Multivariable unconstrained optimization

December 10, 2018

1 Introduction

Consider the problem of maximizing a *concave* function $f(\mathbf{x})$ of *multiple* variables $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ where there are no constraints on the feasible values. A number of search procedures are available for solving such a problem numerically. One of these, the *gradient search procedure*, is an especially important one because it identifies and uses the direction of movement from the current trial solution that maximizes the rate at which $f(\mathbf{x})$ is increased.

2 3D Plotting and Typesetting

The following will initialize the notebook, define a typesetting for plotting, and a function `plot3d` that creates plots in 3D.

```
In [1]: %pylab inline

Populating the interactive namespace from numpy and matplotlib
```

```
In [2]: import matplotlib.pyplot as plt
        import numpy as np
        from mpl_toolkits.mplot3d import Axes3D
        from numpy import *
        from matplotlib.patches import FancyArrowPatch
        from mpl_toolkits.mplot3d import proj3d
        font = {'size': 14}
        matplotlib.rc('font', **font)
```

```
In [3]: class Arrow3D(FancyArrowPatch):
            def __init__(self, xs, ys, zs, *args, **kwargs):
                FancyArrowPatch.__init__(self, (0,0), (0,0), *args, **kwargs)
                self._verts3d = xs, ys, zs

            def draw(self, renderer):
                xs3d, ys3d, zs3d = self._verts3d
                xs, ys, zs = proj3d.proj_transform(xs3d, ys3d, zs3d, renderer.M)
                self.set_positions((xs[0],ys[0]),(xs[1],ys[1]))
                FancyArrowPatch.draw(self, renderer)
```

```
In [4]: def plot3d(f,lim=(-5,5),title='Surface plot',detail=0.05,path=[]):
            fig = plt.figure(figsize=(16,12))
            ax = fig.add_subplot(111, projection='3d')
            xs = ys = np.arange(lim[0],lim[1], detail)
            X, Y = np.meshgrid(xs, ys)
            zs = np.array([f(x, y) for x, y in zip(np.ravel(X), np.ravel(Y))])
            Z = zs.reshape(X.shape)
            surf = ax.plot_surface(X, Y, Z, cmap=cm.Blues)
            fig.colorbar(surf, shrink=0.5, aspect=5)
            xlabel('X-axis');ylabel('Y-axis');ax.set_zlabel('Z-axis')
            plt.title(title);
            if len(path):
                xs, ys, zs = path[0], path[1], path[2]
                ax.plot(xs,ys,zs, 'o', color='r', markersize=5, zorder=3)
                for i in range(len(xs)-1):
                    ax.add_artist(Arrow3D([xs[i], xs[(i+1)%len(xs)]],
                                          [ys[i], ys[(i+1)%len(xs)]],
                                          [zs[i], zs[(i+1)%len(xs)]],
                                          mutation_scale=15, lw=2, arrowstyle="-|>",
                                          color='r', zorder=4))
            return fig, ax
```

3 Convexity/Concavity Tests

The following function calculates the Hessian for a function $f(\mathbf{x})$ to determine if it is convex.

```
In [63]: def hessian(f):
             x, y = symbols('x y')
             f = f(x,y)
             pf_px = f.diff(x)
             pf_py = f.diff(y)
             p2f_pxpy = pf_px.diff(y)
             return Matrix([[pf_px.diff(x), p2f_pxpy], [p2f_pxpy, pf_py.diff(y)]])
```

If the determinant of the Hessian is positive and $\frac{\partial f}{\partial x \partial y}$ is also positive, the function is convex.

```
In [73]: def is_convex(f):
             H = hessian(f)
             return H.det() > 0 and H[0,0] > 0
```

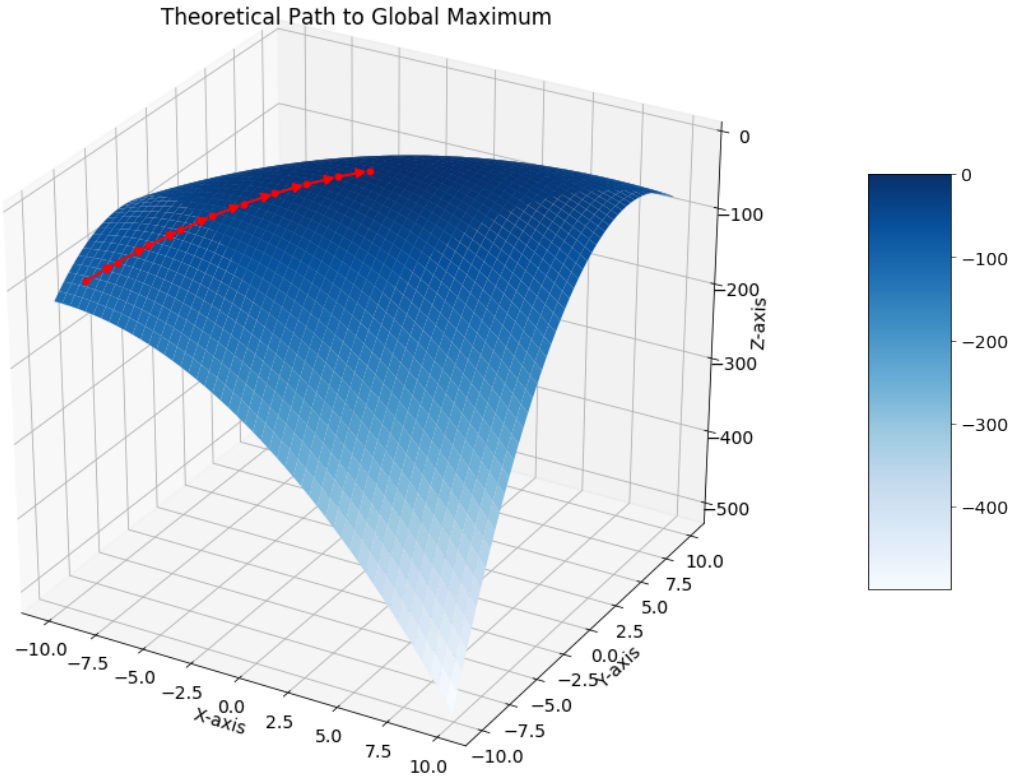
If the determinant of the Hessian is positive and $\frac{\partial f}{\partial x \partial y}$ is negative, the function is concave.

```
In [74]: def is_concave(f):
             H = hessian(f)
             return H.det() > 0 and H[0,0] < 0
```

4 The Gradient Search Procedure

For a single variable function, there are only two possible directions (increase x or decrease x) in which to move from the current trial solution to the next one. The goal was to reach a point where eventually the derivative is (essentially) 0. Now, there are *innumerable* possible directions in which to move; they correspond to the possible *proportional rates* at which the respective variables can be changed. The goal is to reach a point eventually where all the partial derivates are (essentially) 0.

```
In [90]: n = 10
         f = lambda x,y: 2*x*y + 2*y - x**2 - 2*y**2
         xs = np.linspace(-9,0,n)
         ys = np.linspace(-9,0,n)
         zs = f(xs,ys)
         plot3d(f, lim=(-10,10), path=[xs,ys,zs],
                title='Theoretical Path to Global Maximum');
```



Because the objective function $f(\mathbf{x})$ is assumed to be differentiable, it possesses a gradient, denoted by $\nabla f(\mathbf{x})$, at each point $\mathbf{x}$. In particular, the **gradient** at a specific point $\mathbf{x} = \mathbf{x}'$ is the *vector* whose elements are the respective *partial derivatives* evaluated at $\mathbf{x} = \mathbf{x}'$, so that:

$$\nabla f(\mathbf{x}') = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right) \quad \text{at } \mathbf{x} = \mathbf{x}'.$$

The significance of the gradient is that the (infinitesimal) change in $\mathbf{x}$ that *maximizes* the rate at which $f(\mathbf{x})$ increases is the change *proportional* to $\nabla f(\mathbf{x})$. It may be said that the rate at which $f(\mathbf{x})$ increases is maximized if (infinitesimal) changes in $\mathbf{x}$ are in the *direction* of the gradient $\nabla f(\mathbf{x})$. Because the current problem has no constraints, this interpretation of the gradient suggests that an efficient search procedure should keep moving in the direction of the gradient until it (essentially) reaches an optimal solution $\mathbf{x}^*$, where $\nabla f(\mathbf{x}^*) = 0$.

5 Summary of the Gradient Search Procedure

Add a summary of the gradient search procedure.

6 Example

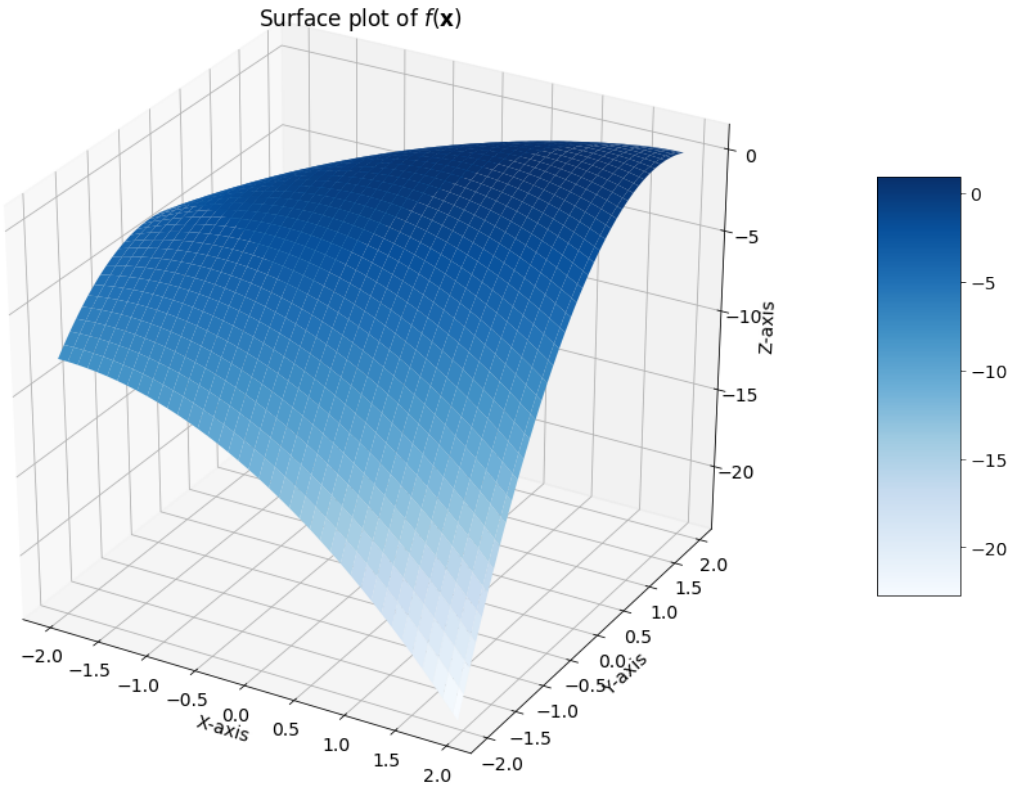
Consider the following two-variable problem:

$$\max f(\mathbf{x}) = 2xy + 2y - x^2 - 2y^2$$

```
In [154]: f = lambda x,y: 2*x*y + 2*y - x**2 - 2*y**2
          lim=(-2,2)
```

Let's take a look at a surface plot of $f(\mathbf{x})$:

```
In [155]: plot3d(f, lim=lim, title='Surface plot of $f(\\mathbf{x})$');
```



First we will check if $f(\mathbf{x})$ is a concave function, for $f(\mathbf{x})$ to be concave, the determinant of the Hessian of $f(\mathbf{x})$ should be positive, and any second-order partial derivate $f(\mathbf{x})_{xy}$ or $f(\mathbf{x})_{yx}$ must be negative.

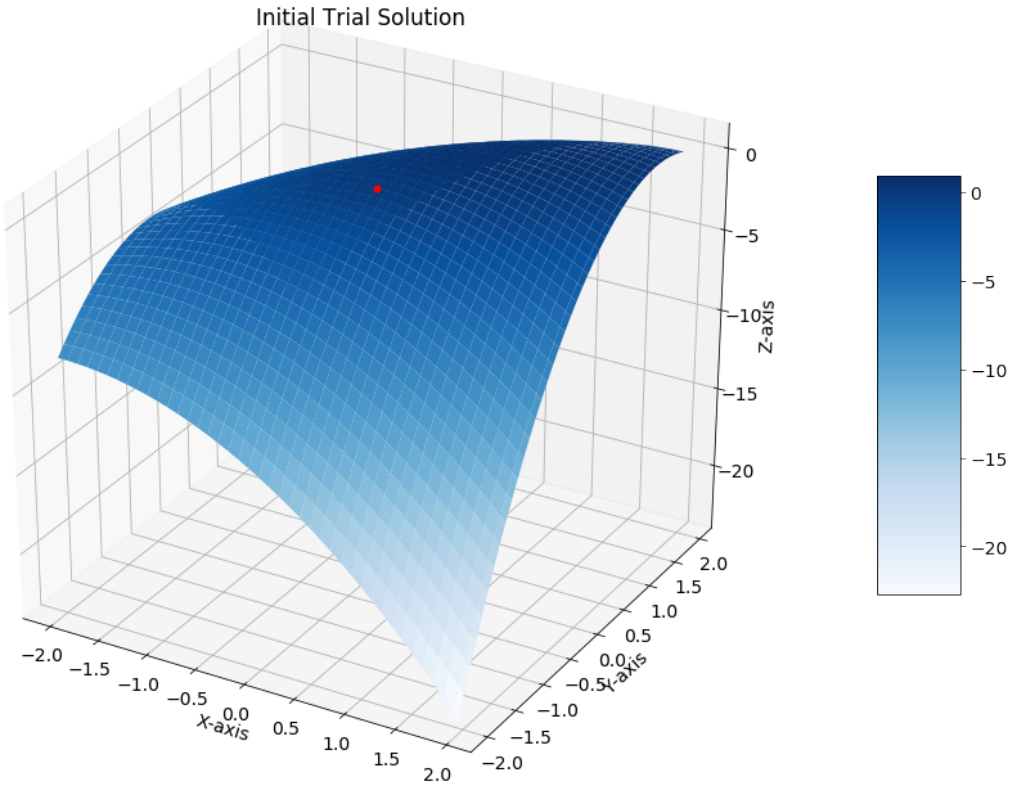
```
In [156]: hessian(f)

Out[156]: Matrix([
          [-2,  2],
          [ 2, -4]])

In [157]: is_concave(f)

Out[157]: True
```

From $f(\mathbf{x})$ we can find that $\frac{\partial f}{\partial x} = 2y - 2x$ and $\frac{\partial f}{\partial y} = 2x + 2 - 4y$. This gives us the gradient $\nabla f(\mathbf{x})$:

```
In [172]: grad = lambda x: np.array([2*x[1] - 2*x[0],
                                     2*x[0] + 2 - 4*x[1]]) # gradient of f

          xp = np.array([0, 0]) # trial solution
```

To begin the gradient search procedure, after choosing a suitably small value of ϵ (normally well under 0.1) suppose that $\mathbf{x} = (0, 0)$ is selected as the initial trial solution.

```
In [159]: plot3d(f, lim=lim, path=[[0],[0],[f(0,0)]],
                 title='Initial Trial Solution');
```



Because the respective partial derivatives are 0 and 2 at this point, the gradient is $\nabla f(0,0) = (0,2)$. With $\epsilon < 2$, the stopping rule then says to perform an iteration.

```
In [160]: grad([0,0])
```

```
Out[160]: array([0, 2])
```

Iteration 1: With values of 0 and 2 for the respective partial derivatives, the first iteration begins by setting:

$$x = 0 + t(0) = 0y = 0 + t(2) = 2t$$

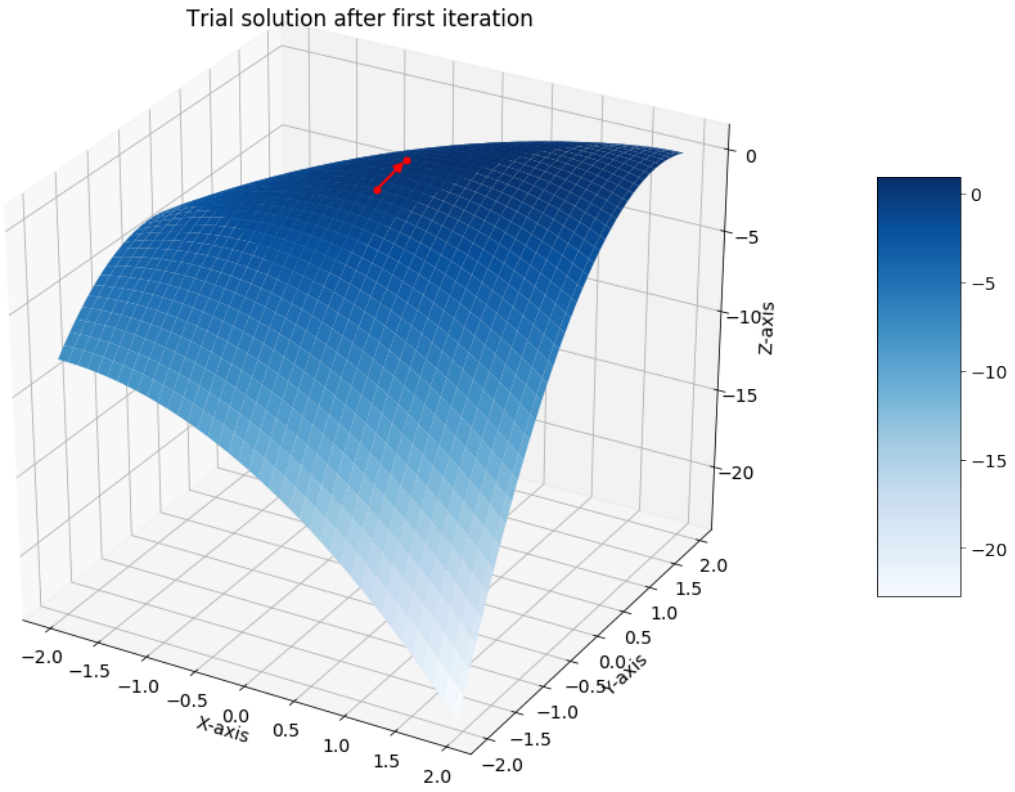
and then substituting these expressions into $f(\mathbf{x})$ to obtain

$$f(\mathbf{x}' + t\nabla f(\mathbf{x}')) = f(0, 2t) = 4t - 8t^2.$$

Now we need to find a maximum in a function of one-variable, we can do this by differentiating $4t - 8t^2$ which is $4 - 16t$, from this it follows that $t^* = \frac{1}{4}$, so we reset: $\mathbf{x}' = (0,0) + \frac{1}{4}(0,2) = (0, \frac{1}{2})$. This completes the iteration. For this new trial solution, the gradient is $\nabla f\left(0, \frac{1}{2}\right) = (1,0)$.

```
In [161]: xp = xp + (1/4) * grad(xp)
          print('x={} with gradient ({})'.format(xp, grad(xp)))
          plot3d(f, lim=lim, path=[[0, 0],[0, 1/2],[f(0,0),f(0,1/2)]],
                 title='Trial solution after first iteration');
```

```
x=[0.  0.5] with gradient ([1. 0.])
```



With $\epsilon < 1$, the stopping rule now says to perform another iteration.
Iteration 2: This time we will do the iteration with help of *SymPy*:

```
In [162]: t = symbols('t') # define a symbol t
          u = grad(xp) * t + xp # find gradient vector
          u
```

```
Out[162]: array([1.0*t, 0.500000000000000], dtype=object)
```

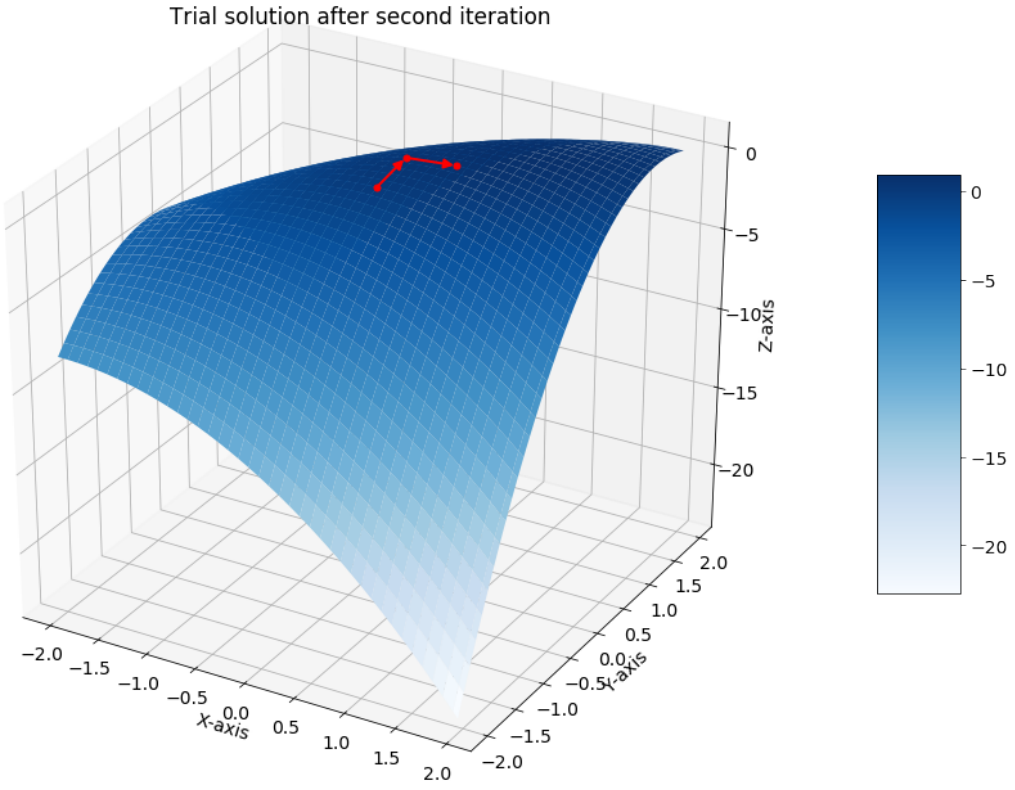
```
In [171]: solve(f(u[0], u[1]).diff()) # substitute the gradient vector into f(x),
                                      # then differentiate and solve for t (max. point).
```

```
Out[171]: [0.500000000000000]
```

So $t^* = \frac{1}{2}$

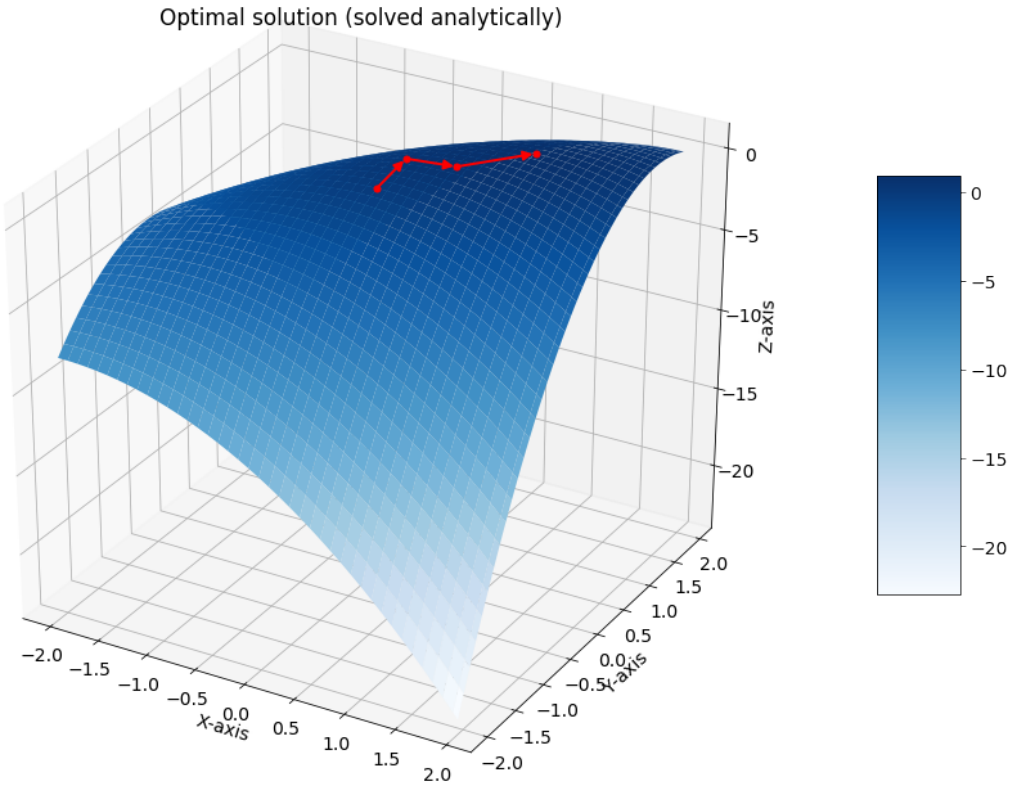
```
In [170]: xp = xp + (1/2) * grad(xp)
          print('x={} with gradient ({})'.format(xp, grad(xp)))
          plot3d(f, lim=lim, path=[[0, 0, .5],[0, 1/2, .5],
                                   [f(0,0),f(0,1/2),f(.5,.5)]],
                 title='Trial solution after second iteration');
```

```
x=[0.5 1. ] with gradient ([ 1. -1.])
```



This completed the second iteration. With a typically small value of ϵ, the procedure would now continue on to several more iterations in similar fashion. If we find the maximum analytically, we find that optimal value is at $(1, 1)$:

```
In [169]: plot3d(f, lim=lim, path=[[0, 0, .5, 1],[0, 1/2, .5, 1],
                                   [f(0,0),f(0,1/2),f(.5,.5),f(1,1)]],
                 title='Optimal solution (solved analytically)');
```



7 Algorithm implementation

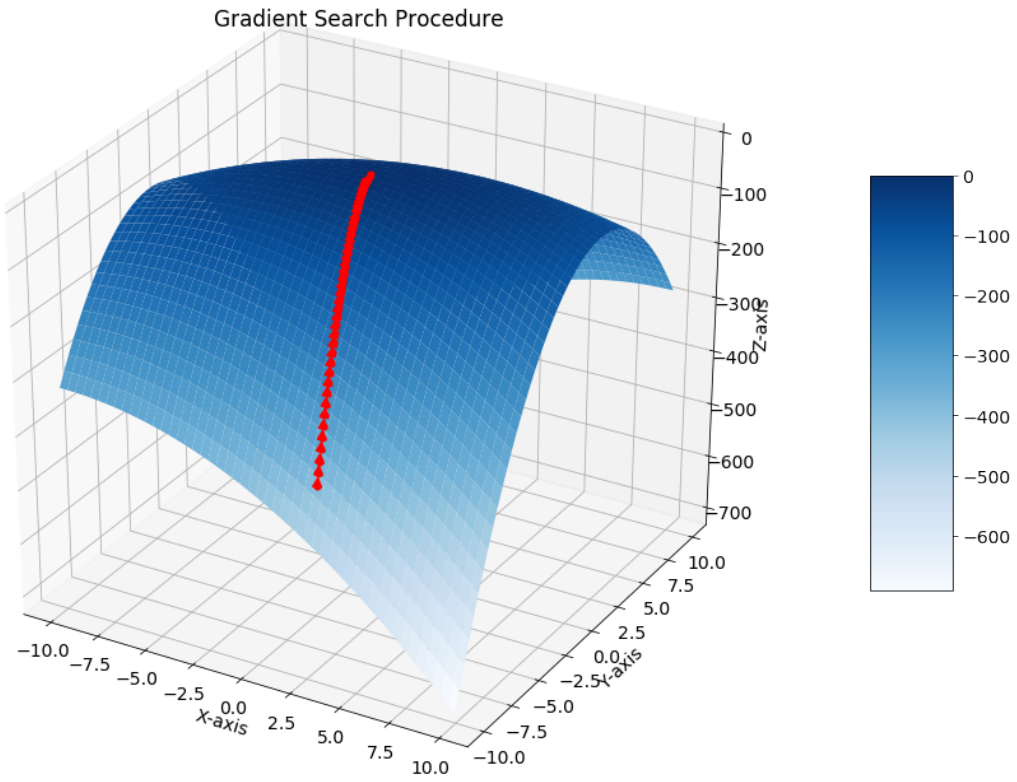
The following implementation of the gradient search procedure will find the optimum, assuming that the Hessian only consists of constants, and that f is concave.

```
In [311]: # function with powers higher than two are not supported.
          f = lambda x1,x2: 2*x1*x2 + 2*x2 - x1**2 - 4*x2**2

          # check if the hessian is positive.
          if hessian(f).det() < 0: raise ValueError('negative hessian!')

          # create a sympy equation from f
          x, y, t = symbols('x y t')
          f_alg = f(x,y)

          # plot limit
          lim = (-10,10)

          # max error
          eps = 5
```

```
# initial trial solution x'
xp = np.array([2.5, -9])

# stores the path (xyz values) to plot
xs = [xp[0]]; ys = [xp[1]]; zs =[f(xp[0], xp[1])]

# get partial derivatives for calculating the gradient
grad = np.array([f_alg.diff(x), f_alg.diff(y)])

# calculate the gradient vector at x'
u = np.array([g.subs(x,xp[0]).subs(y,xp[1]) for g in grad])

# in case it takes forever
bail = 100

# keep iterating while the partials are not (near) zero.
while (abs(u[0]) > eps or abs(u[1]) > eps) and bail > 0:

    d = t*u # multiply t into the gradient vector
    tp = solve(f(d[0], d[1]).diff(t))[0].evalf() # substitute the
                                                 # gradient vector
                                                 # into f, diff, and
                                                 # solve for t.
    xp = xp + tp*u  # set x' to the new x'

    # append path for plotting
    xs.append(xp[0]); ys.append(xp[1]); zs.append(f(xp[0], xp[1]))

    # calculate the gradient at x'
    u = np.array([g.subs(x,xp[0]).subs(y,xp[1]) for g in grad])

    bail -= 1 # in case it takes forever

# force casting to float64
xs = np.array(xs, dtype='float64')
ys = np.array(ys, dtype='float64')
zs = np.array(zs, dtype='float64')

# plot as a surface with the path
plot3d(f, lim=lim, path=[xs,ys,zs],
       title='Gradient Search Procedure')

# show all the variables
f, 'concave: {}'.format(is_concave(f)), f_alg, lim, xp,
grad, u, d, tp, xp, xs, ys, zs, bail
```

```
Out[311]: (array([-2*x + 2*y, 2*x - 8*y + 2], dtype=object),
```

```
array([-0.405601964235434, 3.19945289217659], dtype=object),
array([-0.943729784648726*t, 5.0998725103051*t], dtype=object),
0.0445204527159744,
array([0.0704924941275241, -0.132308487990193], dtype=object),
array([2.5       , 2.43761802, 2.37527779, 2.31298121, 2.2507303 ,
       2.18852723, 2.12637433, 2.06427414, 2.00222937, 1.94024295,
       1.87831809, 1.81645825, 1.75466721, 1.6929491 , 1.63130845,
       1.56975024, 1.50827996, 1.44690368, 1.38562816, 1.32446091,
       1.26341039, 1.20248608, 1.14169873, 1.08106057, 1.02058561,
       0.96028999, 0.90019246, 0.84031498, 0.78068355, 0.7213292 ,
       0.66228947, 0.60361041, 0.54534938, 0.48757917, 0.43039425,
       0.3739205 , 0.31833124, 0.26387512, 0.21092851, 0.16010396,
       0.11250777, 0.07049249]),
array([-9.        , -8.78573145, -8.57143537, -8.35711051, -8.14275554,
       -7.92836905, -7.71394949, -7.49949522, -7.28500446, -7.07047528,
       -6.8559056 , -6.64129314, -6.42663543, -6.21192977, -5.9971732 ,
       -5.78236248, -5.56749403, -5.35256388, -5.13756764, -4.9225004 ,
       -4.70735669, -4.49213031, -4.27681431, -4.06140075, -3.84588055,
       -3.63024326, -3.41447678, -3.19856691, -2.98249693, -2.76624687,
       -2.54979262, -2.33310473, -2.11614654, -1.89887171, -1.6812203 ,
       -1.46311281, -1.24444031, -1.02504755, -0.80470146, -0.58302695,
       -0.35935712, -0.13230849]),
array([-3.93250000e+02, -3.75102268e+02, -3.57381912e+02, -3.40088966e+02,
       -3.23223462e+02, -3.06785435e+02, -2.90774922e+02, -2.75191961e+02,
       -2.60036591e+02, -2.45308856e+02, -2.31008799e+02, -2.17136468e+02,
       -2.03691912e+02, -1.90675184e+02, -1.78086338e+02, -1.65925434e+02,
       -1.54192535e+02, -1.42887707e+02, -1.32011022e+02, -1.21562557e+02,
       -1.11542394e+02, -1.01950621e+02, -9.27873342e+01, -8.40526380e+01,
       -7.57466455e+01, -6.78694805e+01, -6.04212791e+01, -5.34021916e+01,
       -4.68123850e+01, -4.06520457e+01, -3.49213839e+01, -2.96206382e+01,
       -2.47500821e+01, -2.03100324e+01, -1.63008617e+01, -1.27230143e+01,
       -9.57703059e+00, -6.86358416e+00, -4.58354053e+00, -2.73805872e+00,
       -1.32878334e+00, -3.58261822e-01]),
59)
```



8 References

[1] Multivariable unconstrained optimization, *Introduction to Operations Research*, 10th ed., Hillier and Lieberman.

9 Appendix: SymPy Testing

```
In [41]: from sympy import symbols
         from sympy import diff
         from sympy import solve
         from sympy import Matrix

In [9]: x,y,z,t = symbols('x y z t')

In [10]: expr = f(t, t)
         expr

Out[10]: -t**2 + 2*t

In [152]: a = 2*t - t**2
          a > 0
```

```
Out[152]: -t**2 + 2*t > 0
```

```
In [11]: expr.subs(t, 1)
```

```
Out[11]: 1
```

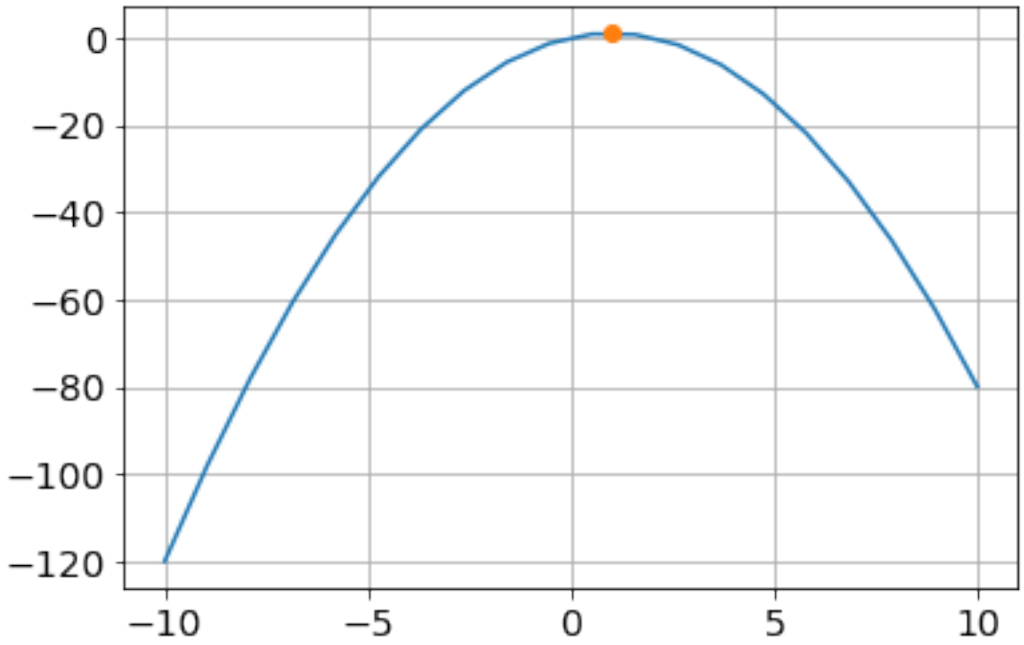
```
In [12]: d_dt = diff(expr)
         d_dt
```

```
Out[12]: -2*t + 2
```

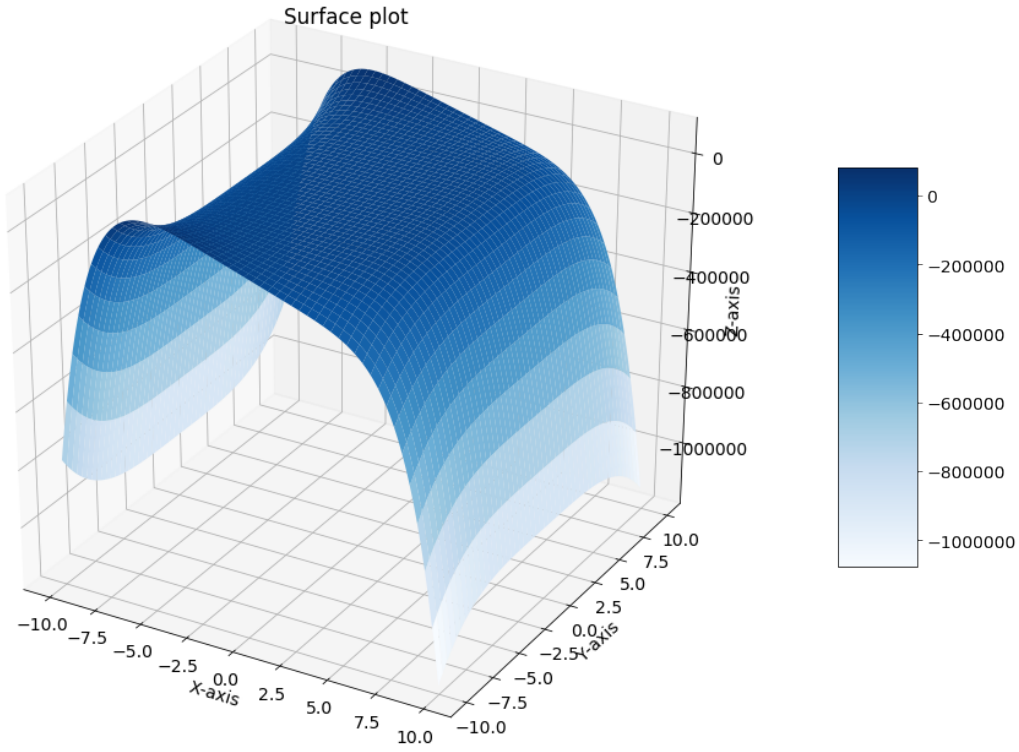
```
In [13]: solve(d_dt, t)
```

```
Out[13]: [1]
```

```
In [14]: xs = np.linspace(-10,10,20)
         ys = [expr.subs(t, x) for x in xs]
         plot(xs,ys)
         plot(1, expr.subs(t, 1), 'o')
         grid();
```




```
In [44]: plot3d(lambda x,y: 2*x*y**3 + 2*y**3 - x**6 - 2*y**4*x, lim=(-10,10));
```




```
In [15]: f = 2*x*y**3 + 2*y**3 - x**6 - 2*y**4*x
         diff(f, x)

Out[15]: -6*x**5 - 2*y**4 + 2*y**3

In [ ]: type(f)

In [45]: diff(f, y)

Out[45]: -8*x*y**3 + 6*x*y**2 + 6*y**2

In [60]: def hessian(f):
             x, y = symbols('x y')
             f = f(x,y)
             pf_px = f.diff(x)
             pf_py = f.diff(y)
             p2f_pxpy = pf_px.diff(y)
             return Matrix([[pf_px.diff(x), p2f_pxpy], [p2f_pxpy, pf_py.diff(y)]])

In [61]: h = hessian(f)
         h
```

```
Out[61]: Matrix([
         [-2,  2],
         [ 2, -4]])

In [37]: (h.det() > 0)

Out[37]: -30*x**4*(-24*x*y**2 + 12*x*y + 12*y) - (-8*y**3 + 6*y**2)**2 > 0

In [91]: type(int(h.det()))

Out[91]: int

In [92]: int(h.det())

Out[92]: 4
```