Lecture 6: Optimization

Jacob Adenbaum

University of Edinburgh

Spring 2024

Logistics

- Lectures: cover the material – no code, mostly theory
- Labs: Cover examples and code, work on problem sets. Run through Week 10
- Problem Sets: you have four more to go
- Take Home Exam: we will be announcing the dates shortly!
- Office Hours (Jacob): Fridays 11:00 - 12:00, Location: 31 Buccleuch Place, 2.11

Numerical Optimization is Essential

- Nearly every economics problem that we write down involves some sort of optimization problem
  - Utility maximization: Demand for several goods, consumption vs. savings, leisure vs. consumption,
  - Profit maximization
  - Planners' problem
  - Best response in a game (nash equilibria)
  - etc...
- But optimization shows up in lots of other places!
  - Most (all?) estimators in statistics/econometrics are parameters that solve an appropriate minimization problem
- It shouldn't surprise you that when we can't solve these problems analytically, we turn to a numerical analysis of the problem

Types of Optimization Problems

- Local vs. Global
- Univariate vs. Multivariate
- Linear vs. Nonlinear
- Constrained vs. Unconstrained

Section 1

Unconstrained Optimization: Univariate

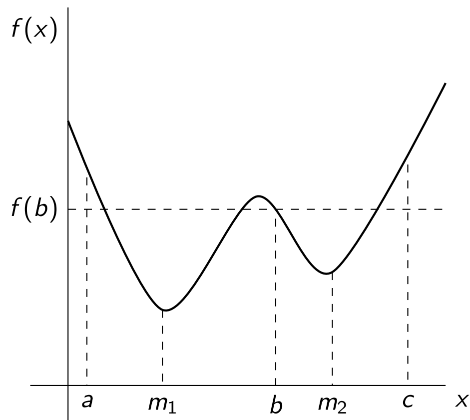
Problem Setup

Suppose we're given some function $f : \mathbb{R} \to \mathbb{R}$, and we're asked to solve:

$$\min_{x \in \mathbb{R}} \; f(x) \tag{1}$$

- We haven't been told anything yet about f.
  - It's not necessarily differentiable
  - We certainly don't have enough to get an analytic solution
- This problem seems simple, but a surprising number of economic problems can be solved with just univariate optimization
- These methods also form the building blocks for some multivariate algorithms, so it is important to have fast methods to solve problems like this

Bracketing Method

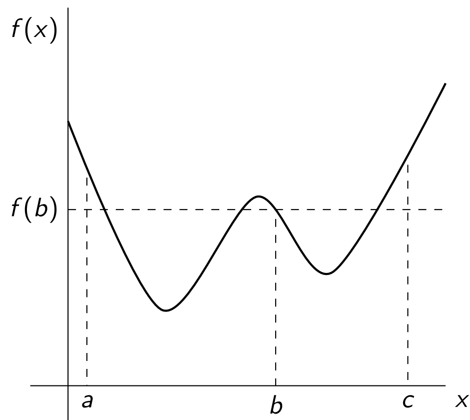


- Suppose that $f : \mathbb{R} \to \mathbb{R}$ is continuous and,
- Suppose we've found points a, b and $c \in \mathbb{R}$ such that

$$f(a) > f(b) \quad \text{and} \quad f(c) > f(b) \tag{2}$$

- We know that somewhere in (a, c) there is a minimum value Why? Well, since f is continuous, and $[a, c]$ is compact, we know a minimum exists, and since $f(b) < f(a)$ and $f(b) < f(c)$, it cannot attain its minimum at the boundary
- Can we refine this to find a minimum?
- Note: ideally, we want to find m_1 since it is the global minimum

Bracketing Method: Algorithm

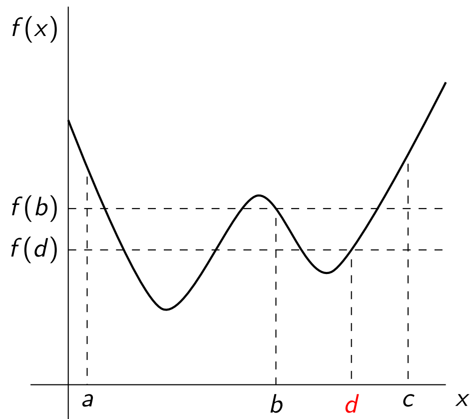


Algorithm 1 Bracketing Algorithm

Require: $a < b < c$ such that $f(a) > f(b)$ and $f(c) > f(b)$

1: **while** $c - a > \epsilon$ **do** ▷ For $\epsilon > 0$
2:     **if** $b - a < c - b$ **then** ▷ Choose a test point
3:         $d \leftarrow (b + c)/2$
4:     **else**
5:         $d \leftarrow (a + b)/2$
6:     **end if**
7:     **if** $d > b$ **then**
8:         **if** $f(d) \geq f(b)$ **then**
9:             $(a, b, c) \leftarrow (a, b, d)$ ▷ d is our new c
10:         **else if** $f(d) < f(b)$ **then**
11:             $(a, b, c) \leftarrow (b, d, c)$ ▷ d is our new candidate
12:         **end if**
13:     **else if** $d < b$ **then**
14:         **if** $f(d) \geq f(b)$ **then**
15:             $(a, b, c) \leftarrow (d, b, c)$ ▷ d is our new a
16:         **else if** $f(d) < f(b)$ **then**
17:             $(a, b, c) \leftarrow (a, d, b)$ ▷ d is our new candidate
18:         **end if**
19:     **end if**
20: **end while**

Bracketing Method: Algorithm

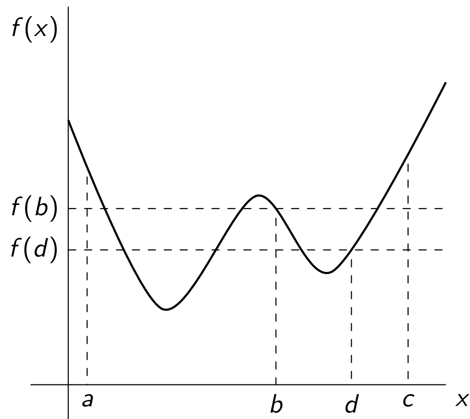


Algorithm 1 Bracketing Algorithm

Require: $a < b < c$ such that $f(a) > f(b)$ and $f(c) > f(b)$

1: **while** $c - a > \epsilon$ **do** ▷ For $\epsilon > 0$
2: **if** $b - a < c - b$ **then** ▷ Choose a test point
3: $d \leftarrow (b + c)/2$
4: **else**
5: $d \leftarrow (a + b)/2$
6: **end if**
7: **if** $d > b$ **then**
8: **if** $f(d) \geq f(b)$ **then**
9: $(a, b, c) \leftarrow (a, b, d)$ ▷ d is our new c
10: **else if** $f(d) < f(b)$ **then**
11: $(a, b, c) \leftarrow (b, d, c)$ ▷ d is our new candidate
12: **end if**
13: **else if** $d < b$ **then**
14: **if** $f(d) \geq f(b)$ **then**
15: $(a, b, c) \leftarrow (d, b, c)$ ▷ d is our new a
16: **else if** $f(d) < f(b)$ **then**
17: $(a, b, c) \leftarrow (a, d, b)$ ▷ d is our new candidate
18: **end if**
19: **end if**
20: **end while**

Bracketing Method: Algorithm

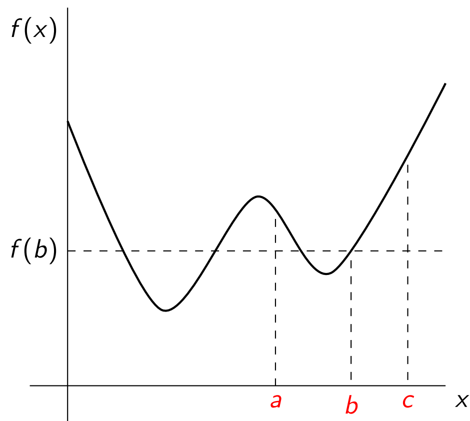


Algorithm 1 Bracketing Algorithm

Require: $a < b < c$ such that $f(a) > f(b)$ and $f(c) > f(b)$

1: **while** $c - a > \epsilon$ **do** ▷ For $\epsilon > 0$
2: **if** $b - a < c - b$ **then** ▷ Choose a test point
3: $d \leftarrow (b + c)/2$
4: **else**
5: $d \leftarrow (a + b)/2$
6: **end if**
7: **if** $d > b$ **then**
8: **if** $f(d) \geq f(b)$ **then**
9: $(a, b, c) \leftarrow (a, b, d)$ ▷ d is our new c
10: **else if** $f(d) < f(b)$ **then**
11: $(a, b, c) \leftarrow (b, d, c)$ ▷ d is our new candidate
12: **end if**
13: **else if** $d < b$ **then**
14: **if** $f(d) \geq f(b)$ **then**
15: $(a, b, c) \leftarrow (d, b, c)$ ▷ d is our new a
16: **else if** $f(d) < f(b)$ **then**
17: $(a, b, c) \leftarrow (a, d, b)$ ▷ d is our new candidate
18: **end if**
19: **end if**
20: **end while**

Bracketing Method: Algorithm

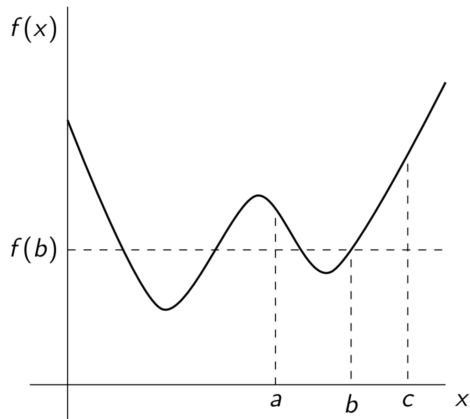


Algorithm 1 Bracketing Algorithm

Require: $a < b < c$ such that $f(a) > f(b)$ and $f(c) > f(b)$

1: **while** $c - a > \epsilon$ **do** ▷ For $\epsilon > 0$
2: **if** $b - a < c - b$ **then** ▷ Choose a test point
3: $d \leftarrow (b + c)/2$
4: **else**
5: $d \leftarrow (a + b)/2$
6: **end if**
7: **if** $d > b$ **then**
8: **if** $f(d) \geq f(b)$ **then**
9: $(a, b, c) \leftarrow (a, b, d)$ ▷ d is our new c
10: **else if** $f(d) < f(b)$ **then**
11: $(a, b, c) \leftarrow (b, d, c)$ ▷ d is our new candidate
12: **end if**
13: **else if** $d < b$ **then**
14: **if** $f(d) \geq f(b)$ **then**
15: $(a, b, c) \leftarrow (d, b, c)$ ▷ d is our new a
16: **else if** $f(d) < f(b)$ **then**
17: $(a, b, c) \leftarrow (a, d, b)$ ▷ d is our new candidate
18: **end if**
19: **end if**
20: **end while**

Bracketing Method: Algorithm

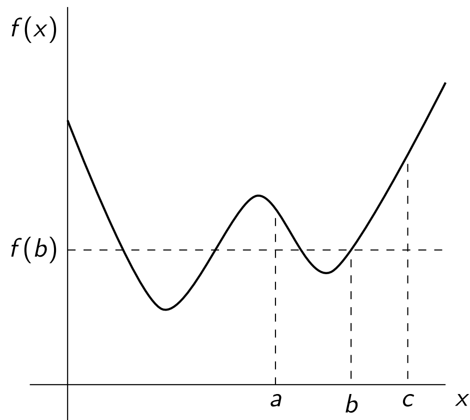


Algorithm 1 Bracketing Algorithm

Require: $a < b < c$ such that $f(a) > f(b)$ and $f(c) > f(b)$

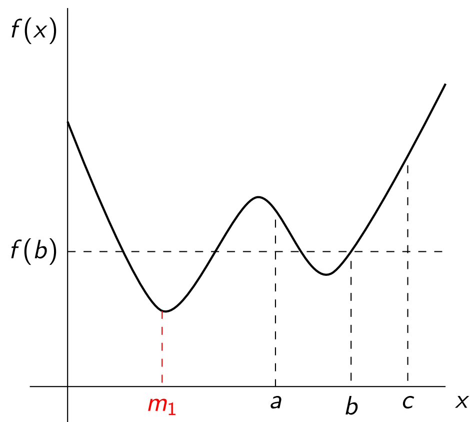
```
 1: while c − a > ε do                              ▷ For ε > 0
 2:     if b − a < c − b then                       ▷ Choose a test point
 3:         d ← (b + c)/2
 4:     else
 5:         d ← (a + b)/2
 6:     end if
 7:     if d > b then
 8:         if f(d) ≥ f(b) then
 9:             (a, b, c) ← (a, b, d)               ▷ d is our new c
10:         else if f(d) < f(b) then
11:             (a, b, c) ← (b, d, c)               ▷ d is our new candidate
12:         end if
13:     else if d < b then
14:         if f(d) ≥ f(b) then
15:             (a, b, c) ← (d, b, c)               ▷ d is our new a
16:         else if f(d) < f(b) then
17:             (a, b, c) ← (a, d, b)               ▷ d is our new candidate
18:         end if
19:     end if
20: end while
```

Bracketing Method: Algorithm



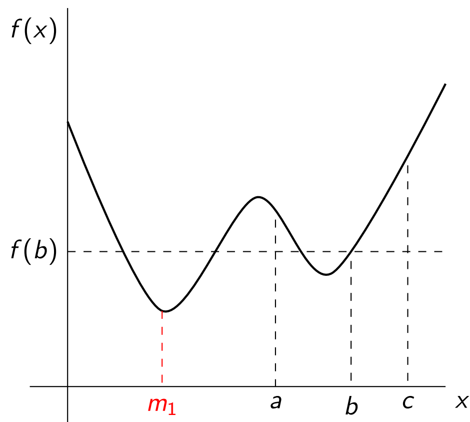
- If we continue this process, the algorithm will eventually converge.
  - At every step, we always choose a and c so that we are bracketing a minimum
  - Our candidate point b always has $f(b) < f(a)$ and $f(c)$.
- The convergence rate is quite slow compared to other algorithms, but it will always work for any continuous, bounded function on a finite interval.
- Notice, however, that we missed the true global minimum of the function.
- The bracketing algorithm is guaranteed to find a local minimum, but not necessarily the global minimum.
- This is not a problem if the objective function is convex (since the minimum is unique).

Bracketing Method: Algorithm



- If we continue this process, the algorithm will eventually converge.
  - At every step, we always choose a and c so that we are bracketing a minimum
  - Our candidate point b always has $f(b) < f(a)$ and $f(c)$.
- The convergence rate is quite slow compared to other algorithms, but it will always work for any continuous, bounded function on a finite interval.
- Notice, however, that we missed the true global minimum of the function.
- The bracketing algorithm is guaranteed to find a local minimum, but not necessarily the global minimum.
- This is not a problem if the objective function is convex (since the minimum is unique).

Bracketing Method: Algorithm



- If we continue this process, the algorithm will eventually converge.
  - At every step, we always choose a and c so that we are bracketing a minimum
  - Our candidate point b always has $f(b) < f(a)$ and $f(c)$.
- The convergence rate is quite slow compared to other algorithms, but it will always work for any continuous, bounded function on a finite interval.
- Notice, however, that we missed the true global minimum of the function.
- The bracketing algorithm is guaranteed to find a local minimum, but not necessarily the global minimum.
- This is not a problem if the objective function is convex (since the minimum is unique).

Other Gradient Free Algorithms

Bracketing is slow, but we can often do better, even without derivative based methods

- Golden-section search
  - This is the same as the bracketing algorithm, but chooses our candidate and test points differently
  - Instead of choosing the midpoint of the intervals, we choose $b = c - (b - a)/\phi$ and $d = a + (c - b)/\phi$ where $\phi = (1 + \sqrt{5})/2$

    Note that ϕ is the golden ratio
  - This keeps the size of the intervals at a constant ratio, which ensures that we are doing a better job of sampling the interval for the minimum (and avoids taking a large number of steps that are inefficiently close together)
  - Still has slow convergence, but it's guaranteed just like in the bracketing case.

Other Gradient Free Algorithms

- Successive Parabolic Interpolation
  - Generates new candidate points by fitting a parabola to the last three points and choosing the minimum of the parabola as the new candidate
  - In general, this is a much faster algorithm than the bracketing methods, however it can often fail to converge if the function is not well behaved
- Brent's method: combines SPI with Golden Section Search
  - Attempts to take steps with successive parabolic interpolation
  - Falls back to the Golden-section search method if that fails.
  - This means it retains the best case properties of SPI, and often is your best bet for derivative free optimization over a single variable.

Newton's Method

- We can do much better, however, if we are willing to use the derivatives of f
- Consider a point x_k, and let's do a second order Taylor approximation of f around x_k:

$$p_k(x) := f(x_k) + f'(x_k)(x - x_k) + \frac{1}{2} f''(x_k)(x - x_k)^2 \tag{3}$$

- Suppose that $f''(x_k) > 0$: then $p_k(x)$ is convex, and the minimum of $p_k(x)$ is at

$$x_{k+1} = x_k - \frac{f'(x_k)}{f''(x_k)} \tag{4}$$

- If p_k is a good global approximation of f, then x_{k+1} should be close to the true minimum

  Or at least, closer than x_k

- If we start with a good enough guess x_0, the sequence defined by this procedure will converge to the true minimum of f

  Note, however, that with a bad guess, Newton's method does not need to converge at all

Section 2

Unconstrained Optimization: Multivariate

Problem

- Suppose that $f : \mathbb{R}^n \to \mathbb{R}$, and we want to solve the problem

$$\max_{x \in \mathbb{R}^n} f(x) \tag{5}$$

- We know that if f is differentiable, a necessary condition for optimality is that:

$$Df(x) = 0 \tag{6}$$

If you haven't seen this notation before, read $Df(x)^T = \nabla f(x)$ if $f : \mathbb{R}^n \to \mathbb{R}$, or as $f'(x)$ if $n = 1$.

In general, if $f : \mathbb{R}^n \to \mathbb{R}^m$, then $Df(x)$ is an $m \times n$ matrix, where $(Df(x))_{ij} = \frac{\partial f_i}{\partial x_j}$

- There are several main approaches here
  1. Newton's Method: Attack eq. (6) directly as a root finding problem. This will require calculating the Hessian $D^2 f(x)$
  2. Gradient Based: Use information encoded in the gradient $Df(x)$ to either climb down the hill towards the minimum, or to approximate $D^2 f(x)$.
  3. Gradient Free: Many different approaches.

Newton's Method in Several Dimensions

- We can do the same 2nd order approximation as before:

$$p_k(x) = f(x_k) + Df(x_k)(x - x_k) + \frac{1}{2}(x - x_k)^T D^2 f(x_k)(x - x_k) \tag{7}$$

- If we differentiate this with respect to x, and set it equal to zero, we can obtain:

$$x^\star = x_k - \left(D^2 f(x_k)\right)^{-1} Df(x_k) \tag{8}$$

  In general, $x^\star$ need not be a minimum if f is not convex

- Newton's method: set $x_{k+1} = x^\star(x_k)$ and keep iterating until convergence
- You can prove that this will converge if you start with a good enough guess (close to the true minimum)
- Problems: Newton's method takes itself too seriously
  - Treats a **local** 2nd order approximation as though it's good **globally**
  - With a bad initial guess, tends to take extremely large step sizes, which take you even farther away from the true optimum, and the whole thing breaks down

Newton's Method in Several Dimensions

- We can do the same 2nd order approximation as before:

$$p_k(x) = f(x_k) + Df(x_k)(x - x_k) + \frac{1}{2}(x - x_k)^T D^2 f(x_k)(x - x_k) \tag{7}$$

- If we differentiate this with respect to x, and set it equal to zero, we can obtain:

$$x^\star = x_k - \left(D^2 f(x_k)\right)^{-1} Df(x_k) \tag{8}$$

In general, $x^\star$ need not be a minimum if f is not convex

- Newton's method: set $x_{k+1} = x^\star(x_k)$ and keep iterating until convergence
- You can prove that this will converge if you start with a good enough guess (close to the true minimum)
- Problems: Newton's method takes itself too seriously
  - Treats a **local** 2nd order approximation as though it's good **globally**
  - With a bad initial guess, tends to take extremely large step sizes, which take you even farther away from the true optimum, and the whole thing breaks down

Newton's Method in Several Dimensions

- We can do the same 2nd order approximation as before:

$$p_k(x) = f(x_k) + Df(x_k)(x - x_k) + \frac{1}{2}(x - x_k)^T D^2 f(x_k)(x - x_k) \tag{7}$$

- If we differentiate this with respect to x, and set it equal to zero, we can obtain:

$$x^\star = x_k - \left(D^2 f(x_k)\right)^{-1} Df(x_k) \tag{8}$$

In general, $x^\star$ need not be a minimum if f is not convex

- Newton's method: set $x_{k+1} = x^\star(x_k)$ and keep iterating until convergence
- You can prove that this will converge if you start with a good enough guess (close to the true minimum)
- Problems: Newton's method takes itself too seriously
  - Treats a **local** 2nd order approximation as though it's good **globally**
  - With a bad initial guess, tends to take extremely large step sizes, which take you even farther away from the true optimum, and the whole thing breaks down

Newton's Method in Several Dimensions

- We can do the same 2nd order approximation as before:

$$p_k(x) = f(x_k) + Df(x_k)(x - x_k) + \frac{1}{2}(x - x_k)^T D^2 f(x_k)(x - x_k) \tag{7}$$

- If we differentiate this with respect to x, and set it equal to zero, we can obtain:

$$x^\star = x_k - \left(D^2 f(x_k)\right)^{-1} Df(x_k) \tag{8}$$

  In general, $x^\star$ need not be a minimum if f is not convex

- Newton's method: set $x_{k+1} = x^\star(x_k)$ and keep iterating until convergence
- You can prove that this will converge if you start with a good enough guess (close to the true minimum)
- Problems: Newton's method takes itself too seriously
  - Treats a **local** 2nd order approximation as though it's good **globally**
  - With a bad initial guess, tends to take extremely large step sizes, which take you even farther away from the true optimum, and the whole thing breaks down

Newton's Method with Line Searches

- What if instead of taking the approximation p_k so seriously, we just used it to get a good search direction?
- Remember that the FOCs for eq. (7) are $D^2 f(x_k)(x^\star - x_k) = -Df(x_k)$
- Let's define our **search direction** $s_k = (x^\star - x_k)$ as the solution to this system of linear equations
  - Write our update step as $x_{k+1} = x_k + \alpha s_k$
  - When $\alpha = 1$, this is just regular Newton's Method
- What if we solve the **univariate** problem

  $$\max_{\alpha_k \geq 0} f(x_k + \alpha_k s_k)$$

  and set $x_{k+1} = x_k + \alpha_k^\star s_k$? This is called a **Line Search**

  Use your favorite derivative free method from before (usually you want Brent's method)
- It tends to be much less sensitive to the initial conditions than regular Newton's method
- You can plug this line search idea back in to most search algorithms: as long as they give you a proposed point, you can try searching *in that direction* instead

Other Methods (non-exhaustive)

1. Gradient Descent: take successive steps downhill (follow the gradient) until you converge
2. Simplex Based Methods: (Nelder-Mead) uses a local simplicial approximation of the objective and a set of heuristic rules to "approximate" the gradient and follow it downhill
3. Quasi-Newton Methods: Instead of calculating the Hessian $D^2 f(x)$ directly, build up an approximation using information just from the derivative.
   - There are many options in this space, but you probably want to be using L-BFGS.
4. Conjugate Gradient: extremely efficient algorithm but it only works if the hessian $D^2 f(x)$ is positive definite (i.e., the function is convex)
5. Global Methods:
   - Simulated Annealing/Monte Carlo Markov Chains: Take random steps, with a probability of sometimes stepping downhill (to help escape local minima)
   - Differential Evolution: Track many different candidate points, and sometimes "mutate" coordinates from the best ones to get better potential test points
   - And many many more...

Section 3

A brief aside: Differentiation

Why do we need gradients?

- We've seen already that derivatives show up all over the place.
- Any time we have a maximization problem

$$\max_{x \in \mathbb{R}^n} f(x)$$

  we know that the solution $x^\star$ satisfies $Df(x^\star) = 0$

- You need to calculate the gradient to use gradient descent, L-BFGS, or any other gradient based method
- You even need second derivatives $D^2 f(x)$ (also called the Hessian) if you want to use Newton's method or other similar approaches

How do we calculate derivatives in practice?

There are three main approaches:

1. Write them down in closed form
   - Nice if you can, but requires a lot of work
   - Hard if you're rapidly iterating on a model
   - Extremely error prone
2. Finite differences
   - Evaluate the function multiple times at small perturbations to the point of intersection
   - Usually quite robust
   - Requires very little work, but is often the slowest approach
3. Automatic Differentiation
   - Have the computer differentiate your program for you
   - Usually works, and calculates exact derivatives
   - Relies on programming language and compiler features – sometimes you're limited in how you write your code, especially in python

How do we calculate derivatives in practice?

There are three main approaches:

1. Write them down in closed form
   - Nice if you can, but requires a lot of work
   - Hard if you're rapidly iterating on a model
   - Extremely error prone
2. Finite differences
   - Evaluate the function multiple times at small perturbations to the point of intersection
   - Usually quite robust
   - Requires very little work, but is often the slowest approach
3. Automatic Differentiation
   - Have the computer differentiate your program for you
   - Usually works, and calculates exact derivatives
   - Relies on programming language and compiler features – sometimes you're limited in how you write your code, especially in python

How do we calculate derivatives in practice?

There are three main approaches:

1. Write them down in closed form
   - Nice if you can, but requires a lot of work
   - Hard if you're rapidly iterating on a model
   - Extremely error prone
2. Finite differences
   - Evaluate the function multiple times at small perturbations to the point of intersection
   - Usually quite robust
   - Requires very little work, but is often the slowest approach
3. Automatic Differentiation
   - Have the computer differentiate your program for you
   - Usually works, and calculates exact derivatives
   - Relies on programming language and compiler features – sometimes you're limited in how you write your code, especially in python

How do we calculate derivatives in practice?

There are three main approaches:

1. Write them down in closed form
    - Nice if you can, but requires a lot of work
    - Hard if you're rapidly iterating on a model
    - Extremely error prone
2. Finite differences
    - Evaluate the function multiple times at small perturbations to the point of intersection
    - Usually quite robust
    - Requires very little work, but is often the slowest approach
3. Automatic Differentiation
    - Have the computer differentiate your program for you
    - Usually works, and calculates exact derivatives
    - Relies on programming language and compiler features – sometimes you're limited in how you write your code, especially in python

Finite Differences

The basic theory of finite differences is pretty simple

- Recall the definition of a derivative for $f : \mathbb{R} \to \mathbb{R}$:

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

- Rather than actually taking the limit, we can try to do this with a really small perturbation:

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

  where h is small.

Benefits:

- In principle, this is pretty easy to implement (although you shouldn't – there are lot's of issues with numerical stability, so you should use someone else's package rather than writing it yourself)
- Conceptually straightforward
- When implemented properly, can be fairly robust

Finite Differences: Drawbacks

All that being said, you probably want to avoid finite differences for your actual code. Why?

- It adds a source of approximation error

  This can be a serious issue for poorly conditioned problems, or problems with very high curvature in f (i.e, $D^2 f(x)$ is large)

- It is really slow compared to other options – it requires a lot more evaluations of your function, which can be quite costly

Consider what we need to do when $f : \mathbb{R}^n \to \mathbb{R}$ where n is nontrivially large:

$$Df_i(x) \approx \frac{f(x + h_i e_i) - f(x)}{h_i} \;\; \forall i = 1, \dots, n$$

where $e_i = (0, \dots, 0, 1, 0 \dots, 0)$ is the ith basis vector for $\mathbb{R}^n$

- This requires two function evaluations of f for every dimension of x

  In practice, it actually involves three, since people tend to use two sided finite differences to reduce the approximation error, or adaptive schemes which require even more function evaluations

- So if $n = 10$, you have to evaluate f at least 21 times every time you want the derivative

- You can see how this gets very expensive very fast as n gets large

Finite Differences: Drawbacks

All that being said, you probably want to avoid finite differences for your actual code. Why?

- It adds a source of approximation error

  This can be a serious issue for poorly conditioned problems, or problems with very high curvature in f (i.e, $D^2 f(x)$ is large)

- It is really slow compared to other options – it requires a lot more evaluations of your function, which can be quite costly

Consider what we need to do when $f : \mathbb{R}^n \to \mathbb{R}$ where n is nontrivially large:

$$Df_i(x) \approx \frac{f(x + h_i e_i) - f(x)}{h_i} \quad \forall i = 1, \ldots, n$$

where $e_i = (0, \ldots, 0, 1, 0 \ldots, 0)$ is the ith basis vector for $\mathbb{R}^n$

- This requires two function evaluations of f for every dimension of x

  In practice, it actually involves three, since people tend to use two sided finite differences to reduce the approximation error, or adaptive schemes which require even more function evaluations

- So if $n = 10$, you have to evaluate f at least 21 times every time you want the derivative

- You can see how this gets very expensive very fast as n gets large

Finite Differences: Drawbacks

All that being said, you probably want to avoid finite differences for your actual code. Why?

- It adds a source of approximation error

  This can be a serious issue for poorly conditioned problems, or problems with very high curvature in f (i.e, $D^2 f(x)$ is large)

- It is really slow compared to other options – it requires a lot more evaluations of your function, which can be quite costly

Consider what we need to do when $f : \mathbb{R}^n \to \mathbb{R}$ where n is nontrivially large:

$$Df_i(x) \approx \frac{f(x + h_i e_i) - f(x)}{h_i} \quad \forall i = 1, \ldots, n$$

where $e_i = (0, \ldots, 0, 1, 0 \ldots, 0)$ is the ith basis vector for $\mathbb{R}^n$

- This requires two function evaluations of f for every dimension of x

  In practice, it actually involves three, since people tend to use two sided finite differences to reduce the approximation error, or adaptive schemes which require even more function evaluations

- So if $n = 10$, you have to evaluate f at least 21 times every time you want the derivative
- You can see how this gets very expensive very fast as n gets large

Autodifferentiation

- In practice you can do much, much better by having the computer write your derivatives for you
- How does this work in practice?
  - Start with the chain rule:
  $$D(f \circ g)(x) = Df(g(x)) \cdot Dg(x)$$
  - A program is just a sequence of elementary operations chained together (all of which have known derivatives/gradients)
  - In principle, if you write down primitive rules for all of the elementary functions, and apply the chain rule all the way through, you can get the computer to write the correct answer
  - Recursive applications of simple rules (like the chain rule) is something computers are really good at!
- It's slightly more complicated than this, but you mostly just need to know that this exists and can work like magic

Autodifferentiation

- In practice you can do much, much better by having the computer write your derivatives for you
- How does this work in practice?
  - Start with the chain rule:
    $$D(f \circ g)(x) = Df(g(x)) \cdot Dg(x)$$
  - A program is just a sequence of elementary operations chained together (all of which have known derivatives/gradients)
  - In principle, if you write down primitive rules for all of the elementary functions, and apply the chain rule all the way through, you can get the computer to write the correct answer
  - Recursive applications of simple rules (like the chain rule) is something computers are really good at!
- It's slightly more complicated than this, but you mostly just need to know that this exists and can work like magic

Autodifferentiation

- In practice you can do much, much better by having the computer write your derivatives for you
- How does this work in practice?
  - Start with the chain rule:
    $$D(f \circ g)(x) = Df(g(x)) \cdot Dg(x)$$
  - A program is just a sequence of elementary operations chained together (all of which have known derivatives/gradients)
  - In principle, if you write down primitive rules for all of the elementary functions, and apply the chain rule all the way through, you can get the computer to write the correct answer
  - Recursive applications of simple rules (like the chain rule) is something computers are really good at!
- It's slightly more complicated than this, but you mostly just need to know that this exists and can work like magic

Autodifferentiation

- In practice you can do much, much better by having the computer write your derivatives for you
- How does this work in practice?
  - Start with the chain rule:
    $$D(f \circ g)(x) = Df(g(x)) \cdot Dg(x)$$
  - A program is just a sequence of elementary operations chained together (all of which have known derivatives/gradients)
  - In principle, if you write down primitive rules for all of the elementary functions, and apply the chain rule all the way through, you can get the computer to write the correct answer
  - Recursive applications of simple rules (like the chain rule) is something computers are really good at!
- It's slightly more complicated than this, but you mostly just need to know that this exists and can work like magic

Autodifferentiation: Benefits/Drawbacks

Benefits:

- Extremely easy – you just call a package and it calculates your derivatives for you
- No approximation error (unlike finite differences)
- Much less error prone than writing derivatives by hand
- It works especially well for large complicated functions

Drawbacks:

- You need to have derivatives written for every primitive function that you use. This means that sometimes you're constrained in the functions you can write without breaking autodiff
- In some cases, you can achieve better performance writing derivatives by hand
- Bad performance when the function you're differentiating involves an iterative procedure
  - For instance, you should never attempt to autodifferentiate the solution to an optimization problem
  - Instead, you can exploit results (like the Envelope Theorem) to get the derivative to those problems in closed form, or just use finite differences

Autodifferentiation: Benefits/Drawbacks

Benefits:

- Extremely easy – you just call a package and it calculates your derivatives for you
- No approximation error (unlike finite differences)
- Much less error prone than writing derivatives by hand
- It works especially well for large complicated functions

Drawbacks:

- You need to have derivatives written for every primitive function that you use. This means that sometimes you're constrained in the functions you can write without breaking autodiff
- In some cases, you can achieve better performance writing derivatives by hand
- Bad performance when the function you're differentiating involves an iterative procedure
  - For instance, you should never attempt to autodifferentiate the solution to an optimization problem
  - Instead, you can exploit results (like the Envelope Theorem) to get the derivative to those problems in closed form, or just use finite differences

Practical Guide to Differentiation

What packages should I use and when??

Finite Differences

- In Julia, consider either `FiniteDiff.jl` or `FiniteDifferences.jl`
  - Both have similar functionality, and will probably work for you
  - `FiniteDifferences.jl` can sometimes be better if you want higher order derivatives
- However, you should only be using these
  1. To double check derivatives you wrote by hand
  2. As a last resort, or
  3. If you're *sure* you know what you're doing

Autodifferentiation

- In Julia, use either `ForwardDiff.jl` or `Zygote.jl`
  - For the kinds of functions we will be dealing with in this course, `ForwardDiff.jl` is probably your best bet
  - Really robust for smaller generic functions

Practical Guide to Differentiation

What packages should I use and when??

Finite Differences

- In Julia, consider either `FiniteDiff.jl` or `FiniteDifferences.jl`
  - Both have similar functionality, and will probably work for you
  - `FiniteDifferences.jl` can sometimes be better if you want higher order derivatives
- However, you should only be using these
  1. To double check derivatives you wrote by hand
  2. As a last resort, or
  3. If you're *sure* you know what you're doing

Autodifferentiation

- In Julia, use either `ForwardDiff.jl` or `Zygote.jl`
  - For the kinds of functions we will be dealing with in this course, `ForwardDiff.jl` is probably your best bet
  - Really robust for smaller generic functions

Practical Guide to Differentiation

What packages should I use and when??

Finite Differences

- In Julia, consider either `FiniteDiff.jl` or `FiniteDifferences.jl`
  - Both have similar functionality, and will probably work for you
  - `FiniteDifferences.jl` can sometimes be better if you want higher order derivatives
- However, you should only be using these
  1. To double check derivatives you wrote by hand
  2. As a last resort, or
  3. If you're *sure* you know what you're doing

Autodifferentiation

- In Julia, use either `ForwardDiff.jl` or `Zygote.jl`
  - For the kinds of functions we will be dealing with in this course, `ForwardDiff.jl` is probably your best bet
  - Really robust for smaller generic functions

Section 4

Constrained Optimization

Problem Setup

- Let's consider an optimization problem of the form

$$\max_{x \in \mathbb{R}^n} f(x) \quad \text{s.t. } g(x) = 0 \tag{9}$$

  for some functions $f : \mathbb{R}^n \to \mathbb{R}$ (objective) and $g : \mathbb{R}^n \to \mathbb{R}^m$ (constraints).

- Extremely general formulation – most interesting problems in economics involve some form of constrained optimization
  - An risk-averse investor chooses a portfolio of assets, subject to a budget constraint
  - A social planner chooses a set of labor and consumption allocations to maximize welfare, subject to incentive compatibility constraints (what is the optimal tax rate?)
  - A social planner chooses which medical students get matched to which schools to maximize total match utility, subject to adding-up constraints (each person is assigned to one and only one match)
- **How do we augment our unconstrained methods in order to enforce the constraints at our solution?**

Problem Setup

- Let's consider an optimization problem of the form

$$\max_{x \in \mathbb{R}^n} f(x) \quad \text{s.t. } g(x) = 0 \tag{9}$$

  for some functions $f : \mathbb{R}^n \to \mathbb{R}$ (objective) and $g : \mathbb{R}^n \to \mathbb{R}^m$ (constraints).

- Extremely general formulation – most interesting problems in economics involve some form of constrained optimization
  - An risk-averse investor chooses a portfolio of assets, subject to a budget constraint
  - A social planner chooses a set of labor and consumption allocations to maximize welfare, subject to incentive compatibility constraints (what is the optimal tax rate?)
  - A social planner chooses which medical students get matched to which schools to maximize total match utility, subject to adding-up constraints (each person is assigned to one and only one match)
- **How do we augment our unconstrained methods in order to enforce the constraints at our solution?**

Problem Setup

- Let's consider an optimization problem of the form

$$\max_{x \in \mathbb{R}^n} f(x) \quad \text{s.t. } g(x) = 0 \tag{9}$$

  for some functions $f : \mathbb{R}^n \to \mathbb{R}$ (objective) and $g : \mathbb{R}^n \to \mathbb{R}^m$ (constraints).
- Extremely general formulation – most interesting problems in economics involve some form of constrained optimization
  - An risk-averse investor chooses a portfolio of assets, subject to a budget constraint
  - A social planner chooses a set of labor and consumption allocations to maximize welfare, subject to incentive compatibility constraints (what is the optimal tax rate?)
  - A social planner chooses which medical students get matched to which schools to maximize total match utility, subject to adding-up constraints (each person is assigned to one and only one match)
- **How do we augment our unconstrained methods in order to enforce the constraints at our solution?**

Problem Setup

- Let's consider an optimization problem of the form

$$\begin{aligned} &\max_{x \in \mathbb{R}^n} f(x) \\ &\text{s.t. } g(x) = 0 \end{aligned} \tag{9}$$

  for some functions $f : \mathbb{R}^n \to \mathbb{R}$ (objective) and $g : \mathbb{R}^n \to \mathbb{R}^m$ (constraints).

- Extremely general formulation – most interesting problems in economics involve some form of constrained optimization
  - An risk-averse investor chooses a portfolio of assets, subject to a budget constraint
  - A social planner chooses a set of labor and consumption allocations to maximize welfare, subject to incentive compatibility constraints (what is the optimal tax rate?)
  - A social planner chooses which medical students get matched to which schools to maximize total match utility, subject to adding-up constraints (each person is assigned to one and only one match)
- **How do we augment our unconstrained methods in order to enforce the constraints at our solution?**

Problem Setup

- Let's consider an optimization problem of the form

$$\begin{aligned} &\max_{x \in \mathbb{R}^n} f(x) \\ &\text{s.t. } g(x) = 0 \end{aligned} \tag{9}$$

  for some functions $f : \mathbb{R}^n \to \mathbb{R}$ (objective) and $g : \mathbb{R}^n \to \mathbb{R}^m$ (constraints).
- Extremely general formulation – most interesting problems in economics involve some form of constrained optimization
  - An risk-averse investor chooses a portfolio of assets, subject to a budget constraint
  - A social planner chooses a set of labor and consumption allocations to maximize welfare, subject to incentive compatibility constraints (what is the optimal tax rate?)
  - A social planner chooses which medical students get matched to which schools to maximize total match utility, subject to adding-up constraints (each person is assigned to one and only one match)
- How do we augment our unconstrained methods in order to enforce the constraints at our solution?

Problem Setup

- Let's consider an optimization problem of the form

$$\max_{x \in \mathbb{R}^n} f(x) \quad \text{s.t. } g(x) = 0 \tag{9}$$

  for some functions $f : \mathbb{R}^n \to \mathbb{R}$ (objective) and $g : \mathbb{R}^n \to \mathbb{R}^m$ (constraints).
- Extremely general formulation – most interesting problems in economics involve some form of constrained optimization
  - An risk-averse investor chooses a portfolio of assets, subject to a budget constraint
  - A social planner chooses a set of labor and consumption allocations to maximize welfare, subject to incentive compatibility constraints (what is the optimal tax rate?)
  - A social planner chooses which medical students get matched to which schools to maximize total match utility, subject to adding-up constraints (each person is assigned to one and only one match)
- **How do we augment our unconstrained methods in order to enforce the constraints at our solution?**

Best Approach: Reformulate the Problem

Oftentimes you can simply reformulate the constraint/problem and choose an easier domain to search over instead

- Simplex Constraints (probabilities sum to 1): suppose $g(x) = \sum x_i - 1$ and we know $0 \leq x_i \leq 1$
  - Use a known bijection from $\mathbb{R}^{n-1} \to \Delta^n$
  - See `UnitSimplex` transformation in `TransformVariables.jl`
- Consumption Savings with Labor Choice:

$$\max_{c,n} \; u(c,n) + V(b') \quad \text{s.t.} \; c + b' \leq wn \tag{10}$$

Choose consumption share of income rather than consumption directly to ensure that $c \geq 0$ always

Best Approach: Reformulate the Problem

Oftentimes you can simply reformulate the constraint/problem and choose an easier domain to search over instead

- Simplex Constraints (probabilities sum to 1): suppose $g(x) = \sum x_i - 1$ and we know $0 \leq x_i \leq 1$
  - Use a known bijection from $\mathbb{R}^{n-1} \to \Delta^n$
  - See `UnitSimplex` transformation in `TransformVariables.jl`
- Consumption Savings with Labor Choice:

$$\begin{aligned} \max_{c,n} \; & u(c, n) + V(b') \\ \text{s.t.} \, & c + b' \leq wn \end{aligned} \tag{10}$$

  Choose consumption share of income rather than consumption directly to ensure that $c \geq 0$ always

Best Approach: Reformulate the Problem (cont.)

- Linear Equality Constraints: Suppose $g(x) = Ax - b$ where A is an $m \times n$ matrix
  - Let M be any matrix whose columns span the nullspace of A (i.e, $AM = 0$, where M is $n \times p$) and let $\hat{x}$ be any solution to $Ax = b$
  - Search over $z \in \mathbb{R}^p$, and solve the reformulated problem

$$\max_{z \in \mathbb{R}^p} f(Mz + \hat{x})$$

    Notice that by construction, $A(Mz + \hat{x}) = A\hat{x} = b$, so this lets us search in an unconstrained way while ensuring that the constraint always holds
- In general, this approach will always be better than trying to handle your constraints with your solution algorithm
  - It's much much easier to solve unconstrained problems (or problems with simple box-bounds) than problems with nonlinear constraints
  - If you can exploit the structure of your problem, that's always better

Best Approach: Reformulate the Problem (cont.)

- Linear Equality Constraints: Suppose $g(x) = Ax - b$ where A is an $m \times n$ matrix
  - Let M be any matrix whose columns span the nullspace of A (i.e, $AM = 0$, where M is $n \times p$) and let $\hat{x}$ be any solution to $Ax = b$
  - Search over $z \in \mathbb{R}^p$, and solve the reformulated problem

    $$\max_{z \in \mathbb{R}^p} f(Mz + \hat{x})$$

    Notice that by construction, $A(Mz + \hat{x}) = A\hat{x} = b$, so this lets us search in an unconstrained way while ensuring that the constraint always holds
- In general, this approach will always be better than trying to handle your constraints with your solution algorithm
  - It's much much easier to solve unconstrained problems (or problems with simple box-bounds) than problems with nonlinear constraints
  - If you can exploit the structure of your problem, that's always better

Lagrangian Approach

- The strategy you've probably seen before is to set up a Lagrangian:

$$\mathcal{L}(x, \lambda) := f(x) - \lambda \cdot g(x)$$

  where $\lambda \in \mathbb{R}^m$.

- We know that a solution $(x^\star, \lambda^\star)$ to eq. (9) satisfies $D\mathcal{L}(x^\star, \lambda^\star) = 0$
- This suggests the approach: Define $h : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ as $h(x, \lambda) := D\mathcal{L}(x, \lambda)$.
  - We want to find $(x^\star, \lambda^\star)$ such that $h(x^\star, \lambda^\star) = 0$
  - This is a root-finding problem, so we can just attack that directly, with your favorite approach (say, Newton's method)
- You can extend this to inequality constraints by adding the additional equality constraints:

$$\lambda_i g_i(x) = 0 \;\; \forall\, i \in 1, \ldots, m$$

- Solving this system with Newton's Method is called **Sequential Quadratic Programming**

This can be an extremely effective strategy, as long as the problem is not too large, since it requires you to solve a system of equations involving the dense hessian $D^2 f(x)$ (which is $O(n^3)$)

Lagrangian Approach

- The strategy you've probably seen before is to set up a Lagrangian:

$$\mathcal{L}(x, \lambda) := f(x) - \lambda \cdot g(x)$$

  where $\lambda \in \mathbb{R}^m$.
- We know that a solution $(x^\star, \lambda^\star)$ to eq. (9) satisfies $D\mathcal{L}(x^\star, \lambda^\star) = 0$
- This suggests the approach: Define $h : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ as $h(x, \lambda) := D\mathcal{L}(x, \lambda)$.
  - We want to find $(x^\star, \lambda^\star)$ such that $h(x^\star, \lambda^\star) = 0$
  - This is a root-finding problem, so we can just attack that directly, with your favorite approach (say, Newton's method)
- You can extend this to inequality constraints by adding the additional equality constraints:

$$\lambda_i g_i(x) = 0 \;\; \forall \; i \in 1, \ldots, m$$

- Solving this system with Newton's Method is called **Sequential Quadratic Programming**

  This can be an extremely effective strategy, as long as the problem is not too large, since it requires you to solve a system of equations involving the dense hessian $D^2 f(x)$ (which is $O(n^3)$)

Lagrangian Approach

- The strategy you've probably seen before is to set up a Lagrangian:

$$\mathcal{L}(x, \lambda) := f(x) - \lambda \cdot g(x)$$

  where $\lambda \in \mathbb{R}^m$.
- We know that a solution $(x^\star, \lambda^\star)$ to eq. (9) satisfies $D\mathcal{L}(x^\star, \lambda^\star) = 0$
- This suggests the approach: Define $h : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ as $h(x, \lambda) := D\mathcal{L}(x, \lambda)$.
  - We want to find $(x^\star, \lambda^\star)$ such that $h(x^\star, \lambda^\star) = 0$
  - This is a root-finding problem, so we can just attack that directly, with your favorite approach (say, Newton's method)
- You can extend this to inequality constraints by adding the additional equality constraints:

$$\lambda_i g_i(x) = 0 \;\; \forall \; i \in 1, \ldots, m$$

- Solving this system with Newton's Method is called **Sequential Quadratic Programming**

  This can be an extremely effective strategy, as long as the problem is not too large, since it requires you to solve a system of equations involving the dense hessian $D^2 f(x)$ (which is $O(n^3)$)

Lagrangian Approach

- The strategy you've probably seen before is to set up a Lagrangian:

$$\mathcal{L}(x, \lambda) := f(x) - \lambda \cdot g(x)$$

  where $\lambda \in \mathbb{R}^m$.
- We know that a solution $(x^\star, \lambda^\star)$ to eq. (9) satisfies $D\mathcal{L}(x^\star, \lambda^\star) = 0$
- This suggests the approach: Define $h : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ as $h(x, \lambda) := D\mathcal{L}(x, \lambda)$.
  - We want to find $(x^\star, \lambda^\star)$ such that $h(x^\star, \lambda^\star) = 0$
  - This is a root-finding problem, so we can just attack that directly, with your favorite approach (say, Newton's method)
- You can extend this to inequality constraints by adding the additional equality constraints:

$$\lambda_i g_i(x) = 0 \;\; \forall \; i \in 1, \ldots, m$$

- Solving this system with Newton's Method is called **Sequential Quadratic Programming**

  This can be an extremely effective strategy, as long as the problem is not too large, since it requires you to solve a system of equations involving the dense hessian $D^2 f(x)$ (which is $O(n^3)$)

Penalty Methods

- Intuition here is simple: **make it painful for the optimizer to violate the constraints**
- Consider the following unconstrained problem:

$$\max_{x \in R^n} \quad f(x) - P \sum_{i=1}^{m} g_i(x)^2 \tag{11}$$

  for some penalty parameter P

  Note: if you're solving a minimization problem, add the penalty term rather than subtract it

  - As $P \to \infty$, the solution to this problem converges to the true solution
  - For inequality constraints $g(x) \leq 0$, replace $g(x)$ with $g^+(x)$ where $g_i^+(x) = \max\{g_i(x), 0\}$

- Problem: when P is very large, this problem is badly conditioned
- This makes our solution algorithms converge very slowly, and can introduce numerical instabilities to our problem

Penalty Methods

- Intuition here is simple: **make it painful for the optimizer to violate the constraints**
- Consider the following unconstrained problem:

$$\max_{x \in R^n} \; f(x) - P \sum_{i=1}^{m} g_i(x)^2 \tag{11}$$

  for some penalty parameter P

  Note: if you're solving a minimization problem, add the penalty term rather than subtract it

  - As $P \to \infty$, the solution to this problem converges to the true solution
  - For inequality constraints $g(x) \leq 0$, replace $g(x)$ with $g^+(x)$ where $g_i^+(x) = \max\{g_i(x), 0\}$
- Problem: when P is very large, this problem is badly conditioned
- This makes our solution algorithms converge very slowly, and can introduce numerical instabilities to our problem

Penalty Methods

- Intuition here is simple: **make it painful for the optimizer to violate the constraints**

- Consider the following unconstrained problem:

$$\max_{x \in R^n} \quad f(x) - P \sum_{i=1}^{m} g_i(x)^2 \tag{11}$$

  for some penalty parameter P

  Note: if you're solving a minimization problem, add the penalty term rather than subtract it

  - As $P \to \infty$, the solution to this problem converges to the true solution
  - For inequality constraints $g(x) \leq 0$, replace $g(x)$ with $g^+(x)$ where $g_i^+(x) = \max\{g_i(x), 0\}$

- Problem: when P is very large, this problem is badly conditioned

- This makes our solution algorithms converge very slowly, and can introduce numerical instabilities to our problem

Penalty Methods: Solution

Instead of starting with a very large P, instead solve a sequence of problems where P gets larger and larger

- Choose a sequence P_k which starts small and diverges to ∞:
  - Let $P_k = \gamma P_{k-1}$ for some $\gamma > 1$, with $P_0 = 1$
  - Common choice is to use $\gamma = 10$ but you may need to vary it by problem
- Let x_k be the solution to

$$\max_{x \in R^n} \; f(x) - P_k \sum_{i=1}^{m} g_i(x)^2 \tag{12}$$

- The trick: use x_k as the initial guess for the $k+1$th problem
- This helps with convergence speed, but does not fix the numerical instabilities caused by the poor conditioning of the problem
- You still have to solve the problem when P_k has grown very, very large

Penalty Methods: Solution

Instead of starting with a very large P, instead solve a sequence of problems where P gets larger and larger

- Choose a sequence P_k which starts small and diverges to ∞:
  - Let $P_k = \gamma P_{k-1}$ for some $\gamma > 1$, with $P_0 = 1$
  - Common choice is to use $\gamma = 10$ but you may need to vary it by problem
- Let x_k be the solution to

$$\max_{x \in R^n} \; f(x) - P_k \sum_{i=1}^{m} g_i(x)^2 \tag{12}$$

- The trick: use x_k as the initial guess for the $k+1$th problem
- This helps with convergence speed, but does not fix the numerical instabilities caused by the poor conditioning of the problem
- You still have to solve the problem when P_k has grown very, very large

Augmented Lagrangian Method

- Similar to the penalty method approach, but we use the Lagrangian of the problem, rather than just the penalty functions, to enforce the constraint
- Consider the sequence of problems for $k = 0, \ldots, \infty$:

$$\max_{x \in \mathbb{R}^n} \quad \underbrace{f(x)}_{\text{Main Objective}} \quad \underbrace{-\frac{P_k}{2} \sum_{i=1}^{m} g_i(x)^2}_{\text{Penalty Function}} \quad \underbrace{-\sum_{i=1}^{m} \lambda_i^k g_i(x)}_{\text{Lagrange Multiplier Penalty}} \tag{13}$$

  where $P_k \to \infty$ is a sequence of increasing penalties Note: if you're solving a minimization problem, add the penalty term rather than subtract it

- We can't pick λ_i as part of the optimization: the true optimum is a saddle point of the Lagrangian, not a minimum
- Use the following update rule for λ_i:

$$\lambda_i^{k+1} = \lambda_i^k + P_k g_i(x_k) \tag{14}$$

  At every iteration, we're updating λ_i with exactly the "contribution" of the penalty function in the previous iteration

- You don't need to keep going until P_k is really large – you can stop if λ has converged!

Section 5

Special Cases

Convexity is your friend

Recall some definitions:

- A function $f : \mathbb{R}^n \to \mathbb{R}$ is convex if for any $\lambda \in (0,1)$, and any points $x, y \in \mathbb{R}^n$
$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y)$$
- A set $A \subset \mathbb{R}^n$ is convex if for every $x, y \in A$, and every $\lambda \in (0,1)$,
$$\lambda x + (1-\lambda)y \in A$$

Why is convexity important for optimization?

1. It has sharp implications about where the extrema can be
2. There are usually a set of specialized methods you can use once you know that your problem is convex (which tend to perform *much* better)

Convexity is your friend

Recall some definitions:

- A function $f : \mathbb{R}^n \to \mathbb{R}$ is convex if for any $\lambda \in (0, 1)$, and any points $x, y \in \mathbb{R}^n$
$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y)$$

- A set $A \subset \mathbb{R}^n$ is convex if for every $x, y \in A$, and every $\lambda \in (0, 1)$,
$$\lambda x + (1 - \lambda)y \in A$$

Why is convexity important for optimization?

1. It has sharp implications about where the extrema can be
2. There are usually a set of specialized methods you can use once you know that your problem is convex (which tend to perform *much* better)

Convexity is your friend

Recall some definitions:

- A function $f : \mathbb{R}^n \to \mathbb{R}$ is convex if for any $\lambda \in (0, 1)$, and any points $x, y \in \mathbb{R}^n$
$$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y)$$

- A set $A \subset \mathbb{R}^n$ is convex if for every $x, y \in A$, and every $\lambda \in (0, 1)$,
$$\lambda x + (1-\lambda) y \in A$$

Why is convexity important for optimization?

1. It has sharp implications about where the extrema can be
2. There are usually a set of specialized methods you can use once you know that your problem is convex (which tend to perform *much* better)

Convexity is your friend

Recall some definitions:

- A function $f : \mathbb{R}^n \to \mathbb{R}$ is convex if for any $\lambda \in (0, 1)$, and any points $x, y \in \mathbb{R}^n$
$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y)$$

- A set $A \subset \mathbb{R}^n$ is convex if for every $x, y \in A$, and every $\lambda \in (0, 1)$,
$$\lambda x + (1 - \lambda)y \in A$$

Why is convexity important for optimization?

1. It has sharp implications about where the extrema can be
2. There are usually a set of specialized methods you can use once you know that your problem is convex (which tend to perform *much* better)

Convexity is your friend

Recall some definitions:

- A function $f : \mathbb{R}^n \to \mathbb{R}$ is convex if for any $\lambda \in (0, 1)$, and any points $x, y \in \mathbb{R}^n$
$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y)$$

- A set $A \subset \mathbb{R}^n$ is convex if for every $x, y \in A$, and every $\lambda \in (0, 1)$,
$$\lambda x + (1 - \lambda)y \in A$$

Why is convexity important for optimization?

1. It has sharp implications about where the extrema can be
2. There are usually a set of specialized methods you can use once you know that your problem is convex (which tend to perform *much* better)

Convex Functions have Unique Global Minima

Theorem 1

Let $f : \mathbb{R}^n \to \mathbb{R}$ be a strictly convex function. Then every local minima of f is a global minimum.

Proof. Suppose not. I.e, there is $x^\star \neq x_m$ with $f(x^\star) < f(x_m)$

- Let x_m be a local minimum of f (there exists a neighborhood U such that if $y \in U$, then $f(y) \geq f(x_m)$)
- Pick λ sufficiently small that $x' = \lambda x_m + (1 - \lambda)x^\star \in U$, and so $f(x') \geq f(x_m)$
- But we also know that

$$\begin{aligned} f(x') &< \lambda f(x_m) + (1-\lambda)f(x^\star) && \text{By strict convexity of } f \\ &< \lambda f(x_m) + (1-\lambda)f(x_m) && \text{Since } x^\star \text{ is a global minimum} \\ &= f(x_m) \end{aligned}$$

  This is a contradiction, so no such $x^\star$ can exist.

Convex Problems have Unique Minima

Theorem 2

Let $f : \mathbb{R}^n \to \mathbb{R}$ be a strictly convex, continuous function, and let $D \subset \mathbb{R}^n$ be a compact, convex set. Then, the constrained optimization problem

$$\min_x f(x)$$
$$s.t.\ x \in \mathcal{D}$$

has a unique minimum.

Proof. We know a minimum exists because continuous functions on compact sets always have a minimum.

Suppose we have two: x_1 and x_2 with $f(x_1) = f(x_2) \leq f(x)$ for all $x \in \mathcal{D}$

Then consider $x' = \frac{x_1+x_2}{2}$. We know $x' \in \mathcal{D}$ because $\mathcal{D}$ is convex.

But then strict convexity implies

$$f(x') < \frac{1}{2}f(x_1) + \frac{1}{2}f(x_2) = f(x_1)$$

which contradicts that $f(x_1)$ was a minimum to begin with. □

So what does convexity buy us?

- Peace of Mind
  - Theorem 1 means that with a convex objective, if we find a local minimum (which satisfies our constraints) then we're done. We can never do better
  - Theorem 2 tells us that we'll never wind up with an indeterminate solution – there's always a unique minimum
  - This means that if our objective is convex, we can check the interior for a single unique global minimum, and then just check points on the boundary of $\mathcal{D}$ if we can't find one
- Speed
  - For general, nonlinear convex problems, consider looking into the Conjugate Gradient method
  - Many other convex problems have special structures that specific solvers can exploit.
  - Taking advantage of it often gets you huge speed gains, or takes an infeasible problem and makes it feasible
- Note: All of this goes through, exactly the same, if you replace min with max and convex function with concave function.

So what does convexity buy us?

- Peace of Mind
  - Theorem 1 means that with a convex objective, if we find a local minimum (which satisfies our constraints) then we're done. We can never do better
  - Theorem 2 tells us that we'll never wind up with an indeterminate solution – there's always a unique minimum
  - This means that if our objective is convex, we can check the interior for a single unique global minimum, and then just check points on the boundary of $\mathcal{D}$ if we can't find one
- Speed
  - For general, nonlinear convex problems, consider looking into the Conjugate Gradient method
  - Many other convex problems have special structures that specific solvers can exploit.
  - Taking advantage of it often gets you huge speed gains, or takes an infeasible problem and makes it feasible
- Note: All of this goes through, exactly the same, if you replace min with max and convex function with concave function.

So what does convexity buy us?

- Peace of Mind
  - Theorem 1 means that with a convex objective, if we find a local minimum (which satisfies our constraints) then we're done. We can never do better
  - Theorem 2 tells us that we'll never wind up with an indeterminate solution – there's always a unique minimum
  - This means that if our objective is convex, we can check the interior for a single unique global minimum, and then just check points on the boundary of $\mathcal{D}$ if we can't find one
- Speed
  - For general, nonlinear convex problems, consider looking into the Conjugate Gradient method
  - Many other convex problems have special structures that specific solvers can exploit.
  - Taking advantage of it often gets you huge speed gains, or takes an infeasible problem and makes it feasible
- Note: All of this goes through, exactly the same, if you replace min with max and convex function with concave function.

Special Case: Linear Programs

Consider a problem of the form

$$\begin{aligned} \min_{x \in \mathbb{R}^n} \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned} \tag{15}$$

where $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, and A is an $m \times n$ matrix.

- This is known as a Linear Program (LP)
- It has a linear objective function and affine (linear plus a constant) constraint set
- Whenever you're working on a problem, you should always be on the lookout for the possibility that you can reformulate it as an LP

  There are *extremely* efficient numerical algorithms to solve these problems
- Simplex Method: Intuition
  - The constraint set of this problem looks like a polytope (in 2-d, polygon)
  - One can show that a minimum to the problem will always lie on one of the corners, so its sufficient to just check all the corners in a smart way

Special Case: Linear Programs

Consider a problem of the form

$$\begin{aligned} \min_{x\in\mathbb{R}^n} \quad & c^T x \\ \text{s.t.} \quad & Ax \leq b \\ & x \geq 0 \end{aligned} \tag{15}$$

where $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, and A is an $m \times n$ matrix.

- This is known as a Linear Program (LP)
- It has a linear objective function and affine (linear plus a constant) constraint set
- Whenever you're working on a problem, you should always be on the lookout for the possibility that you can reformulate it as an LP

  There are *extremely* efficient numerical algorithms to solve these problems
- Simplex Method: Intuition
  - The constraint set of this problem looks like a polytope (in 2-d, polygon)
  - One can show that a minimum to the problem will always lie on one of the corners, so its sufficient to just check all the corners in a smart way

Operationalizing It All: What packages do I use in Julia?

- Univariate Optimization: use `Optim.jl` – defaults to Brent's method
- Multivariate Optimization:
  - Nelder-Mead, Newton, Conjugate Gradient, and L-BFGS: use `Optim.jl`

    This is usually the fastest and easiest way to get off the ground, since Optim will automatically compute the derivaties you need with `ForwardDiff.jl`
  - If you need nonlinear constraints, use `NLopt.jl`
    - Gradient Based: MMA (Method of Moving Asymptotes) is extremely robust, but also has SLSQP and L-BFGS
    - COBYLA and BOBYQA are good for derivative free (and BOBYQA works particularly well when your objective function looks quadratic)
    - Allows you to embed any method inside an Augmented Lagrangian problem, which lets you do L-BFGS with nonlinear constraints, for instance
- Linear Programming: Use `JuMP.jl`

  In principle, if you like using `JuMP.jl`, you can use it to set up and solve any of these problems. However it's particularly useful for LPs and other special convex problems