opt.grad Gradient descent

Consider a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$ that represents some cost or value. This is called an **objective function**, and we often want to find an $\boldsymbol{X} \in \mathbb{R}^n$ that yields f's **extremum**: minimum or maximum, depending on whichever is desirable.

It is important to note however that some functions have no finite extremum. Other functions have multiple. Finding a **global extremum** is generally difficult; however, many good methods exist for finding a **local extremum**: an extremum for some region $R \subset \mathbb{R}^n$.

The method explored here is called **gradient descent**. It will soon become apparent why it has this name.

Stationary points

Recall from basic calculus that a function f of a single variable had potential local extrema where $df(x)/dx = 0$. The multivariate version of this, for multivariate function f, is

$$\operatorname{grad} f = \boldsymbol{0}. \tag{1}$$

A value $\boldsymbol{X}$ for which Eq. 1 holds is called a **stationary point**. However, as in the univariate case, a stationary point may not be a local extremum; in these cases, it called a **saddle point**.

Consider the **hessian matrix** H with values, for independent variables x_i,

$$H_{ij} = \partial^2_{x_i x_j} f. \tag{2}$$

For a stationary point $\boldsymbol{X}$, the **second partial derivative test** tells us if it is a local maximum, local minimum, or saddle point:

minimum If $H(\boldsymbol{X})$ is **positive definite** (all its eigenvalues are positive),

$\mathbf{X}$ is a local minimum.

maximum If $H(\mathbf{X})$ is **negative definite** (all its eigenvalues are negative),
$\mathbf{X}$ is a local maximum.

saddle If $H(\mathbf{X})$ is **indefinite** (it has both positive and negative eigenvalues),
$\mathbf{X}$ is a saddle point.

These are sometimes called tests for concavity: minima occur where f is **convex** and maxima where f is **concave** (i.e. where $-f$ is convex).
It turns out, however, that solving Eq. 1 directly for stationary points is generally hard. Therefore, we will typically use an iterative technique for estimating them.

The gradient points the way

Although Eq. 1 isn't usually directly useful for computing stationary points, it suggests iterative techniques that are. Several techniques rely on the insight that **the gradient points toward stationary points**. Recall from Lec. vecs.grad that $\operatorname{grad} f$ is a vector field that points in the direction of greatest increase in f.
Consider starting at some point $\mathbf{x}_0$ and wanting to move iteratively closer to a stationary point. So, if one is seeking a maximum of f, then choose $\mathbf{x}_1$ to be in the direction of $\operatorname{grad} f$. If one is seeking a minimum of f, then choose $\mathbf{x}_1$ to be opposite the direction of $\operatorname{grad} f$.
The question becomes: *how far* α should we go in (or opposite) the direction of the gradient? Surely too-small α will require more iteration and too-large α will lead to poor convergence or missing minima altogether. This framing of the problem is called **line search**. There are a few common methods for choosing α, called the **step size**, some more computationally efficient than others.
Two methods for choosing the step size are described below. Both are framed as

minimization methods, but changing the sign of the step turns them into maximization methods.

The classical method

Let

$$\mathbf{g}_k = \operatorname{grad} f(\mathbf{x}_k), \tag{3}$$

the gradient at the algorithm's current estimate $\mathbf{x}_k$ of the minimum. The classical method of choosing α is to attempt to solve analytically for

$$\alpha_k = \underset{\alpha}{\operatorname{argmin}}\, f(\mathbf{x}_k - \alpha \mathbf{g}_k). \tag{4}$$

This solution approximates the function f as one varies α. It is approximate because as α varies, so should $\mathbf{x}$. But even with α as the only variable, Eq. 4 may be difficult or impossible to solve. However, this is sometimes called the "optimal" choice for α. Here "optimality" refers not to practicality but to ideality. This method is rarely used to solve practical problems.
The algorithm of the classical gradient descent method can be summarized in the pseudocode of Algorithm grad.1. It is described further in Kreyszig (2011, § 22.1).

Algorithm grad.1 Classical gradient descent

```
1: procedure classical_minimizer($f$,$\mathbf{x}_0$,$T$)
2:     while $\delta\mathbf{x} > T$ do        ▷ until threshold $T$ is met
3:         $\mathbf{g}_k \leftarrow \operatorname{grad} f(\mathbf{x}_k)$
4:         $\alpha_k \leftarrow \operatorname{argmin}_\alpha f(\mathbf{x}_k - \alpha\mathbf{g}_k)$
5:         $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \alpha_k\mathbf{g}_k$
6:         $\delta\mathbf{x} \leftarrow \|\mathbf{x}_{k+1} - \mathbf{x}_k\|$
7:         $k \leftarrow k + 1$
8:     end while
9:     return $\mathbf{x}_k$        ▷ the threshold was reached
10: end procedure
```

The Barzilai and Borwein method

In practice, several non-classical methods are used for choosing step size α. Most of these

construct criteria for step sizes that are too small and too large and prescribe choosing some α that (at least in certain cases) must be in the sweet-spot in between. Barzilai **and** Borwein (1988) developed such a prescription, which we now present.
Let $\Delta\mathbf{x}_k = \mathbf{x}_k - \mathbf{x}_{k-1}$ and $\Delta\mathbf{g}_k = \mathbf{g}_k - \mathbf{g}_{k-1}$. This method minimizes $\|\Delta x - \alpha\Delta g\|^2$ by choosing

$$\alpha_k = \frac{\Delta\mathbf{x}_k \cdot \Delta\mathbf{g}_k}{\Delta\mathbf{g}_k \cdot \Delta\mathbf{g}_k}. \tag{5}$$

The algorithm of this gradient descent method can be summarized in the pseudocode of Algorithm grad.2. It is described further in Barzilai **and** Borwein (**ibidem**).

Algorithm grad.2 Barzilai and Borwein gradient descent

```
procedure barzilai_minimizer($f$,$\mathbf{x}_0$,$T$)
    while $\delta\mathbf{x} > T$ do                    ▷ until threshold $T$ is met
        $\mathbf{g}_k \leftarrow \text{grad}\, f(\mathbf{x}_k)$
        $\Delta\mathbf{g}_k \leftarrow \mathbf{g}_k - \mathbf{g}_{k-1}$
        $\Delta\mathbf{x}_k \leftarrow \mathbf{x}_k - \mathbf{x}_{k-1}$
        $\alpha_k \leftarrow \frac{\Delta\mathbf{x}_k \cdot \Delta\mathbf{g}_k}{\Delta\mathbf{g}_k \cdot \Delta\mathbf{g}_k}$
        $\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k - \alpha_k \mathbf{g}_k$
        $\delta\mathbf{x} \leftarrow \|\mathbf{x}_{k+1} - \mathbf{x}_k\|$
        $k \leftarrow k + 1$
    end while
    return $\mathbf{x}_k$                        ▷ the threshold was reached
end procedure
```

Example opt.grad-1 re: Barzilai and Borwein gradient descent

Consider the functions (a) $f_1 : \mathbb{R}^2 \to \mathbb{R}$ and (b) $f_2 : \mathbb{R}^2 \to \mathbb{R}$ defined as

$$f_1(\mathbf{x}) = (x_1 - 25)^2 + 13(x_2 + 10)^2 \tag{6}$$

$$f_2(\mathbf{x}) = \frac{1}{2}\mathbf{x} \cdot A\mathbf{x} - \mathbf{b} \cdot \mathbf{x} \tag{7}$$

where

$$A = \begin{bmatrix} 20 & 0 \\ 0 & 10 \end{bmatrix} \quad \text{and} \tag{8a}$$

$$b = \begin{bmatrix} 1 & 1 \end{bmatrix}^\top. \tag{8b}$$

Use the method of Barzilai **and** Borwein (1988) starting at some x_0 to find a minimum of each function.

First, load some Python packages.

```
from sympy import *
import numpy as np
import matplotlib.pyplot as plt
from IPython.display import display, Markdown, Latex
from tabulate import tabulate
```

We begin by writing a class `gradient_descent_min` to perform the gradient descent. This is not optimized for speed.

```
class gradient_descent_min():
  """ A Barzilai and Borwein gradient descent class.

  Inputs:
    * f:  Python function of x variables
    * x:  list of symbolic variables (eg [x1, x2])
    * x0: list of numeric initial guess of a min of f
    * T: step size threshold for stopping the descent

  To execute the gradient descent call descend method.

  nb: This is only for gradients in cartesian
      coordinates! Further work would be to implement
      this in multiple or generalized coordinates.
      See the grad method below for implementation.
  """

  def __init__(self,f,x,x0,T):
    self.f = f
    self.x = Array(x)
    self.x0 = np.array(x0)
    self.T = T
    self.n = len(x0) # size of x
    self.g = lambdify(x,self.grad(f,x),'numpy')
    self.xk = np.array(x0)
    self.table = {}

  def descend(self):
    # unpack variables
    f = self.f
    x = self.x
    x0 = self.x0
    T = self.T
    g = self.g
    # initialize variables
    N = 0
    x_k = x0
    dx = 2*T # can't be zero
    x_km1 = .9*x0-.1 # can't equal x0
    g_km1 = np.array(g(*x_km1))
    N_max = 100 # max iterations
    table_data = [[N,x0,np.array(g(*x0)),0]]
    while (dx > T and N < N_max) or N < 1:
      N += 1 # increment index
      g_k = np.array(g(*x_k))
      dg_k = g_k - g_km1
      dx_k = x_k - x_km1
      alpha_k = abs(dx_k.dot(dg_k)/dg_k.dot(dg_k))
      x_km1 = x_k # store
      x_k = x_k - alpha_k*g_k
      # save
      table_data.append([N,x_k,g_k,alpha_k])
      self.xk = np.vstack((self.xk,x_k))
      # store other variables
      g_km1 = g_k
      dx = np.linalg.norm(x_k - x_km1) # check
    self.tabulater(table_data)

  def tabulater(self,table_data):
    np.set_printoptions(precision=2)
    tabulate.LATEX_ESCAPE_RULES={}
    self.table['python'] = tabulate(
      table_data,
      headers=["N","x_k","g_k","alpha"],
    )
    self.table['latex'] = tabulate(
      table_data,
      headers=[
        "$N$","$\\bm{x}_k$","$\\bm{g}_k$","$\\alpha$"
      ],
      tablefmt="latex_raw",
    )

  def grad(self,f,x): # cartesian coord's gradient
    return derive_by_array(f(x),x)
```

First, consider f_1.

```
var('x1 x2')
x = Array([x1,x2])
f1 = lambda x: (x[0]-25)**2 + 13*(x[1]+10)**2
gd = gradient_descent_min(f=f1,x=x,x0=[-50,40],T=1e-8)
```

Perform the gradient descent.

```
gd.descend()
```

Print the interesting variables.

```
print(gd.table['python'])
```

N	x_k	g_k	α
0	[-50 40]	[-150 1300]	0
1	[-43.65 -15.03]	[-150 1300]	0.0423296
2	[-38.36 -10.]	[-137.3 -130.74]	0.0384979
3	[-33.11 -10.]	[-1.27e+02 1.24e-01]	0.041454
4	[24.99 -10.]	[-1.16e+02 -9.62e-03]	0.499926
5	[25. -10.05]	[-0.02 0.12]	0.499999
6	[25. -10.]	[-1.84e-08 -1.38e+00]	0.0385225
7	[25. -10.]	[-1.70e-08 2.19e-03]	0.0384615
8	[25. -10.]	[-1.57e-08 0.00e+00]	0.0384615

Now let's `lambdify` the function `f1` so we can plot.

```
f1_lambda = lambdify((x1,x2),f1(x),'numpy')
```

Now let's plot a contour plot with the gradient descent overlaid.

```
fig, ax = plt.subplots()
# contour plot
X1 = np.linspace(-100,100,100)
X2 = np.linspace(-50,50,100)
X1, X2 = np.meshgrid(X1,X2)
F1 = f1_lambda(X1,X2)
plt.contourf(X1,X2,F1)
plt.colorbar()
# gradient descent plot
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.collections import LineCollection
xX1 = gd.xk[:,0]
xX2 = gd.xk[:,1]
points = np.array([xX1, xX2]).T.reshape(-1, 1, 2)
segments = np.concatenate(
  [points[:-1], points[1:]], axis=1
)
lc = LineCollection(
  segments,
  cmap=plt.get_cmap('Reds')
)
lc.set_array(np.linspace(0,1,len(xX1))) # color segs
lc.set_linewidth(3)
ax.autoscale(False) # avoid the scatter changing lims
ax.add_collection(lc)
ax.scatter(
  xX1,xX2,
  zorder=1,
  marker="o",
  color=plt.cm.Reds(np.linspace(0,1,len(xX1))),
  edgecolor='none'
)
plt.show()
```

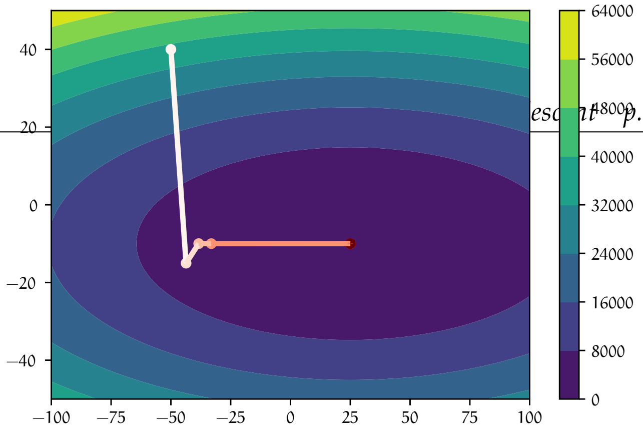

Figure grad.1:

Now consider f_2.

```
A = Matrix([[10,0],[0,20]])
b = Matrix([[1,1]])
def f2(x):
  X = Array([x]).tomatrix().T
  return 1/2*X.dot(A*X) - b.dot(X)
gd = gradient_descent_min(f=f2,x=x,x0=[50,-40],T=1e-8)
```

Perform the gradient descent.

```
gd.descend()
```

Print the interesting variables.

```
print(gd.table['python'])
```

N	x_k	g_k	α
0	[50 -40]	[499. -801.]	0
1	[17.58 12.04]	[499. -801.]	0.0649741
2	[8.07 -1.01]	[174.78 239.88]	0.0544221
3	[3.62 0.17]	[79.66 -21.22]	0.0558582
4	[0.49 -0.05]	[35.16 2.49]	0.0889491
5	[0.1 0.14]	[3.89 -1.94]	0.0990201
6	[0.1 0.]	[0.04 1.9]	0.0750849
7	[0.1 0.05]	[0.01 -0.95]	0.050005
8	[0.1 0.05]	[4.74e-03 9.58e-05]	0.0500012
9	[0.1 0.05]	[2.37e-03 -2.38e-09]	0.0999186
10	[0.1 0.05]	[1.93e-06 2.37e-09]	0.1
11	[0.1 0.05]	[0.00e+00 -2.37e-09]	0.0999997

Now let's `lambdify` the function `f2` so we can plot.

```
f2_lambda = lambdify((x1,x2),f2(x),'numpy')
```

Now let's plot a contour plot with the gradient descent overlaid.

```
fig, ax = plt.subplots()
# contour plot
X1 = np.linspace(-100,100,100)
X2 = np.linspace(-50,50,100)
X1, X2 = np.meshgrid(X1,X2)
F2 = f2_lambda(X1,X2)
plt.contourf(X2,X1,F2)
plt.colorbar()
# gradient descent plot
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.collections import LineCollection
xX1 = gd.xk[:,0]
xX2 = gd.xk[:,1]
points = np.array([xX1, xX2]).T.reshape(-1, 1, 2)
segments = np.concatenate(
  [points[:-1], points[1:]], axis=1
)
lc = LineCollection(
  segments,
  cmap=plt.get_cmap('Reds')
)
lc.set_array(np.linspace(0,1,len(xX1))) # color segs
lc.set_linewidth(3)
ax.autoscale(False) # avoid the scatter changing lims
ax.add_collection(lc)
ax.scatter(
  xX1,xX2,
  zorder=1,
  marker="o",
  color=plt.cm.Reds(np.linspace(0,1,len(xX1))),
  edgecolor='none'
)
plt.show()
```

. Python code in this section was generated from a Jupyter notebook named `gradient_descent.ipynb` with a `python3` kernel.

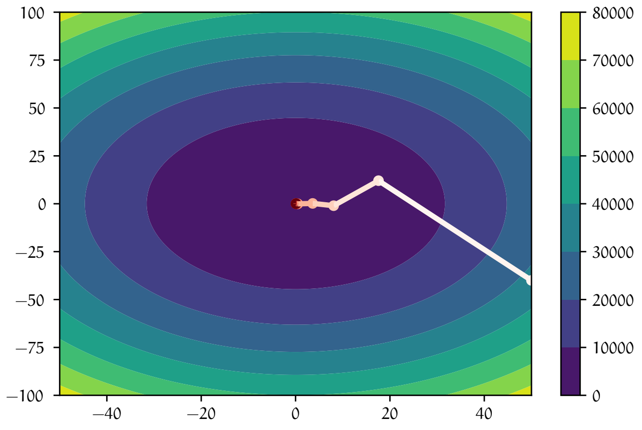

Figure grad.2: