Optimization with Scipy (2)

Unconstrained Optimization Cont'd & 1D optimization

Harry Lee

February 5, 2018

CEE 696

Table of contents

Unconstrained Optimization

Minimization routine inputs

For a general (black-box) optimization program, what inputs do you need?

- objective function
- constraint functions
- optimization method/solver
- additional parameters:
  - solution accuracy (numerical precision)
  - maximum number of function evaluations
  - maximum number of iterations

Optimization problem

Find values of the variable $\mathbf{x}$ to give minimum of an objective function $f(x)$ subject to any constraints (restrictions) $g(x), h(x)$

$$\begin{aligned} \min_{x} \quad & f(\mathbf{x}) \\ \text{subject to} \quad & g_i(\mathbf{x}) \geq 0,\ i = 1, \dots, m \\ & h_j(\mathbf{x}) = 0,\ i = 1, \dots, p \\ & \mathbf{x} \in \mathbf{X} \end{aligned}$$

Assume $\mathbf{X}$ be a subset of $\mathbb{R}^n$

$\mathbf{x}$: $n \times 1$ vector of decision variables, i.e., $\mathbf{x} = [x_1, x_2, \cdots, x_n]$

$f(\mathbf{x})$: objective function, $\mathbb{R}^n \rightarrow \mathbb{R}$

$g(\mathbf{x})$: m inequality constraints $\mathbb{R}^n \rightarrow \mathbb{R}$

$h(\mathbf{x})$: p equality constraints $\mathbb{R}^n \rightarrow \mathbb{R}$

Unconstrained optimization (1)

Find values of the variable x to give minimum of an objective function $f(x)$

$$\min_{x} \quad f(\mathsf{x})$$
$$\mathsf{x} \in \mathsf{X}$$

Assume X be a subset of $\mathbb{R}^n$

x : $n \times 1$ vector of decision variables, i.e., $\mathsf{x} = [x_1, x_2, \cdots, x_n]$

$f(\mathsf{x})$: objective function, $\mathbb{R}^n \to \mathbb{R}$

Unconstrained optimization (2)

For now, we will assume

- continuous, smooth objective function f(x): $\mathbb{R}^n \to \mathbb{R}$
- smoothness, by which we mean that the second derivatives $f''(x)$ $(\nabla^2 f)$ exist and are continuous.
- Constrained optimization problems can be reformulated as unconstrained Optimization
- In specific, the constraints of an optimization problem becomes penalized terms in the objective function and we can solve the problem as an unconstrained problem.

Solution to optimization problem : global vs local optimum

A point x^* is a global minimizer if $f(x^*) \leq f(x)$ for all x

A point x^* is a local minimizer if there is a neighborhood $\mathbb{N}$ of x^* such that $f(x^*) \leq f(x)$ for all $x \in \mathbb{N}$

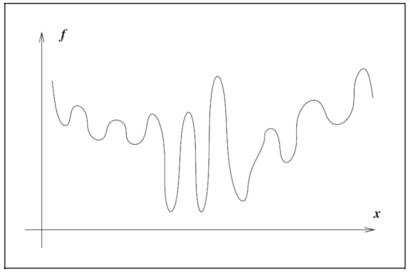


Figure 1: Fig 2.2 from Nocedal & Wright [2006]

Gradient

The vector of first-order derivatives of f is the gradient

$$\nabla f(x) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \cdots, \frac{\partial f}{\partial x_n}\right)^T$$

where

$$\frac{\partial f}{\partial x_i} = \lim_{\epsilon \to 0} \frac{f(x + \epsilon e_i) - f(x)}{\epsilon}$$

and e_i is the $n \times 1$ vector consisting all zeros, except for a 1 in position i

ex) $f(x) = x_1^2 + x_2^2$

$$\nabla f(x) = (2x_1, 2x_2)^T$$

Hessian

The matrix of second-order derivatives of f is the Hessian

$$\nabla^2 f(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

ex) $f = x_1^2 + x_2^2$

$$\nabla^2 f = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$$

Before we start

always add this!

```
import scipy.optimize as opt
import numpy as np
import matplotlib.pyplot as plt
```

Objective function

We will use this 2D example problem:

```
def func(x):
    """2D function x^2 + y^2"""
    return x[0]**2 + x[1]**2

x = np.linspace(-5, 5, 50)
y = np.linspace(-5, 5, 50)

X,Y = np.meshgrid(x,y)  # meshgrid
XY = np.vstack([X.ravel(), Y.ravel()]) # 2D to 1D vertic
Z = func(XY).reshape(50,50) # back to 2D

plt.contour(X, Y, Z) # plot contour
plt.text(0, 0, 'x', va='center', ha='center',
         color='red', fontsize=20)
plt.gca().set_aspect('equal', adjustable='box')#equal ax
plt.show()
```

Gradient and Hessian Information

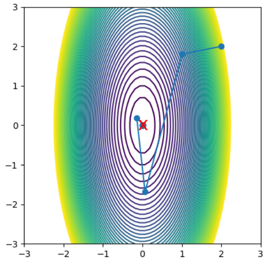
```python
def func_grad(x):
    '''derivative of x^2 + y^2'''
    grad = np.zeros_like(x)
    grad[0] = 2*x[0]
    grad[1] = 2*x[1]
    return grad

def func_hess(x):
    '''hessian of x^2 + y^2 '''
    n = np.size(x) # we assume this a n x 1 or 1 x n vec
    hess = np.zeros((n,n),'d')
    hess[0,0] = 2.
    hess[1,0] = 0.
    hess[0,1] = 0.
    hess[1,1] = 2.
    return hess
```

```
def reporter(x):
    """Capture intermediate states of optimization"""
    global xs
    xs.append(x)

x0 = np.array([2.,2.])
xs = [x0]
opt.minimize(func,x0,jac=func_grad,callback=reporter)
#opt.minimize(func,x0,jac=func_grad,hess=func_hess,
#             callback=reporter)

xs = np.array(xs)
plt.figure()
plt.contour(X, Y, Z, np.linspace(0,25,50))
plt.text(0, 0, 'x', va='center', ha='center',
         color='red', fontsize=20)
plt.plot(xs[:, 0], xs[:, 1], '-o')
plt.gca().set_aspect('equal', adjustable='box')
plt.show()
```

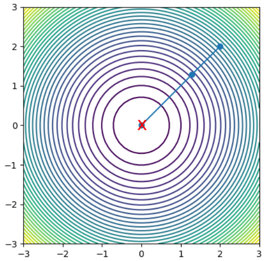
How about $f = 5x^2 + y^2$?

```
def func(x):
    """2D function 5*x^2 + y^2"""
    return 5*x[0]**2 + x[1]**2

def func_grad(x):
    '''derivative of 5*x^2 + y^2'''
    grad[0] = 10*x[0]

def func_hess(x):
    '''hessian of 5*x^2 + y^2 '''
    hess[0,0] = 10.
```

Results



(a) $x^2 + y^2$



(b) $5x^2 + y^2$

Unconstrained optimization

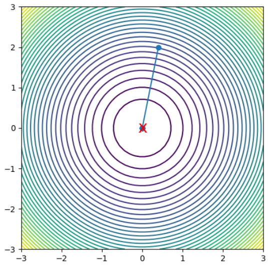
So, general procedure is:

1. Choose an initial guess/starting point x_0
2. Beginning at x_0, find a sequence of iterates x_k $(x_1, x_2, x_3, \cdots, x_\infty)$ with non-increasing function $(f(x_k))$ value until a solution point with sufficient accuracy is found or until no further progress can be made.

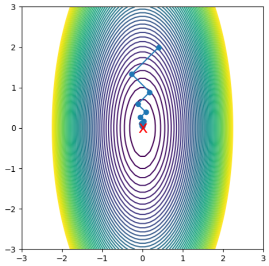
To find the next intermediate solution x_{k+1}, the algorithm routines use information about the function at x_k (and possibly earlier iterates).

Steepest gradient

What if we use gradient information alone? Initial guess is (0.4,2.0).



(c) $x^2 + y^2$



(d) $5x^2 + y^2$

We will discuss this more once we finish 1D optimization

1D Optimization

1D optimization (1)

Why is 1D optimization important?

- Easy to understand with straightforward visualization
- Many optimization algorithms are based on the optimization of a 1D continuous function.
- In fact, many multi-dimensional problems are solved or accelerated by sequentially solving 1D problems.
- ex) Line search

1D optimization (2)

- (Like multi-dimensional problems) no universal algorithm rather a collection of algorithms
- Each algorithm tailored to a particular type of optimization problem
- Generally, more information on objective function means better optimization (with increased number of function evaluation)
- Then, we have to make sure the solution we found is really optimal one: check optimality conditions
- If not satisfied, function will give information

1D optimization - Optimality conditions

$$\min f(x), x \in \mathbb{R}$$

If f is twice-continuously differentiable, and a local minimum of f exists at a point x^*, two conditions hold at x^*:

Necessary conditions	Sufficient conditions
1) $f'(x^*) = 0$	1) $f'(x^*) = 0$
2) $f''(x^*) \geq 0$	2) $f''(x^*) > 0$

Thus finding a minimum becomes “finding a zero” of $f'(x)$

How to find zero?

Numerically, one never finds exactly a zero, but can identify only a small interval such that

$$f(a)f(b) < 0, |a - b| < \mathrm{tol}$$

zero will be within [a,b], thus we say that the zero is “bracketed”.

Note that $f(a)f(b) < 0$ is a sufficient condition for bracketing a zero, not a necessary one.

From this, we can construct zero-finding algorithms:

What is the simplest, straightforward zero-finding approach?

Before we start

Always add these lines for the class!

```
import scipy.optimize as opt
import numpy as np
import matplotlib.pyplot as plt
```

Finding zero - (1) Bisection Method

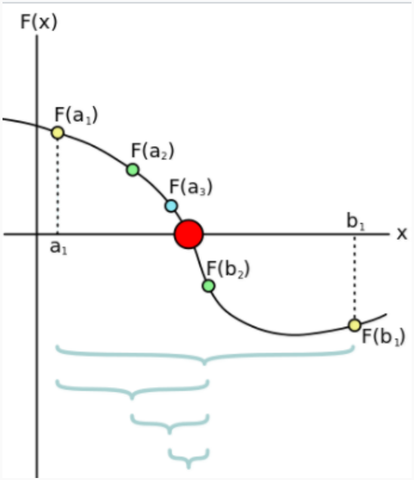
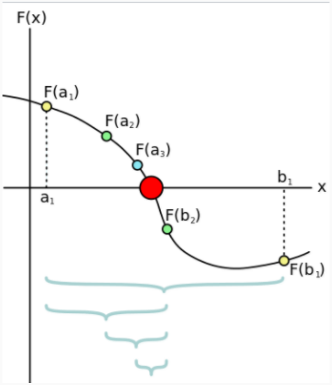


Figure 2: bisection/interval halving from wikipedia

Finding zero - (1) Bisection Method



1. input objective function *f*, endpoints x_{left}, x_{right}, tolerance *tol*, maximum iteration *maxiter*
2. cut interval in half each time
3. compute $x_{mid} = \frac{1}{2}\left(x_{left} + x_{right}\right)$
4. Based on the sign of $f(x_{mid})$, replace x_{left} or x_{right} with x_{mid}
5. stop $x_{right} - x_{left}$ is sufficiently small

Finding zero - (1) Bisection method

scipy.optimize.bisect(f, a, b, args=(), xtol=1e-12, rtol=4.440892098500626e-16, maxiter=100, full_output=False, disp=True)

output:

x0 : (float) zero of f

res : (RootResults) object for detailed output (only if full_output = True)

```python
def func(x):
    return np.cos(x)

# a root of cos is x=np.pi/2
x0, res = opt.bisect(func, 1, 2, full_output = True)
print("exact value = %f" % (np.pi/2.))
```

Finding zero - (1) Bisection method

HW 2.1 write your own bisection method!

```
    INPUT: function $f$, endpoints $a$, $b$, tol, maxiter
Require: $a < b$
    $i \leftarrow 0$, $x_{left} \leftarrow a$, $x_{right} \leftarrow b$
Ensure: $f(x_{left})f(x_{right}) < 0$

    while $i \leq maxiter$ do
        $x_{mid} \leftarrow 0.5(x_{left} + x_{right})$
        if $f(x_{mid}) = 0$ or $x_{right} - x_{left} < tol$ then
            return $x_{mid}$
        $i \leftarrow i + 1$
        if $f(x_{left})f(x_{mid}) > 0$ then
            $x_{left} = x_{mid}$
        else
            $x_{right} = x_{mid}$
```

Rate of Convergence

The bisection method only uses the sign values instead of the function values, so it tends to converge slowly.

One of the key measures of performance of an optimization algorithm is its rate of convergence:

$$\frac{\|x_{k+1} - x^*\|}{\|x_k - x^*\|^p} \leq M \text{ for all } k \text{ sufficiently large}$$

where M is a constant and p indicates rates of convergence. for p = 1, it has linear convergence, for 1< p < 2, superlinear convergence, for p = 2, quadratic convergence

Finding zero - (2) Secant method

We can improve the rate of convergence by using additional information - approximate the objective function f(x) with linear interpolation:

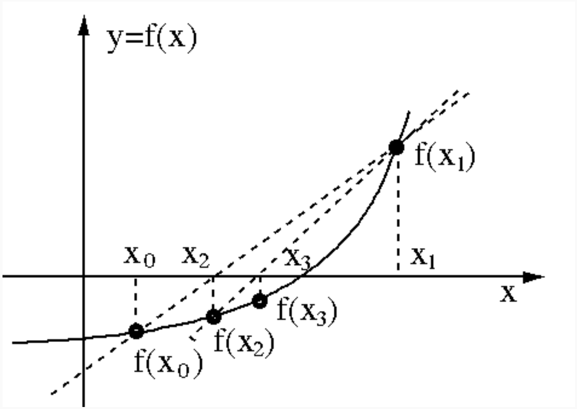


Figure 3

Finding zero - (3) Brent method

Combine bisection and secant methods.

The method approximates the function f(x) with parabolic interpolation.

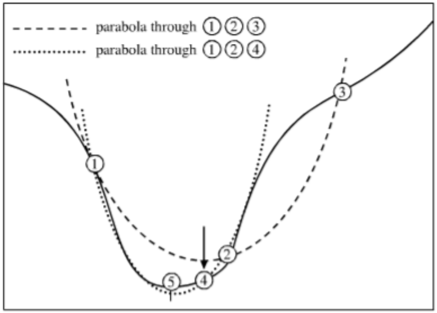


Figure 10.3.1. Convergence to a minimum by inverse parabolic interpolation. A parabola (dashed line) is drawn through the three original points 1,2,3 on the given function (solid line). The function is evaluated at the parabola's minimum, 4, which replaces point 3. A new parabola (dotted line) is drawn through points 1,4,2. The minimum of this parabola is at 5, which is close to the minimum of the function.

Finding zero - (3) Brent method

scipy.optimize.brent(func, args=(), brack=None, tol=1.48e-08, full_output=0, maxiter=500)

```
def f(x):
  return x**2

xmin, fval, iter, nfuncalls
     = opt.brent(f,brack=(-5,0,5),full_output=True)
```

Finding zero - (4) Newton-Raphson Method

If derivative of f (this is actually second order derivative of the original function!) is available, we can do linear interpolation:

$$f(x_{k+1}) \approx f(x_k) + f'(x_k)(x_{k+1} - x_k)$$

if we rearrange the equation above with $f(x_{k+1}) = 0$,

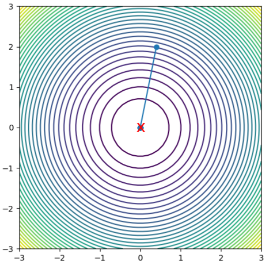
$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

Do this until it converges.

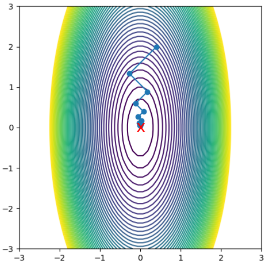
See the movie below
`https://en.wikipedia.org/wiki/Newton%27s_method#/media/File:NewtonIteration_Ani.gif`

Multi-dimensional unconstrained optimization

Steepest gradient (1)



(a) $x^2 + y^2$



(b) $5x^2 + y^2$

1. input objective function f and starting point x_0
2. at iteration k, compute search direction $p_k = -\nabla f$ at x_k
3. set $\alpha_k = 1$ or perform line search: $\min_{\alpha_k} f(x_k + \alpha_k p_k)$
4. update $x_{k+1} = x_k + \alpha_k p_k$
5. repeat 2-4 until it converges

Steepest gradient (2) - HW2.2

HW 2.2 write your own steepest gradient method!

INPUT: function f, gradient ∇f, initial point x_0, tolerance *tol*, maximum iteration *maxiter*, boolean *linesearch*

```
while $k \leq maxiter$ and $\|x_k - x_{k-1}\| \geq tol$ do
    $g \leftarrow \nabla f_{x_k}$
    $p \leftarrow -g_k$
    if linesearch then
        $\alpha_k \leftarrow \underset{\alpha}{argmin}\, f(x_k + \alpha p_k)$
        $x_{k+1} \leftarrow x_k + \alpha_k p_k$
    else
        $x_{k+1} \leftarrow x_k + p_k$
    $k \leftarrow k + 1$
```

Newton's method (1)

1. The steepest descent method uses only first derivatives
2. Newton-Raphson method or so-called Newton's method uses both first and second derivatives and performs better!

Recall from 1D Newton-Raphson method (for minimization instead of zero-finding):

$$f(x_{k+1}) \approx f(x_k) + f^{'}(x_k)(x_{k+1} - x_k) + \frac{1}{2}f^{''}(x_k)(x_{k+1} - x_k)^2$$

if we rearrange the equation above with $f(x_{k+1}) = 0$,

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)}$$

Newton’s method (2)

Like 1D, Newton’s method forms a quadratic model of the objective function around the current iterate

$$f(x) \approx f(x_k) + (x - x_k)^T \mathbf{g}_k + \frac{1}{2}(x - x_k)^T \mathsf{H}_k (x - x_k)$$

where we use $\mathbf{g}_k = \nabla f(x_k)$, $\mathsf{H}_k = \nabla f^2(x_k)$ for simplicity.

Then we can find the local optimum by $\frac{\partial f}{\partial x} = 0$:

$$0 = \mathbf{g}_k + \mathsf{H}_k(x - x_k)$$

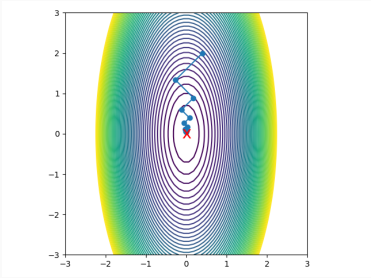
The next iterate is then

$$x_{k+1} = x_k - \mathsf{H}_k^{-1}\mathbf{g}_k$$

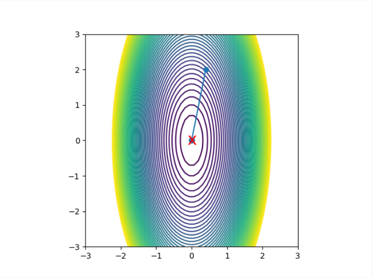
Newton's method (3)

$$\begin{aligned} x_{k+1} &= x_k - \mathbf{H_k}^{-1}\mathbf{g_k} \\ x_{k+1} &= x_k - \alpha_k \mathbf{H_k}^{-1}\mathbf{g_k} \end{aligned}$$

This is called the Newton step.



(c) $5^2 + y^2$ with steepest gradient



(d) $5x^2 + y^2$ with Newton's method

Newton's method (4)

HW 2.3 write your own Newton's method:

```
INPUT: function $f$, gradient $\nabla f$, Hessian $\nabla^2 f$, initial point $x_0$,
tolerance $tol$, maximum iteration $maxiter$, boolean $linesearch$

while $k \leq maxiter$ and $\|x_k - x_{k-1}\| \geq tol$ do
    $g \leftarrow \nabla f_{x_k}$, $H \leftarrow \nabla f^2_{x_k}$
    $p_k \leftarrow -H^{-1} g$
    if $linesearch$ then
        $\alpha_k \leftarrow \underset{\alpha}{argmin}\, f(x_k + \alpha p_k)$
        $x_{k+1} \leftarrow x_k + \alpha_k p_k$
    else
        $x_{k+1} \leftarrow x_k + p_k$
    $k \leftarrow k + 1$
```

For $\mathbf{H}^{-1}\mathbf{g}$, use numpy.linalg.solve(H,g)

Optimization with MODFLOW

my_first_opt_flopy_optimization.py

Our first application is groundwater supply maximization while keeping minimal head drops.

We reuse our first flopy example with constant head = 0 m at left and right boundaries, assume a pumping well at the center of domain and allow 1 m head drop.

Then our optimization problem is formulated as

$$\boxed{\begin{aligned} \max_{Q} \quad & Q \\ \text{subject to} \quad & h_i(\mathrm{x}) \geq -1, \; i = 1, \ldots, m \end{aligned}}$$

How can we perform this optimization?

Back to Flopy

We need to evaluate head distribution using MODFLOW. See Flopy well modeling slide first:

`https://www2.hawaii.edu/~jonghyun/classes/S18/CEE696/files/09_flopy_example2.pdf`

Reformulation to unconstrained optimization

What we learned so far is unconstrained optimization while our problem is constrained optimization. But we can convert the constrained optimization to unconstrained optimization by adding a penalty term!

$$\begin{aligned} \max_{Q} \quad & Q \\ \text{subject to} \quad & h_i(\mathrm{x}) \geq -1, \; i = 1, \ldots, m \end{aligned}$$

$$\min_{Q} \; -Q + \mathbb{1}_{\{h_i(x) < -1\}} \lambda (h_i(x) + 1)^2$$

where $\mathbb{1}_{condition}$ is an indicator function (1 for condition = true, otherwise 0) and λ is a big number for penalty.

Run this

https://www2.hawaii.edu/~jonghyun/classes/S18/CEE696/files/my_first_opt_with_flopy.py