Lecture 5: Optimization

Programming and Numerical Methods for Economics—ECNM10115

Juan Zurita & Zhuotong Liu
School of Economics, The University of Edinburgh

Autumn 2025

Types of Optimization Problems

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

Section 1

Unconstrained Optimization: Univariate

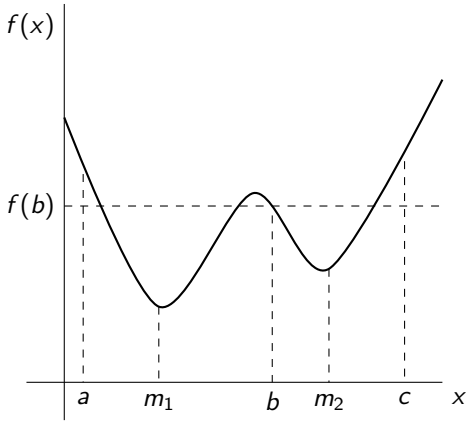
Problem Setup

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

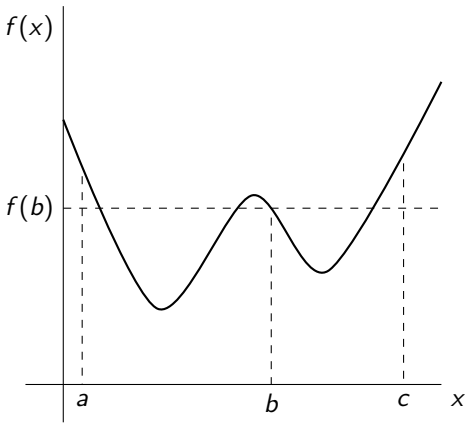


- Suppose that $f : \mathbb{R} \to \mathbb{R}$ is continuous and,
- Suppose we've found points a, b and $c \in \mathbb{R}$ such that

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

- 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

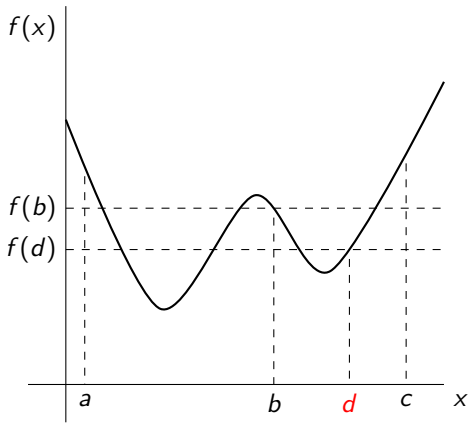


Algorithm 1 Bracketing Algorithm

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

while $c - a > \epsilon$ **do** ▷ For $\epsilon > 0$
  if $b - a < c - b$ **then** ▷ Choose a test point
    $d \leftarrow (b + c)/2$
  else
    $d \leftarrow (a + b)/2$
  end if
  if $d > b$ **then**
    if $f(d) \geq f(b)$ **then**
      $(a, b, c) \leftarrow (a, b, d)$ ▷ d is our new c
    else if $f(d) < f(b)$ **then**
      $(a, b, c) \leftarrow (b, d, c)$ ▷ d is our new candidate
    end if
  else if $d < b$ **then**
    if $f(d) \geq f(b)$ **then**
      $(a, b, c) \leftarrow (d, b, c)$ ▷ d is our new a
    else if $f(d) < f(b)$ **then**
      $(a, b, c) \leftarrow (a, d, b)$ ▷ d is our new candidate
    end if
  end if
end while

Bracketing Method: Algorithm

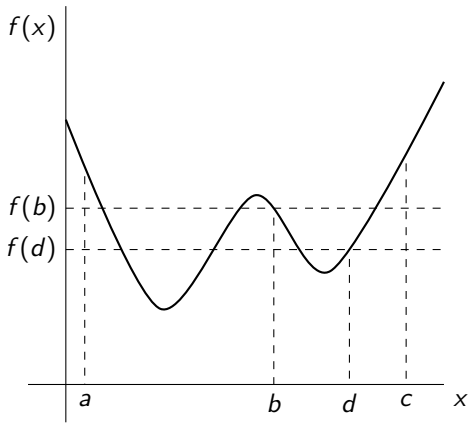


Algorithm 1 Bracketing Algorithm

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

while $c - a > \epsilon$ **do** ▷ For $\epsilon > 0$
  if $b - a < c - b$ **then** ▷ Choose a test point
    $d \leftarrow (b + c)/2$
  else
    $d \leftarrow (a + b)/2$
  end if
  if $d > b$ **then**
    if $f(d) \geq f(b)$ **then**
      $(a, b, c) \leftarrow (a, b, d)$ ▷ d is our new c
    else if $f(d) < f(b)$ **then**
      $(a, b, c) \leftarrow (b, d, c)$ ▷ d is our new candidate
    end if
  else if $d < b$ **then**
    if $f(d) \geq f(b)$ **then**
      $(a, b, c) \leftarrow (d, b, c)$ ▷ d is our new a
    else if $f(d) < f(b)$ **then**
      $(a, b, c) \leftarrow (a, d, b)$ ▷ d is our new candidate
    end if
  end if
end while

Bracketing Method: Algorithm

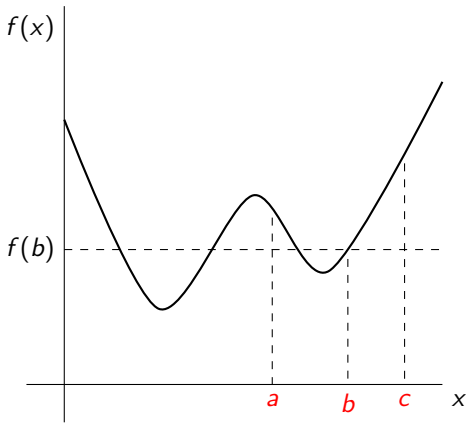


Algorithm 1 Bracketing Algorithm

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

while $c - a > \epsilon$ **do** ▷ For $\epsilon > 0$
  if $b - a < c - b$ **then** ▷ Choose a test point
    $d \leftarrow (b + c)/2$
  else
    $d \leftarrow (a + b)/2$
  end if
  if $d > b$ **then**
    if $f(d) \geq f(b)$ **then**
      $(a, b, c) \leftarrow (a, b, d)$ ▷ d is our new c
    else if $f(d) < f(b)$ **then**
      $(a, b, c) \leftarrow (b, d, c)$ ▷ d is our new candidate
    end if
  else if $d < b$ **then**
    if $f(d) \geq f(b)$ **then**
      $(a, b, c) \leftarrow (d, b, c)$ ▷ d is our new a
    else if $f(d) < f(b)$ **then**
      $(a, b, c) \leftarrow (a, d, b)$ ▷ d is our new candidate
    end if
  end if
end while

Bracketing Method: Algorithm

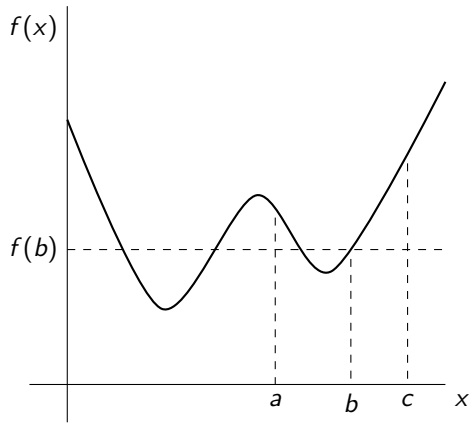


Algorithm 1 Bracketing Algorithm

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

while $c - a > \epsilon$ **do** ▷ For $\epsilon > 0$
  if $b - a < c - b$ **then** ▷ Choose a test point
    $d \leftarrow (b + c)/2$
  else
    $d \leftarrow (a + b)/2$
  end if
  if $d > b$ **then**
    if $f(d) \geq f(b)$ **then**
      $(a, b, c) \leftarrow (a, b, d)$ ▷ d is our new c
    else if $f(d) < f(b)$ **then**
      $(a, b, c) \leftarrow (b, d, c)$ ▷ d is our new candidate
    end if
  else if $d < b$ **then**
    if $f(d) \geq f(b)$ **then**
      $(a, b, c) \leftarrow (d, b, c)$ ▷ d is our new a
    else if $f(d) < f(b)$ **then**
      $(a, b, c) \leftarrow (a, d, b)$ ▷ d is our new candidate
    end if
  end if
end while

Bracketing Method: Algorithm

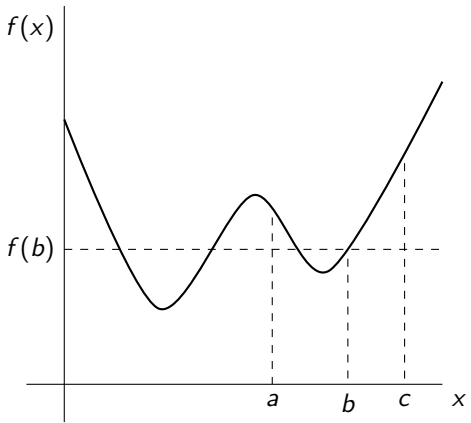


Algorithm 1 Bracketing Algorithm

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

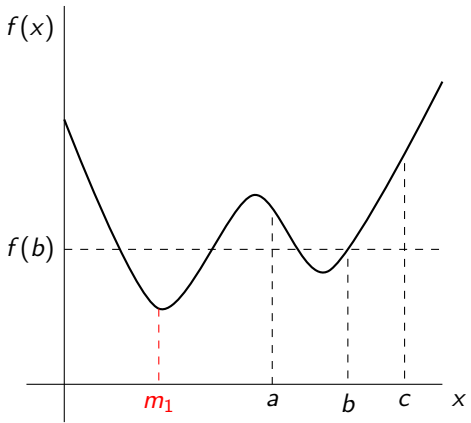
while $c - a > \epsilon$ **do** ▷ For $\epsilon > 0$
  if $b - a < c - b$ **then** ▷ Choose a test point
    $d \leftarrow (b + c)/2$
  else
    $d \leftarrow (a + b)/2$
  end if
  if $d > b$ **then**
    if $f(d) \geq f(b)$ **then**
      $(a, b, c) \leftarrow (a, b, d)$ ▷ d is our new c
    else if $f(d) < f(b)$ **then**
      $(a, b, c) \leftarrow (b, d, c)$ ▷ d is our new candidate
    end if
  else if $d < b$ **then**
    if $f(d) \geq f(b)$ **then**
      $(a, b, c) \leftarrow (d, b, c)$ ▷ d is our new a
    else if $f(d) < f(b)$ **then**
      $(a, b, c) \leftarrow (a, d, b)$ ▷ d is our new candidate
    end if
  end if
end while

Bracketing Method: Algorithm



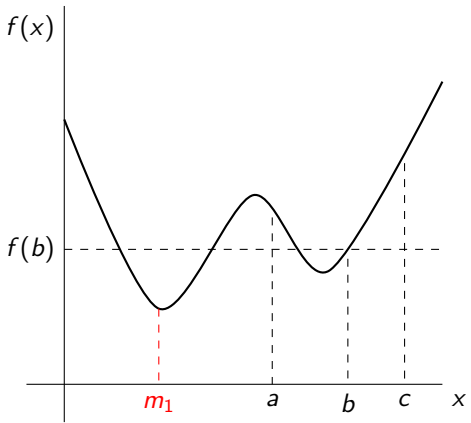
- If we continue this process, the algorithm will eventually converge.
  - At every step, we always choose a and c so that we are bracketing a minimum
  - Our candidate point b always has $f(b) < f(a)$ and $f(c)$.
- The convergence rate is quite slow compared to other algorithms, but it will always work 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).

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

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:

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

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

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