnd2D}]
    }
   ];

(*Generate plots*)
Show[fighSurface, fighpt3D]
Show[fighContour, fighpt2D, fighArrow2D]

(*Zoom into to area of interest*)
Print["Zoomed in view"]
deltaX1 = 2;
deltaX2 = 2;
deltaF0 = 20;

Show[fighContour, fighpt2D, fighArrow2D,
 PlotRange →
   {{x0[[1, 1]] - deltaX1, x0[[1, 1]] + deltaX1}, {x0[[2, 1]] - deltaX2, x0[[2, 1]] + deltaX2}}]
```


f0(x1,x2)
400
300
200
100
0
−10
0
x1
10
10
0
x2
−10



Zoomed in view



This may have problems of slow convergence if the cost function is shaped in a certain way (for example long and narrow functions)