Data Science for Economists

Lecture 14: Intro to Numerical Optimization

Drew Van Kuiken
University of North Carolina | ECON 370

Table of contents

Introduction

Agenda

Today we will cover numerical methods and optimization.

This lecture might be a bit more theory heavy than programming heavy.

What is important here is to understand the concepts rather than the details.

- Unless you specialize in computational methods for economics later on, you will likely not need to understand the details.

Motivation

Most models in data science require you to solve a maximization or minimization problem to estimate the model

- "Maximize the out-of-sample classification accuracy"
- "Minimize the variance of observations within the same cluster"

Similarly, sometimes one must find the solution to some equation in order to do some calculation

For most interesting problems, it is impossible to find an answer "analytically"

- That is, find a formula that gives the solution

Instead, we will have to use "numerical methods" to find the solutions

- Use a computer to find the solution using numbers rather than formulas

Today we will cover a handful of numerical methods that you might encounter in the future

Optimization Overview

(Some important concepts and terminology)

Optimization Overview

A basic optimization problem looks something like this:

$$\min_{x_1, x_2, x_3} f(x_1, x_2, x_3)$$

- f is a function that takes in three inputs and returns a single number as output
- x_1, x_2, x_3 are inputs/arguments to the function f (could be any number of inputs)
- f is referred to as the *objective function*

Sometimes the solution is written like this:

$$\boldsymbol{x}^* = \arg\min_{x_1, x_2, x_3} f(x_1, x_2, x_3)$$

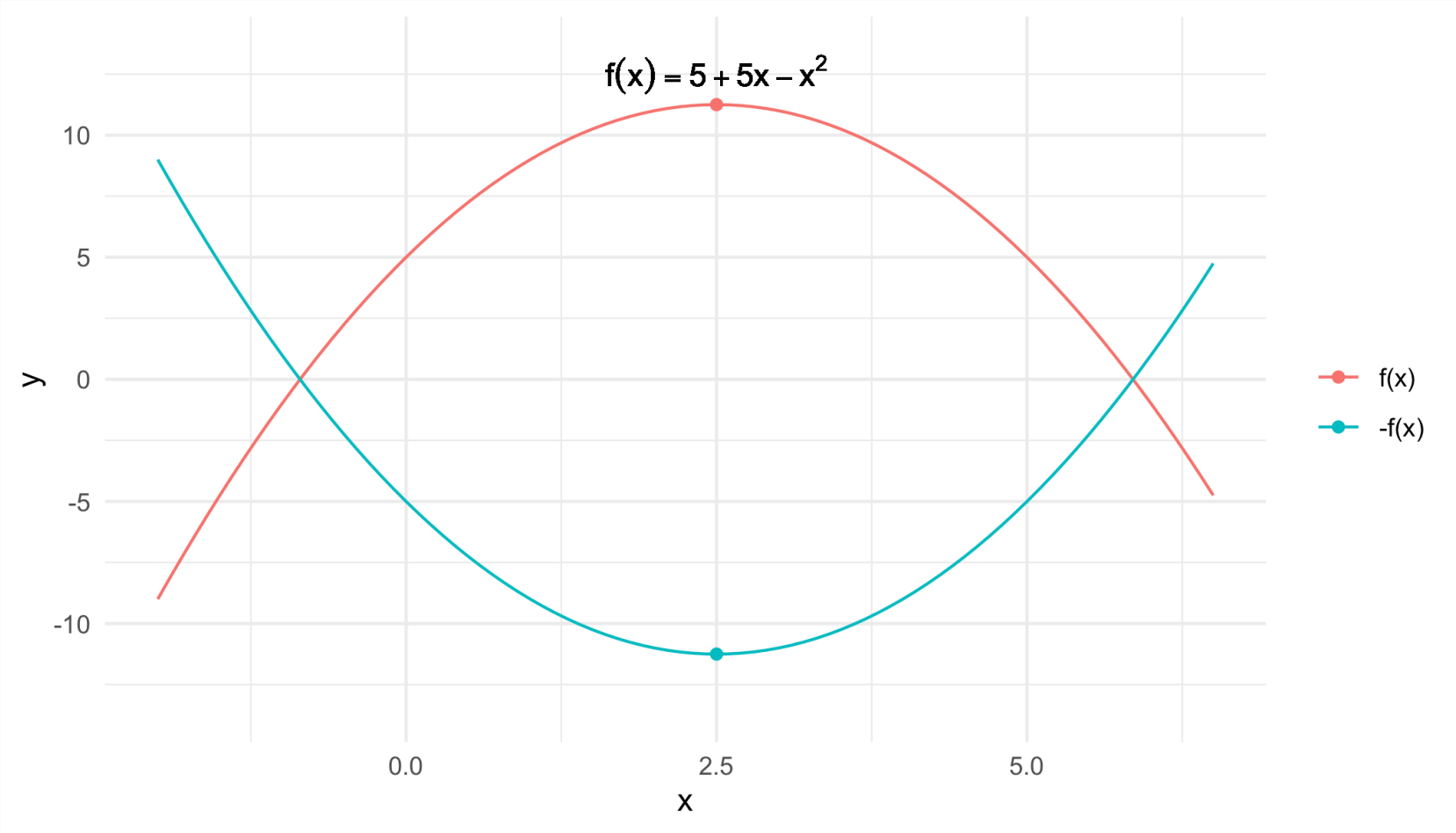
- $\boldsymbol{x}^* = (x_1^*, x_2^*, x_3^*)$
- The optimal inputs $\boldsymbol{x}^*$ are the arguments that minimize f (arg min)

Maximization problems are also optimization problems:

$$\max_{x_1, x_2, x_3} f(x_1, x_2, x_3)$$

- Fact: Maximizing f is the same thing as minimizing $-f$

Max $f(x)$ vs Min $-f(x)$



x^* is same whether maximizing $f(x)$ or minimizing $-f(x)$

- Since we can always min $-f$ to solve max f, we will assume minimization

Constrained Optimization

The optimization problems we have seen so far are "unconstrained"

- There are no limitations ("constraints") on the $\boldsymbol{x}$ values

A constrained optimization problem looks something like this:

$$\min_{x_1,x_2,x_3} f(x_1, x_2, x_3) \text{ s.t. } g(x_1, x_2, x_3) \geq 0$$

- g is a constraint function
- s.t. means "such that"
- E.g. if constraint is $x_1 + x_2 + x_3 \leq 1$, then $g(x_1, x_2, x_3) = 1 - x_1 - x_2 - x_3$

These are generally harder to solve in `R`

- Can rewrite the problem if we know the constraint "binds" i.e. $g(\boldsymbol{x}^*) = 0$
- Solve $g(x_1, x_2, x_3) = 0$ for one of the inputs and substitute into objective function
- Using same g as above, if $g(\boldsymbol{x}^*) = 0$, then $x_1 = 1 - x_2 - x_3$ and then

$$\min_{x_2,x_3} f(1 - x_2 - x_3, x_2, x_3)$$

Let's see two econ examples of optimization problems

Optimization Example 1

The Consumer's Problem: Utility Maximization

Example 1: Utility Maximization from 410

A consumer is choosing how much to consume of goods (x_1, x_2) when faced with prices (p_1, p_2) and income w, e.g. x_1 could be apples and x_2 could be oranges

- (x_1, x_2) is a generic bundle of x_1 and x_2
- $(2.5, 3)$ would be 2.5 apples and 3 oranges
- Prices and income are assumed to be fixed
- She gets no benefit from leftover income

She has a utility function $u(x_1, x_2)$ that *summarizes* her preferences over bundles (x_1, x_2)

- $u(x_1, x_2)$ returns a number that indicates how much she likes the bundle (x_1, x_2)

She wants to choose some combination of x_1 and x_2 that makes her as "well off" as possible given prices (p_1, p_2) and income w

That is she chooses values of x_1 and x_2 to solve

$$\max_{x_1 \geq 0, x_2 \geq 0} u(x_1, x_2) \text{ s.t. } x_1 p_1 + x_2 p_2 \leq w$$

This is referred to as the consumer's Utility Maximization Problem (**UMP**)

Example 1: Utility Maximization from 410

$$\max_{x_1 \geq 0, x_2 \geq 0} u(x_1, x_2) \text{ s.t. } x_1 p_1 + x_2 p_2 \leq w$$

Using the terminology from the previous slides:

- The **UMP** is a constrained optimization problem:
  - $g(x_1, x_2) = w - x_1 p_1 - x_2 p_2 \geq 0$; this is the "budget constraint" (**BC**)
  - There are also "box constraints" i.e. $x_1 \geq 0, x_2 \geq 0$, more on these later
- $u(x_1, x_2)$ is the objective function

To solve with `R` using the `optim` function, rewrite as a minimization problem:

$$\min_{x_1 \geq 0, x_2 \geq 0} -u(x_1, x_2) \text{ s.t. } x_1 p_1 + x_2 p_2 \leq w$$

The `optim` function can only solve unconstrained optimization problems

- Need to rewrite problem using the **BC**
- Turns **UMP** from a constrained optimization problem into an unconstrained problem.

To do this, must discuss $u(x_1, x_2)$ and its properties.

Utility Function and Prefenences

The utility function $u(x_1, x_2)$ is said to summarize the consumer's preferences

- That is, (x_1, x_2) is preferred to $(\tilde{x}_1, \tilde{x}_2)$ if and only if $u(x_1, x_2) \geq u(\tilde{x}_1, \tilde{x}_2)$

Can take many different forms depending upon preferences between x_1 and x_2

- Cobb-Douglas: $u(x_1, x_2) = x_1^{\alpha_1} x_2^{\alpha_2}$ where $\alpha_1, \alpha_2 > 0$ and $\alpha_1 + \alpha_2 = 1$
- Quasi-linear: $u(x_1, x_2) = x_1 + v(x_2)$ where $v(x)$ is some increasing function e.g. $\sqrt{x}$
- (Perfect) Substitutes: $u(x_1, x_2) = \alpha_1 x_1 + \alpha_2 x_2$ where $\alpha_1, \alpha_2 > 0$
- (Perfect) Complements: $u(x_1, x_2) = \min\{\alpha_1 x_1, \alpha_2 x_2\}$ where $\alpha_1, \alpha_2 > 0$

Choice depends upon what is being modeled.

- Cobb-Douglas: Consumption of all goods increases proportionally with income
- Quasi-linear: Spending a fixed amount on one good and all other income on the other
- Substitutes: The goods can be substituted perfectly in a fixed ratio
- Complements: Must have more of all goods in a fixed ratio to be better off

$u(x_1, x_2)$ is "monotonic" in good j if having more of good j is always better

- $u(x_1 + \varepsilon, x_2) > u(x_1, x_2)$ for $\varepsilon > 0$

Budget Constraint

Again, the following is known as the budget constraint (**BC**):

$$x_1 p_1 + x_2 p_2 \leq w$$

In words, this says a consumer cannot spend more than her income

- $x_j p_j$ is how much she spends on good j
- Doesn't have to spend all income but doesn't value leftover income

The **BC** will "bind" if $u(x_1, x_2)$ is (strictly) monotonic in at least one good

- That is, the consumer spends all her income: $x_1 p_1 + x_2 p_2 = w$
- Can solve **BC** for either x_1 or x_2; we'll always choose x_1 today

$$x_1 = \frac{w - x_2 p_2}{p_1}$$

Can now rewrite **UMP** as an unconstrained choice of only x_2:

$$\min_{0 \leq x_2 \leq \frac{w}{p_2}} -u\left(\frac{w - x_2 p_2}{p_1}, x_2\right)$$

Rewriting **UMP**

Using $x_1 = \frac{w - x_2 p_2}{p_1}$, can rewrite **UMP** as an unconstrained choice of only x_2

$$\min_{0 \leq x_2 \leq \frac{w}{p_2}} -u\left(\frac{w - x_2 p_2}{p_1}, x_2\right)$$

- Notice there are still constraints on x_2, that is $0 \leq x_2 \leq \frac{w}{p_2}$
- These are "box constraints" as they form a box around the possible values of x's
- `optim` can handle box constraints

The box constraints come from the following reasoning: The consumer could,

1. Not buy any x_2 at all, so $x_2 = 0$
2. Buy only x_2, so $x_1 = 0$ and all money is spent on $x_2 = \frac{w}{p_2}$ (see **BC**)
3. Buy any amount of x_2 between 0 and $\frac{w}{p_2}$

So it must be that

$$0 \leq x_2 \leq \frac{w}{p_2}$$

Solving Consumer's Problem

Using the methods developed in 410, could solve the **UMP** using calculus to find a formula for the choice of x_1 and x_2

- We denote the optimal amounts of x_1 and x_2 as x_1^* and x_2^*
- Cobb-Douglas: $x_1^* = \alpha_1 \frac{w}{p_1}$, $x_2^* = \alpha_2 \frac{w}{p_2}$
- Quasi-linear: $x_2^* = \left(\frac{p_1}{2p_2}\right)^2$, $x_1^* = \frac{w - x_2^* p_2}{p_1}$ if $v(x) = \sqrt{x}$ and w is "large enough"

Instead, let's use numerical optimization to do it and review the function `optim`

- par: "Initial values for the parameters to be optimized over."
  - i.e. initial guesses for $\boldsymbol{x}^*$
- fn: "A function to be minimized (or maximized), with first argument the vector of parameters over which minimization is to take place. It should return a scalar result."
  - i.e. the objective function f
- method: "The method to be used." More on this later
- lower, upper: lower and upper bounds for the values of `par`
  - These are for box constraints!

Solve Cobb-Douglas Numerically

```r
## ---- Utility functions
U_CD = function(x,alpha){ # Cobb-Douglas
  alpha = alpha/sum(alpha) # normalize alpha so that they sum to 1
  prod(abs(x)^alpha) }

## ---- Objective function to minimize
x1Solve = function(x2,ps,w){(w-x2*ps[2])/ps[1]}

f_opt = function(x2,ps,w,U){
  x1 = x1Solve(x2,ps,w)   # use BC to solve for x1 as a function of x2, ps, w
  -U(c(x1,x2))            # get negative U(x1,x2) to solve problem via minimize
}

## ---- Set parameters and solve UMP
w        = 100           # set income
ps       = c(2,5)        # set prices
alphas   = c(1/3,2/3)    # set alphas
x2_guess = (0+w/ps[2])/2 # set first guess of x2 to be midpoint of [0,w/p2]

optoutCD = optim(x2_guess,fn=f_opt,
                 ps=ps,w=w,U=function(x){U_CD(x,alphas)},
                 method="Brent",lower=0,upper=w/ps[2])
```

Solve Cobb-Douglas Numerically

Before moving on, lets look at the output from `optim`

```
optoutCD
```

```
## $par
## [1] 13.33333
##
## $value
## [1] -14.3629
##
## $counts
## function gradient
##       NA       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

`$par` is the (hopefully) optimal value of x_2, `$value` is the value of fn(par), and `$convergence` tells us whether it successfully "converged"

Solve Cobb-Douglas Numerically

Let's look at the optimal bundle

```
x2star = optoutCD$par # store optimal value of x2
xOptCD = c("x1_star"= x1Solve(x2star,ps,w), # calc optimal x1 using BC
           "x2_star"= x2star)               # add optimal x2
xOptCD
```

```
##  x1_star  x2_star
## 16.66667 13.33333
```

Let's compare the solution when using `optim` to the solution learned in 410

```
alphas*w/ps # using CD formula from 410
```

```
## [1] 16.66667 13.33333
```

Same result!

Solve Quasi-Linear Numerically

```
U_QL = function(x,qlid=1,v=function(x){sqrt(prod(x))}) {  # Quasi-linear
  sum(x[qlid])+v(x[-qlid])}

## ---- Set parameters and solve UMP
w        = 100           # set income
ps       = c(5,2)        # set prices
x2_guess = (0+w/ps[2])/2 # set first guess of x2 to be midpoint of [0,w/p2]

optoutQL = optim(x2_guess,fn=f_opt,
                 ps=ps,w=w,U=function(x){U_QL(x)},
                 method="Brent",lower=0,upper=w/ps[2])
```

Solve Quasi-Linear Numerically

```
x2star = optoutQL$par # store optimal value of x2
xOptQL = c("x1_star"= x1Solve(x2star,ps,w), # calc optimal x1 using BC
           "x2_star"= x2star)               # add optimal x2

xOptQL
```

```
## x1_star x2_star
## 19.3750  1.5625
```

Now let's compare this answer with the answer using the 410 formulas

```
x2_410 = (ps[1]/(ps[2]*2))^2    # solve x2 using 410 formula
c(x1Solve(x2_410,ps,w), x2_410) # optimal bundle with 410 formulas
```

```
## [1] 19.3750  1.5625
```

Again, same result! Let's do a few more things

Solve Demand for K Goods

Can easily generalize our good to have more than two goods

```
## ---- Function to solve for x1
x1Solve = function(xsm1,ps,w){(w-sum(xsm1*ps[-1]))/ps[1]}

## ---- Objective function to minimize
f_opt = function(xsm1,ps,w,U){
  # xsm1 is "x's without (minus) good 1 (first element)"
  # ps is price of the goods (including good 1), so length(xsm1)==length(ps)+1
  # w is income and U is utility function
  x1 = x1Solve(xsm1,ps,w) # use BC to solve for x1
  -U(c(x1,xsm1))          # get negative U(x1,x2)
}

## ---- Set parameters and solve UMP for Cobb Douglas
w      = 100            # set income
ps     = c(5,2,3)       # set prices
alphas = c(1/7,2/7,4/7) # set alphas
x_guess = rep(1,2)      # set first guess of xsm1

optoutCD3 = optim(x_guess,fn=f_opt,
                  ps=ps,w=w,U=function(x){U_CD(x,alphas)},
                  method="L-BFGS-B",lower=rep(0,2),upper=w/ps[-1])
```

Solve Cobb-Douglas: K Goods

Let's look at the optimal bundle

```
xstar = optoutCD3$par # store optimal value of x2
xOptCD3 = c(x1Solve(xstar,ps,w), # calc optimal x1 using BC
              xstar)               # add optimal x2
names(xOptCD3) = paste0("x",1:length(ps),"_star")
xOptCD3
```

```
##   x1_star   x2_star   x3_star
##  2.857143 14.285714 19.047619
```

Let's compare the solution when using `optim` to the solution learned in 410

```
alphas*w/ps # using CD formula from 410
```

```
## [1]  2.857143 14.285714 19.047619
```

Again, same result! Well... almost. We will discuss why it's not exactly the same later.

Demand Functions

$x_1^*(p_1, p_2, w)$ and $x_2^*(p_1, p_2, w)$ are demand *functions* given prices and income.

Lets write a these functions in `R`

```
SolveDemand = function(ps,w,U){
  # ps are prices, w is income, and U is the utility function
  ## ---- Set parameters for optimization problem
  Np           = length(ps) # set number of prices
  methd        = "L-BFGS-B" # initialize method
  methd[Np==2] = "Brent"    # if Np==2 (so only solving one good), method is Brent
  ubs = w/ps[-1]            # set upper bounds on box constraints
  lbs = rep(0,Np-1)         # set lower bounds on box constraints

  ## ---- Solve optimization problem
  optout = optim(rep(1,Np-1),fn=f_opt, ps=ps,w=w,U=U,
                 method=methd,lower=lbs,upper=ubs)

  ## ---- Format output
  xOpt   = c(x1Solve(optout$par,ps,w),optout$par) # create optimal x bundle
  names(xOpt) = paste0("x",1:Np)          # set names of xOpt
  xOpt
}
```

Demand Functions: Cobb Douglas

Let's test these demand functions with Cobb-Doug:

```
SolveDemand(c(5,2,3),100,U=function(x){U_CD(x,c(1/7,2/7,4/7))})
```

```
##        x1        x2        x3
##  2.857143 14.285714 19.047619
```

```
SolveDemand(c(5,2,3),200,U=function(x){U_CD(x,c(1/7,2/7,4/7))})
```

```
##        x1        x2        x3
##  5.714286 28.571426 38.095239
```

```
SolveDemand(c(2,5,4),100,U=function(x){U_CD(x,c(1/7,2/7,4/7))})
```

```
##        x1        x2        x3
##  7.142875  5.714309 14.285676
```

```
SolveDemand(c(5,2,3,4),100,U=function(x){U_CD(x,c(1/9,2/9,4/9,2/9))})
```

```
##        x1        x2        x3        x4
##  2.222218 11.111108 14.814909  5.555492
```

Demand Functions: Quasi-linear

Let's test these demand functions:

```
SolveDemand(c(5,2),100,U=function(x){U_QL(x)})
```

```
##       x1       x2
## 19.3750  1.5625
```

```
SolveDemand(c((5/2)*75,75),100,U=function(x){U_QL(x)})
```

```
##            x1            x2
## 1.855565e-08 1.333333e+00
```

```
SolveDemand(c(5,2,3),100,U=function(x){U_QL(x)})
```

```
##         x1         x2         x3
## -20.00000  50.00000  33.33333
```

```
SolveDemand(c(5,2,3,5),100,U=function(x){U_QL(x,qlid=c(1,4))})
```

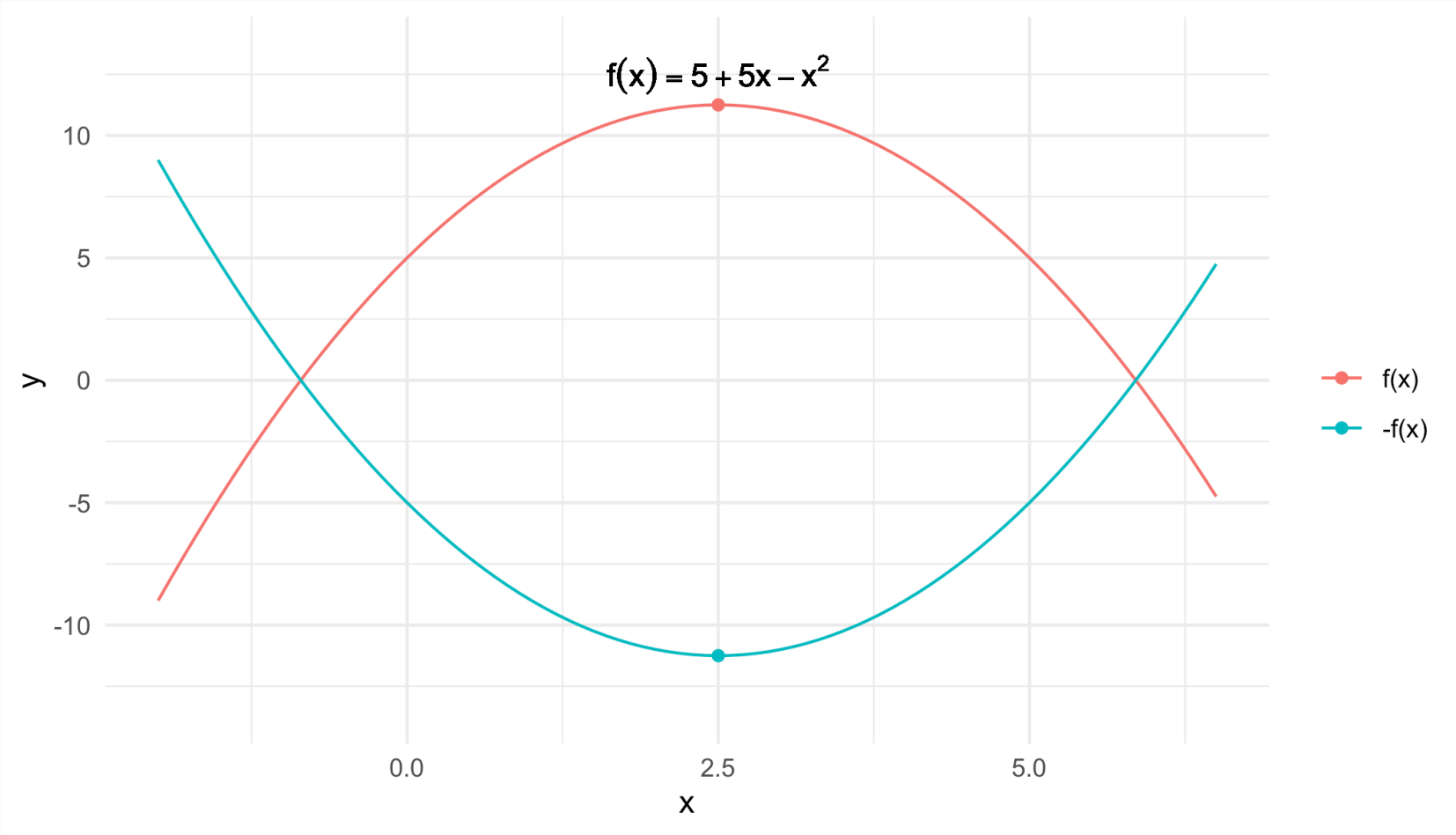
```
##         x1         x2         x3         x4
## -21.00000  50.00000  33.33333   1.00000
```

Math Review

(Some important calculus concepts)

Math Review: Optimization

What makes a function special at the max or min?



Math Review: Optimization

The function f is "flat" around x^* i.e. the derivative of the function f is 0: $f'(x^*) = 0$

Math Review: Derivative

- The derivative of a function tells you about it's rate of change.
  - "Instantaneous slope"
- We denote the derivative of f with respect to x a few different ways:
  - $f'(x)$ or $\frac{df}{dx}$
  - Note that the derivative of a function is also a function: $g(x) = f'(x)$
- The derivative tells us more than just the slope:
  - $f'(x) > 0 \rightarrow$ function is increasing
  - $f'(x) < 0 \rightarrow$ function is decreasing
  - $f'(x) = 0 \rightarrow$ function is not changing i.e. "critical point"
- The formal definition of a derivative is as follows:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

  - We will actually be using this definition!
- Note: Not all functions have a derivative everywhere (e.g. $f(x) = |x|$).
- All intuition here will be for differentiable functions.
  - Optimization methods exist for non-differentiable functions

Math Review: Optimization

Solving an optimization problem is finding the values of x that solve the following equation

$$g(x) = 0$$

where $g(x) = f'(x)$ and $f(x)$ is the function we want to optimize

The x values that solve $g(x) = 0$ are referred to as "critical points" and we denote them x^*

- A function can have many critical points
- A critical point is not always a solution to an optimization problem
  - Critical points can indicate a "local maximum," "local minimum," or "saddle point."
- Think of critical points as "candidate points."

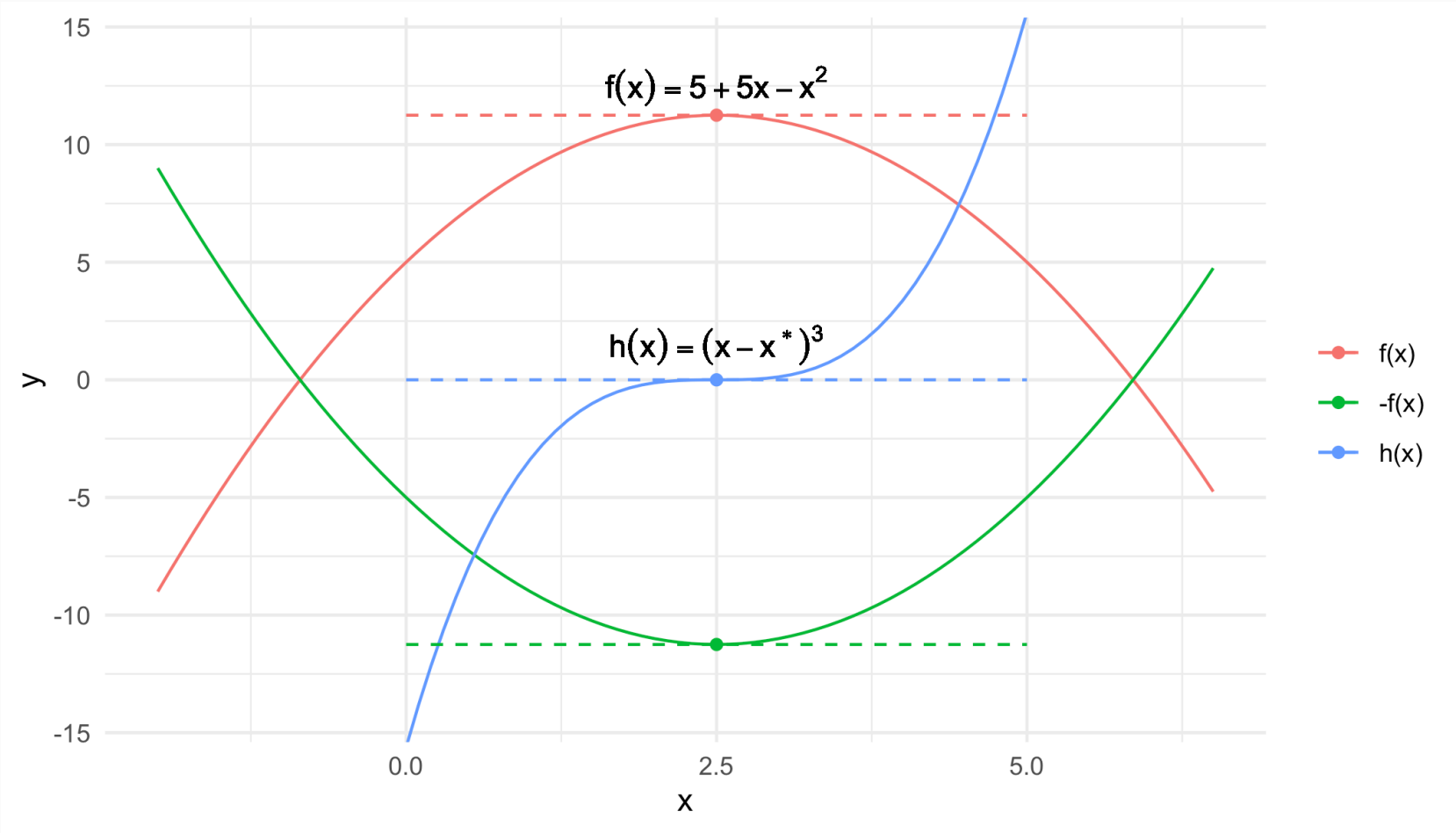
To find critical points numerically, need to be able to

1. Calculate (or approximate) the derivative $f'(x)$
2. Solve the equation $g(x) = f'(x) = 0$

We will discuss numerical methods to do both

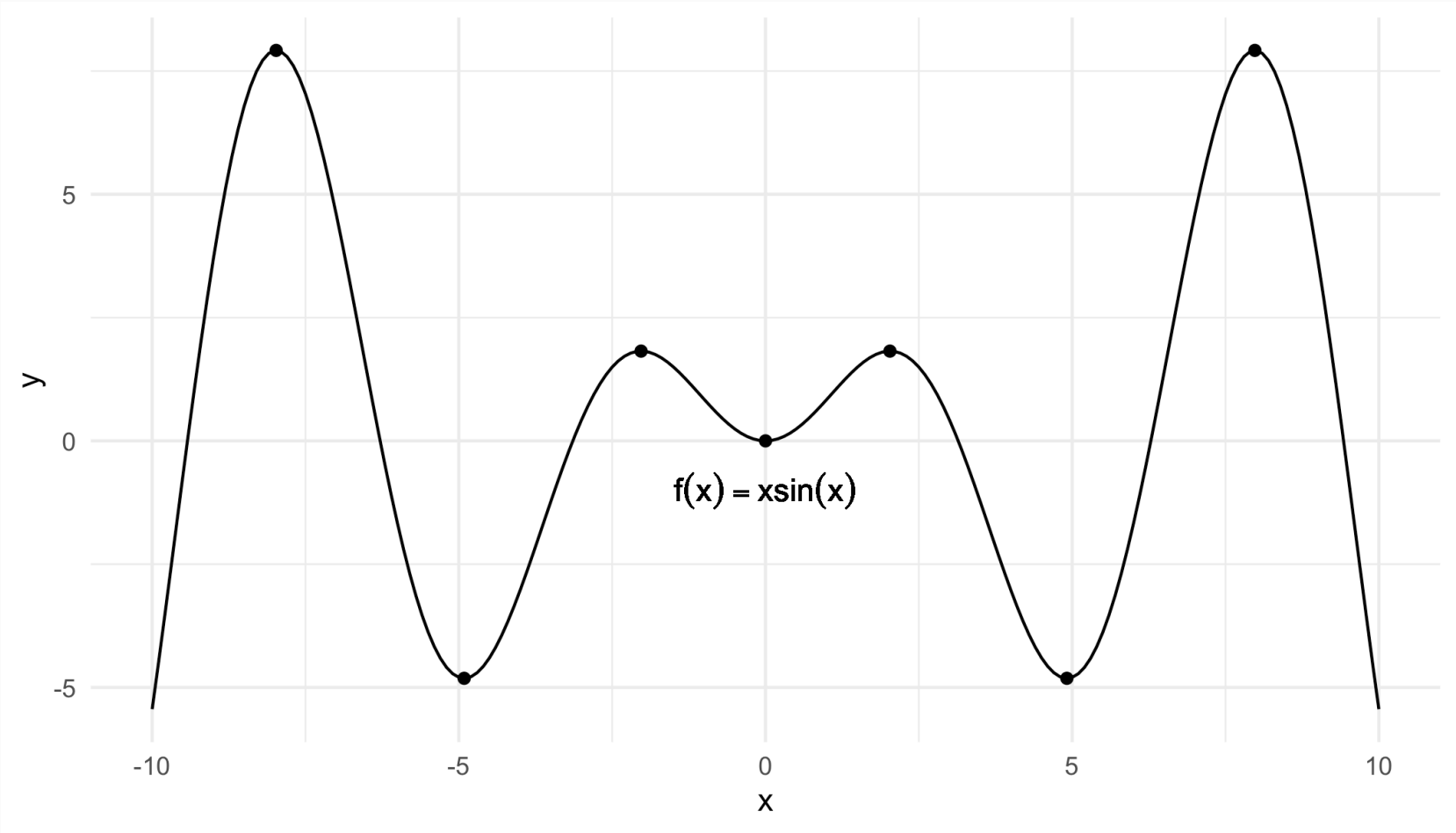
- The two are related!

Critical Point Meaning



x^* can indicate a maximum $(f(x^*))$, minimum $(-f(x^*))$, or "saddle point" $(h(x^*))$

Multiple Critical Points



The function $f(x) = x\sin(x)$ has many critical points and many local maxima and minima!

Selecting Critical Points

Need to determine which critical point results in the ***global minimum*** (or maximum)

Can determine if maximum, minimum, or saddle point by checking second derivative

- Second derivative is the "derivative of the derivative"
- Denoted $f''(x)$

Second Derivative Test

1. $f''(x^*) > 0$ indicates a minimum
2. $f''(x^*) < 0$ indicates a maximum
3. $f''(x^*) = 0$ indicates a saddle point

A few notes:

- In real-world problems, second derivative can be hard to calculate
- Second derivative test still doesn't guarantee a global optimum
- Many optimization algorithms have been developed to resolve these issues

Numerical Derivatives

Approximating Derivatives

While getting the formula for $f'(x)$ is ideal due to accuracy and computational demand, sometimes it is not feasible to do so.

- Instead, we can approximate the derivative!

Remember the definition of the derivative; as h "gets small", the expression on RHS "approaches" a unique value $f'(x)$

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

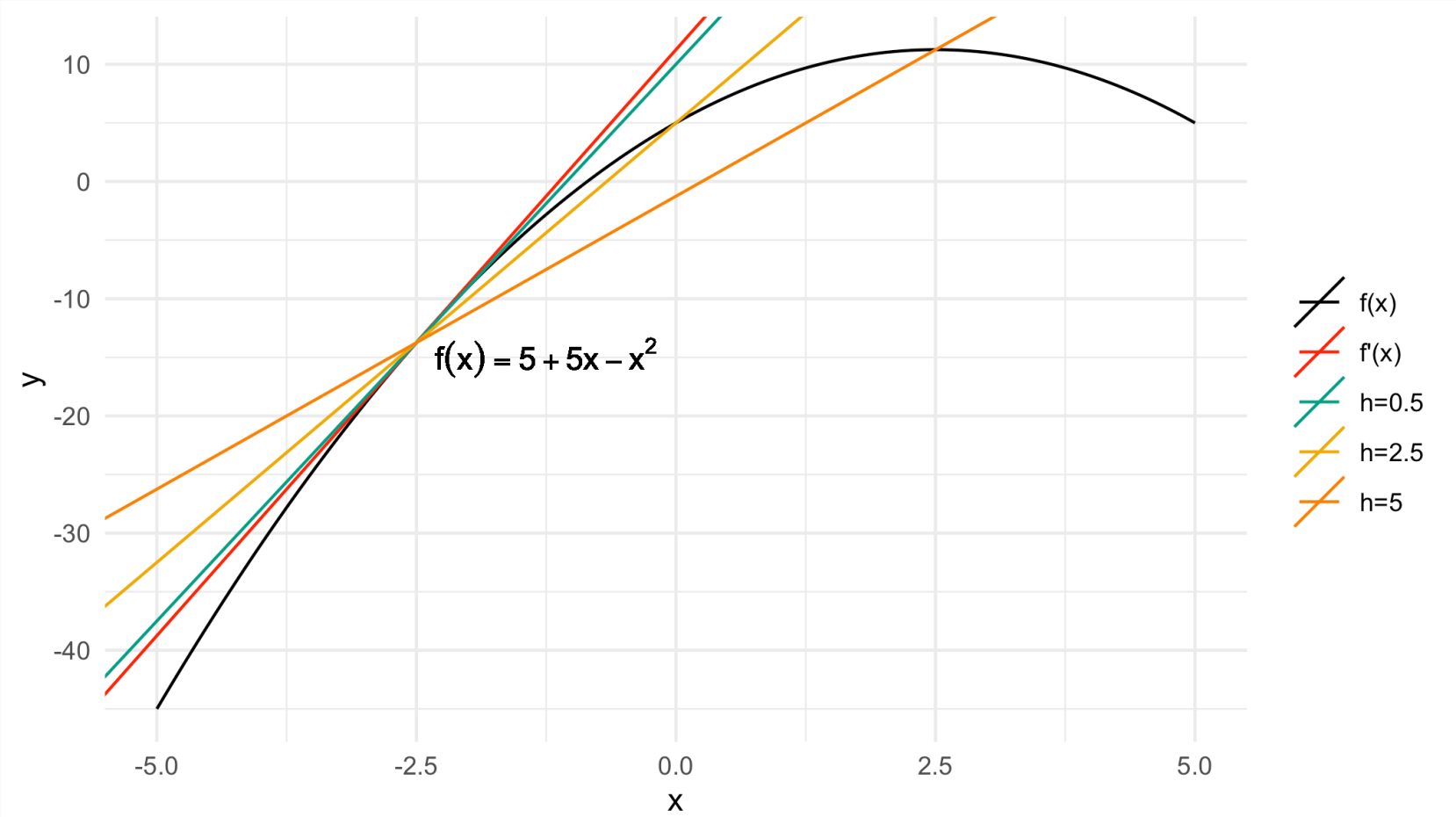
So for a "small" $h > 0$,

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}$$

- This is the "forward differencing" approach; we can also "backward difference"

$$f'(x) \approx \frac{f(x) - f(x-h)}{h}$$

Approximating Derivatives



As $h \to 0$, $\frac{f(x+h)-f(x)}{h} \to f'(x)$

Approximating Derivatives (Cont.)

Forward and backwards differencing work, but we can get a more accurate approximation using the ***same*** h

Instead of favoring one side, we are approximating the derivative equally around a "neighborhood" of x.

Center difference:

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

- Can show that the error generator by center differencing is smaller than foward or backwards differencing.

There are even more accurate methods that are bit more computationally demanding.

- For those curious, look up Richardson's Extrapolation Method.

Example

Let's keep things simple and look at the derivative of $f(x) = x^2$. The analytical expression of the derivative is $f'(x) = 2x$.

```
f            = function(x){x^2}         #make function f(x)
fp           = function(x){2*x}         #make f'(x)
xs           = seq(0,2,0.5)             #store x's
h            = 0.01                     #store step size
der          = fp(xs)                   #calc actual derivatives
center_der   = (f(xs+h)-f(xs-h))/(2*h) #calc center differenced
forward_der  = (f(xs+h)-f(xs))/(h)      #calc forward differenced
backward_der = (f(xs)-f(xs-h))/(h)      #calc backwards differenced
deriv_data   = data.table("f'(x)"=der,"Center"=center_der,
                          "Foward"=forward_der,"Backward"=backward_der)
deriv_data
```

```
##    f'(x) Center Foward Backward
##    <num>  <num>  <num>    <num>
## 1:     0      0   0.01    -0.01
## 2:     1      1   1.01     0.99
## 3:     2      2   2.01     1.99
## 4:     3      3   3.01     2.99
## 5:     4      4   4.01     3.99
```

Derivatives in Multiple Dimensions

If instead of $f(x)$, we have something like $f(x_1, x_2, \ldots, x_n) = f(\boldsymbol{x})$, not much changes.

The derivative will be a vector called the gradient denoted

$$\nabla f(\boldsymbol{x}) = \Big(\frac{\partial f(\boldsymbol{x})}{\partial x_1}, \ldots, \frac{\partial f(\boldsymbol{x})}{\partial x_n} \Big)$$

The gradient at a point $\boldsymbol{x}$ is interpreted as a direction if moving in each dimension holding fixed the other values of $\boldsymbol{x}$

To approximate each (partial) derivative, follow the same approach.

The partial derivative of f in the x_i dimension at $\boldsymbol{x} = (x_1, \ldots, x_n)$ is,

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i} \approx \frac{f(x_1, \ldots, x_i + h, \ldots, x_n) - f(x_1, \ldots, x_i - h, \ldots, x_n)}{2h}$$

- Do this for all inputs and then collect them in a vector.
- Note: This requires evaluating the function $2n$ times.

Example

Suppose that $f(x,y) = x^2 + \sin(y)$. $\frac{\partial f}{\partial x} = 2x$ and $\frac{\partial f}{\partial y} = \cos(y)$

```
f            = function(x){x[1]^2+sin(x[2])}        #make function f(x)
fp           = function(x){c(2*x[1],cos(x[2]))}     #make gradient of f(x)
h            = 0.01                                 #store step size
fp(c(1,0.5))
```

```
## [1] 2.0000000 0.8775826
```

```
c((f(c(1+h,0.5))-f(c(1-h,0.5)))/(2*h),(f(c(1,0.5+h))-f(c(1,0.5-h)))/(2*h))
```

```
## [1] 2.0000000 0.8775679
```

```
c((f(c(1+h,0.5))-f(c(1,0.5)))/h,(f(c(1,0.5+h))-f(c(1,0.5)))/h)
```

```
## [1] 2.0100000 0.8751708
```

```
c((f(c(1,0.5))-f(c(1-h,0.5)))/h,(f(c(1,0.5))-f(c(1,0.5-h)))/h)
```

```
## [1] 1.990000 0.879965
```

Solving Equations Numerically

(i.e. Finding Roots)

Solving for Roots

Numerical optimization comes down to solving the following equation:

$$g(x) = 0$$

where $g(x)$ is the derivative of a function $f(x)$ or $g(x) = f'(x)$

Need methods to solve equations numerically

- There exist many methods (e.g. bisection), but we will focus on Newton's Method.

Suppose we have a function $f(x)$ and we want to find a value of x such that $f(x) = 0$.

- The x values that solve $f(x) = 0$ are known as the "roots" of f.

Idea Behind Newton's Method:

- Start with an initial guess x_0
- Use the derivative of f to tell us which direction to move in and update x_0 to x_1.
- Continue updating x_n to x_{n+1} until $|f(x_{n+1})| \leq \varepsilon$

Newton's Method: Intuition

$$f'(x_0) \approx \frac{f(x) - f(x_0)}{x - x_0}$$

Remember that we want $f(x) = 0$, so

$$f'(x_0) \approx \frac{0 - f(x_0)}{x - x_0}$$

$$x - x_0 \approx -\frac{f(x_0)}{f'(x_0)}$$

$$x \approx x_0 - \frac{f(x_0)}{f'(x_0)}$$

Steps for Newton's Method:

1. Choose x_0 i.e. an initial guess for the x that solves $f(x) = 0$
2. Update guess $x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}$ and check if $|f(x_1)| \leq \varepsilon$
3. If not, continue updating $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ until $|f(x_{n+1})| \leq \varepsilon$

Newton's Method: Square-Root Example

Let $x = \sqrt{a}$ i.e. x is the square-root of some number $a > 0$

- What is x for a given a?
- x solves $x^2 = a$ or $x^2 - a = 0$ i.e. x solves $f_a(x) \equiv x^2 - a = 0$

How to solve with Newton's Method?

1. Make an initial guess of the solution x_0 (could be $x_0 = a$)
2. Update $x_{n+1} = x_n - \frac{f_a(x_n)}{f'_a(x_n)} \equiv x_n - \frac{x_n^2 - a}{2x_n} \equiv \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$ for $n = 0$
3. If $\left|f_a(x_{n+1})\right| < \varepsilon$, $x_{n+1} = \sqrt{a}$, otherwise continue updating

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$$

Newton's Method: Square-Root Example

```
fa = function(x,a) x^2-a
a  = 2
xn = a
eps = 1e-15

xnp1 = 0.5*(xn+a/xn)
while(abs(fa(xnp1,a))>eps){
  xn   = xnp1
  xnp1 = 0.5*(xn+a/xn)
}
c("Newtons"=xnp1,"sqrt_fun"=sqrt(a))
```

```
##  Newtons sqrt_fun
## 1.414214 1.414214
```

Newton's Method: General Function

```
FindRootsNewton = function(x0,f, ... ,eps=1e-15,MaxIt = 10000,h=0.01){

  f2 = function(x){f(x, ... )}; n = 1; xn = x0
  gr2 = function(x) (f2(x+h)-f2(x-h))/(2*h)
  if(abs(f2(x0))<eps){
    x0
  }else{
  xnp1 = xn - f2(xn)/gr2(xn)
  while(abs(f2(xnp1)) ⩾ eps & n ⩽ MaxIt){
    n    = n+1; xn = xnp1      # advance counter and update xn with xnp1
    xnp1 = xn - f2(xn)/gr2(xn) # update xnp1
  }
  warntxt = "Max iterations reached."
  warntxt = paste(warntxt,"|f(xnp1)|=",abs(f2(xnp1)))
  if(abs(f2(xnp1)) ⩾ eps) warning(warntxt)
  xnp1 # return xnp1
  }
}
```

Newton's: Square-Root Example

```
## --- create grid of as and a function fapp to pass to sapply
as = seq(0,10,0.5)
fsqrta = function(xa) FindRootsNewton(x0=xa,f=function(x,a){x^2-a},a=xa)
## --- calc sqrt(as) using Newton's Method and built-in sqrt function
sqrtnewt = sapply(as,fsqrta); sqrtfun  = sqrt(as)
```

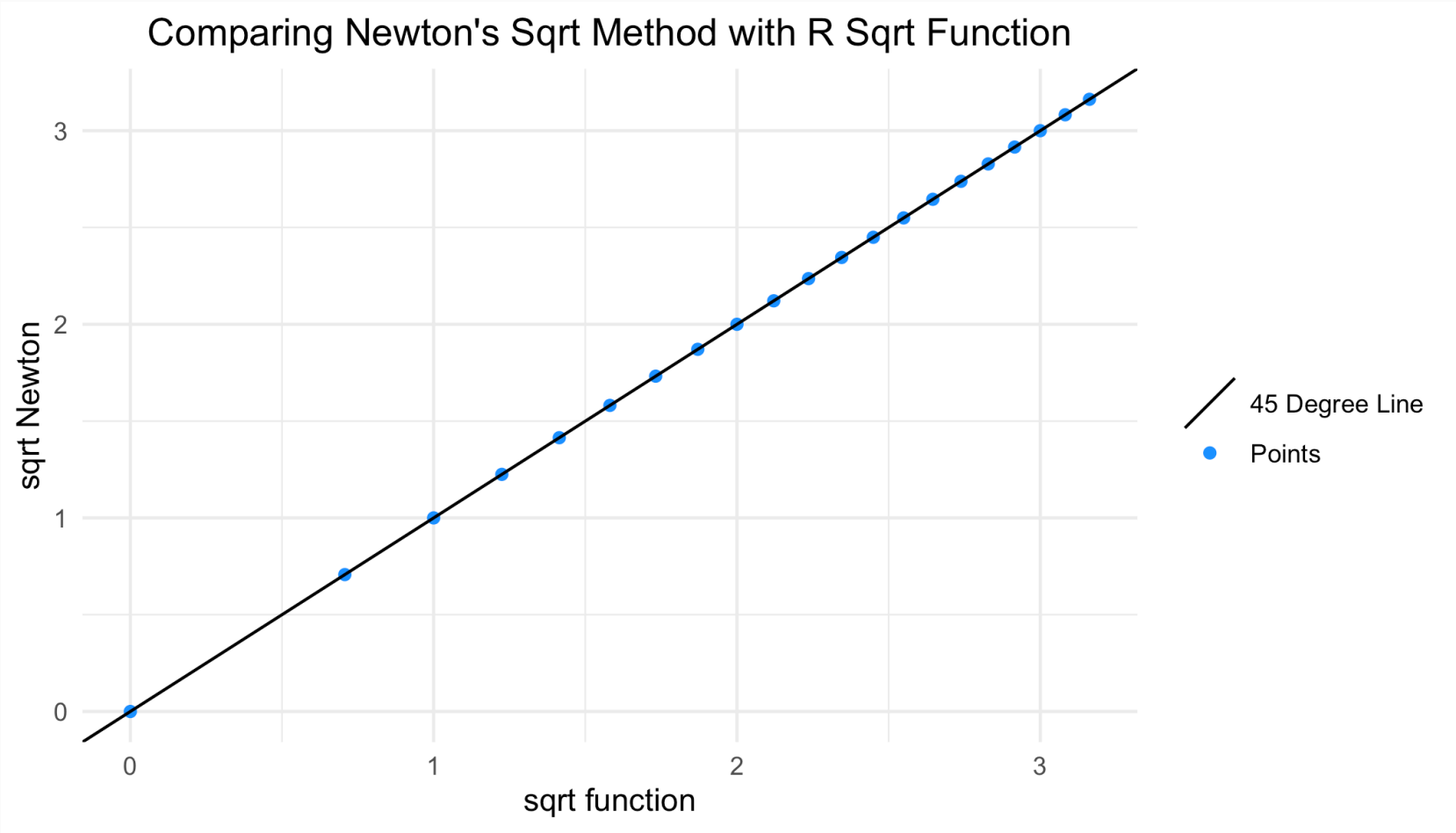
```
## Warning in FindRootsNewton(x0 = xa, f = function(x, a) {: Max iterations
## reached. |f(xnp1)|= 1.77635683940025e-15
## Warning in FindRootsNewton(x0 = xa, f = function(x, a) {: Max iterations
## reached. |f(xnp1)|= 1.77635683940025e-15
## Warning in FindRootsNewton(x0 = xa, f = function(x, a) {: Max iterations
## reached. |f(xnp1)|= 1.77635683940025e-15
```

```
## --- give names to elements of sqrtnewt/sqrtfun & return summary stats
names(sqrtnewt) = paste0("a=",as); names(sqrtfun)=names(sqrtnewt)
summary(sqrtnewt-sqrtfun)
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 0.000e+00 0.000e+00 0.000e+00 4.229e-17 0.000e+00 4.441e-16
```

Values for $\sqrt{a}$ are very similar for Newton's Method and the R square-root function

Newton's: Square-Root Example



Numericl Optimization

A Naive Approach: Grid Search

- If the dimension of x is small enough, you can make a grid of points to search on and see which set of values minimizes f.
- While this seems simple, in practice, you rarely want to do it.
- If x has more than two dimensions, must search many points.
  - Particularly bad of f takes awhile to run.
- Must make grid fine enough so that you're not skipping over too many points, but too fine runs into the same problem as before.

```
x1 = seq(0,5,0.05)
x2 = seq(0,5,0.05)
x3 = seq(0,5,0.05)
nrow(expand.grid(x1,x2))
```

```
## [1] 10201
```

```
nrow(expand.grid(x1,x2,x3))
```

```
## [1] 1030301
```

Newton's Method for Optimization

First, note that if we don't have a formula for $f'(x)$, must approximate it using the methods we already discuss.

Second, let's return to our original problem: solve $g(x) = 0$ where $g(x) = f'(x)$ for some function f.

This is essentially a root finding problem, so can use Newton's Method:

$$x_{n+1} = x_n - \frac{g(x_n)}{g'(x_n)}$$

Since $g(x) = f'(x)$, this means our method is the same as

$$x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$$

Newton's Method Example

```
NewtonOpt = function(x0,f,... ,h=0.01,eps=1e-15,MaxIt=10000){
  ## --- Set initial x, counter, g(x)=f'(x), and g'(x)=f''(x)
  xn = x0; n = 1
  g  = function(x){(f(x+h,... )-f(x-h,... ))/(2*h)}
  gp = function(x){(g(x+h)-g(x-h))/(2*h)}

  ## ---- Code for Newton's Method
  xnp1 = xn - g(xn)/gp(xn) # update x_n to x_{n+1}
  while(abs(f(xnp1)) ⩾ eps & n ⩽ MaxIt){
    xn = xnp1; n = n + 1 # update initial guess and counter
    xnp1 = xn - g(xn)/gp(xn) # update x_n to x_{n+1}
  }
  warntxt = "Max iterations reached."
  warntxt = paste(warntxt,"|f'(xnp1)|=",abs(g(xnp1)))

  ## ---- Test for Convergence
  if(abs(f(xnp1))⩾eps){
    warning(warntxt)
  }
  xnp1 # return xn1
}
```

Newton Optimization: Our Quadratic

We've seen $f(x) = 5 + 5x - x^2$ a lot now. Let's minimize it using Newton's Method.

We will use different values for x_0 to see if the initial x_0 matters when minimizing $f(x)$.

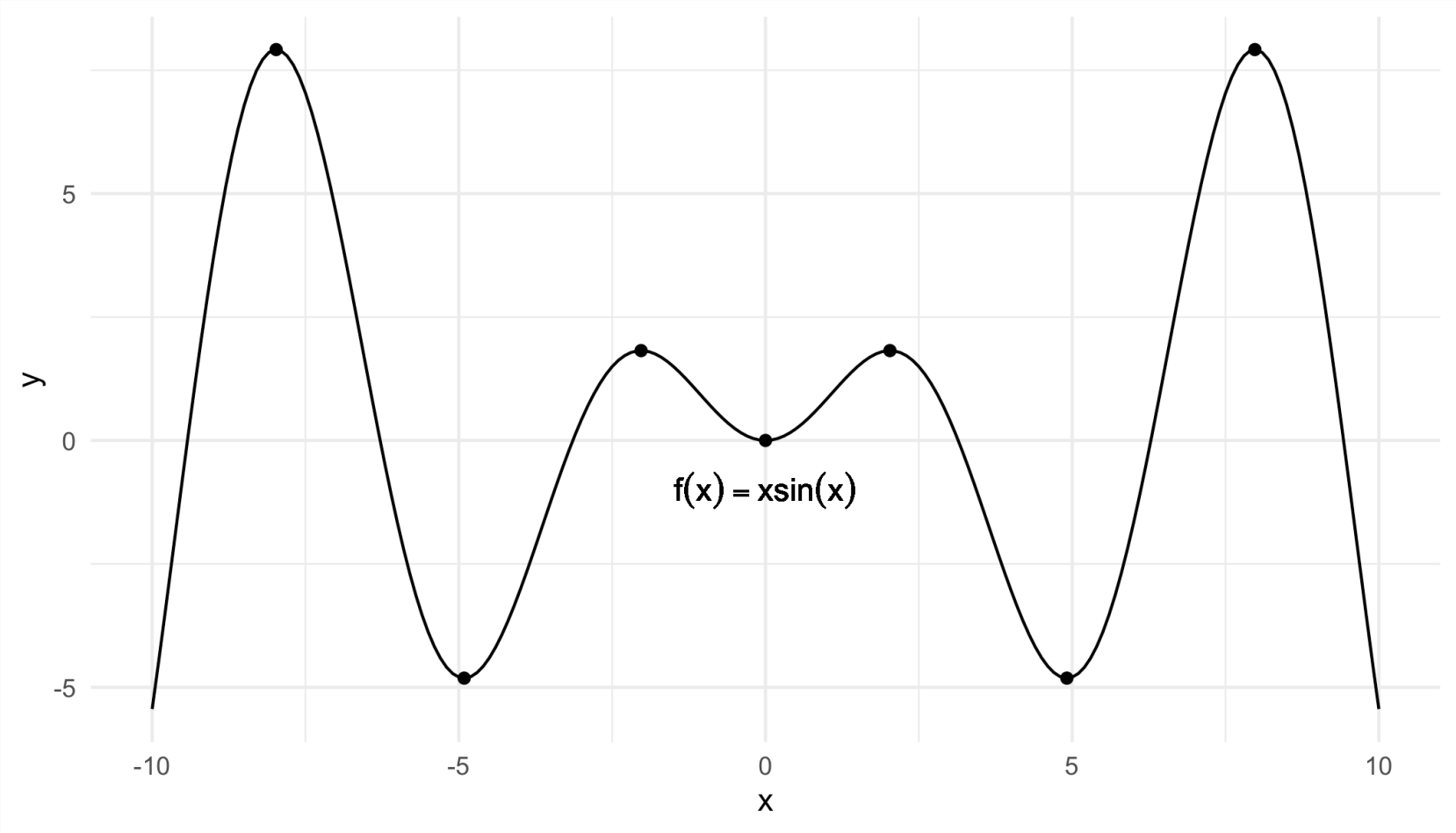
```
## --- Example Using Different x0 values and f(x)=5+5x-x^2
x0s = seq(-5,3,0.5)                              # grid of x0 vals
sapply(x0s,NewtonOpt,f=function(x){5+5*x-x^2}) # min f(x) with each x0
```

```
##  [1] 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5 2.5
```

The starting value for x (that is x_0) does not seem to matter here!

- All values of x_0 result in same x^*
- When might the exact value of x_0 matter?

When Starting Values Matter



Multiple critical points that result in a local minimum!

- Potential critical points are $x^* = 0$, $x^* \approx -4.91$, and $x^* \approx 4.91$

When Starting Values Matter

Create grid of values for x_0 and see which minimum optim finds.

```
fxsin           = function(x){x*sin(x)}          # create f(x)=x*sin(x)
x0s             = seq(-9,9,0.75)                 # create grid of x0's
crtpnts         = sapply(x0s,NewtonOpt,f=fxsin) # min f(x) using each x0
names(crtpnts) = paste0("x0=",x0s)               # give names to crtpnts as x0s
round(crtpnts, digits = 2)                       # round crtpnts for display
```

```
##    x0=-9 x0=-8.25  x0=-7.5 x0=-6.75    x0=-6 x0=-5.25  x0=-4.5 x0=-3.75
##    -7.98    -7.98    -7.98   -11.09    -4.91    -4.91    -4.91   -11.09
##    x0=-3 x0=-2.25  x0=-1.5 x0=-0.75     x0=0  x0=0.75   x0=1.5  x0=2.25
##    -2.03    -2.03    -2.03     0.00     0.00     0.00     2.03     2.03
##     x0=3  x0=3.75   x0=4.5  x0=5.25     x0=6  x0=6.75   x0=7.5  x0=8.25
##     2.03    11.09     4.91     4.91     4.91    11.09     7.98     7.98
##     x0=9
##     7.98
```

Different starting value of x_0 result in different solutions to $f'(x) = 0$!

- Note that it finds critical points that are **not** *local minima*

When Starting Values Matter

What to do? Could plug critical points x^* into f and see which value $f(x^*)$ is the smallest.

```
f       = function(x) x*sin(x)
crtpnts = round(crtpnts,digits=10)
crtpnts = sort(unique(crtpnts))
round(crtpnts,digits=3)
```

```
## [1] -11.086  -7.979  -4.913  -2.029   0.000   2.029   4.913   7.979  11.086
```

9 different critical points!

```
fctpt = f(crtpnts)
vals  = crtpnts[c(which.min(fctpt),which.max(fctpt))]
names(vals) = c("x_star_min","x_star_max")
round(fctpt,digits=2)
```

```
## [1] -11.04   7.92  -4.81   1.82   0.00   1.82  -4.81   7.92 -11.04
```

```
vals
```

```
## x_star_min x_star_max
## -11.085535  -7.978662
```

Reflection

While Newton's Method seems to work pretty well, there are some problems:

- Can find min, max, & saddle points depending upon x_0 if f is not "convex"
  - Minimizing non-convex functions can be very challenging: check out this best practices paper for one model used in my subfield.
  - Can solve problem multiple times, but this might be challenging if time consuming.
- Second derivative is hard to approximate numerically when f has more than one input.
  - If $f(x_1, \ldots, x_n)$, then second derivative is the "Hessian" & is an $n \times n$ matrix.
  - To approximate, must evaluate the objective function $4n^2/2$ times.

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

Methods have been developed to get around the second point & kinda the first.

Quasi-Newton: Gradient Descent

The second derivative only scales the updating process.

- So we don't "learn" too fast or too slow.

Methods have been developed to approximate the Hessian or scale the updating process.

- "Quasi-Newton"

Idea: Simply choose a scaling parameter $\alpha > 0$ to update

$$x_{n+1} = x_n - \alpha \nabla f(x_n)$$

This is the idea behind gradient descent. How to choose α?

- Can pick a fixed value or pick an optimal α:

1. Calculate $\nabla f(x_n)$ and save it.
2. Then choose α_n to minimize $f(x_n - \alpha_n \nabla f(x_n))$.
3. Repeat this step each time you update x_{n+1}.
    - Whether this is beneficial is problem specific.
    - Smaller optimization problem during your larger optimization problem.

Quasi-Newton: BFGS

The limitation of gradient descent is that even if $\nabla f(x)$ is a vector, only one value for α

- Might want to scale each element of $\nabla f(x)$ differently.

Instead, we might be able to approximate the Hessian using previous values of $\nabla f(x_n)$.

This method is called BFGS.

- BFGS stands for the last names of the authors.
- We aren't going to discuss the details.
- Understand that BFGS approximates the second derivative (Hessian) using previous values of the first derivative (gradient) in a special way.

Extensions of BFGS that allow for box-constraints and limited memory of the Hessian approximation: L-BFGS-B.

- B stands for box-constraints
- L stands for limited memory. BFGS uses all previous values of $\nabla f(x_n)$. L-BFGS-B will "forgot" (i.e. not use) older values of $\nabla f(x_n)$.

Derivative Free Methods

There are numerical optimization methods that don't use the derivative in case the objective function is not differentiable.

1. Nelder-Mead
2. Simulated annealing
3. BOBYQA, COBYLA

While these seem safer to use, they typically don't find the solutions as fast and we know less about their "convergence properties."

Avoid these unless the gradient of your objective function is fragile/non-existent.

- Simulated annealing is useful when optimizing certain estimators used in my sub field as the objective function is usually not differentiable everywhere

Which Method?

So... of all these methods, which one to choose?

Problem specific!

- Depends on the properties of f, how long it takes f to run, how many inputs f has, etc.
- Unfortunately, there is no one answer.
- Optimization can make or break a project.

Usually a trade-off between the following three properties:

1. Robustness: Performs well for various problems and starting values.
2. Efficiency: Achieves the solution relatively quickly.
3. Accuracy: Identify a solution with precision, not sensitive to starting values.

Numerical Optimization in R

Numerical Optimization in R

Let's return to our OLS example:

```
x = advert_data$TV
y = advert_data$Sales

f = function(theta) sum( (y - theta[1]-theta[2]*x)^2)

theta0 = c(mean(y),0)
optim_out = optim(par=theta0,fn = f,method="BFGS")
```

Numerical Optimization in R

```
optim_out
```

```
## $par
## [1] 7.03259355 0.04753664
##
## $value
## [1] 2102.531
##
## $counts
## function gradient
##       34        4
##
## $convergence
## [1] 0
##
## $message
## NULL
```

```
coef(lm(Sales~TV,data=advert_data))
```

```
## (Intercept)          TV
##  7.03259355  0.04753664
```

optim Using Other Methods

```
theta0    = c(mean(y),0)
optim_out = optim(par=theta0,fn = f,method="Nelder-Mead")
optim_out
```

```
## $par
## [1] 7.03247736 0.04753491
##
## $value
## [1] 2102.531
##
## $counts
## function gradient
##       75       NA
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Custom Gradients

```r
# requires the numDeriv package
f      = function(theta) sum((y-theta[1]-theta[2]*x)^2) # define f
fgr    = function(theta) numDeriv::grad(f,theta)        # define grad of f
theta0 = c(mean(y),0)                                   # set theta0
optim(par=theta0,fn = f,gr=fgr,method="BFGS")           # solve opt problem
```

```
## $par
## [1] 7.03259355 0.04753664
##
## $value
## [1] 2102.531
##
## $counts
## function gradient
##       34        4
##
## $convergence
## [1] 0
##
## $message
## NULL
```

Multivariate Regression

$$\text{Sales}_i = \alpha + \beta_1 \text{TV}_i + \beta_2 \text{Radio}_i + \beta_3 \text{Newspaper}_i + \varepsilon_i$$

```
x1 = advert_data$TV        # store TV var in data as x1
x2 = advert_data$Radio     # store Radio var in data as x2
x3 = advert_data$Newspaper # store Newspaper var in data as x3

f = function(theta){ # define objective function for multivariate regression
  sum( (y - theta[1] - theta[2]*x1 - theta[3]*x2 - theta[4]*x3)^2  )}

fgrad     = function(theta) numDeriv::grad(f,theta)          # define grad of f
theta0    = c(mean(y),rep(0,3))                              # set theta0
optimBFGS = optim(par=theta0,fn = f,gr=fgrad,method="BFGS") # solve using BFGS
optimNM   = optim(par=theta0,fn = f,gr=fgrad)                # solve with Nelder-Mead
```

Multivariate Regression

```
coef(lm(Sales~TV+Radio+Newspaper,data=advert_data)) # solution from OLS formula
```

```
##  (Intercept)           TV        Radio    Newspaper
##  2.938889369  0.045764645  0.188530017 -0.001037493
```

```
optimBFGS$par # solution using optim with BFGS
```

```
## [1]  2.938889370  0.045764645  0.188530017 -0.001037493
```

```
optimNM$par # solution using optim with Nelder-Mead
```

```
## [1]  2.939224869  0.045758581  0.188544858 -0.001041921
```

Up Next: Application - Simulation