Optimization

Chris Paciorek

2025-08-13

Table of contents

Overview

References:

- Gentle: *Computational Statistics*
- Lange: *Optimization*
- Monahan: *Numerical Methods of Statistics*
- Givens and Hoeting: *Computational Statistics*
- Materials online from Stanford's EE364a course on convex optimization, including Boyd and Vandenberghe's (online) book Convex Optimization.

Videos (optional):

There are various videos from 2020 in the bCourses Media Gallery that you can use for reference if you want to.

- Video 1. Convergence in optimization
- Video 2. Profiling
- Video 3. Multivariate Newton-Raphson
- Video 4. Descent methods and Newton-like methods
- Video 5. Stochastic gradient descent
- Video 6. Coordinate descent
- Video 7. Nelder-Mead
- Video 8. Optimization in practice
- Video 9. Introduction to optimization under constraints

- Video 10. Optimization under equality constraints
- Video 11. Barrier method for constrained optimization

1. Notation

We'll make use of the first derivative (the gradient) and second derivative (the Hessian) of functions. We'll generally denote univariate and multivariate functions (without distinguishing between them) as $f(x)$ with $x = (x_1, \ldots, x_p)$. The (column) vector of first partial derivatives (the gradient) is $f'(x) = \nabla f(x) = (\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_p})^\top$ and the matrix of second partial derivatives (the Hessian) is

$$f''(x) = \nabla^2 f(x) = H_f(x) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_p} \\ \frac{\partial^2 f}{\partial x_1 \partial x_2} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_p} \\ \vdots & \vdots & \ddots & \\ \frac{\partial^2 f}{\partial x_1 \partial x_p} & \frac{\partial^2 f}{\partial x_2 \partial x_p} & \cdots & \frac{\partial^2 f}{\partial x_p^2} \end{pmatrix}.$$

In considering iterative algorithms, I'll use x_0, $x_1, \ldots, x_t$, x_{t+1} to indicate the sequence of values as we search for the optimum, denoted x^*. x_0 is the starting point, which we must choose (often carefully). If it's unclear at any point whether I mean a value of x in the sequence or a sub-element of the x vector, let me know, but hopefully it will be clear from context most of the time.

I'll try to use x (or if we're talking explicitly about a likelihood, θ) to indicate the argument with respect to which we're optimizing and Y to indicate data involved in a likelihood. I'll try to use z to indicate covariates/regressors so there's no confusion with x.

2. Overview

The basic goal here is to optimize a function numerically when we cannot find the maximum (or minimum) analytically. Some examples:

1. Finding the MLE for a GLM
2. Finding least squares estimates for a nonlinear regression model,

$$Y_i \sim \mathcal{N}(g(z_i; \beta), \sigma^2)$$

   where $g(\cdot)$ is nonlinear and we seek to find the value of $\theta = (\beta, \sigma^2)$ that best fits the data.
3. Maximizing a likelihood under constraints
4. Fitting a machine learning prediction method

Maximum likelihood estimation and variants thereof is a standard situation in which optimization comes up.

We'll focus on **minimization**, since any maximization of f can be treated as minimization of $-f$. The basic setup is to find the *argument*, x, that minimizes $f(x)$:

$$x^* = \arg\min_{x \in D} f(x)$$

where D is the domain. Sometimes $D = \Re^p$ but other times it imposes constraints on x. When there are no constraints, this is unconstrained optimization, where any x for which $f(x)$ is defined is a possible solution. We'll assume that f is continuous as there's little that can be done systematically if we're dealing with a discontinuous function.

In one dimension, minimization is the same as root-finding with the derivative function, since the minimum of a differentiable function can only occur at a point at which the derivative is zero. So with differentiable functions we'll seek to find x^* s.t. $f'(x^*) = \nabla f(x^*) = 0$. To ensure a minimum, we want that for all y in a neighborhood of x^*, $f(y) \geq f(x^*)$, or (for twice differentiable functions) $f''(x^*) \geq 0$.

In more than one dimension, we want that the Hessian evaluated at x^* is positive semi-definite, which tells us that moving in any direction away from x^* would not go downhill.

Different strategies are used depending on whether D is discrete and countable, or continuous, dense and uncountable. We'll concentrate on the continuous case but the discrete case can arise in statistics, such as in doing variable selection.

In general we rely on the fact that we can evaluate f. Often we make use of analytic or numerical derivatives of f as well, or derivatives from packages that provide automatic differentiation (AD).

To some degree, optimization is a solved problem, with good software implementations, so it raises the question of how much to discuss in this class. The basic motivation for going into some of the basic classes of optimization strategies is that the function being optimized changes with each problem and can be tricky to optimize, and I want you to know something about how to choose a good approach when you find yourself with a problem requiring optimization. Finding global, as opposed to local, minima can also be an issue.

Note that I'm not going to cover MCMC (Markov chain Monte Carlo) methods, which are used for approximating integrals and sampling from posterior distributions in a Bayesian context and in a variety of ways for optimization. If you take a Bayesian course you'll cover this in detail, and if you don't do Bayesian work, you probably won't have much need for MCMC, though it comes up in MCEM (Monte Carlo EM) and simulated annealing, among other places.

Goals for the unit

Optimization is a big topic. Here's what I would like you to get out of this:

1. an understanding of line searches (one-dimensional optimization),
2. an understanding of multivariate derivative-based optimization and how line searches are useful within this,
3. an understanding of derivative-free methods,
4. an understanding of the methods used in Python's optimization routines, their strengths and weaknesses, and various tricks for doing better optimization in Python, and
5. a basic idea of what convex optimization is and when you might want to go learn more about it.

3. Univariate function optimization

We'll start with some strategies for univariate functions. These can be useful later on in dealing with multivariate functions.

Golden section search

This strategy requires only that the function be unimodal.

Assume we have a single minimum, in $[a, b]$. We choose two points in the interval and evaluate them, $f(x_1)$ and $f(x_2)$. If $f(x_1) < f(x_2)$ then the minimum must be in $[a, x_2]$, and if the converse in $[x_1, b]$. We proceed by choosing a new point in the new, smaller interval and iterate. At each step we reduce the length of the interval in which the minimum must lie. The primary question involves what is an efficient rule to use to choose the new point at each iteration.

Suppose we start with x_1 and x_2 s.t. they divide $[a, b]$ into three equal segments. Then we use $f(x_1)$ and $f(x_2)$ to rule out either the leftmost or rightmost segment based on whether $f(x_1) < f(x_2)$. If we have divided equally, we cannot place the next point very efficiently because either x_1 or x_2 equally divides the remaining space, so we are forced to divide the remaining space into relative lengths of 0.25, 0.25, and 0.5. The next time around, we may only rule out the shorter segment, which leads to inefficiency.

The efficient strategy is to maintain the *golden ratio* between the distances between the points using $\phi = (\sqrt{5} - 1)/2 \approx .618$ (the golden ratio), which is determined by solving for ϕ in this equation: $\phi - \phi^2 = 2\phi - 1$. We start with $x_1 = a + (1 - \phi)(b - a)$ and $x_2 = a + \phi(b - a)$. Then suppose $f(x_1) < f(x_2)$ so the minimum must be in $[a, x_2]$. Since $x_1 - a > x_2 - x_1$, we now choose x_3 in the interval $[a, x_1]$ to produce three subintervals, $[a, x_3]$, $[x_3, x_1]$, $[x_1, x_2]$. We choose to place x_3 s.t. it uses the golden ratio in the interval $[a, x_1]$, namely $x_3 = a + (1 - \phi)(x_2 - a)$. This means that the length of the first subinterval is $(\phi - \phi^2)(b - a)$ and the length of the third subinterval is $(2\phi - 1)(b - a)$, but those lengths are equal because we found ϕ to satisfy $\phi - \phi^2 = 2\phi - 1$.

The careful choice of ϕ allows us to narrow the search interval by an equal proportion,$1 - \phi$, in each iteration. Eventually we have narrowed the minimum to between x_{t-1} and x_t, where the difference $|x_t - x_{t-1}|$ is sufficiently small (within some tolerance - see Section 4 for details), and we report $(x_t + x_{t-1})/2$.

Bisection method

The bisection method requires the existence of the first derivative but has the advantage over the golden section search of halving the interval at each step. We again assume unimodality.

We start with an initial interval (a_0, b_0) and proceed to shrink the interval. Let's choose a_0 and b_0, and set x_0 to be the mean of these endpoints. Now we update according to the following algorithm, assuming our current interval is $[a_t, b_t]$.

- If $f'(a_t)f'(x_t) < 0$, then $[a_{t+1}, b_{t+1}] = [a_t, x_t]$
- If $f'(a_t)f'(x_t) > 0$, then $[a_{t+1}, b_{t+1}] = [x_t, b_t]$

and set x_{t+1} to the mean of a_{t+1} and b_{t+1}. The basic idea is that if the derivative at both a_t and x_t is negative, then the minimum must be between x_t and b_t, based on the intermediate value theorem. If the derivatives at a_t and x_t are of different signs, then the minimum must be between a_t and x_t.

Since the bisection method reduces the size of the search space by one-half at each iteration, one can work out that each decimal place of precision requires 3-4 iterations. Obviously bisection is more efficient than the golden section search because we reduce by $0.5 > 0.382 = 1 - \phi$, so we've gained

information by using the derivative. It requires an evaluation of the derivative however, while golden section just requires an evaluation of the original function.

Bisection is an example of a *bracketing* method, in which we trap the minimum within a nested sequence of intervals of decreasing length. These tend to be slow, but if the first derivative is continuous, they are robust and don't require that a second derivative exist.

Newton-Raphson (Newton's method)

Overview

We'll talk about Newton-Raphson (N-R) as an optimization method rather than a root-finding method, but they're just different perspectives on the same algorithm.

For N-R, we need two continuous derivatives that we can evaluate. The benefit is speed, relative to bracketing methods. We again assume the function is unimodal. The minimum must occur at x^* s.t. $f'(x^*) = 0$, provided the second derivative is non-negative at x^*. So we aim to find a zero (a root) of the first derivative function. Assuming that we have an initial value x_0 that is close to x^*, we have the Taylor series approximation

$$f'(x) \approx f'(x_0) + (x - x_0)f''(x_0).$$

Now set $f'(x) = 0$, since that is the condition we desire (the condition that holds when we are at x^*), and solve for x to get

$$x_1 = x_0 - \frac{f'(x_0)}{f''(x_0)},$$

and iterate, giving us updates of the form $x_{t+1} = x_t - \frac{f'(x_t)}{f''(x_t)}$. What are we doing intuitively? Basically we are taking the tangent to $f(x)$ at x_0 and extrapolating along that line to where it crosses the x-axis to find x_1. We then reevaluate $f(x_1)$ and continue to travel along the tangents.

One can prove that if $f'(x)$ is twice continuously differentiable, is convex, and has a root, then N-R converges from any starting point.

Note that we can also interpret the N-R update as finding the analytic minimum of the quadratic Taylor series approximation to $f(x)$.

> ⚠ Convergence of Newton's method
>
> Newton's method converges very quickly (as we'll discuss in Section 4), but if you start too far from the minimum, you can run into serious problems.

Secant method variation on N-R

Suppose we don't want to calculate the second derivative required in the divisor of N-R. We might replace the analytic derivative with a discrete difference approximation based on the secant line joining $(x_t, f'(x_t))$ and $(x_{t-1}, f'(x_{t-1}))$, giving an approximate second derivative:

$$f''(x_t) \approx \frac{f'(x_t) - f'(x_{t-1})}{x_t - x_{t-1}}.$$

For this variant on N-R, we need two starting points, x_0 and x_1.

An alternative to the secant-based approximation is to use a standard discrete approximation of the derivative such as

$$f''(x_t) \approx \frac{f'(x_t + h) - f'(x_t - h)}{2h}.$$

How can Newton's method go wrong?

Let's think about what can go wrong - namely when we could have $f(x_{t+1}) > f(x_t)$? To be concrete (and without loss of generality), let's assume that $f(x_t) > 0$, in other words that $x^* < x_t$.

1. As usual, we can develop some intuition by starting with the worst case that $f''(x_t)$ is 0, in which case the method would fail as x_{t+1} would be $-\infty$.
2. Now suppose that $f''(x_t)$ is a small positive number. Basically, if $f'(x_t)$ is relatively flat, we can get that $|x_{t+1} - x^*| > |x_t - x^*|$ because we divide by a small value for the second derivative, causing x_{t+1} to be far from x_t (though it does at least go in the correct direction). We'll see an example on the board and the demo code (see below).
3. Newton's method can also go uphill (going in the wrong direction, away from x^*) when the second derivative is negative, with the method searching for a maximum, since we would have $x_{t+1} > x_t$. Another way to think of this is that Newton's method does not automatically minimize the function, rather it finds local optima.

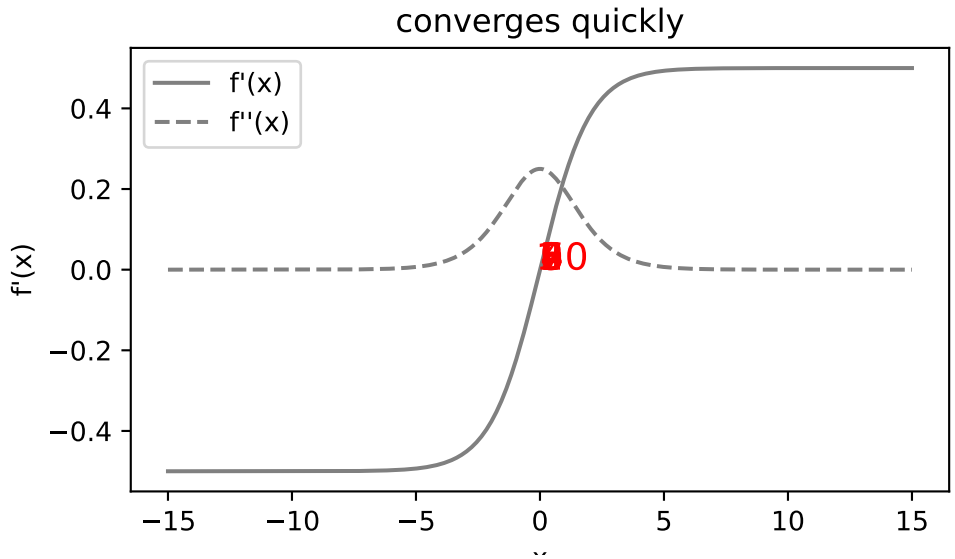
In all these cases Newton's method could diverge, failing to converge on the optimum.

Divergence

First let's see an example of divergence. The first and second plots show two cases of convergence, while the third plot panel shows divergence. In the third plot, the initial second derivative value is small enough that x_2 is further from x^* than x_1 and then x_3 is yet further away. In all cases the sequence of x values is indicated by the red letters.

```
import numpy as np
import matplotlib.pyplot as plt

def f_deriv1(x, theta=1):
    ## First derivative - we want the root of this.
    return np.exp(x * theta) / (1 + np.exp(x * theta)) - 0.5

def f_deriv2(x, theta=1):
    ## Second derivative - used to scale the optimization steps.
    return np.exp(x * theta) / ((1 + np.exp(x * theta)) ** 2)

def make_plot(xs, xvals, f_deriv1, f_deriv2, subplot, title):
    plt.plot(xs, f_deriv1(xs), '-', label="f'(x)", color = 'grey')
    plt.plot(xs, f_deriv2(xs), '--', label="f''(x)", color = 'grey')
    for i in range(len(xvals)):
        plt.text(xvals[i], 0, i, fontsize=14, color = 'red')
    plt.xlabel("x")
```

```
    plt.ylabel("f'(x)")
    plt.title(title)
    plt.legend(loc='upper left')


xs = np.linspace(-15, 15, 300)

n = 10
xvals = np.zeros(n)

## Good starting point
x0 = 1

xvals[0] = x0
for t in range(1,10):
    xvals[t] = xvals[t-1] - f_deriv1(xvals[t-1]) / f_deriv2(xvals[t-1])

make_plot(xs, xvals, f_deriv1, f_deriv2, 1, "converges quickly")
plt.show(block=False)
```

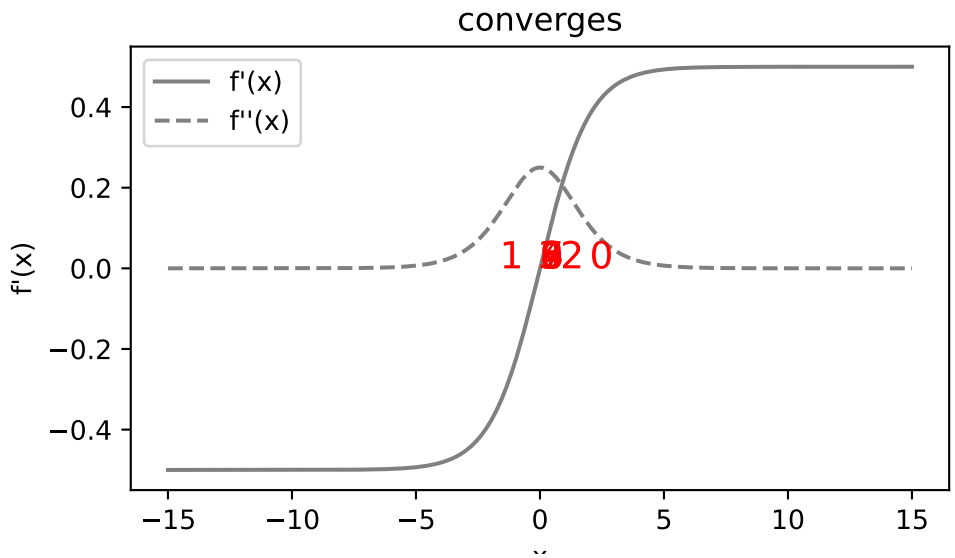



```
print(np.round(xvals,3))
```

```
[ 1.    -0.175  0.001 -0.     0.     0.     0.     0.     0.     0.   ]
```

```
## Ok starting point
x0 = 2

xvals[0] = x0
```

```
for t in range(1,10):
    xvals[t] = xvals[t-1] - f_deriv1(xvals[t-1]) / f_deriv2(xvals[t-1])

make_plot(xs, xvals, f_deriv1, f_deriv2, 2, "converges")
plt.show(block=False)
```



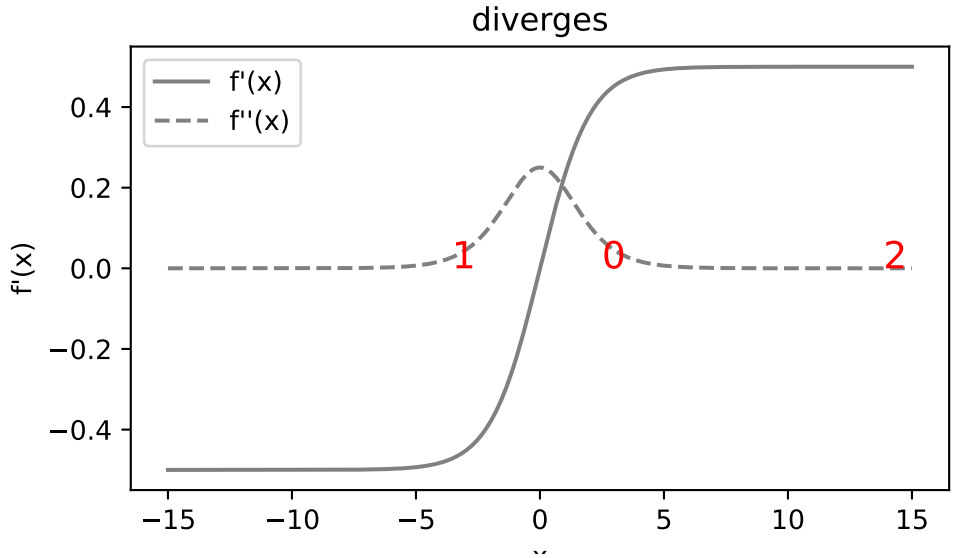

```
print(np.round(xvals,3))
```

```
[ 2.    -1.627  0.819 -0.095  0.    -0.     0.     0.     0.     0.   ]
```

```
## Bad starting point

x0 = 2.5

xvals[0] = x0
for t in range(1,10):
    xvals[t] = xvals[t-1] - f_deriv1(xvals[t-1]) / f_deriv2(xvals[t-1])
```

```
<string>:2: RuntimeWarning: divide by zero encountered in scalar divide
<string>:4: RuntimeWarning: invalid value encountered in scalar divide
```

```
make_plot(xs, xvals[np.abs(xvals) < 15], f_deriv1, f_deriv2, 3, "diverges")
plt.show(block=False)
```




```
## whoops!

print(np.round(xvals,3))
```

```
[ 2.50000000e+00 -3.55000000e+00  1.38460000e+01 -5.15287628e+05
             inf              nan              nan              nan
             nan              nan]
```

In the last case the divergence quickly leads to numerical overflow and then NaNs (resulting from trying to use infinity in calculations).

Multiple optima: converging to the wrong optimum

In the first row of the next figure, let's see an example of climbing uphill and finding a local maximum rather than minimum. The other rows show convergence. In all cases the minimum is at $x^* \approx 3.14$

```
# Define the original function
def f(x):
    return np.cos(x)

# Define the gradient
def f_deriv1(x):
    return -np.sin(x)

# Define the second derivative
def f_deriv2(x):
    return -np.cos(x)# original fxn

def make_plot2(xs, xvals, f, f_deriv1, num, title):
```

```
    # gradient subplot
    plt.subplot(3, 2, num)
    plt.plot(xs, f_deriv1(xs), '-', label="f'(x)")
    plt.scatter(np.pi, f_deriv1(np.pi))
    for i in range(len(xvals)):
        plt.text(xvals[i], 0, i, fontsize=12, color = 'red')
    plt.xlabel('x')
    plt.ylabel("f'(x)")
    plt.title(title[0])
    plt.legend(loc='lower right')
    plt.tight_layout()
    # function subplot
    plt.subplot(3, 2, num+1)
    plt.plot(xs, f(xs), '-', label="f(x)")
    plt.scatter(np.pi, f(np.pi))
    for i in range(len(xvals)):
        plt.text(xvals[i], 0, i, fontsize=12, color = 'red')
    plt.xlabel('x')
    plt.ylabel("f(x)")
    plt.title(title[1])
    plt.legend(loc='lower right')
    plt.tight_layout()


xs = np.linspace(0, 2 * np.pi, num=300)

x0 = 5.5 # starting point

## f_deriv1(x0) # positive
## f_deriv2(x0) # negative
## x1 = x0 - f_deriv1(x0)/f_deriv2(x0) # whoops, we've gone uphill
## because of the negative second derivative
xvals = np.zeros(n)

xvals[0] = x0
for t in range(1,10):
    xvals[t] = xvals[t-1] - f_deriv1(xvals[t-1]) / f_deriv2(xvals[t-1])
## print(xvals)

plt.figure(figsize=(10, 7))

make_plot2(xs, xvals, f, f_deriv1, 1, title =
    ['uphill to local maximum, gradient view', 'uphill to local maximum, function view'])

## In contrast, with better starting points we can find the minimum
## (but this nearly diverges).
```

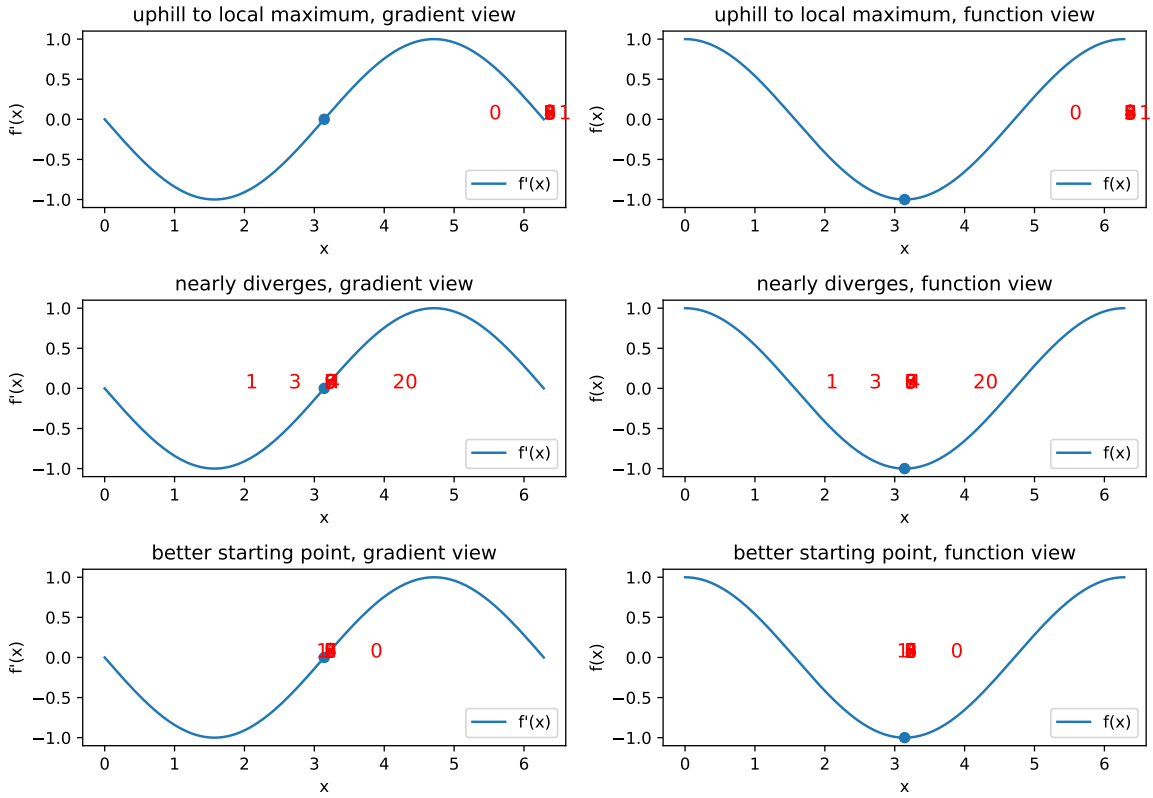
```
x0 = 4.3 # ok starting point
## f_deriv1(x0)
## f_deriv2(x0)
## x1 = x0 - f_deriv1(x0)/f_deriv2(x0)  # going downhill

xvals[0] = x0
for t in range(1,10):
    xvals[t] = xvals[t-1] - f_deriv1(xvals[t-1]) / f_deriv2(xvals[t-1])
## print(xvals)

make_plot2(xs, xvals, f, f_deriv1, 3, title =
    ['nearly diverges, gradient view', 'nearly diverges, function view'])

## With a better starting point, we converge quickly.

x0 = 3.8 # good starting point
f_deriv1(x0)
```

```
np.float64(0.6118578909427189)
```

```
f_deriv2(x0)
```

```
np.float64(0.7909677119144168)
```

```
x1 = x0 - f_deriv1(x0)/f_deriv2(x0)   # going downhill

xvals[0] = x0
for t in range(1,10):
    xvals[t] = xvals[t-1] - f_deriv1(xvals[t-1]) / f_deriv2(xvals[t-1])
## print(xvals)

make_plot2(xs, xvals, f, f_deriv1, 5, title =
    ['better starting point, gradient view', 'better starting point, function view'])

plt.show(block=False)
```



Improving Newton's method

One nice, general idea is to use a fast method such as Newton's method *safeguarded* by a robust, but slower method. Here's how one can do this for N-R, safeguarding with a bracketing method such as bisection. Basically, we check the N-R proposed move to see if N-R is proposing a step outside of where the root is known to lie based on the previous steps and the gradient values for those steps. If so, we could choose the next step based on bisection.

Another approach is backtracking. If a new value is proposed that yields a larger value of the function, backtrack to find a value that reduces the function. One possibility is a line search but given that we're trying to reduce computation, a full line search is often unwise computationally (also in the multivariate Newton's method, we are in the middle of an iterative algorithm for which we will just be going off in another direction anyway at the next iteration). A basic approach is to keep backtracking in halves. A nice alternative is to fit a polynomial to the known information about that slice of the function, namely $f(x_{t+1})$, $f(x_t)$, $f'(x_t)$ and $f''(x_t)$ and find the minimum of the polynomial approximation.

4. Convergence ideas

Convergence metrics

We might choose to assess whether $f'(x_t)$ is near zero, which should assure that we have reached the critical point. However, in parts of the domain where $f(x)$ is fairly flat, we may find the derivative is near zero even though we are far from the optimum. Instead, we generally monitor $|x_{t+1} - x_t|$ (for the moment, assume x is scalar). We might consider absolute convergence: $|x_{t+1} - x_t| < \epsilon$ or relative convergence, $\frac{|x_{t+1} - x_t|}{|x_t|} < \epsilon$. Relative convergence is appealing because it accounts for the scale of x, but it can run into problems when x_t is near zero, in which case one can use $\frac{|x_{t+1} - x_t|}{|x_t| + \epsilon} < \epsilon$. We would want to account for machine precision in thinking about setting ϵ. For relative convergence a reasonable choice of ϵ would be to use the square root of machine epsilon or about 1×10^{-8}.

Problems with the optimization may show up in a convergence measure that fails to decrease or cycles (oscillates). Software generally has a stopping rule that stops the algorithm after a fixed number of iterations; these can generally be changed by the user. When an algorithm stops because of the stopping rule before the convergence criterion is met, we say the algorithm has failed to converge. Sometimes we just need to run it longer, but often it indicates a problem with the function being optimized or with your starting value.

For multivariate optimization, we use a distance metric between x_{t+1} and x_t, such as $\|x_{t+1} - x_t\|_p$, often with $p = 1$ or $p = 2$.

Starting values

Good starting values are important because they can improve the speed of optimization, prevent divergence or cycling, and prevent finding local optima.

Using random or selected multiple starting values can help with multiple optima (aka multimodality).

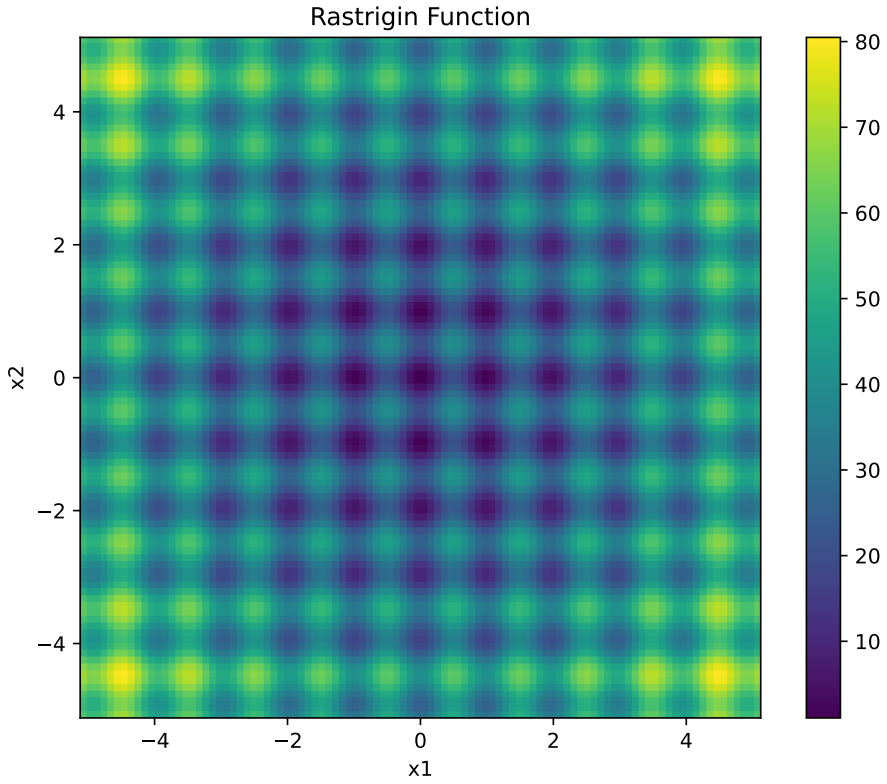
Here's a function (the Rastrigin function) with multiple optima that is commonly used for testing methods that claim to work well for multimodal problems. This is a hard function to optimize with respect to, particularly in higher dimensions (one can do it in higher dimensions than 2 by simply making the x vector longer but having the same structure). In particular Rastrigin with 30 dimensions is considered to be very hard.

```
def rastrigin(x):
    A = 10
    n = len(x)
    return A * n + np.sum(x**2 - A * np.cos(2 * np.pi * x))

const = 5.12
nGrid = 100
gr = np.linspace(-const, const, num=nGrid)

# Create a grid of x values
x1, x2 = np.meshgrid(gr, gr)
xs = np.column_stack((x1.ravel(), x2.ravel()))
```

```
# Calculate the Rastrigin function for each point in the grid
y = np.apply_along_axis(rastrigin, 1, xs)

# Create a plot
plt.figure(figsize=(8, 6))
plt.imshow(y.reshape((nGrid, nGrid)), extent=[-const, const, -const, const], origin='lower', cmap='vi
plt.colorbar()
```

```
<matplotlib.colorbar.Colorbar object at 0x787221185160>
```

```
plt.title('Rastrigin Function')
plt.xlabel('x1')
plt.ylabel('x2')
plt.show()
```



Convergence rates

Let $\epsilon_t = |x_t - x^*|$. If the limit

$$\lim_{t\to\infty} \frac{|\epsilon_{t+1}|}{|\epsilon_t|^\beta} = c$$

exists for $\beta > 0$ and $c \neq 0$, then a method is said to have order of convergence β. This basically measures how big the error at the $t+1$th iteration is relative to that at the tth iteration, with the approximation that $|\epsilon_{t+1}| \approx c|\epsilon_t|^\beta$.

Bisection doesn't formally satisfy the criterion needed to make use of this definition, but roughly speaking it has linear convergence ($\beta = 1$), so the magnitude of the error decreases by a factor of c at each step. Next we'll see that N-R has quadratic convergence ($\beta = 2$), which is fast.

To analyze convergence of N-R, we'll assume that $f'(x)$ is twice continuously differentiable and consider a Taylor expansion of the gradient at the minimum, x^*, around the current value, x_t:

$$f'(x^*) = f'(x_t) + (x^* - x_t)f''(x_t) + \frac{1}{2}(x^* - x_t)^2 f'''(\xi_t) = 0,$$

for some $\xi_t \in [x^*, x_t]$. Making use of the N-R update equation: $x_{t+1} = x_t - \frac{f'(x_t)}{f''(x_t)}$ to substitute , and some algebra, we have

$$\frac{|\epsilon_{t+1}|}{|\epsilon_t|^\beta} = \frac{|x^* - x_{t+1}|}{(x^* - x_t)^2} = \left|\frac{1}{2}\frac{f'''(\xi_t)}{f''(x_t)}\right|.$$

If the limit of the ratio on the right hand side exists (note the assumption of twice continuous differentiability) and is equal to c:

$$c = \lim_{x_t \to x^*} \left|\frac{1}{2}\frac{f'''(\xi_t)}{f''(x_t)}\right| = \left|\frac{1}{2}\frac{f'''(x^*)}{f''(x^*)}\right|$$

then we see that $\beta = 2$.

If c were one, then we see that if we have k digits of accuracy at t, we'd have $2k$ digits at $t+1$ (e.g., $|\epsilon_t| = 0.01$ results in $|\epsilon_{t+1}| = 0.0001$), which justifies the characterization of quadratic convergence being fast. In practice c will moderate the rate of convergence. The smaller c the better, so we'd like to have the second derivative be large and the third derivative be small. The expression also indicates we'll have a problem if $f''(x_t) = 0$ at any point (think about what this corresponds to graphically - what is our next step when $f''(x_t) = 0$?). The characteristics of the derivatives determine the domain of attraction (the region in which we'll converge rather than diverge) of the minimum.

Givens and Hoeting show that using the secant-based approximation to the second derivative in N-R has order of convergence, $\beta \approx 1.62$.

Here's an example of convergence comparing bisection and N-R. First, Newton-Raphson:

```
np.set_printoptions(precision=10)

# Define the original function
def f(x):
    return np.cos(x)
```

```
# Define the gradient
def f_deriv1(x):
    return -np.sin(x)

# Define the second derivative
def f_deriv2(x):
    return -np.cos(x)

xstar = np.pi  # known minimum

## Newton-Raphson (N-R) method
x0 = 2
n_it = 10
xvals = np.zeros(n_it)
xvals[0] = x0
for t in range(1, n_it):
    xvals[t] = xvals[t - 1] - f_deriv1(xvals[t - 1]) / f_deriv2(xvals[t - 1])

print(xvals)
```

```
[2.           4.1850398633 2.4678936745 3.2661862776 3.1409439123
 3.1415926537 3.1415926536 3.1415926536 3.1415926536 3.1415926536]
```

Next, here is bisection:

```
## Bisection method
def bisec_step(interval, f_deriv1):
    interval = interval.copy()
    xt = np.mean(interval)
    if f_deriv1(interval[0]) * f_deriv1(xt) <= 0:
        interval[1] = xt
    else:
        interval[0] = xt
    return interval

n_it = 30
a0 = 2
b0 = (3 * np.pi / 2) - (xstar - a0)
interval = np.zeros((n_it, 2))
interval[0,:] = [a0, b0]

for t in range(1, n_it):
    interval[t,:] = bisec_step(interval[t-1,:], f_deriv1)

print(np.mean(interval, axis=1))
```

```
[2.7853981634 3.1780972451 2.9817477042 3.0799224747 3.1290098599
```

```
3.1535535525 3.1412817062 3.1474176293 3.1443496678 3.142815687
3.1420486966 3.1416652014 3.1414734538 3.1415693276 3.1416172645
3.141593296  3.1415813118 3.1415873039 3.1415903    3.141591798
3.141592547  3.1415929215 3.1415927343 3.1415926406 3.1415926875
3.1415926641 3.1415926524 3.1415926582 3.1415926553 3.1415926538]
```

5. Multivariate optimization

Optimizing as the dimension of the space gets larger becomes increasingly difficult:

1. In high dimensions, there are many possible directions to go.
2. One can end up having to do calculations with large vectors and matrices.
3. Multimodality increasingly becomes a concern (and can be hard to detect).

First we'll discuss the idea of profiling to reduce dimensionality and then we'll talk about various numerical techniques, many of which build off of Newton's method (using second derivative information). We'll finish by talking about methods that only use the gradient (and not the second derivative) and methods that don't use any derivative information.

Profiling

A core technique for likelihood optimization is to analytically maximize over any parameters for which this is possible. Suppose we have two sets of parameters, θ_1 and θ_2, and we can analytically maximize w.r.t θ_2. This will give us $\widehat{\theta}_2(\theta_1)$, a function of the remaining parameters over which analytic maximization is not possible. Plugging in $\widehat{\theta}_2(\theta_1)$ into the objective function (in this case generally the likelihood or log likelihood) gives us the profile (log) likelihood solely in terms of the obstinant parameters. For example, suppose we have the regression likelihood with correlated errors:

$$Y \sim \mathcal{N}(X\beta, \sigma^2\Sigma(\rho)),$$

where $\Sigma(\rho)$ is a correlation matrix that is a function of a parameter, ρ. The maximum w.r.t. β is easily seen to be the GLS estimator $\widehat{\beta}(\rho) = (X^\top\Sigma(\rho)^{-1}X)^{-1}X^\top\Sigma(\rho)^{-1}Y$. (In general such a maximum is a function of all of the other parameters, but conveniently it's only a function of ρ here.) This gives us the initial profile likelihood

$$\frac{1}{(\sigma^2)^{n/2}|\Sigma(\rho)|^{1/2}} \exp\left(-\frac{(Y - X\widehat{\beta}(\rho))^{-\top}\Sigma(\rho)^{-1}(Y - X\widehat{\beta}(\rho))}{2\sigma^2}\right).$$

We then notice that the likelihood is maximized w.r.t. σ^2 at

$$\widehat{\sigma}^2(\rho) = \frac{(Y - X\widehat{\beta}(\rho))^\top\Sigma(\rho)^{-1}(Y - X\widehat{\beta}(\rho))}{n}.$$

This gives us the final profile likelihood,

$$\frac{1}{|\Sigma(\rho)|^{1/2}} \frac{1}{(\widehat{\sigma}^2(\rho))^{n/2}} \exp(-\frac{1}{2}n),$$

a function of ρ only, for which numerical optimization is much simpler.

Newton-Raphson (Newton's method)

For multivariate x we have the Newton-Raphson update $x_{t+1} = x_t - f''(x_t)^{-1} f'(x_t)$, or in our other notation,

$$x_{t+1} = x_t - H_f(x_t)^{-1} \nabla f(x_t).$$

Let's consider a very simple example of nonlinear least squares. We'll use the famous Mauna Loa atmospheric carbon dioxide record.

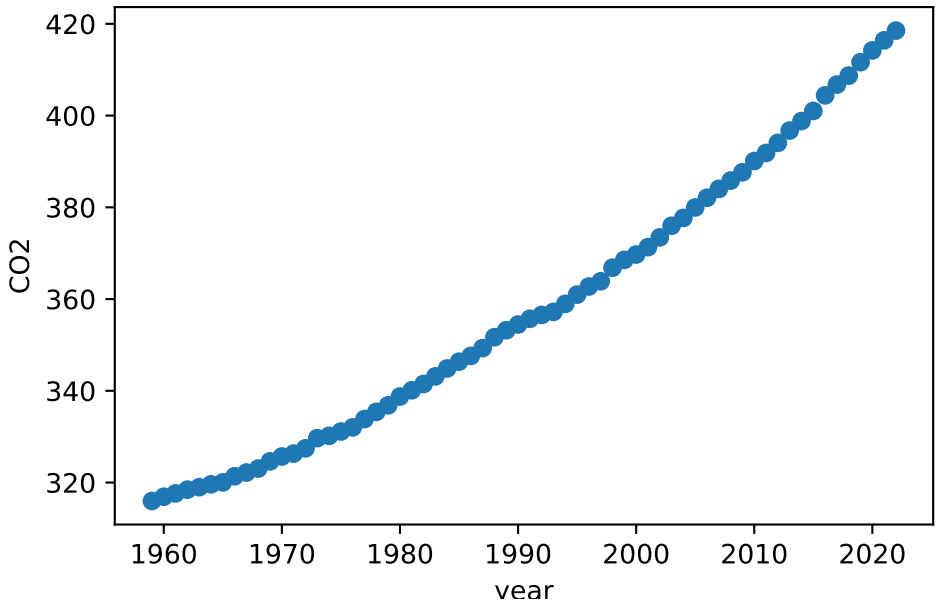
Let's suppose (I have no real reason to think this) that we think that the data can be well-represented by this nonlinear model:

$$Y_i = \beta_0 + \beta_1 \exp(t_i/\beta_2) + \epsilon_i.$$

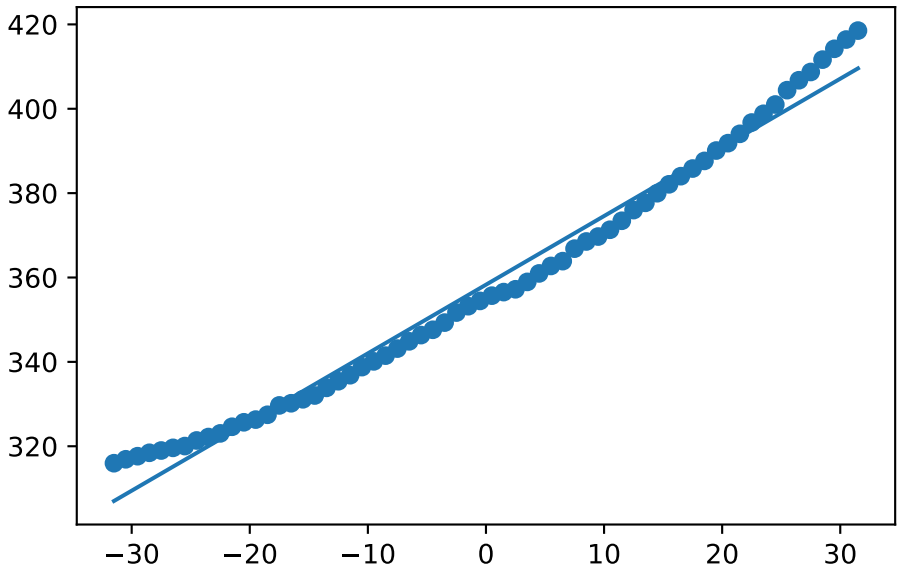
Some of the things we need to worry about with Newton's method in general about are (1) good starting values, (2) positive definiteness of the Hessian, and (3) avoiding errors in deriving the derivatives.

A note on the positive definiteness: since the Hessian may not be positive definite (although it may well be, provided the function is approximately locally quadratic), one can consider modifying the Cholesky decomposition of the Hessian to enforce positive definiteness by adding diagonal elements to H_f as necessary.

```
import os
import pandas as pd
import statsmodels.api as sm
data = pd.read_csv(os.path.join('..','data', 'co2_annmean_mlo.csv'),
    header = 0, names = ['year','co2','unc'])

plt.scatter(data.year, data.co2)
plt.xlabel('year')
plt.ylabel("CO2")
plt.show(block=False)
```

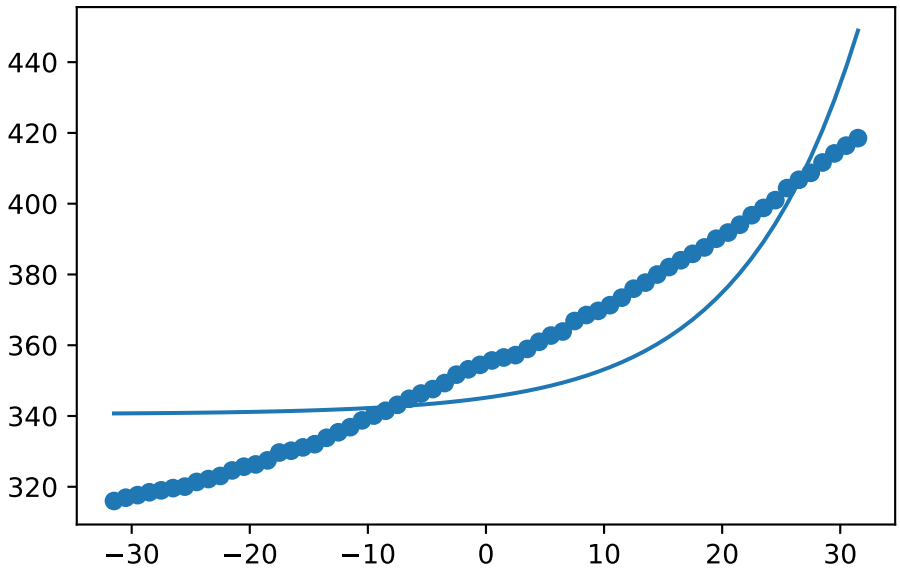



```
## Center years for better numerical behavior
data.year = data.year - np.mean(data.year)
### Linear fit - not a good model
X = sm.add_constant(data.year)
model = sm.OLS(data.co2, X).fit()

plt.scatter(data.year, data.co2)
plt.plot(data.year, model.fittedvalues, '-')

plt.show(block=False)
```



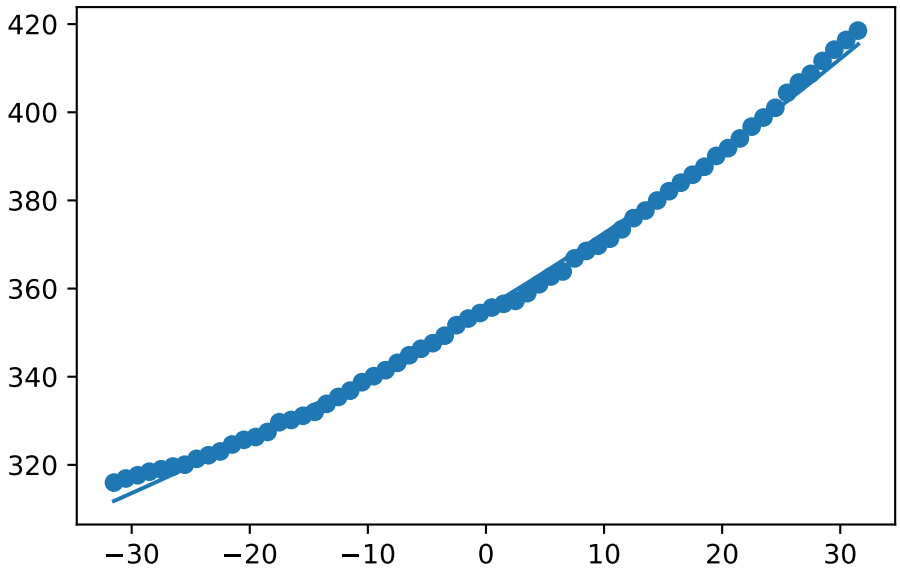
We need some starting values. Having centered the year variable, β_2 seems plausibly like it would be order of magnitude of 10, which is about the magnitude of the year values.

```
beta2_init = 10
implicit_covar = np.exp(data.year/beta2_init)

X = sm.add_constant(implicit_covar)
model = sm.OLS(data.co2, X).fit()
beta0_init, beta1_init = model.params

plt.scatter(data.year, data.co2)

def fit(params):
    return params[0] + params[1] * np.exp(data.year / params[2])

beta = (beta0_init, beta1_init, beta2_init)
plt.plot(data.year, fit(beta), '-')
plt.show(block=False)
```



That's not great. How about changing the scale of beta2 more?

```
beta2_init = 100
implicit_covar = np.exp(data.year/beta2_init)

X = sm.add_constant(implicit_covar)
model = sm.OLS(data.co2, X).fit()
beta0_init, beta1_init = model.params

plt.scatter(data.year, data.co2)
beta = (beta0_init, beta1_init, beta2_init)
plt.plot(data.year, fit(beta), '-')

plt.show(block=False)
```



Let's get derivative information using automatic differentation (the algorithmic implementation of the chain rule for derivatives also used in gradient descent in deep learning, as well as various other contexts). We'll use JAX, but PyTorch or Tensorflow are other options. We need to use the JAX versions of various numpy operations in order to be able to get the derivatives.

```
import jax.numpy as jnp
import jax

def loss(params):
    fitted = params[0] + params[1]*jnp.exp(jnp.array(data.year)/params[2])
    return jnp.sum((fitted - jnp.array(data.co2))**2.0)

deriv1 = jax.grad(loss)
deriv2 = jax.hessian(loss)

deriv1(jnp.array([beta0_init, beta1_init, beta2_init]))
```

```
Array([-9.1552734e-04, -8.9168549e-04,  4.7961845e+00], dtype=float32)
```

```
hess = deriv2(jnp.array([beta0_init, beta1_init, beta2_init]))
hess
```

```
Array([[128.      , 130.1952  ,  -7.139159],
       [130.1952  , 136.91656 , -14.690589],
       [ -7.139159, -14.69059 ,  12.677566]], dtype=float32)
```

```
np.linalg.eig(hess)[0]
```

```
array([ 2.6369040e+02, -2.5785028e-03,  1.3906311e+01], dtype=float32)
```

The Hessian is not positive definite. We could try tricks such as adding to the diagonal of the Hessian or using the pseudo-inverse (i.e., setting all negative eigenvalues in the inverse to zero).

Instead, let's try a bit more to find starting values where the Hessian is positive definite. The order of magnitude of our initial value for β_2 seems about right, so let's try halving or doubling it.

```
beta2_init = 50
implicit_covar = np.exp(data.year/beta2_init)

X = sm.add_constant(implicit_covar)
model = sm.OLS(data.co2, X).fit()
beta0_init, beta1_init = model.params

hess = deriv2(jnp.array([beta0_init, beta1_init, beta2_init]))

np.linalg.eig(hess)[0]
```

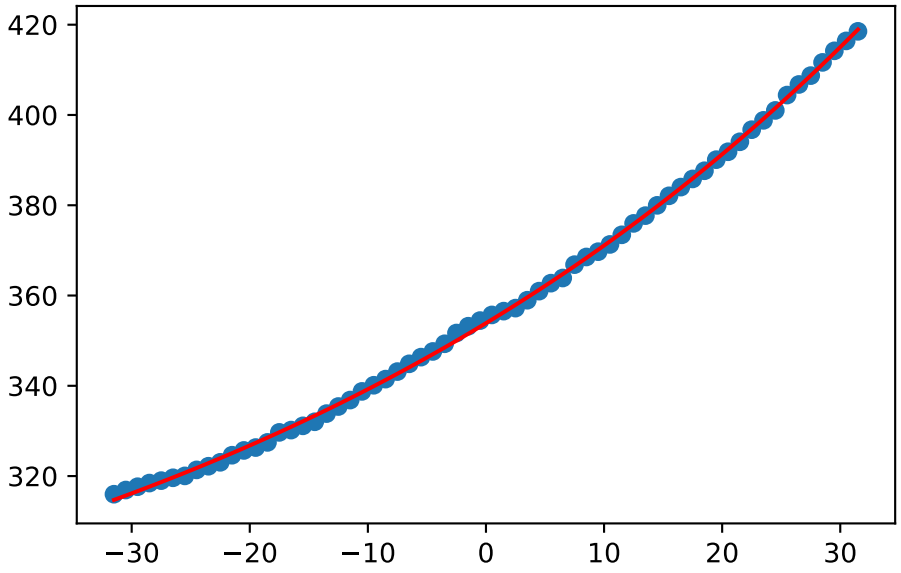
```
array([3.0403909e+02, 2.4108765e-01, 5.5710758e+01], dtype=float32)
```

That seems better. Let's try with that.

```
n_it = 10
xvals = np.zeros(shape = (n_it, 3))
xvals[0, :] = (beta0_init, beta1_init, beta2_init)

for t in range(1, n_it):
  jxvals = jnp.array(xvals[t-1, :])
  hess = deriv2(jxvals)
  e = np.linalg.eig(hess)
  if(np.any(e[0] < 0)):
    raise ValueError("not positive definite")
  xvals[t, :] = xvals[t-1, :] - np.linalg.solve(hess, deriv1(jxvals))
  print(loss(xvals[t,:]))
```

```
38.304234
30.817396
30.443848
30.442596
30.442549
30.442583
30.442507
30.44248
30.442472
```

```
beta_hat = xvals[t,:]

plt.scatter(data.year, data.co2)
plt.plot(data.year, fit(beta_hat), 'r-')

plt.show(block=False)
```



That looks pretty good, but the lack of positive definiteness/sensitivity to starting values should make us cautious. That said, in this case we can visually assess the fit and see that it looks pretty good.

Next we'll see that some optimization methods used commonly for statistical models (in particular Fisher scoring and iterative reweighted least squares (IRLS or IWLS) are just Newton-Raphson in disguise.

Fisher scoring variant on N-R (optional)

The Fisher information (FI) is the expected value of the outer product of the gradient of the log-likelihood with itself

$$I(\theta) = E_f(\nabla f(y) \nabla f(y)^\top),$$

where the expected value is with respect to the data distribution. Under regularity conditions (true for exponential families), the expectation of the Hessian of the log-likelihood is minus the Fisher information, $E_f H_f(y) = -I(\theta)$. We get the observed Fisher information by plugging the data values into either expression instead of taking the expected value.

Thus, standard N-R can be thought of as using the observed Fisher information to find the updates. Instead, if we can compute the expectation, we can use minus the FI in place of the Hessian. The result is the Fisher scoring (FS) algorithm. Basically instead of using the Hessian for a given set of data, we are using the FI, which we can think of as the average Hessian over repeated samples of data from the data distribution. FS and N-R have the same convergence properties (i.e., quadratic convergence) but in a given problem, one may be computationally or analytically easier. Givens and Hoeting comment that FS works better for rapid improvements at the beginning of iterations and N-R better for refinement at the end.

$$\begin{aligned}(NR): \theta_{t+1} = &\quad \theta_t - H_f(\theta_t)^{-1}\nabla f(\theta_t) \\ (FS): \theta_{t+1} = &\quad \theta_t + I(\theta_t)^{-1}\nabla f(\theta_t)\end{aligned}$$

The Gauss-Newton algorithm for nonlinear least squares involves using the FI in place of the Hessian in determining a Newton-like step. `nls()` in R uses this approach.

Connections between statistical uncertainty and ill-conditionedness

When either the observed or expected FI matrix is nearly singular this means we have a small eigenvalue in the inverse covariance (the precision), which means a large eigenvalue in the covariance matrix. This indicates some linear combination of the parameters has low precision (high variance), and that in that direction the likelihood is nearly flat. As we've seen with N-R, convergence slows with shallow gradients, and we may have numerical problems in determining good optimization steps when the likelihood is sufficiently flat. So convergence problems and statistical uncertainty go hand in hand. One, but not the only, example of this occurs when we have nearly collinear regressors.

IRLS (IWLS) for Generalized Linear Models (GLMs)

As many of you know, iterative reweighted least squares (also called iterative weighted least squares) is the standard method for estimation with GLMs. It involves linearizing the model and using working weights and working variances and solving a weighted least squares (WLS) problem (recalling that the generic WLS solution is $\hat{\beta} = (X^\top WX)^{-1}X^\top WY$).

Exponential families can be expressed as

$$f(y;\theta,\phi) = \exp((y\theta - b(\theta))/a(\phi) + c(y,\phi)),$$

with $E(Y) = b'(\theta)$ and $\text{Var}(Y) = b''(\theta)$. If we have a GLM in the canonical parameterization (log link for Poisson data, logit for binomial), we have the natural parameter θ equal to the linear predictor, $\theta = \eta$. A standard linear predictor would simply be $\eta = X\beta$.

Considering N-R for a GLM in the canonical parameterization (and ignoring $a(\phi)$, which is one for logistic and Poisson regression), one can show that the gradient of the GLM log-likelihood is the inner product of the covariates and a residual vector, $\nabla l(\beta) = (Y - E(Y))^\top X$, and the Hessian is $H_l(\beta) = -X^\top WX$ where W is a diagonal matrix with $\{\text{Var}(Y_i)\}$ on the diagonal (the working weights). Note that both $E(Y)$ and the variances in W depend on β, so these will change as we iteratively update β. Therefore, the N-R update is

$$\beta_{t+1} = \beta_t + (X^\top W_t X)^{-1}X^\top(Y - E(Y)_t)$$

where $E(Y)_t$ and W_t are the values at the current parameter estimate, β_t . For example, for logistic regression (here with $n_i = 1$), $W_{t,ii} = p_{ti}(1 - p_{ti})$ and $E(Y)_{ti} = p_{ti}$ where $p_{ti} = \frac{\exp(X_i^\top \beta_t)}{1+\exp(X_i^\top \beta_t)}$. In the canonical parameterization of a GLM, the Hessian does not depend on the data, so the observed and expected FI are the same, and therefore N-R and FS are the same.

The update above can be rewritten in the standard form of IRLS as a WLS problem,

$$\begin{aligned}
\beta_{t+1} &= \beta_t + (X^\top W_t X)^{-1} X^\top (Y - E(Y)_t) \\
&= (X^\top W_t X)^{-1} (X^\top W_t X)\beta_t + (X^\top W_t X)^{-1} X^\top (Y - E(Y)_t) \\
&= (X^\top W_t X)^{-1} X^\top W_t \left[X\beta_t + W_t^{-1}(Y - E(Y)_t)\right] \\
&= (X^\top W_t X)^{-1} X^\top W_t \tilde{Y}_t,
\end{aligned}$$

where the so-called working observations are $\tilde{Y}_t = X\beta_t + W_t^{-1}(Y - E(Y)_t)$. Note that these are on the scale of the linear predictor. The interpretation is that the working observations are equal to the current fitted values, $X\beta_t$, plus weighted residuals where the weight (the inverse of the variance) takes the actual residuals and scales to the scale of the linear predictor.

While IRLS is standard for GLMs, you can also use general purpose optimization routines.

IRLS is a special case of the general Gauss-Newton method for nonlinear least squares.

Descent methods and Newton-like methods

More generally a Newton-like method has updates of the form

$$x_{t+1} = x_t - \alpha_t M_t^{-1} f'(x_t).$$

We can choose M_t in various ways, including as an approximation to the second derivative.

This opens up several possibilities:

1. using more computationally efficient approximations to the second derivative,
2. avoiding steps that do not go in the correct direction (i.e., go uphill when minimizing), and
3. scaling by α_t so as not to step too far.

Let's consider a variety of strategies.

Descent methods

The basic strategy is to choose a good direction and then choose the longest step for which the function continues to decrease. Suppose we have a direction, p_t. Then we need to move $x_{t+1} = x_t + \alpha_t p_t$, where α_t is a scalar, choosing a good α_t. We might use a line search (e.g., bisection or golden section search) to find the local minimum of $f(x_t + \alpha_t p_t)$ with respect to α_t. However, we often would not want to run to convergence, since we'll be taking additional steps anyway.

Steepest descent chooses the direction as the steepest direction downhill, setting $M_t = I$, since the gradient gives the steepest direction uphill (the negative sign in the equation below has us move directly downhill rather than directly uphill). Given the direction, we want to scale the step

$$x_{t+1} = x_t - \alpha_t f'(x_t)$$

where the contraction, or step length, parameter α_t is chosen sufficiently small to ensure that we descend, via some sort of line search. The critical downside to steepest descent is that when the contours are elliptical, it tends to zigzag; here's an example.

My original code for this was in R, so I'm just leaving it that way rather than having to do a lot of fine-tuning to get the image to display the way I want in Python.

(Note that I do a full line search (using the golden section method via `optimize()`) at each step in the direction of steepest descent - this is generally computationally wasteful, but I just want to illustrate how steepest descent can go wrong, even if you go the "right" amount in each direction.)

```
par(mai = c(.5,.4,.1,.4))
f <- function(x){
    x[1]^2/1000 + 4*x[1]*x[2]/1000 + 5*x[2]^2/1000
}
f_deriv1 <- function(x){
    c(2 * x[1]/1000 + 4 * x[2]/1000,
    4 * x[1]/1000 + 10 * x[2]/1000)
}
lineSearch <- function(alpha, xCurrent, direction, FUN){
    newx <- xCurrent + alpha * direction
    FUN(newx)
}
nIt <- 50
xvals <- matrix(NA, nr = nIt, nc = 2)
xvals[1, ] <- c(7, -4)
for(t in 2:50){
    newalpha <- optimize(lineSearch, interval = c(-5000, 5000),
        xCurrent = xvals[t-1, ], direction = f_deriv1(xvals[t-1, ]),
        FUN = f)$minimum
    xvals[t, ] <- xvals[t-1, ] + newalpha * f_deriv1(xvals[t-1, ])
}
x1s <- seq(-5, 8, len = 100); x2s = seq(-5, 2, len = 100)
fx <- apply(expand.grid(x1s, x2s), 1, f)
## plot f(x) surface on log scale
fields::image.plot(x1s, x2s, matrix(log(fx), 100, 100),
    xlim = c(-5, 8), ylim = c(-5,2))
lines(xvals) ## overlay optimization path
```

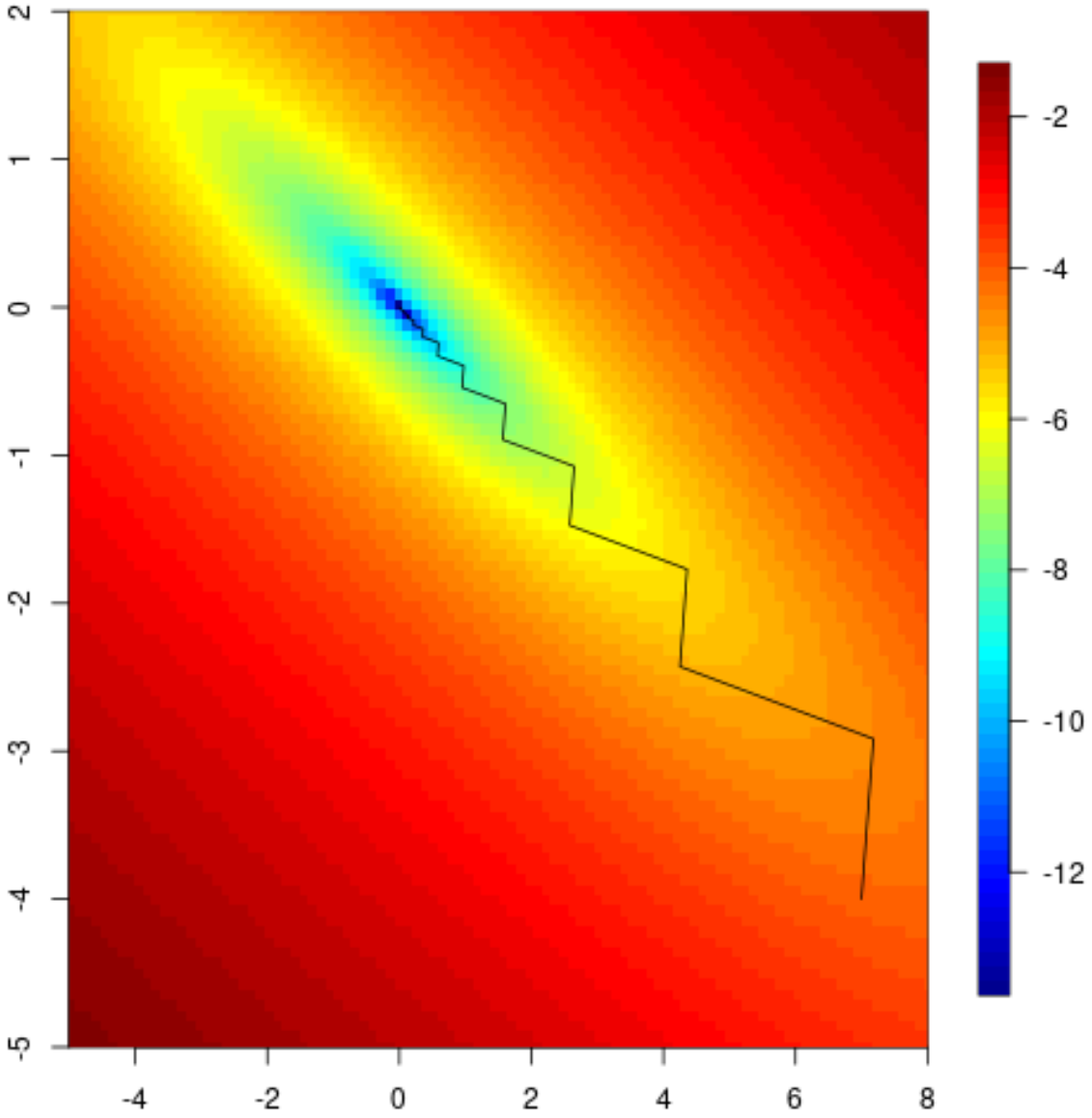


Figure 1: Path of steepest descent

If the contours are circular, steepest descent works well. Newton's method deforms elliptical contours based on the Hessian. Another way to think about this is that steepest descent does not take account of the rate of change in the gradient, while Newton's method does.

The general descent algorithm is

$$x_{t+1} = x_t - \alpha_t M_t^{-1} f'(x_t),$$

where M_t is generally chose to approximate the Hessian and α_t allows us to adjust the step in a smart way. Basically, since the negative gradient tells us the direction that descends (at least within a small neighborhood), if we don't go too far, we should be fine and should work our way downhill. One can work this out formally using a Taylor approximation to $f(x_{t+1}) - f(x_t)$ and see that we make use of M_t being positive definite. (Unfortunately backtracking with positive definite M_t does not give a theoretical guarantee that the method will converge. We also need to make sure that the steps descend sufficiently quickly and that the algorithm does not step along a level contour of f.)

The conjugate gradient algorithm for iteratively solving large systems of equations is all about choosing the direction and the step size in a smart way given the optimization problem at hand.

Quasi-Newton methods such as BFGS

Other replacements for the Hessian matrix include estimates that do not vary with t and finite difference approximations. When calculating the Hessian is expensive, it can be very helpful to substitute an approximation.

A basic finite difference approximation requires us to compute finite differences in each dimension, but this could be computationally burdensome. A more efficient strategy for choosing M_{t+1} is to (1) make use of M_t and (2) make use of the most recent step to learn about the curvature of $f'(x)$ in the direction of travel. One approach is to use a rank one update to M_t.

A basic strategy is to choose M_{t+1} such that the secant condition is satisfied:

$$M_{t+1}(x_{t+1} - x_t) = \nabla f(x_{t+1}) - \nabla f(x_t),$$

which is motivated by the fact that the secant approximates the gradient in the direction of travel. Basically this says to modify M_t in such a way that we incorporate what we've learned about the gradient from the most recent step. M_{t+1} is not fully determined based on this, and we generally impose other conditions, in particular that M_{t+1} is symmetric and positive definite. Defining $s_t = x_{t+1} - x_t$ and $y_t = \nabla f(x_{t+1}) - \nabla f(x_t)$, the unique, symmetric rank one update (why is the following a rank one update?) that satisfies the secant condition is

$$M_{t+1} = M_t + \frac{(y_t - M_t s_t)(y_t - M_t s_t)^\top}{(y_t - M_t s_t)^\top s_t}.$$

If the denominator is positive, M_{t+1} may not be positive definite, but this is guaranteed for non-positive values of the denominator. One can also show that one can achieve positive definiteness by shrinking the denominator toward zero sufficiently.

A standard approach to updating M_t is a commonly-used rank two update that generally results in M_{t+1} being positive definite is

$$M_{t+1} = M_t - \frac{M_t s_t (M_t s_t)^\top}{s_t^\top M_t s_t} + \frac{y_t y_t^\top}{s_t^\top y_t},$$

which is known as the Broyden-Fletcher-Goldfarb-Shanno (BFGS) update. This is one of the methods used in R in *optim()*.

Question: how can we update M_t^{-1} to M_{t+1}^{-1} efficiently? It turns out there is a way to update the Cholesky of M_t efficiently and this is a better approach than updating the inverse.

The order of convergence of quasi-Newton methods is generally slower than the quadratic convergence of N-R because of the approximations but still faster than linear. In general, quasi-Newton methods will do much better if the scales of the elements of x are similar. Lange suggests using a starting point for which one can compute the expected information, to provide a good starting value M_0.

Note that for estimating a covariance based on the numerical information matrix, we would not want to rely on M_t from the final iteration, as the approximation may be poor. Rather we would spend the effort to better estimate the Hessian directly at x^*.

Stochastic gradient descent

Stochastic gradient descent (SGD) is a well-known method in machine learning, commonly used for fitting deep neural networks. It allows you to optimize an objective function with respect to what is often a very large number of parameters even when the data size is huge.

Gradient descent is a simplification of Newton's method that does not rely on the second derivative, but rather chooses the direction using the gradient and then a step size, α_t:

$$x_{t+1} = x_t - \alpha_t f'(x_t)$$

The basic idea of stochastic gradient descent is to replace the gradient with a function whose expected value is the gradient, $E(g(x_t)) = f'(x_t)$:

$$x_{t+1} = x_t - \alpha_t g(x_t)$$

Thus on average we should go in a good (downhill) direction. Given that we know that strictly following the gradient can lead to slow convergence, it makes some intuitive sense that we could still do ok without using the exact gradient. One can show formally that SGD will converge for convex functions.

SGD can be used in various contexts, but the common one we will focus on is when

$$f(x) = \sum_{i=1}^{n} f_i(x)$$

$$f'(x) = \sum_{i=1}^{n} f_i'(x)$$

for large n. Thus calculation of the gradient is $O(n)$, and we may not want to incur that computational cost. How could we implement SGD in such a case? At each iteration we could randomly choose an observation and compute the contribution to the gradient from that data point, or we could choose a random subset of the data (this is *mini-batch SGD*), or there are variations where we systematically cycle through the observations or cycle through subsets. However, in some situations, convergence is actually much faster when using randomness. And if the data are ordered in some meaningful way we definitely do not want to cycle through the observations in that order, as this can result in a biased estimate of the gradient and slow convergence. So one generally randomly shuffles the data before starting SGD. Note that using subsets rather than individual observations is likely to be more effective as it can allow us to use optimized matrix/vector computations.

One thing to note is that often one would scale the objective function (and therefore the gradient) by dividing by the number of observations. This of course doesn't change the optimum or the directions involved, but it does mean that the magnitude of the estimated gradient won't change with the batch size. And it means that the expected gradient is equal to the true gradient, rather than a scaled version of the true gradient.

How should one choose the step size, α_t (also called the learning rate)? One might think that as one gets close to the optimum, if one isn't careful, one might simply bounce around near the optimum in a random way, without actually converging to the optimum. So intuition suggests that α_t should decrease with t. Some choices of step size have included:

- $\alpha_t = 1/t$
- set a schedule, such that for T iterations, $\alpha_t = \alpha$, then for the next T, $\alpha_t = \alpha\gamma$, then for the next T, $\alpha_t = \alpha\gamma^2$. A heuristic is for $\gamma \in (0.8, 0.9)$.
- run with $\alpha_t = \alpha$ for T iterations, then with $\alpha_t = \alpha/2$ for $2T$, then with $\alpha_t = \alpha/4$ for $4T$ and so forth.

Coordinate descent (Gauss-Seidel)

Gauss-Seidel is also known a back-fitting or cyclic coordinate descent. The basic idea is to work element by element rather than having to choose a direction for each step. For example backfitting used to be used to fit generalized additive models of the form $E(Y) = f_1(z_1) + f_2(z_2) + ... + f_p(z_p)$.

The basic strategy is to consider the jth component of $f'(x)$ as a univariate function of x_j only and find the root, $x_{j,t+1}$ that gives $f_j'(x_{j,t+1}) = 0$. One cycles through each element of x to complete a single cycle and then iterates. The appeal is that univariate root-finding/minimization is easy, often more stable than multivariate, and quick.

However, Gauss-Seidel can zigzag, since you only take steps in one dimension at a time, as we see here. (Again the code is in R.)

```
f <- function(x){
    return(x[1]^2/1000 + 4*x[1]*x[2]/1000 + 5*x[2]^2/1000)
}
f1 <- function(x1, x2){ # f(x) as a function of x1
    return(x1^2/1000 + 4*x1*x2/1000 + 5*x2^2/1000)
}
f2 <- function(x2, x1){ # f(x) as a function of x2
    return(x1^2/1000 + 4*x1*x2/1000 + 5*x2^2/1000)
}
x1s <- seq(-5, 8, len = 100); x2s = seq(-5, 2, len = 100)
fx <- apply(expand.grid(x1s, x2s), 1, f)
fields::image.plot(x1s, x2s, matrix(log(fx), 100, 100))
nIt <- 49
xvals <- matrix(NA, nr = nIt, nc = 2)
xvals[1, ] <- c(7, -4)
## 5, -10
for(t in seq(2, nIt, by = 2)){
    ## Note that full optimization along each axis is unnecessarily
```

```
    ## expensive (since we are going to just take another step in the next
    ## iteration. Just using for demonstration here.
    newx1 <- optimize(f1, x2 = xvals[t-1, 2], interval = c(-40, 40))$minimum
    xvals[t, ] <- c(newx1, xvals[t-1, 2])
    newx2 <- optimize(f2, x1 = newx1, interval = c(-40, 40))$minimum
    xvals[t+1, ] <- c(newx1, newx2)
}
lines(xvals)
```

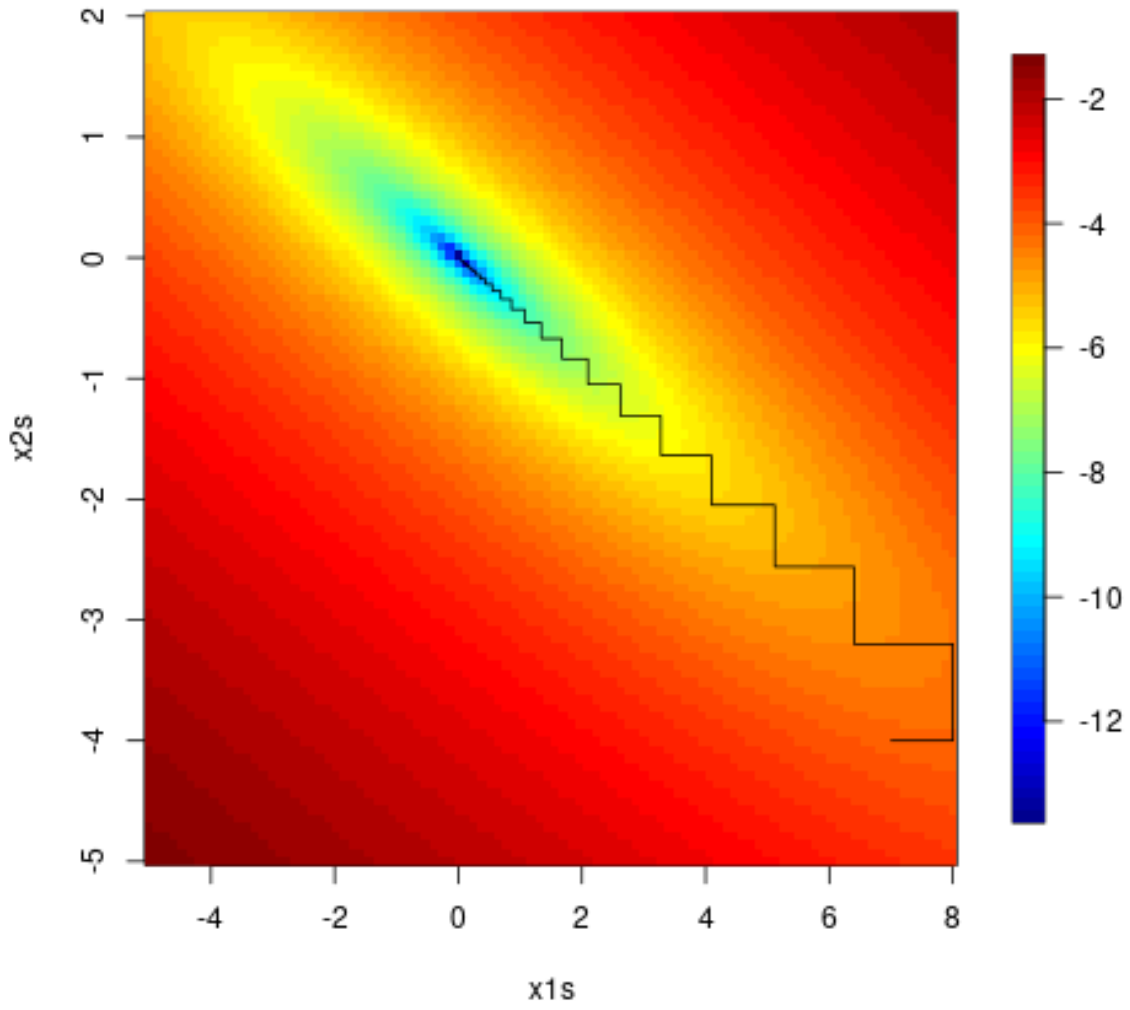


Figure 2: Coordinate descent

In the notes for Unit 9 on linear algebra, I discussed the use of Gauss-Seidel to iteratively solve $Ax = b$ in situations where factorizing A (which of course is $O(n^3)$) is too computationally expensive.

The lasso

The *lasso* uses an L1 penalty in regression and related contexts. A standard formulation for the lasso

in regression is to minimize

$$\|Y - X\beta\|_2^2 + \lambda \sum_j |\beta_j|$$

to find $\hat{\beta}(\lambda)$ for a given value of the penalty parameter, λ. A standard strategy to solve this problem is to use coordinate descent, either cyclically, or by using directional derivatives to choose the coordinate likely to decrease the objective function the most (a greedy strategy). We need to use directional derivatives because the penalty function is not differentiable, but does have directional derivatives in each direction. The directional derivative of the objective function for β_j is

$$-2 \sum_i x_{ij}(Y_i - X_i^\top \beta) \pm \lambda$$

where we add λ if $\beta_j \geq 0$ and you subtract λ if $\beta_j < 0$. If $\beta_{j,t}$ is 0, then a step in either direction contributes $+\lambda$ to the derivative as the contribution of the penalty.

Once we have chosen a coordinate, we set the directional derivative to zero and solve for β_j to obtain $\beta_{j,t+1}$.

The `glmnet` package in R (described in this Journal of Statistical Software paper) implements such optimization for a variety of penalties in linear model and GLM settings, including the lasso. This Mittal et al. paper describes similar optimization for survival analysis with very large p, exploiting sparsity in the X matrix for computational efficiency; note that they do not use Newton-Raphson because the matrix operations are infeasible computationally.

One nice idea that is used in lasso and related settings is the idea of finding the regression coefficients for a variety of values of λ, combined with "warm starts". A general approach is to start with a large value of λ for which all the coefficients are zero and then decrease λ. At each new value of λ, use the estimated coefficients from the previous value as the starting values. This should allow for fast convergence and gives what is called the "solution path". Often λ is chosen based on cross-validation.

The LARS (least angle regression) algorithm uses a similar strategy that allows one to compute $\hat{\beta}_\lambda$ for all values of λ at once.

The lasso can also be formulated as the constrained minimization of $\|Y - X\beta\|_2^2$ s.t. $\sum_j |\beta_j| \leq c$, with c now playing the role of the penalty parameter. Solving this minimization problem would take us in the direction of quadratic programming, a special case of convex programming, discussed in Section 9.

Nelder-Mead

This approach avoids using derivatives or approximations to derivatives. This makes it robust, but also slower than Newton-like methods. The basic strategy is to use a simplex, a polytope of $p + 1$ points in p dimensions (e.g., a triangle when searching in two dimensions, tetrahedron in three dimensions...) to explore the space, choosing to shift, expand, or contract the polytope based on the evaluation of f at the points.

The algorithm relies on four tuning factors: a reflection factor, $\alpha > 0$; an expansion factor, $\gamma > 1$; a contraction factor, $0 < \beta < 1$; and a shrinkage factor, $0 < \delta < 1$. First one chooses an initial simplex: $p + 1$ points that serve as the vertices of a convex hull.

1. Evaluate and order the points, $x_1, \ldots, x_{p+1}$ based on $f(x_1) \leq \ldots \leq f(x_{p+1})$. Let $\bar{x}$ be the average of the first p x's.
2. (Reflection) Reflect x_{p+1} across the hyperplane (a line when $p+1=3$) formed by the other points to get x_r, based on α.
    - $x_r = (1+\alpha)\bar{x} - \alpha x_{p+1}$
3. If $f(x_r)$ is between the best and worst of the other points, the iteration is done, with x_r replacing x_{p+1}. We've found a good direction to move.
4. (Expansion) If $f(x_r)$ is better than all of the other points, expand by extending x_r to x_e based on γ, because this indicates the optimum may be further in the direction of reflection. If $f(x_e)$ is better than $f(x_r)$, use x_e in place of x_{p+1}. If not, use x_r. The iteration is done.
    - $x_e = \gamma x_r + (1-\gamma)\bar{x}$
5. If $f(x_r)$ is worse than all the other points, but better than $f(x_{p+1})$, let $x_h = x_r$. Otherwise $f(x_r)$ is worse than $f(x_{p+1})$ so let $x_h = x_{p+1}$. In either case, we want to concentrate our polytope toward the other points.
    a. (Contraction) Contract x_h toward the hyperplane formed by the other points, based on β, to get x_c. If the result improves upon $f(x_h)$ replace x_{p+1} with x_c. Basically, we haven't found a new point that is better than the other points, so we want to contract the simplex away from the bad point.
        - $x_c = \beta x_h + (1-\beta)\bar{x}$
    b. (Shrinkage) Otherwise (if x_c is not better than x_h), replace x_{p+1} with x_h and shrink the simplex toward x_1. Basically this suggests our step sizes are too large and we should shrink the simplex, shrinking towards the best point.
        - $x_i = \delta x_i + (1-\delta)x_1$ for $i = 2, \ldots, p+1$

Convergence is assessed based on the sample variance of the function values at the points, the total of the norms of the differences between the points in the new and old simplexes, or the size of the simplex. In class we'll work through some demo code (click below to show the code in the html version of this document) that illustrates the individual steps in an iteration of Nelder-Mead.

```
alpha = 1
gamma = 2
beta = 0.5
delta = 0.5

# Auxiliary function to plot line segments
def plotseg(ind1, ind2, col=1):
    if not isinstance(ind1, np.ndarray):
        plt.plot([xs[ind1, 0], xs[ind2, 0]], [xs[ind1, 1], xs[ind2, 1],], color=col)
    else:
        plt.plot([ind1[0], xs[ind2, 0]], [ind1[1], xs[ind2, 1]], color=col)

# Initial polytope
```

```
xs = np.array([[-2, 3], [-6, 4], [-4, 2]], dtype=np.float64)

plt.figure()
plt.plot(xs[:, 0], xs[:, 1], 'o-', color = 'black')
plt.xlim(-7, -1)
plt.ylim(1, 8)

plotseg(0, 1, 'black')
plotseg(0, 2, 'black')
plotseg(1, 2, 'black')

xbar = np.mean(xs[:2], axis=0)
plt.text(xs[2, 0], xs[2, 1], 'x[p+1]')
plt.plot(xbar[0], xbar[1], 'ro')


# Reflection
xr = (1 + alpha) * xbar - alpha * xs[2]
plt.plot(xr[0], xr[1], 'ro', marker='o', markersize=4, color='red')
plt.text(xr[0], xr[1], 'x[r]')

plotseg(xr, 0, 'red')
plotseg(xr, 1, 'red')
plotseg(0, 1, 'red')

# Consider expansion
xe = gamma * xr + (1 - gamma) * xbar
plt.plot(xe[0], xe[1], 'ro', marker='o', markersize=4, color='green')
plt.text(xe[0], xe[1], 'x[e]')

plotseg(xe, 0, 'green')
plotseg(xe, 1, 'green')
plotseg(0, 1, 'green')

# Consider contraction
xh = xs[2,:]  # Suppose the original point is better than the reflection
xc = beta * xh + (1 - beta) * xbar
plt.plot(xc[0], xc[1], 'ro', marker='o', markersize=4, color='blue')
plt.text(xc[0], xc[1], 'x[c]')

plotseg(xc, 0, 'blue')
plotseg(xc, 1, 'blue')
plotseg(0, 1, 'blue')

# Shrinkage
```

```
xs[1,:] = delta * xs[1,:] + (1 - delta) * xs[0,:]
xs[2,:] = delta * xs[2,:] + (1 - delta) * xs[0,:]

plt.plot(xs[1, 0], xs[1, 1], 'ro', marker='o', markersize=4, color='purple')
plt.plot(xs[2, 0], xs[2, 1], 'ro', marker='o', markersize=4, color='purple')

plotseg(0, 1, 'purple')
plotseg(0, 2, 'purple')
plotseg(1, 2, 'purple')

plt.show(block=False)
```

We can see the points at which the function was evaluated in the same quadratic example we saw in previous sections. The left hand panel shows the steps from a starting point somewhat far from the optimum, with the first 9 points numbered. In this case, we start with points 1, 2, and 3. Point 4 is a reflection. At this point, it looks like point 5 is a contraction but that doesn't exactly follow the algorithm above (since Point 4 is between Points 2 and 3 so the iteration should end without a contraction), so perhaps the algorithm as implemented is a bit different than as described above. In any event, the new set is (2, 3, 4). Then point 6 and point 7 are reflection and expansion steps and the new set is (3, 4, 6). Points 8 and 9 are again reflection and expansion steps. The right hand panel shows the steps from a starting point near (actually at) the optimum. Points 4 and 5 are reflection and expansion steps, with the next set being (1, 2, 5). Now step 6 is a reflection but it is the worst of all the points, so point 7 is a contraction of point 2 giving the next set (1, 5, 7). Point 8 is then a reflection and point 9 is a contraction of point 5.

(Again some code in R.)

```
f <- function(x, plot = TRUE, verbose = FALSE) {
    result <- x[1]^2/1000 + 4*x[1]*x[2]/1000 + 5*x[2]^2/1000
    if(verbose) print(result)
    if(plot && cnt < 10) {
        points(x[1], x[2], pch = as.character(cnt))
        if(cnt < 10) cnt <<- cnt + 1 else cnt <<- 1
        if(interactive())
            invisible(readline(prompt = "Press <Enter> to continue..."))
    } else if(plot) points(x[1], x[2])
    return(result)
}

par(mfrow = c(1,2), mgp = c(1.8,.7,0), mai = c(.5,.45,.1,.5), cex = 0.7)

x1s <- seq(-5, 10, len = 100); x2s = seq(-5, 2, len = 100)
fx <- apply(expand.grid(x1s, x2s), 1, f, FALSE)
cnt <- 1
fields::image.plot(x1s, x2s, matrix(log(fx), 100, 100))
init <- c(7, -4)
optim(init, f, method = "Nelder-Mead", verbose = FALSE)
```

```
par(cex = 0.7)
x1s <- seq(-.2, .2, len = 100); x2s = seq(-.12, .12, len = 100)
fx <- apply(expand.grid(x1s, x2s), 1, f, FALSE)
cnt <- 1
fields::image.plot(x1s, x2s, matrix(log(fx), 100, 100))
init <- c(-0, 0)
optim(init, f, method = "Nelder-Mead", verbose = FALSE)
```

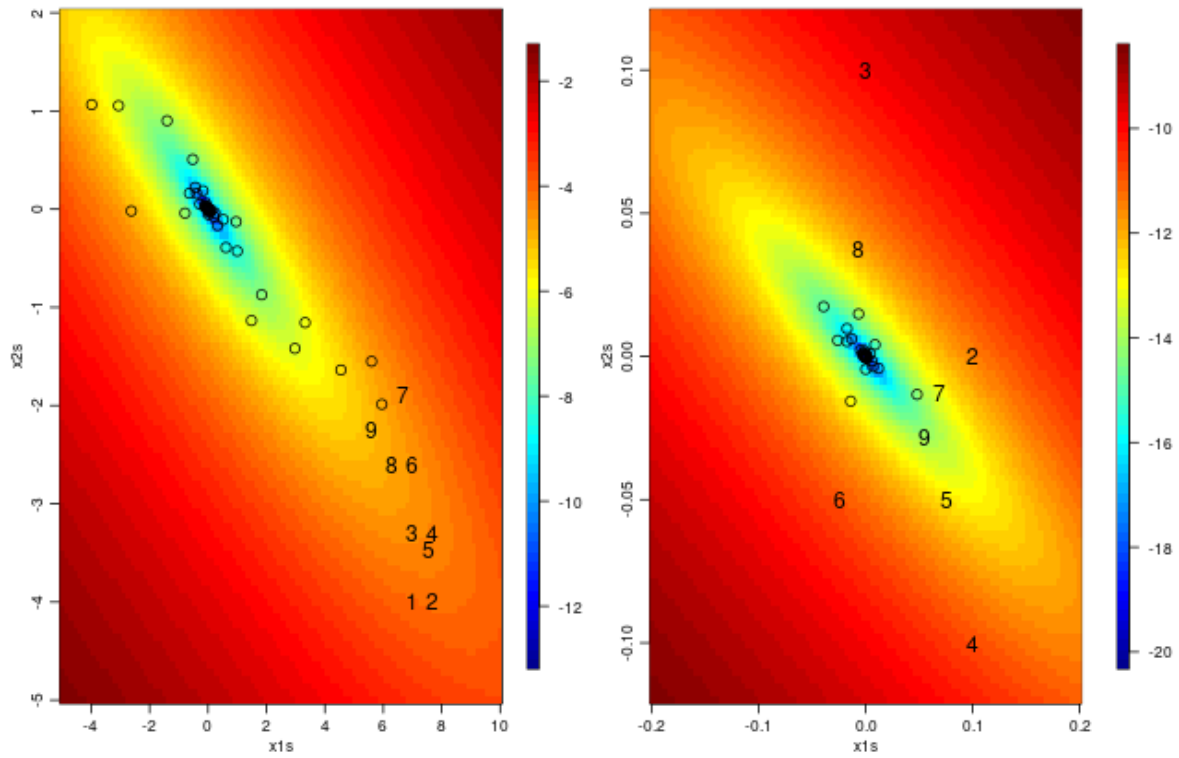


Figure 3: Nelder-Mead

Here's an online graphical illustration of Nelder-Mead.

This is the default in `optim()` in R. It is an option (by specifying `method='Nelder-mead'`) for `scipy.optimize.minimize` (BFGS or a variant is the default).

Simulated annealing (SA) (optional)

Simulated annealing is a *stochastic* descent algorithm, unlike the deterministic algorithms we've already discussed. It has a couple critical features that set it aside from other approaches. First, uphill moves are allowed; second, whether a move is accepted is stochastic, and finally, as the iterations proceed the

algorithm becomes less likely to accept uphill moves.

Assume we are minimizing a negative log likelihood as a function of θ, $f(\theta)$.

The basic idea of simulated annealing is that one modifies the objective function, f in this case, to make it less peaked at the beginning, using a "temperature" variable that changes over time. This helps to allow moves away from local minima, when combined with the ability to move uphill. The name comes from an analogy to heating up a solid to its melting temperature and cooling it slowly - as it cools the atoms go through rearrangements and slowly freeze into the crystal configuration that is at the lowest energy level.

Here's the algorithm. We divide up iterations into stages, $j = 1, 2, ...$ in which the temperature variable, τ_j, is constant. Like MCMC, we require a proposal distribution to propose new values of θ.

1. Propose to move from θ_t to $\tilde{\theta}$ from a proposal density, $g_t(\cdot|\theta_t)$, such as a normal distribution centered at θ_t.
2. Accept $\tilde{\theta}$ as θ_{t+1} according to the probability $\min(1, \exp((f(\theta_t) - f(\tilde{\theta}))/\tau_j)$ - i.e., accept if a uniform random deviate is less than that probability. Otherwise set $\theta_{t+1} = \theta_t$. Notice that for larger values of τ_j the differences between the function values at the two locations are reduced (just like a large standard deviation spreads out a distribution). So the exponentiation smooths out the objective function when τ_j is large.
3. Repeat steps 1 and 2 m_j times.
4. Increment the temperature and cooling schedule: $\tau_j = \alpha(\tau_{j-1})$ and $m_j = \beta(m_{j-1})$. Back to step 1.

The temperature should slowly decrease to 0 while the number of iterations, m_j, should be large. Choosing these 'schedules' is at the core of implementing SA. Note that we always accept downhill moves in step 2 but we sometimes accept uphill moves as well.

For each temperature, SA produces an MCMC based on the Metropolis algorithm. So if m_j is long enough, we should sample from the stationary distribution of the Markov chain, $\exp(-f(\theta)/\tau_j))$. Provided we can move between local minima, the chain should gravitate toward the global minima because these are increasingly deep (low values) relative to the local minima as the temperature drops. Then as the temperature cools, θ_t should get trapped in an increasingly deep well centered on the global minimum. There is a danger that we will get trapped in a local minimum and not be able to get out as the temperature drops, so the temperature schedule is quite important in trying to avoid this.

A wide variety of schedules have been tried. One approach is to set $m_j = 1 \forall j$ and $\alpha(\tau_{j-1}) = \frac{\tau_{j-1}}{1+a\tau_{j-1}}$ for a small a. For a given problem it can take a lot of experimentation to choose τ_0 and m_0 and the values for the scheduling functions. For the initial temperature, it's a good idea to choose it large enough that $\exp((f(\theta_i) - f(\theta_j))/\tau_0) \approx 1$ for any pair $\{\theta_i, \theta_j\}$ in the domain, so that the algorithm can visit the entire space initially.

Simulated annealing can converge slowly. Multiple random starting points or stratified starting points can be helpful for finding a global minimum. However, given the slow convergence, these can also be computationally burdensome.

6. Basic optimization in Python

Core optimization functions

Scipy provides various useful optimization functions via `scipy.optimize`, including many of the algorithms discussed in this unit.

- `minimize_scalar` implements golden section search (`golden`) and interpolation combined with golden section search (`brent`, akin to `optimize` in R).
- `minimize` implements various methods for multivariate optimization including Nelder-Mead and BFGS. You can choose which method you prefer and can try multiple methods. You can supply a gradient function for use with the Newton-related methods but it can also calculate numerical derivatives on the fly.
- One can provide a variety of nonlinear, linear, and simple bounds constraints as well, though certain types of constraints can only be used with certain algorithms.

Here's a very basic example of using `minimize` with the Mauna Loa CO2 example we saw earlier when hand-coding Newton-Raphson. Here we'll include the unknown variance as an additional parameter so we have a full likelihood. And as mentioned previously, we could profile out β_0 and β_1 and σ^2, but we won't do that here so as to illustrate multivariate Newton-Raphson.

```
import os
import pandas as pd
import numpy as np
import statsmodels.api as sm
from scipy.optimize import minimize

data = pd.read_csv(os.path.join('..','data', 'co2_annmean_mlo.csv'),
    header = 0, names = ['year','co2','unc'])

## Center years for better numerical behavior
data.year = data.year - np.mean(data.year)

beta2_init = 50
implicit_covar = np.exp(data.year/beta2_init)

X = sm.add_constant(implicit_covar)
model = sm.OLS(data.co2, X).fit()
beta0_init, beta1_init = model.params

def nll(params, data):
    # params[3] is log of sigma^2 to address constraint
    n = len(data.year)
    fitted = params[0] + params[1] * np.exp(data.year / params[2])
    return (n/2)*params[3] + 0.5 * np.sum((data.co2 - fitted)**2) / np.exp(params[3])

sigma2_init = np.mean((data.co2-model.fittedvalues)**2)
```

```
inits = (beta0_init, beta1_init, beta2_init, np.log(sigma2_init))

# Optimization using Nelder-Mead
start_time = time.time()
fit1 = minimize(nll, inits, args=(data), method='Nelder-Mead', options={'disp': True})
end_time = time.time()
print("Nelder-Mead Optimization:")
print(fit1)
print("Execution Time:", end_time - start_time, "seconds")

# Optimization using BFGS
start_time = time.time()
fit2 = minimize(nll, inits, args=(data), method='BFGS', options={'disp': True})
end_time = time.time()
print("\nBFGS Optimization:")
print(fit2)
print("Execution Time:", end_time - start_time, "seconds")

## IMPORTANT: `hess_inv` is just the final estimate from BFGS not
## a direct numerical estimate of the Hessian at the optimum.
## If you need the Hessian, calculate it directly, e.g, using `numdifftools`.

# BFGS with specific relative tolerance on 'x', given precision loss message.
## See `https://docs.scipy.org/doc/scipy/reference/optimize.minimize-bfgs.html`.

fit2_alt = minimize(nll, inits, args=(data), method='BFGS',
    options={'disp': True, 'xrtol': 1e-6})


# Different starting value (recall non-positive definite Hessian)


beta2_init = 100
implicit_covar = np.exp(data.year/beta2_init)
X = sm.add_constant(implicit_covar)
model = sm.OLS(data.co2, X).fit()
beta0_init, beta1_init = model.params
sigma2_init = np.mean((data.co2-model.fittedvalues)**2)
inits = (beta0_init, beta1_init, beta2_init, np.log(sigma2_init))
fit3 = minimize(nll, inits, args=(data), method='BFGS', options={'disp': True})

beta2_init = 10
implicit_covar = np.exp(data.year/beta2_init)
X = sm.add_constant(implicit_covar)
model = sm.OLS(data.co2, X).fit()
```

```
beta0_init, beta1_init = model.params
sigma2_init = np.mean((data.co2-model.fittedvalues)**2)
inits = (beta0_init, beta1_init, beta2_init, np.log(sigma2_init))
fit4 = minimize(nll, inits, args=(data), method='BFGS', options={'disp': True})

# Arbitrarily bad starting values
inits = (10, 0, 10000, 0.1)
fit5 = minimize(nll, inits, args=(data), method='Nelder-Mead', options={'disp': True})
fit6 = minimize(nll, inits, args=(data), method='BFGS', options={'disp': True})
```

Automatic differentiation (AD)

Optimizers that use derivative information often allow you to pass in a gradient function and possibly a Hessian function.

Automatic differentiation is basically the implementation of the chain rule on a computer, building up derivative information for a potentially complicated calculation from the known derivatives of basic functions (such as multiplication, exponentiation, etc.). This involves some careful software engineering that generates the code that will calculate the derivative via the chain rule. Given that it's *just* the chain rule, it gets surprisingly complicated.

However, from a user perspective, it's often simple to use if you are able to write your calculation using JAX or PyTorch or another AD-enabled package.

So if you have an optimizer that takes gradient/Hessian functions, you can probably pass in JAX or PyTorch versions of those functions.

And of course if you're implementing something yourself, you may want to consider making use of AD rather than using numerical differentiation. We saw an example of this in Section 5.

Various considerations in using the Python functions

As we've seen, initial values are important both for avoiding divergence (e.g., in N-R), for increasing speed of convergence, and for helping to avoid local optima. So it is well worth the time to try to figure out a good starting value or multiple starting values for a given problem.

Scaling can be important. One useful step is to make sure the problem is well-scaled, namely that a unit step in any parameter has a comparable change in the objective function, preferably approximately a unit change at the optimum. Basically if x_j is varying at p orders of magnitude smaller than the other xs, we want to reparameterize to $\tilde{x}_j = x_j \cdot 10^p$ and then convert back to the original scale after finding the answer. Or we may want to work on the log scale for some variables, reparameterizing as $\tilde{x}_j = \log(x_j)$.

If the function itself gives very large or small values near the solution, you may want to rescale the entire function to avoid calculations with very large or small numbers. This can avoid problems such as having apparent convergence because a gradient is near zero, simply because the scale of the function is small. When we use the log of a likelihood (primarily to avoid over/underflow), that often helps in this regard as well even if the function would not over/underflow.

> **!** Check your answer
>
> **Always** consider your answer and make sure it makes sense, in particular that you haven't 'converged' to an extreme value on the boundary of the space.

Venables and Ripley suggest that it is often worth supplying analytic first derivatives rather than having a routine calculate numerical derivatives but not worth supplying analytic second derivatives.

In general for software development it's obviously worth putting more time into figuring out the best optimization approach and supplying derivatives. For a one-off analysis, you can try a few different approaches and assess sensitivity.

The nice thing about likelihood optimization is that the asymptotic theory tells us that with large samples, the likelihood is approximately quadratic (i.e., the asymptotic normality of MLEs), which makes for a nice surface over which to do optimization. When optimizing with respect to variance components and other parameters that are non-negative, one approach to dealing with the constraints is to optimize with respect to the log of the parameter.

7. Combinatorial optimization over discrete spaces

Many statistical optimization problems involve continuous domains, but sometimes there are problems in which the domain is discrete. Variable selection is an example of this.

Simulated annealing can be used for optimizing in a discrete space. Another approach uses *genetic algorithms*, in which one sets up the dimensions as loci grouped on a chromosome and has mutation and crossover steps in which two potential solutions reproduce. An example would be in high-dimensional variable selection.

Stochastic search variable selection is a popular Bayesian technique for variable selection that involves MCMC.

8. Convexity

Many optimization problems involve (or can be transformed into) convex functions. Convex optimization (also called convex programming) is a big topic and one that we'll only brush the surface of in Sections 8 and 9. The goal here is to give you enough of a sense of the topic that you know when you're working on a problem that might involve convex optimization, in which case you'll need to go learn more.

Optimization for convex functions is simpler than for ordinary functions because we don't have to worry about local optima - any stationary point (point where the gradient is zero) is a global minimum. A set S in $\Re^p$ is convex if any line segment between two points in S lies entirely within S. More generally, S is convex if any convex combination is itself in S, i.e., $\sum_{i=1}^m \alpha_i x_i \in S$ for non-negative weights, α_i, that sum to 1. Convex functions are defined on convex sets - f is convex if for points in a convex set, $x_i \in S$, we have $f(\sum_{i=1}^m \alpha_i x_i) \leq \sum_{i=1}^m \alpha_i f(x_i)$. Strict convexity is when the inequality is strict (no equality).

The first-order convexity condition relates a convex function to its first derivative: f is convex if and only if $f(x) \geq f(y) + \nabla f(y)^\top (x - y)$ for y and x in the domain of f. We can interpret this as saying that the first order Taylor approximation to f is tangent to and below (or touching) the function at all points.

The second-order convexity condition is that a function is convex if (provided its first derivative exists), the derivative is non-decreasing, in which case we have $f''(x) \geq 0 \ \forall x$ (for univariate functions). If we have $f''(x) \leq 0 \ \forall x$ (a concave, or convex down function) we can always consider $-f(x)$, which is convex. Convexity in multiple dimensions means that the gradient is nondecreasing in all dimensions. If f is twice differentiable, then if the Hessian is positive semi-definite, f is convex.

There are a variety of results that allow us to recognize and construct convex functions based on knowing what operations create and preserve convexity. The Boyd book is a good source for material on such operations. Note that norms are convex functions (based on the triangle inequality), $\| \sum_{i=1}^n \alpha_i x_i \| \leq \sum_{i=1}^n \alpha_i \| x_i \|$.

We'll talk about a general algorithm that works for convex functions (the MM algorithm) and about the EM algorithm that is well-known in statistics, and is a special case of MM.

MM algorithm

The MM algorithm is really more of a principle for constructing problem specific algorithms. MM stands for majorize-minorize. We'll use the majorize part of it to minimize functions - the minorize part is the counterpart for maximizing functions.

Suppose we want to minimize a convex function, $f(x)$. The idea is to construct a majorizing function, at x_t, which we'll call g. g majorizes f at x_t if $f(x_t) = g(x_t)$ and $f(x) \leq g(x) \forall x$.

The iterative algorithm is as follows. Given x_t, construct a majorizing function $g_t(x)$. Then minimize g_t w.r.t. x (or at least move downhill, such as with a modified Newton step) to find x_{t+1}. Then we iterate, finding the next majorizing function, $g_{t+1}(x)$. The algorithm is obviously guaranteed to go downhill, and ideally we use a function g that is easy to work with (i.e., to minimize or go downhill with respect to). Note that we haven't done any matrix inversions or computed any derivatives of f. Furthermore, the algorithm is numerically stable - it does not over- or undershoot the optimum. The downside is that convergence can be quite slow.

The tricky part is finding a good majorizing function. Basically one needs to gain some skill in working with inequalities. The Lange book has some discussion of this.

An example is for estimating regression coefficients for median regression (aka least absolute deviation regression), which minimizes $f(\theta) = \sum_{i=1}^n |y_i - z_i^\top \theta| = \sum_{i=1}^n |r_i(\theta)|$. Note that $f(\theta)$ is convex because affine functions (in this case $y_i - z_i^\top \theta$) are convex, convex functions of affine functions are convex, and the summation preserves the convexity. We want to minimize

$$\begin{aligned} f(\theta) &= \sum_{i=1}^n |r_i(\theta)| \\ &= \sum_{i=1}^n \sqrt{r_i(\theta)^2} \end{aligned}$$

Next, $h(x) = \sqrt{x}$ is concave, so we can use the following (commonly-used) inequality, $h(x) \leq h(y) + h'(y)(x-y)$ which holds for any concave function, h, and note that we have equality when $y = x$. For $y = \theta_t$, the current value in the iterative optimization, we have:

$$\begin{aligned} f(\theta) &= \sum_{i=1}^{n} \sqrt{r_i(\theta)^2} \\ &\leq \sum_{i=1}^{n} \sqrt{r_i(\theta_t)^2} + \frac{r_i(\theta)^2 - r_i(\theta_t)^2}{2\sqrt{r_i(\theta_t)^2}} \\ &= g_t(\theta) \end{aligned}$$

where the term on the right of the second equation is our majorizing function $g(\theta)$ for the current θ_t. We then have

$$\begin{aligned} g_t(\theta) &= \sum_{i=1}^{n} \sqrt{r_i(\theta_t)^2} + \frac{1}{2} \sum_{i=1}^{n} \frac{r_i(\theta)^2 - r_i(\theta_t)^2}{2\sqrt{r_i(\theta_t)^2}} \\ &= \frac{1}{2} \sum_{i=1}^{n} \sqrt{r_i(\theta_t)^2} + \frac{1}{2} \sum_{i=1}^{n} \frac{r_i(\theta)^2}{\sqrt{r_i(\theta_t)^2}} \end{aligned}$$

Our job in this iteration of the algorithm is to minimize g with respect to θ (recall that θ_t is a fixed value), so we can ignore the first sum, which doesn't involve θ. Minimizing the second sum can be seen as a weighted least squares problem, where the numerator is the usual sum of squared residuals and the weights are $w_i = \frac{1}{\sqrt{(y_i - z_i^\top \theta_t)^2}}$. Intuitively this makes sense: the weight is large when the magnitude of the residual is small this makes up for the fact that we are using least squares when we want to mimimize absolute deviations. So our update is:

$$\theta_{t+1} = (Z^\top W(\theta_t) Z)^{-1} Z^\top W(\theta_t) Y,$$

where $W(\theta_t)$ is a diagonal matrix with elements $w_1, \ldots, w_n$.

As usual, we want to think about what could go wrong numerically. If we have some very small magnitude residuals, they will get heavily upweighted in this procedure, which might cause instability in our optimization.

For an example of MM being used in practice for a real problem, see Jung et al. (2014): Biomarker Detection in Association Studies: Modeling SNPs Simultaneously via Logistic ANOVA, Journal of the American Statistical Association 109:1355.

Expectation-Maximization (EM)

It turns out the EM algorithm that many of you have heard about is a special case of MM. For our purpose here, we'll consider maximization.

The EM algorithm is most readily motivated from a missing data perspective. Suppose you want to maximize $L(\theta|x) = f(x;\theta)$ based on available data in a missing data context. Denote the complete data as $Y = (X, Z)$ with Z is missing. As we'll see, in many cases, Z is actually a set of latent variables that we introduce into the problem to formulate it so we can use EM. The canonical example is when Z are membership indicators in a mixture modeling context. (Note that in the case where you introduce

Z, that also means that one could also just directly maximize $L(\theta|x)$, which in many cases may work better than using the EM algorithm.)

In general, $\log L(\theta|x)$ may be hard to optimize because it involves an integral over the missing data, Z:

$$f(x;\theta) = \int f(x,z;\theta)dz,$$

but the EM algorithm provides a recipe that makes the optimization straightforward for many problems.

The algorithm is as follows. Let θ^t be the current value of θ. Then define

$$Q(\theta|\theta^t) = E(\log L(\theta|Y)|x;\theta^t)$$

.

That expectation is an expectation with respect to the conditional distribution, $f(z|x;\theta = \theta^t)$.

The algorithm is

1. E step: Compute $Q(\theta|\theta^t)$, ideally calculating the expectation over the missing data in closed form. Note that $\log L(\theta|Y)$ is a function of θ so $Q(\theta|\theta^t)$ will involve both θ and θ^t.
2. M step: Maximize $Q(\theta|\theta^t)$ with respect to θ, finding θ^{t+1}.
3. Continue until convergence.

Ideally both the E and M steps can be done analytically. When the M step cannot be done analytically, one can employ some of the numerical optimization tools we've already seen. When the E step cannot be done analytically, one standard approach is to estimate the expectation by Monte Carlo, which produces Monte Carlo EM (MCEM). The strategy is to draw from z_j from $f(z|x,\theta^t)$ and approximate Q as a Monte Carlo average of $\log f(x,z_j;\theta)$, and then optimize over this approximation to the expectation. If one can't draw in closed form from the conditional density, one strategy is to do a short MCMC to draw a (correlated) sample. However, if the E step cannot be done analytically EM often will be very slow. (Even when the E step can be done analytically, EM is often slow.)

EM can be show to increase the value of the function at each step using Jensen's inequality (equivalent to the information inequality that holds with regard to the Kullback-Leibler divergence between two distributions) (Givens and Hoeting, p. 95, go through the details). Furthermore, one can show that it amounts, at each step, to maximizing a minorizing function for $\log L(\theta)$ - the minorizing function (effectively Q) is tangent to $\log L(\theta)$ at θ^t and lies below $\log L(\theta)$.

A standard example is a mixture model. (Here we'll assume a mixture of normal distributions, but other distributions could be used.) Therefore we have

$$f(x;\theta) = \sum_{k=1}^{K} \pi_k f_k(x;\mu_k,\sigma_k)$$

where we have K mixture components and π_k are the (marginal) probabilities of being in each component. The complete parameter vector is $\theta = \{\{\pi_k\},\{\mu_k\},\{\sigma_k\}\}$. Note that the likelihood is a complicated product (over observations) over the sum (over components), so maximization may be difficult. Furthermore, such likelihoods are well-known to be multimodal because of label switching.

To use EM, we take the group membership indicators for each observation as the missing data. For the ith observation, we have $z_i \in \{1, 2, \ldots, K\}$. Introducing these indicators "breaks the mixture". If we know the memberships for all the observations, it's often easy to estimate the parameters for each group based on the observations from that group. For example if the $\{f_k\}$'s were normal densities, then we can estimate the mean and variance of each normal density using the sample mean and sample variance of the x_i's that belong to each mixture component. EM will give us a variation on this that uses "soft" (i.e., probabilistic) weighting.

The complete log likelihood given z and x is

$$\log \prod_i f(x_i|z_i;\theta)\Pr(Z_i = z_i;\theta)$$

which can be expressed as

$$\begin{aligned}\log L(\theta|x,z) = & \sum_i \log f(x_i;\mu_{z_i},\sigma_{z_i}) + \log \pi_{z_i} \\ = & \sum_i \sum_k I(z_i = k)(\log f_k(x_i;\mu_k,\sigma_k) + \log \pi_k)\end{aligned}$$

with Q equal to

$$Q(\theta|\theta^t) = \sum_i \sum_k E(I(z_i = k)|x_i;\theta^t)(\log f_k(x_i;\mu_k,\sigma_k) + \log \pi_k)$$

where $E(I(z_i = k)|x_i;\theta^t)$ is equal to the probability that the ith observation is in the kth group given x_i and θ_t, which is calculated from Bayes theorem as

$$p_{ik}^t = \frac{\pi_k^t f_k(x_i;\mu_k^t,\sigma_k^t)}{\sum_j \pi_j^t f_j(x_i;\mu_k^t,\sigma_k^t)}$$

We can now separately maximize $Q(\theta|\theta^t)$ with respect to π_k and μ_k, σ_k to find π_k^{t+1} and $\mu_k^{t+1}, \sigma_k^{t+1}$, since the expression is the sum of a term involving the parameters of the distributions and a term involving the mixture probabilities. In the latter case, if the f_k are normal distributions, you end up with a weighted sum of normal distributions, for which the estimators of the mean and variance parameters are the weighted mean of the observations and the weighted variance.

9. Optimization under constraints

Constrained optimization is harder than unconstrained, and inequality constraints harder to deal with than equality constraints.

Constrained optimization can sometimes be avoided by reparameterizing. Some examples include: - working on the log scale (e.g., to optimize w.r.t. a variance component or other non-negative parameter) - using the logit transformation to optimize with respect to a parameter on $(0, 1)$ (or more generally some other bounded interval, after shifting and scaling to $(0, 1)$.

Optimization under constraints often goes under the name of 'programming', with different types of programming for different types of objective functions combined with different types of constraints.

Convex optimization (convex programming)

Convex programming minimizes $f(x)$ s.t. $h_j(x) \leq 0$, $j = 1, \ldots, m$ and $a_i^\top x = b_i$, $i = 1, \ldots, q$, where both f and the constraint functions are convex. Note that this includes more general equality constraints, as we can write $g(x) = b$ as two inequalities $g(x) \leq b$ and $g(x) \geq b$. It also includes $h_j(x) \geq b_j$ by taking $-h_j(x)$. Note that we can always have $h_j(x) \leq b_j$ and convert to the above form by subtracting b_j from each side (note that this preserves convexity). A vector x is said to be feasible, or in the feasible set, if all the constraints are satisfied for x.

There are good algorithms for convex programming, and it's possible to find solutions when we have hundreds or thousands of variables and constraints. It is often difficult to recognize if one has a convex program (i.e., if f and the constraint functions are convex), but there are many tricks to transform a problem into a convex program and many problems can be solved through convex programming. So the basic challenge is in recognizing or transforming a problem to one of convex optimization; once you've done that, you can rely on existing methods to find the solution.

Linear programming, quadratic programming, second order cone programming and semidefinite programming are all special cases of convex programming. In general, these types of optimization are progressively more computationally complex.

First let's see some of the special cases and then discuss the more general problem.

Linear programming: Linear system, linear constraints

Linear programming seeks to minimize

$$f(x) = c^\top x$$

subject to a system of m inequality constraints, $a_i^\top x \leq b_i$ for $i = 1, \ldots, m$, where A is of full row rank. This can also be written in terms of generalized inequality notation, $Ax \preceq b$. There are standard algorithms for solving linear programs, including the simplex method and interior point methods.

Note that each equation in the set of equations $Ax = b$ defines a hyperplane, so each inequality in $Ax \preceq b$ defines a half-space. Minimizing a linear function (presuming that the minimum exists) must mean that we push in the correct direction towards the boundaries formed by the hyperplanes, with the solution occuring at a corner (vertex) of the solid formed by the hyperplanes. The simplex algorithm starts with a feasible solution at a corner and moves along edges in directions that improve the objective function.

General system, equality constraints

Linear equality constraints

Suppose we have an objective function $f(x)$ and we have q equality constraints, $Ax = b$. We can manipulate this into an unconstrained problem. The null space of A is the set of δ s.t. $A\delta = 0$. So if we start with a candidate x_c s.t. $Ax_c = b$ (e.g., by using the pseudo inverse, A^+b), we can form all other candidates (a candidate is an x s.t. $Ax = b$) as $x = x_c + \delta = x_c + Bz$ where B is a set of column basis functions for the null space of A and $z \in \Re^{p-q}$. Consider $h(z) = f(x_c + Bz)$ and note that h is a function of $p - q$ rather than p inputs. Namely, we are working in a reduced dimension space with no constraints. If we assume differentiability of f, we can express $\nabla h(z) = B^\top \nabla f(x_c + Bz)$

and $H_h(z) = B^\top H_f(x_c + Bz)B$. Then we can use unconstrained methods to find the point at which $\nabla h(z) = 0$.

How do we find B? One option is to use the $p - m$ columns of V in the SVD of A that correspond to singular values that are zero. A second option is to take the QR decomposition of $A^\top$. Then B is the columns of Q_2, where these are the columns of the (non-skinny) Q matrix corresponding to the rows of R that are zero.

Nonlinear equality constraints

For more general (nonlinear) equality constraints, $g_i(x) = b_i$, $i = 1, \ldots, q$, we can use the Lagrange multiplier approach to define a new objective function,

$$L(x, \lambda) = f(x) + \lambda^\top (g(x) - b)$$

for which, if we set the derivative (with respect to both x and the Lagrange multiplier vector, λ) equal to zero, we have a critical point of the original function and we respect the constraints.

An example occurs with quadratic programming, under the simplification of affine equality constraints (quadratic programming in general optimizes a quadratic function under affine inequality constraints - i.e., constraints of the form $Ax - b \preceq 0$). For example we might solve a least squares problem subject to linear equality constraints, $f(x) = \frac{1}{2}x^\top Qx + m^\top x + c$ s.t. $Ax = b$, where Q is positive semi-definite. The Lagrange multiplier approach gives the objective function

$$L(x, \lambda) = \frac{1}{2}x^\top Qx + m^\top x + c + \lambda^\top (Ax - b)$$

and differentiating gives the equations

$$\frac{\partial L(x, \lambda)}{\partial x} = m + Qx + A^\top \lambda = 0$$
$$\frac{\partial L(x, \lambda)}{\partial \lambda} = Ax = b,$$

which gives us a system of equations that leads to the solution

$$\begin{pmatrix} x \\ \lambda \end{pmatrix} = \begin{pmatrix} Q & A^\top \\ A & 0 \end{pmatrix}^{-1} \begin{pmatrix} -m \\ b \end{pmatrix}.$$

Using known results for inverses of matrices split into blocks, one gets that $x^* = -Q^{-1}m + Q^{-1}A^\top(AQ^{-1}A^\top)^{-1}(AQ^{-1}m + b)$. This can be readily coded up using strategies from Unit 10.

The dual problem (optional)

Sometimes a reformulation of the problem eases the optimization. There are different kinds of dual problems, but we'll just deal with the Lagrangian dual. Let $f(x)$ be the function we want to minimize, under constraints $g_i(x) = 0$; $i = 1, \ldots, q$ and $h_j(x) \leq 0$; $j = 1, \ldots, m$. Here I've explicitly written out the equality constraints to follow the notation in Lange. Consider the Langrangian,

$$L(x, \lambda, \mu) = f(x) + \sum_i \lambda_i g_i(x) + \sum_j \mu_j h_j(x).$$

Solving that can be shown to be equivalent to this optimization:

$$\inf_x \sup_{\lambda,\mu:\mu_j \geq 0} L(x, \lambda, \mu)$$

where the supremum ensures that the constraints are satisfied because the Lagrangian is infinity if the constraints are not satisfied.

Let's consider interchanging the minimization and maximization. For $\mu \succeq 0$, one can show that

$$\sup_{\lambda,\mu:\mu_j \geq 0} \inf_x L(x, \lambda, \mu) \leq \inf_x \sup_{\lambda,\mu:\mu_j \geq 0} L(x, \lambda, \mu),$$

because $\inf_x L(x, \lambda, \mu) \leq f(x^*)$ for the minimizing value x^* (p. 216 of the Boyd book). This gives us the Lagrange dual function:

$$d(\lambda, \mu) = \inf_x L(x, \lambda, \mu),$$

and the Lagrange dual problem is to find the best lower bound:

$$\sup_{\lambda,\mu:\mu_j \geq 0} d(\lambda, \mu).$$

The dual problem is always a convex optimization problem because $d(\lambda, \mu)$ is concave (because $d(\lambda, \mu)$ is a pointwise infimum of a family of affine functions of (λ, μ)). If the optima of the primal (original) problem and that of the dual do not coincide, there is said to be a "duality gap". For convex programming, if certain conditions are satisfied (called *constraint qualifications*), then there is no duality gap, and one can solve the dual problem to solve the primal problem. Usually with the standard form of convex programming, there is no duality gap. Provided we can do the minimization over x in closed form we then maximize $d(\lambda, \mu)$ w.r.t. the Lagrangian multipliers in a new constrained problem that is sometimes easier to solve, giving us (λ^*, μ^*).

One can show (p. 242 of the Boyd book) that $\mu_i^* = 0$ unless the ith constraint is active at the optimum x^* and that x^* minimizes $L(x, \lambda^*, \mu^*)$. So once one has (λ^*, μ^*), one is in the position of minimizing an unconstrained convex function. If $L(x, \lambda^*, \mu^*)$ is strictly convex, then x^* is the unique optimum provided x^* satisfies the constraints, and no optimum exists if it does not.

Here's a simple example: suppose we want to minimize $x^\top x$ s.t. $Ax = b$. The Lagrangian is $L(x, \lambda) = x^\top x + \lambda^\top (Ax - b)$. Since $L(x, \lambda)$ is quadratic in x, the infimum is found by setting $\nabla_x L(x, \lambda) = 2x + A^\top \lambda = 0$, yielding $x = -\frac{1}{2} A^\top \lambda$. So the dual function is obtained by plugging this value of x into $L(x, \lambda)$, which gives

$$d(\lambda) = -\frac{1}{4} \lambda^\top A A^\top \lambda - b^\top \lambda,$$

which is concave quadratic. In this case we can solve the original constrained problem in terms of this unconstrained dual problem.

Another example is the primal and dual forms for finding the SVM classifier (see the Wikipedia article). In this algorithm, we want to develop a classifier using n pairs of $y \in \mathfrak{R}^1$ and $x \in \mathfrak{R}^p$. The dual form is easily derived because the minimization over x occurs in a function that is quadratic in x. Expressing the problem in the primal form gives an optimization in $\mathfrak{R}^p$ while doing so in the dual form gives an optimization in $\mathfrak{R}^n$. So one reason to use the dual form would be if you have $n \ll p$.

KKT conditions (optional)

Karush-Kuhn-Tucker (KKT) theory provides sufficient conditions under which a constrained optimization problem has a minimum, generalizing the Lagrange multiplier approach. The Lange and Boyd books have whole sections on this topic.

Suppose that the function and the constraint functions are continuously differentiable near x^* and that we have the Lagrangian as before:

$$L(x, \lambda, \mu) = f(x) + \sum_i \lambda_i g_i(x) + \sum_j \mu_j h_j(x).$$

For nonconvex problems, if x^* and (λ^*, μ^*) are the primal and dual optimal points and there is no duality gap, then the KKT conditions hold:

$$\begin{aligned} h_j(x^*) \leq &\quad 0 \\ g_i(x^*) = &\quad 0 \\ \mu_j^* \geq &\quad 0 \\ \mu_j^* h_j(x^*) = &\quad 0 \\ \nabla f(x^*) + \sum_i \lambda_i^* \nabla g_i(x^*) + \sum_j \mu_j^* \nabla h_j(x^*) = &\quad 0. \end{aligned}$$

For convex problems, we also have that if the KKT conditions hold, then x^* and (λ^*, μ^*) are primal and dual optimal and there is no duality gap.

We can consider this from a slightly different perspective, in this case requiring that the Lagrangian be twice differentiable.

First we need a definition. A *tangent direction*, w, with respect to $g(x)$, is a vector for which $\nabla g_i(x)^\top w = 0$. If we are at a point, x^*, at which the constraint is satisfied, $g_i(x^*) = 0$, then we can move in the tangent direction (orthogonal to the gradient of the constraint function) (i.e., along the level curve) and still satisfy the constraint. This is the only kind of movement that is legitimate (gives us a feasible solution).

If the gradient of the Lagrangian with respect to x is equal to 0,

$$\nabla f(x^*) + \sum_i \lambda_i \nabla g_i(x^*) + \sum_j \mu_j \nabla h_j(x^*) = 0,$$

and if $w^\top H_L(x^*, \lambda, \mu) w > 0$ (with H_L being the Hessian of the Lagrangian) for all vectors w s.t. $\nabla g(x^*)^\top w = 0$ and, for all active constraints,$\nabla h(x^*)^\top w = 0$, then x^* is a local minimum. An active constraint is an inequality for which $h_j(x^*) = 0$ (rather than $h_j(x^*) < 0$, in which case it is inactive). Basically we only need to worry about the inequality constraints when we are on the boundary, so the goal is to keep the constraints inactive.

Some basic intuition is that we need positive definiteness only for directions that stay in the feasible region. That is, our only possible directions of movement (the tangent directions) keep us in the feasible region, and for these directions, we need the objective function to be increasing to have a minimum.

If we were to move in a direction that goes outside the feasible region, it's ok for the quadratic form involving the Hessian to be negative.

Many algorithms for convex optimization can be interpreted as methods for solving the KKT conditions.

Interior-point methods

We'll briefly discuss one of the standard methods for solving a convex optimization problem. The barrier method is one type of interior-point algorithm. It turns out that Newton's method can be used to solve a constrained optimization problem, with twice-differentiable f and linear equality constraints. So the basic strategy of the barrier method is to turn the more complicated constraint problem into one with only linear equality constraints.

Recall our previous notation, in which convex programming minimizes $f(x)$ s.t. $h_i(x) \leq 0$, $j = 1, \ldots, m$ and $a_i^\top x = b_i$, $i = 1, \ldots, q$, where both f and the constraint functions are convex. The strategy begins with moving the inequality constraints into the objective function:

$$f(x) + \sum_{j=1}^{m} I_-(h_j(x))$$

where $I_-(u) = 0$ if $u \leq 0$ and $I_-(u) = \infty$ if $u > 0$.

This is fine, but the new objective function is not differentiable so we can't use a Newton-like approach. Instead, we approximate the indicator function with a logarithmic function, giving the new objective function

$$\tilde{f}(x) = f(x) + \sum_{j=1}^{m} -(1/t^*) \log(-h_j(x)),$$

which is convex and differentiable. The new term pushes down the value of the overall objective function when x approaches the boundary, nearing points for which the inequality constraints are not met. The $-\sum(1/t^*)\log(-h_j(x))$ term is called the log barrier, since it keeps the solution in the feasible set (i.e., the set where the inequality constraints are satisfied), provided we start at a point in the feasible set. Newton's method with equality constraints ($Ax = b$) is then applied. The key thing is then to have t^* get larger (i.e., t^* is some increasing function of iteration time t) as the iterations proceed, which allows the solution to get closer to the boundary if that is indeed where the minimum lies.

The basic ideas behind Newton's method with equality constraints are (1) start at a feasible point, x_0, such that $Ax_0 = b$, and (2) make sure that each step is in a feasible direction, $A(x_{t+1} - x_t) = 0$. To make sure the step is in a feasible direction we have to solve a linear system similar to that in the simplified quadratic programming problem:

$$\begin{pmatrix} x_{t+1} - x_t \\ \lambda \end{pmatrix} = \begin{pmatrix} H_{\tilde{f}}(x_t) & A^\top \\ A & 0 \end{pmatrix}^{-1} \begin{pmatrix} -\nabla \tilde{f}(x_t) \\ 0 \end{pmatrix},$$

which shouldn't be surprising since the whole idea of Newton's method is to substitute a quadratic approximation for the actual objective function.

Software for constrained and convex optimization

For general convex optimization in Python see the `cvxopt` package. Some other resources to consider are

- MATLAB, in particular the `fmincon()` function, the CVX system, and MATLAB's linear and quadratic programming abilities.
- The CVXR package in R.

I haven't looked into CVXR in detail but given the developers include Stephen Boyd, who is a convex optimization guru, it's worth checking out.

`cvxopt` has specific solves (see `help(cvxopt.solvers)` for different kinds of convex optimization. A general purpose one is `cvxopt.solvers.cp`. Specifying the problem (the objective function, nonlinear constraints, and linear constraints) using the software is somewhat involved, so I haven't worked out an example here.

10. Summary

The different methods of optimization have different advantages and disadvantages.

According to Lange, MM and EM are numerically stable and computationally simple but can converge very slowly. Newton's method shows very fast convergence but has the downsides we've discussed. Quasi-Newton methods fall in between. Convex optimization generally comes up when optimizing under constraints.

One caution about optimizing under constraints is that you just get a point estimate; quantifying uncertainty in your estimator is more difficult. One strategy is to ignore the inactive inequality constraints and reparameterize (based on the active equality constraints) to get an unconstrained problem in a lower-dimensional space. Then you can make use of the Hessian in the usual fashion to estimate the information matrix.