Local Descent

Up to this point, we have focused on optimization involving a single design variable. This chapter introduces a general approach to optimization involving *multivariate* functions, or functions with more than one variable. The focus of this chapter s on how to use *local methods* to incrementally improve a design point until some convergence criterion is met. We being by discussing methods that, at each iteration, choose a descent direction based on a local model and then choose a step size. We then discuss methods that restrict the step to be within a region where the local model is believed to be valid. This chapter concludes with a discussion of convergence conditions.

-1. Descent Direction Iteration

A common approach to optimization is to incrementally improve a design point x by taking a step that minimizers the objective value based on a local model. The local model may be obtained, for example, from a first-or second order Taylor approximation. Optimization algorithms that follow this general approach are referred to as *descent direction methods.* They start with a design point $x^{(1)}$ and then generates a sequence of points, sometimes called *iterates,* to converge to a local minimum.

The iterative descent direction procedure involves the following steps:

1. Check whether $x^{(k)}$ satisfies the termination conditions. If it does, terminate; otherwise proceed to the next step.

2. Determine the descent direction $d^{(k)}$ using local information such as the gradient or Hessian. Some algorithms assume $||d^{(k)}|| = 1,$ but others do note.

3. Determine the step size or learning rate $\alpha^{(k)}$. Some algorithms attempts to optimize the step size so that the step maximally decreases f.

4. Compute the next design point according to:

$$x^{(k+1)} \leftarrow x^{(k)} + \alpha^{(k)} d^{(k)} \tag{1}$$

(We use step size to refer to the magnitude of the overall step. Obtaining a new iterate using equation (1) with a step size $\alpha^{(k)}$ implies that the descent direction $d^{(k)}$ has unit length. We use learning rate to refer to a scalar multiple used on a descent direction vector which does not necessarily have unit length.)

There are many different optimization methods, each of their own ways of determining α and d.

-2. Line Search

For the moment, assume we have chosen a descent direction d. We need to choose the step factor α to obtain our next design point. One approach is to use *line search,* which selects the step factor that minimizer the one-dimensional function:

$$minimize_{\alpha}\ f(x+\alpha d) \tag{2}$$

Line search is a univariate optimization problem. We can apply the univariate optimization method of our choice. To inform the search, we can use the derivative of the line search objective, which is simply the directional derivative along $d\ at\ x+\alpha d$.

Algorithm 4.1

```
def function line_search(f, x, d):
  objective = alpha -> f(x+alpha*d)
  a,b = bracket_minimum(objective)
  alpha = minimize(objective,a,b)
  return x + alpha*d
end.
```

We will be using Bisection method to find the roots of the function.

Link to Colab workbook : Exact Line search

-3. Bisection Method

The bisection method can be used to find *roots* of a function, or points where the function is zero. Such root-finding methods can be used for optimization by applying them to the derivative of the objective, locating where $f'(x) = 0$. In general, we must ensure that the resulting points are indeed local minima.

The *bisection method* maintains a bracket $[a, b]$ in which at least one root is known to exist. If f is continuous on $[a, b]$, and there is some $y \in [f(a), f(b)]$, then the *intermediate value theorem* stipulates that there exists at least one $x \in [a, b]$, such that $f(x) = y$.

The bisection method cuts the bracketed region in half with every iteration. The midpoint $(a+b)/2$ is evaluated, and the new bracket is formed from the midpoint and whichever side that continues to bracket a zero. We can terminate immediately if the midpoint evaluates to zero. Otherwise we can terminate if the midpoint evaluates to zero.
Root-finding algorithms like the bisection method require starting intervals $[a, b]$ on opposite sides

of a zero. That is, $sign(f'(a)) \neq sign(f'(b))$, or equivalently, $f'(a)f'(b) \leq 0$.

The bisection algorithm,where f' is the derivative of the univariate function we seem to optimize. We have $a < b$ that bracket a zero of f'. The interval width tolerance is ϵ. Calling bisection returns the new bracketed interval $[a, b]$ as a tuple.

```
function bisection(f',a,b,eps)
  if a > b; a,b = b,a ; end # ensure a < b

  ya, yb = f'(a), f'(b)
  if ya == 0; b = a; end
  if yb == 0; a = b; end

  while b - a > eps
    x = (a + b)/2
    y = f'(x)
    if y == 0
      a,b = x,x
    else if sign(y) == sign(ya)
      a = x
    else
      b = x
   end
end
  return (a,b)
end
```