8 Optimization

This chapter concerns optimization mathematics.

8.1 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.

8.1.1 Stationary Points

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

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

A value $\boldsymbol{X}$ for which section 8.1.1 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,

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

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), $\boldsymbol{X}$ is a local minimum.

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

saddle If $H(\boldsymbol{X})$ is **indefinite** (it has both positive and negative eigenvalues), $\boldsymbol{X}$ 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 section 8.1.1 directly for stationary points is generally hard. Therefore, we will typically use an iterative technique for estimating them.

8.1.2 The Gradient Points the Way

Although section 8.1.1 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 section 5.3 that grad f is a vector field that points in the direction of greatest increase in f.

Consider starting at some point $\boldsymbol{x}_0$ and wanting to move iteratively closer to a stationary point. So, if one is seeking a maximum of f, then choose $\boldsymbol{x}_1$ to be in the direction of grad f. If one is seeking a minimum of f, then choose $\boldsymbol{x}_1$ to be opposite the direction of 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.

8.1.3 The Classical Method

Let

$$\boldsymbol{g}_k = \text{grad}\, f(\boldsymbol{x}_k),$$

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

$$\alpha_k = \underset{\alpha}{\text{argmin}}\, f(\boldsymbol{x}_k - \alpha \boldsymbol{g}_k).$$

This solution approximates the function f as one varies α. It is approximate because as α varies, so should $\boldsymbol{x}$. But even with α as the only variable, section 8.1.3 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 1. It is described further in (Kreyszig 2011; § 22.1).

Algorithm 1 Classical gradient descent

```
1: procedure classical_minimizer(f, x_0, T)
2:   while δx > T do                              ▷ until threshold T is met
3:     g_k ← grad f(x_k)
4:     α_k ← argmin_α f(x_k − α g_k)
5:     x_{k+1} ← x_k − α_k g_k
6:     δx ← ||x_{k+1} − x_k||
7:     k ← k + 1
8:   return x_k                                    ▷ the threshold was reached
```

8.1.4 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 $\Delta \boldsymbol{x}_k = \boldsymbol{x}_k - \boldsymbol{x}_{k-1}$ and $\Delta \boldsymbol{g}_k = \boldsymbol{g}_k - \boldsymbol{g}_{k-1}$. This method minimizes $\|\Delta x - \alpha \Delta g\|^2$ by choosing

$$\alpha_k = \frac{\Delta \boldsymbol{x}_k \cdot \Delta \boldsymbol{g}_k}{\Delta \boldsymbol{g}_k \cdot \Delta \boldsymbol{g}_k}.$$

The algorithm of this gradient descent method can be summarized in the pseudocode of algorithm 2. It is described further in (Barzilai and Borwein 1988).

Algorithm 2 Barzilai and Borwein gradient descent

```
1: procedure barzilai_minimizer(f, x_0, T)
2:   while δx > T do                              ▷ until threshold T is met
3:     g_k ← grad f(x_k)
4:     Δg_k ← g_k − g_{k−1}
5:     Δx_k ← x_k − x_{k−1}
6:     α_k ← (Δx_k · Δg_k) / (Δg_k · Δg_k)
7:     x_{k+1} ← x_k − α_k g_k
8:     δx ← ||x_{k+1} − x_k||
9:     k ← k + 1
10:  return x_k                                    ▷ the threshold was reached
```

Example 8.1

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

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

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

where

$$A = \begin{bmatrix} 20 & 0 \\ 0 & 10 \end{bmatrix} \quad \text{and} \tag{8.1}$$

$$b = \begin{bmatrix} 1 & 1 \end{bmatrix}^\top . \tag{8.2}$$

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.

```
import sympy as sp
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
pd.set_option("display.precision", 3)  # Show only three decimal places
```

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 = sp.Array(x)
```

```
    self.x0 = np.array(x0)
    self.T = T
    self.n = len(x0) # size of x
    self.g = sp.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
      t_list = [N,x_k,g_k,alpha_k]
      t_list = [
        [f"{t_i:.3g}" for t_i in t] if isinstance(t,np.ndarray) \
        else t for t in t_list]
      table_data.append(t_list)
      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):
    table = pd.DataFrame(table_data,columns=['N','x_k','g_k','alpha_k'])
    self.table['python'] = table
    self.table['latex'] = table.to_latex(index=False)
```

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

First, consider f_1.

```
x1, x2 = sp.symbols('x1, x2')
x = sp.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  alpha_k
0  0     [-50, 40]         [-150, 1300]    0.000
1  1  [-43.7, -15]      [-150, 1.3e+03]    0.042
2  2  [-38.4, -10]         [-137, -131]    0.038
3  3  [-33.1, -10]       [-127, 0.124]    0.041
4  4     [25, -10]     [-116, -0.00962]    0.500
5  5   [25, -10.1]     [-0.0172, 0.115]    0.500
6  6     [25, -10]   [-1.84e-08, -1.38]    0.039
7  7     [25, -10]  [-1.7e-08, 0.00219]    0.038
8  8     [25, -10]       [-1.57e-08, 0]    0.038
```

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

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

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

```python
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.draw()
```

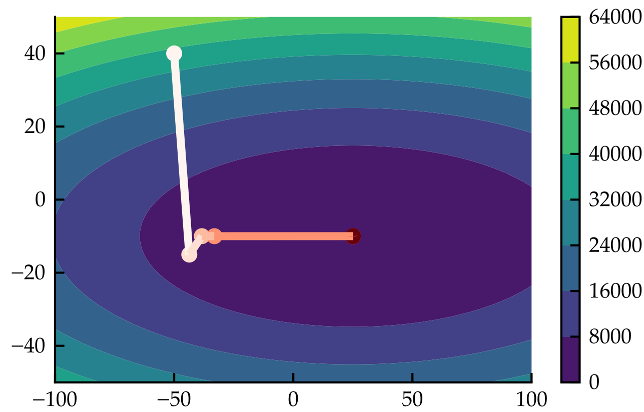

Figure 8.1. Gradient descent on f_1.

Now consider f_2.

```
A = sp.Matrix([[10, 0], [0, 20]])
b = sp.Matrix([[1, 1]])
def f2(x):
  X = sp.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  alpha_k
0    0         [50, -40]       [499.0, -801.0]    0.000
1    1        [17.6, 12]           [499, -801]    0.065
2    2     [8.07, -1.01]            [175, 240]    0.054
3    3     [3.62, 0.174]         [79.7, -21.2]    0.056
4    4  [0.489, -0.0468]          [35.2, 2.49]    0.089
5    5    [0.104, 0.145]         [3.89, -1.94]    0.099
6    6  [0.101, 0.00238]         [0.0381, 1.9]    0.075
7    7       [0.1, 0.05]     [0.00949, -0.952]    0.050
8    8       [0.1, 0.05]   [0.00474, 9.58e-05]    0.050
9    9       [0.1, 0.05]  [0.00237, -2.38e-09]    0.100
10  10       [0.1, 0.05]  [1.93e-06, 2.37e-09]    0.100
11  11       [0.1, 0.05]        [0, -2.37e-09]    0.100
```

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

```
f2_lambda = sp.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()
```

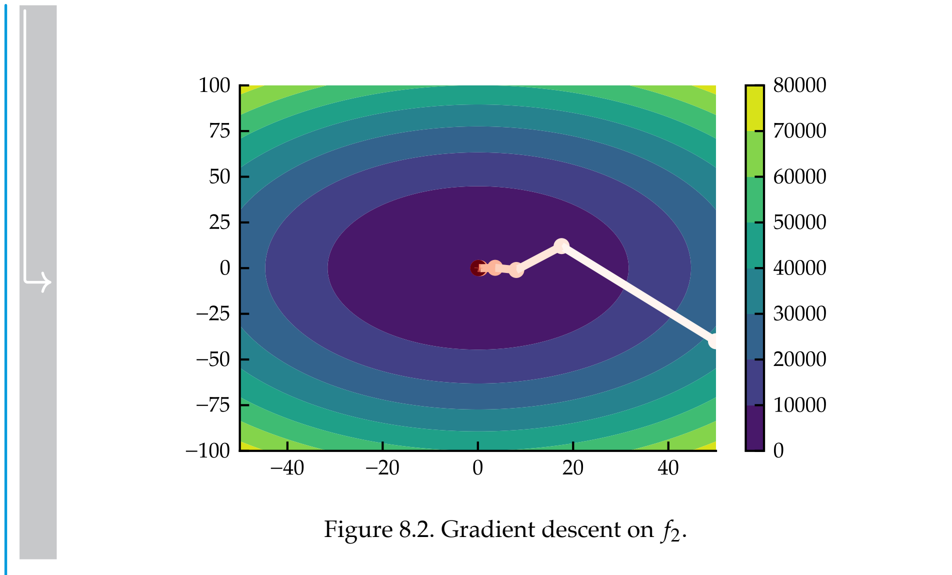

Figure 8.2. Gradient descent on f_2.

8.2 Constrained Linear Optimization

Consider a linear objective function $f : \mathbb{R}^n \to \mathbb{R}$ with variables x_i in vector $\boldsymbol{x}$ and coefficients c_i in vector $\boldsymbol{c}$:

$$f(\boldsymbol{x}) = \boldsymbol{c} \cdot \boldsymbol{x}$$

subject to the linear **constraints**—restrictions on x_i—

$$A\boldsymbol{x} \leq \boldsymbol{a}, \tag{8.3}$$

$$B\boldsymbol{x} = \boldsymbol{b}, \text{ and} \tag{8.4}$$

$$\boldsymbol{l} \leq \boldsymbol{x} \leq \boldsymbol{u} \tag{8.5}$$

where A and B are constant matrices and $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{l}, \boldsymbol{u}$ are n-vectors. This is one formulation of what is called a **linear programming problem**. Usually we want to **maximize** f over the constraints. Such problems frequently arise throughout engineering, for instance in manufacturing, transportation, operations, etc. They are called **constrained** because there are constraints on $\boldsymbol{x}$; they are called **linear** because the objective function and the constraints are linear.

We call a pair $(\boldsymbol{x}, f(\boldsymbol{x}))$ for which $\boldsymbol{x}$ satisfies equation (8.3) a **feasible solution**. Of course, not every feasible solution is **optimal**: a feasible solution is optimal iff there exists no other feasible solution for which f is greater (assuming we're maximizing). We call the vector subspace of feasible solutions $S \subset \mathbb{R}^n$.