# Lecture 01

## Intro to computing

Ivan Rudik
AEM 7130

# Software and stuff

Necessary things to download to follow along today and in the future:

- Git
- Julia or JuliaPro
- Jupyter/Anaconda

# Software and stuff

Necessary things to download to follow along today and in the future:

- Git
- Julia or JuliaPro
- Jupyter/Anaconda
- Also make a GitHub account

# Software and stuff

Necessary things to download to follow along today and in the future:

- Git
- Julia or JuliaPro
- Jupyter/Anaconda
- Also make a GitHub account

For this lecture you will need the following Julia packages

```
import Pkg; Pkg.add("ForwardDiff"); Pkg.add("Distributions"); Pkg.add("BenchmarkTools")
using ForwardDiff, Distributions, BenchmarkTools
```

# What this class is about

1. Learning how to compute dynamic models through approximation and estimation
2. Other useful computational techniques and estimation approaches
3. Learning important details about computing models

# What you need to succeed in this course

1. ECON 6090 and ECON 6170
2. Or potentially some other graduate economics classes (see me if you haven't yet)
3. Previous coding experience or willingness to spend some time learning as you go

# Course materials

1. Everything we use in the course will be **freely available** and posted to the course GitHub (details next class on how to use Git)
2. Books (free from the library or authors' websites):
    1. Judd (1998)
    2. Miranda and Fackler (2002)
    3. Nocedal and Wright (2006)
    4. Karp and Traeger (2013)

# Things to do before next class

- Spend some time reading up on Julia:
  - Learning Julia
  - QuantEcon Julia lectures

# What we will cover in the class

1. Basic computing and things you need to think about
2. Coding, version control, reproducibility, workflow
3. Optimization
4. Solving dynamic models
5. Solving spatial models
6. Machine learning

# What you have to do

- Come to class
- 4 computational problem sets
- Final research project proposal
- Final research project
- One presentation of a paper from the literature

# Important days / times

- Office hours: Tuesday 1:30-2:00, Thursday 1:00-1:30 on Zoom
- Final project proposal: March 23
- Final project paper: May 25

# Grading

- Problem sets: 40% (10% each)
- Final project proposal: 15%
- Final project paper/presentation: 25%
- Class participation: 10%
- Computational paper presentation: 10%

# Problem sets (10% each)

You **must** use Julia and write .jl scripts, no Jupyter notebooks

You can work in groups of up to 3

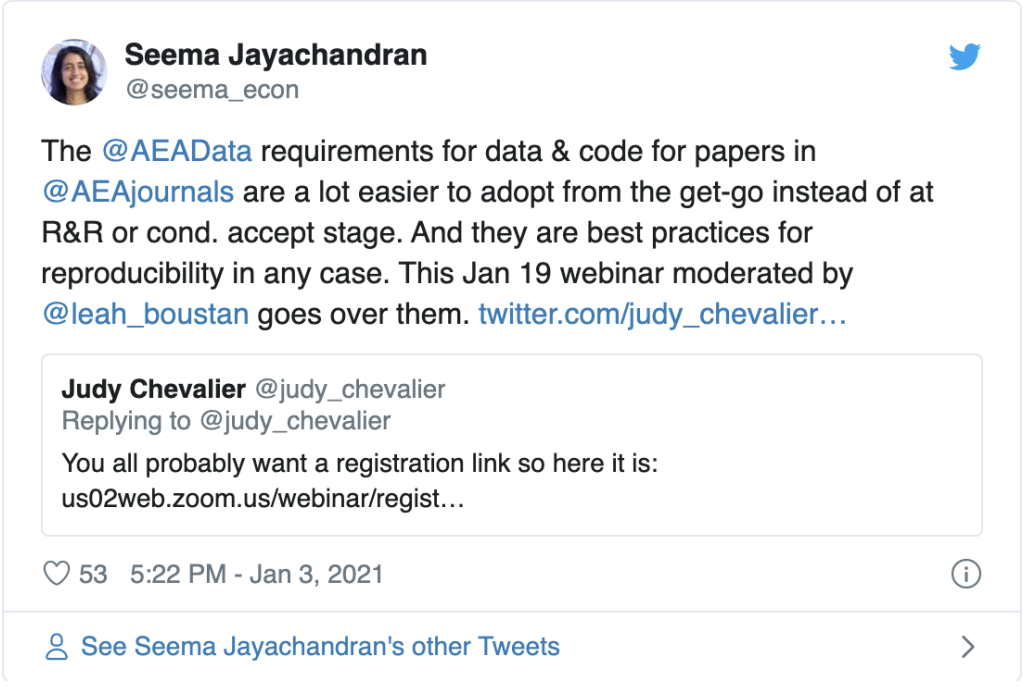
Problem sets will be where you **implement** the techniques we learn in class on your own, but we will be doing our fair share of coding in class

# Problem sets (10% each)

Why am I making you do problem sets this way?

If you want to publish in an AEA journal (amongst others now..) you need to have good practices, other journals are following suit

Julia is very good for reproducibility



# Computational paper presentations (10%)

Everyone will present a paper starting in 2-3 weeks

The paper can apply methods we've learned about (or will learn about), or can be a new method that we have not covered

You must consult with me at least 1 week prior to your scheduled presentation date to ensure the paper is appropriate for a presentation

The syllabus has some pre-approved papers you can choose from under the **Applications** header

# Final project (25% paper, 15% proposal)

The final project will be the beginning of a **computationally-driven** research project or an extension of an existing paper with new methods

Proposals will be due about half way through the class

# Final project (25% paper, 15% proposal)

The only requirement is that the project **cannot be computationally trivial** (i.e. no 600 observation two-way FE papers)

It can be numerical, empirical, whatever

Everyone will present their final projects in the last week of class

More details on the syllabus and to come later

# Slides

All slides will be available on the course GitHub page

# Slides

All slides will be available on the course GitHub page

The slides are made with R Markdown and can run Julia via the JuliaCall package, e.g.

```
# true coefficient
bbeta = π;
# random x data
x = randn(100,1)*5 .+ 3;
# OLS data generating process
y = bbeta.*x .+ randn(100,1)*10;

# OLS estimation
bbeta_hat = inv(x'x)x'y;

println("β-hat is $(round(bbeta_hat[1],digits=3)) and the true β is $(round(bbeta,digits=3)).")
```

```
## β-hat is 3.065 and the true β is 3.142.
```

# Why computational methods?

# Why do we need computational methods?

Everything you've done so far has likely been solvable analytically

# Why do we need computational methods?

Everything you've done so far has likely been solvable analytically

Including OLS: $\hat{\beta} = (X'X)^{-1}X'Y$

# Why do we need computational methods?

Everything you've done so far has likely been solvable analytically

Including OLS: $\hat{\beta} = (X'X)^{-1}X'Y$

**Not all economic models have closed-form solutions, and others can't have closed-form solutions with losing important economic content**

This is generally true for dynamic models

# What can we compute?

We can use computation + theory to answer **quantitative** questions

# What can we compute?

We can use computation + theory to answer **quantitative** questions

Theory can't give us welfare in dollar terms

Theory can't tell us the value of economic primitives

# What can we compute?

Theory often relies on strong assumptions like:

# What can we compute?

Theory often relies on strong assumptions like:

- log utility (lose income vs substitution)
- no transactions costs (important friction)
- strictly concave objectives (natural phenomena don't follow this)
- static decisionmaking

It can be unclear what the cost of these assumptions are

# Example 1

Suppose we have a constant elasticity demand function: $q(p) = p^{-0.2}$

In equilibrium, quantity demanded is $q^* = 2$

**What price clears the market in equilibrium?**

# Example 1

Suppose we have a constant elasticity demand function: $q(p) = p^{-0.2}$

In equilibrium, quantity demanded is $q^* = 2$

**What price clears the market in equilibrium?**

Just invert the demand function:

$2 = p^{-0.2}$

# Example 1

Suppose we have a constant elasticity demand function: $q(p) = p^{-0.2}$

In equilibrium, quantity demanded is $q^* = 2$

**What price clears the market in equilibrium?**

Just invert the demand function:

$2 = p^{-0.2}$

$p^* = 2^{-5}$ ✓

Your calculator can do the rest

# Example 2

Suppose the demand function is now: $q(p) = 0.5p^{-0.2} + 0.5p^{-0.5}$, a weighted average of two CE demand functions

In equilibrium, quantity demanded is $q^* = 2$

**What price clears the market in equilibrium?**

# Example 2

Suppose the demand function is now: $q(p) = 0.5p^{-0.2} + 0.5p^{-0.5}$, a weighted average of two CE demand functions

In equilibrium, quantity demanded is $q^* = 2$

**What price clears the market in equilibrium?**

First, does a solution exist?

# Example 2

Suppose the demand function is now: $q(p) = 0.5p^{-0.2} + 0.5p^{-0.5}$, a weighted average of two CE demand functions

In equilibrium, quantity demanded is $q^* = 2$

**What price clears the market in equilibrium?**

First, does a solution exist?

Yes, why?

# Example 2

$q(p)$ is monotonically decreasing

# Example 2

$q(p)$ is monotonically decreasing

$q(p)$ is greater than 2 at $p = 0.1$ and less than 2 at $p = 0.2$

# Example 2

$q(p)$ is monotonically decreasing

$q(p)$ is greater than 2 at $p = 0.1$ and less than 2 at $p = 0.2$

$\rightarrow$ by intermediate value theorem $q(p) = 2$ somewhere in $(0.1, 0.2)$

# Example 2

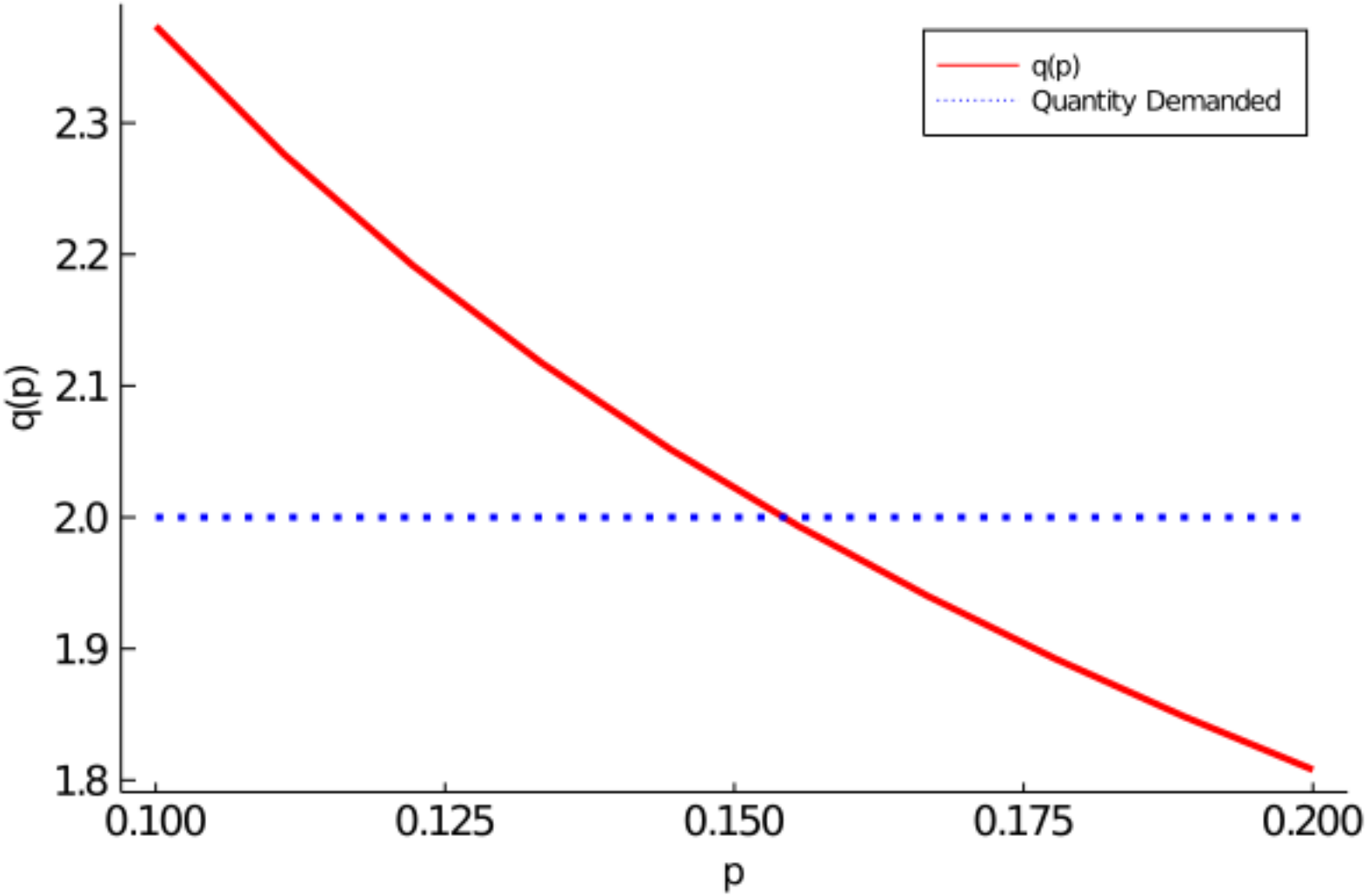
```
# We know solution is between .1 and .2
x = collect(range(.1, stop = .2, length = 10)) # generate evenly spaced grid
q_d = ones(size(x)).*2                         # generate equal length vector of qd=2

# Price function
price(p) = p.^(-0.2)/2 .+ p.^(-0.5)/2

# Get corresponding quantity values at these prices
y = price(x)
```

Now plot $q_d$ and $q(p)$

# Example 2


q(p)
Quantity Demanded
2.3
2.2
2.1
2.0
1.9
1.8
q(p)
0.100
0.125
0.150
0.175
0.200
p

# Example 2

Notice: if we let $t = p^{-0.1}$ then:

$q(t) = 0.5t^2 + 0.5t^5$

# Example 2

Notice: if we let $t = p^{-0.1}$ then:

$q(t) = 0.5t^2 + 0.5t^5$

Can we solve for $t$ now?

# Example 2

Notice: if we let $t = p^{-0.1}$ then:

$q(t) = 0.5t^2 + 0.5t^5$

Can we solve for $t$ now?

No! Closed-form solutions to fifth order polynomials are not guaranteed to exist!

# Example 2

Notice: if we let $t = p^{-0.1}$ then:

$q(t) = 0.5t^2 + 0.5t^5$

Can we solve for $t$ now?

No! Closed-form solutions to fifth order polynomials are not guaranteed to exist!

So how do we solve the problem?

# Newton's method

Iteratively do the following:

1. Guess solution to: $q(p) - q^* = 0 \rightarrow q(p) - 2 = 0$
2. Approximate the function with local second order polynomial around guess
3. Solve this easier equation
4. Solution is the new guess
5. Stop if previous guess and new guess are sufficiently close

We will learn more about this and why it works in a later class

# Newton code

```
# Define demand functions
demand(p) = p^(-0.2)/2 + p^(-0.5)/2 - 2      # quantity minus price
demand_grad(p) = .1*p^(-1.2) + .25*p^(-1.5) # demand gradient
```

# Newton code

```
# Define demand functions
demand(p) = p^(-0.2)/2 + p^(-0.5)/2 - 2      # quantity minus price
demand_grad(p) = .1*p^(-1.2) + .25*p^(-1.5) # demand gradient
```

```
function find_root_newton(demand, demand_grad)
    p = .3         # initial guess
    deltap = 1e10 # initialize stepsize

    while abs(deltap) > 1e-4
        deltap = demand(p)/demand_grad(p)
        p += deltap
        println("Intermediate guess of p = $(round(p,digits=3)).")
    end
    println("The solution is p = $(round(p,digits=3)).")
    return p
end;
```

# Newton code

```
# Solve for price
find_root_newton(demand, demand_grad)
```

```
## Intermediate guess of p = 0.068.
## Intermediate guess of p = 0.115.
## Intermediate guess of p = 0.147.
## Intermediate guess of p = 0.154.
## Intermediate guess of p = 0.154.
## Intermediate guess of p = 0.154.
## The solution is p = 0.154.

## 0.15419764093200633
```

# Example 3

Consider a two period ag commodity market model

Period 1: Farmer makes acreage decisions for planting

Period 2: Per-acre yield realizes, equilibrium crop price clears the market

# Example 3

Consider a two period ag commodity market model

Period 1: Farmer makes acreage decisions for planting

Period 2: Per-acre yield realizes, equilibrium crop price clears the market

The farmer's policy function is: $a(E[p]) = \frac{1}{2} + \frac{1}{2}E[p]$

# Example 3

Consider a two period ag commodity market model

Period 1: Farmer makes acreage decisions for planting

Period 2: Per-acre yield realizes, equilibrium crop price clears the market

The farmer's policy function is: $a(E[p]) = \frac{1}{2} + \frac{1}{2}E[p]$

After planting, yield $\hat{y}$ realizes, producing a total quantity $q = a\hat{y}$ of the crop

# Example 3

Consider a two period ag commodity market model

Period 1: Farmer makes acreage decisions for planting
Period 2: Per-acre yield realizes, equilibrium crop price clears the market

The farmer's policy function is: $a(E[p]) = \frac{1}{2} + \frac{1}{2}E[p]$

After planting, yield $\hat{y}$ realizes, producing a total quantity $q = a\hat{y}$ of the crop

Demand is given by $p(q) = 3 - 2q$

Yield is given by $\hat{y} \sim \mathcal{N}(1, 0.1)$

# How much acreage gets planted?

$$p(\hat{y}) = 3 - 2a\hat{y}$$

# How much acreage gets planted?

$$p(\hat{y}) = 3 - 2a\hat{y}$$

$$a = \frac{1}{2} + \frac{1}{2}(3 - 2aE[\hat{y}])$$

# How much acreage gets planted?

$$p(\hat{y}) = 3 - 2a\hat{y}$$

$$a = \frac{1}{2} + \frac{1}{2}(3 - 2aE[\hat{y}])$$

Rearrange and solve:

$$a^* = 1$$

# How much acreage gets planted?

$p(\hat{y}) = 3 - 2a\hat{y}$

$a = \frac{1}{2} + \frac{1}{2}(3 - 2aE[\hat{y}])$

Rearrange and solve:

$a^* = 1$

Now suppose the government implements a price floor on the crop of $p > 1$ so we have that $p(\hat{y}) = \max(1, 3 - 2a\hat{y})$

**How much acreage does the farmer plant?**

# How much acreage gets planted?

This is analytically intractable

# How much acreage gets planted?

This is analytically intractable

The max operator is non-linear so we can't pass the expectation through

$$E[\max(1, 3 - 2a\hat{y})] \neq \max(1, E[3 - 2a\hat{y}])$$

# How much acreage gets planted?

This is analytically intractable

The max operator is non-linear so we can't pass the expectation through

$$E[\max(1, 3 - 2a\hat{y})] \neq \max(1, E[3 - 2a\hat{y}])$$

$\rightarrow$ we need to solve this numerically

# Function iteration

We can solve this using another technique called **function iteration**

```
# Function iteration method to find a root
function find_root_fi(mn, variance)

    y = randn(1000)*sqrt(variance) .+ mn # draws of the random variable
    a = 1.                               # initial guess
    differ = 100.                        # initialize error
    exp_price = 1.                       # initialize expected price

    while differ > 1e-4
        a_old = a                        # save old acreage
        p = max.(1, 3 .- 2 .*a.*y)       # compute price at all distribution points
        exp_price = mean(p)              # compute expected price
        a = 1/2 + 1/2*exp_price          # get new acreage planted given new price
        differ= abs(a - a_old)           # change in acreage planted
        println("Intermediate acreage guess: $(round(a,digits=3))")
    end

    return a, exp_price
end
```

# Function iteration

```
acreage, expected_price = find_root_fi(1, 0.1)
```

```
## Intermediate acreage guess: 1.133
## Intermediate acreage guess: 1.093
## Intermediate acreage guess: 1.103
## Intermediate acreage guess: 1.1
## Intermediate acreage guess: 1.101
## Intermediate acreage guess: 1.101
## Intermediate acreage guess: 1.101
```

```
## (1.1008333654930316, 1.2016667309860631)
```

```
println("The optimal number of acres to plant is $(round(acreage, digits = 3)).\nThe expected pr
```

```
## The optimal number of acres to plant is 1.101.
## The expected price is 1.202.
```

# Quantifying speed and accuracy

# Big O notation

How do we quantify **speed** and **accuracy** of computational algorithms?

i.e. what is the **computational complexity** of the problem?

# Big O notation

How do we quantify **speed** and **accuracy** of computational algorithms?

i.e. what is the **computational complexity** of the problem?

**General mathematical definition:** Big O notation describes the limiting behavior of a function when the argument tends towards a particular value or infinity

# Big O notation

How do we quantify **speed** and **accuracy** of computational algorithms?

i.e. what is the **computational complexity** of the problem?

**General mathematical definition:** Big O notation describes the limiting behavior of a function when the argument tends towards a particular value or infinity

**Programming context:** Describes the limiting behavior of algorithms in terms of run time/memory/accuracy as input size grows

# Big O notation

How do we quantify **speed** and **accuracy** of computational algorithms?

i.e. what is the **computational complexity** of the problem?

**General mathematical definition:** Big O notation describes the limiting behavior of a function when the argument tends towards a particular value or infinity

**Programming context:** Describes the limiting behavior of algorithms in terms of run time/memory/accuracy as input size grows

You've seen this before in the expression of Taylor series' errors

# Big O Notation

Written as: **O(F(x))**

Here is how to think about it:

# Big O Notation

Written as: **O(F(x))**

Here is how to think about it:

**O(x):** linear

- Time to solve increases linearly in input x
- Accuracy changes linearly in input x

# Big O Notation

Written as: **O(F(x))**

Here is how to think about it:

**O(x):** linear

- Time to solve increases linearly in input x
- Accuracy changes linearly in input x

**Examples?**

# Big O Notation

Written as: **O(F(x))**

Here is how to think about it:

**O(x):** linear

- Time to solve increases linearly in input x
- Accuracy changes linearly in input x

**Examples?**

Time to find a particular (e.g. maximum) value in an unsorted array

$\rightarrow$ For each element, check whether it is the value we want

# Big O Notation

**$O(c^x)$:** exponential

- Time to solve increases exponentially in input x
- Accuracy changes exponentially in input x

# Big O Notation

**$O(c^x)$:** exponential

- Time to solve increases exponentially in input x
- Accuracy changes exponentially in input x

**Examples?**

# Big O Notation

**$O(c^x)$:** exponential

- Time to solve increases exponentially in input x
- Accuracy changes exponentially in input x

**Examples?**

Time to solve a standard dynamic program, ex traveling salesman

$\rightarrow$ For each city $i = 1, \ldots, n$, solve a Bellman as a function of all other cities

# Big O Notation

$\mathbf{O(n!)}$: factorial

- Time to solve increases factorially in input x
- Accuracy changes factorially in input x

# Big O Notation

$\mathbf{O(n!)}$: factorial

- Time to solve increases factorially in input x
- Accuracy changes factorially in input x

**Examples?**

# Big O Notation

$\mathbf{O(n!)}$: factorial

- Time to solve increases factorially in input x
- Accuracy changes factorially in input x

**Examples?**

Solving traveling salesman by brute force

$\rightarrow$ Obtain travel time for all possible combinations of intermediate cities

# Big O Notation: Accuracy example

This is how you have probably seen Big O used before:

# Big O Notation: Accuracy example

This is how you have probably seen Big O used before:

Taylor series for $sin(x)$ around zero:

$$sin(x) \approx x - x^3/3! + x^5/5! + O(x^7)$$

What does $O(x^7)$ mean here?

# Big O Notation: Accuracy example

$$sin(x) \approx x - x^3/3! + x^5/5! + O(x^7)$$

# Big O Notation: Accuracy example

$sin(x) \approx x - x^3/3! + x^5/5! + O(x^7)$

As we move away from $0$ to some $x$, the upper bound of the growth rate in the error of our approximation to $sin(x)$ is $x^7$

We are approximating about zero so $x$ is small and $x^n$ is decreasing in $n$

# Big O Notation: Accuracy example

$sin(x) \approx x - x^3/3! + x^5/5! + O(x^7)$

As we move away from $0$ to some $x$, the upper bound of the growth rate in the error of our approximation to $sin(x)$ is $x^7$

We are approximating about zero so $x$ is small and $x^n$ is decreasing in $n$

For small $x$, higher order polynomials mean the error will grow slower and we have a better local approximation

# Taylor expansions

```
# fifth and third order Taylor approximations
sin_error_5(x) = sin(x) - (x - x^3/6 + x^5/120)
sin_error_3(x) = sin(x) - (x - x^3/6)
```

# Taylor expansions

```
# fifth and third order Taylor approximations
sin_error_5(x) = sin(x) - (x - x^3/6 + x^5/120)
sin_error_3(x) = sin(x) - (x - x^3/6)
```

```
println("Error of fifth-order approximation at x = .001 is: $(sin_error_5(.001))
Error of third-order approximation at x = .001 is: $(sin_error_3(.001))
Error of fifth-order approximation at x = .01 is: $(sin_error_5(.01))
Error of third-order approximation at x = .01 is: $(sin_error_3(.01))
Error of fifth-order approximation at x = .1 is: $(sin_error_5(.1))
Error of third-order approximation at x = .1 is: $(sin_error_3(.1))")
```

```
## Error of fifth-order approximation at x = .001 is: 0.0
## Error of third-order approximation at x = .001 is: 8.239936510889834e-18
## Error of fifth-order approximation at x = .01 is: -1.7347234759768 07e-18
## Error of third-order approximation at x = .01 is: 8.333316675601665e-13
## Error of fifth-order approximation at x = .1 is: -1.983851971587569e-11
## Error of third-order approximation at x = .1 is: 8.331349481138783e-8
```

# Big O Notation: Speed examples

Here are a few examples for fundamental computational methods

# Big O Notation: O(1)

**O(1):** algorithm executes in **constant time**

The size of the input does not affect execution speed

# Big O Notation: O(1)

**O(1):** algorithm executes in **constant time**

The size of the input does not affect execution speed

# Big O Notation: O(1)

**O(1):** algorithm executes in **constant time**

The size of the input does not affect execution speed

accessing a specific location in an array

# Big O Notation: O(x)

**O(x):** algorithm executes in **linear time**

Execution speed grows linearly in input size

Example:

# Big O Notation: O(x)

**O(x):** algorithm executes in **linear time**

Execution speed grows linearly in input size

Example:

inserting an element into an arbitrary location in a 1 dimensional array

Bigger array $\rightarrow$ need to shift around more elements in memory to accommodate the new element

# Big O Notation: $O(x^2)$

$\mathbf{O(x^2)}$ **:** algorithm executes in **quadratic time**

More generally called polynomial time for $x^n$

Execution speed grows quadratically in input size

Example:

# Big O Notation: $O(x^2)$

$\mathbf{O(x^2)}$ **:** algorithm executes in **quadratic time**

More generally called polynomial time for $x^n$

Execution speed grows quadratically in input size

Example:

**bubble sort**, step through a list, compare adjacent elements, swap if in the wrong order

# Computer arithmetic

# Computer arithmetic - storage

**Question:** which numbers can be represented by a computer?

# Computer arithmetic - storage

**Question:** which numbers can be represented by a computer?

Before the answer, **how** are numbers physically represented by a computer?

# Computer arithmetic - storage

**Question:** which numbers can be represented by a computer?

Before the answer, **how** are numbers physically represented by a computer?

**Binary:** a base 2 number system

Each digit can only take on 0 or 1

# Computer arithmetic - storage

**Question:** which numbers can be represented by a computer?

Before the answer, **how** are numbers physically represented by a computer?

**Binary:** a base 2 number system

Each digit can only take on 0 or 1

**Base 10:** each digit can take on 0-9

# Computer arithmetic - storage

**Question**: which numbers can be represented by a computer?

# Computer arithmetic - storage

**Question**: which numbers can be represented by a computer?

**Answer:** a **subset** of the rational numbers

# Computer arithmetic - storage

**Question**: which numbers can be represented by a computer?

**Answer:** a **subset** of the rational numbers

Computers have **finite** memory and hard disk space, there are infinite rational numbers

# Computer arithmetic - storage

**Question**: which numbers can be represented by a computer?

**Answer:** a **subset** of the rational numbers

Computers have **finite** memory and hard disk space, there are infinite rational numbers

This imposes a strict limitation on the storage of numbers

# Computer arithmetic - storage

**Question**: which numbers can be represented by a computer?

**Answer:** a **subset** of the rational numbers

Computers have **finite** memory and hard disk space, there are infinite rational numbers

This imposes a strict limitation on the storage of numbers

Numbers are stored as: $\pm m b^{\pm n}$

$m$ is the mantissa/significand, $b$ is the base, $n$ is the exponent

All three are integers

# Computer arithmetic - storage

$\pm m b^{\pm n}$

The significand typically gives the significant digits

The exponent scales the number up or down in magnitude

# Computer arithmetic - storage

The size of numbers a computer can represent is limited
by how much space is typically allocated for a real number

# Computer arithmetic - storage

The size of numbers a computer can represent is limited
by how much space is typically allocated for a real number

Space allocations are usually 64 bits: 53 for $m$ and 11 for $n$

```
println(typeof(5.0))
```

```
## Float64
```

```
println(typeof(5))
```

```
## Int64
```

# Computer arithmetic - storage

`Int64` means it is a **integer** with 64 bits of storage

`Float64` means it is a **floating point number** with 64 bits of storage

Floating point just means $b^{\pm n}$ can move the decimal point around in the significand

`Int64` and `Float64` are different, this will be important later

# The limits of computers

Limitations on storage suggest three facts

# The limits of computers

Limitations on storage suggest three facts

**1.** There exists a **machine epsilon** which denotes the smallest relative quantity representible by a computer

# The limits of computers

Limitations on storage suggest three facts

**1.** There exists a **machine epsilon** which denotes the smallest relative quantity representible by a computer

Machine epsilon is the smallest $\epsilon$ such that a machine can always distinguish

$$N + \epsilon > N > N - \epsilon$$

# The limits of computers

```
println("Machine epsilon ϵ is $(eps(Float64))")
```

```
## Machine epsilon ϵ is 2.220446049250313e-16
```

```
println("Is 1 + ϵ/2 > 1? $(1 + eps(Float64)/2 > 1)")
```

```
## Is 1 + ϵ/2 > 1? false
```

```
println("Is 1 - ϵ/2 < 1? $(1 - eps(Float64)/2 < 1)")
```

```
## Is 1 - ϵ/2 < 1? true
```

# The limits of computers

```
println("The smallest representable number larger than 1.0 is $(nextfloat(1.0))")
```

```
## The smallest representable number larger than 1.0 is 1.0000000000000002
```

```
println("The largest representable number smaller than 1.0 is $(prevfloat(1.0))")
```

```
## The largest representable number smaller than 1.0 is 0.9999999999999999
```

# The limits of computers

Machine epsilon changes depending on the amount of storage allocated

# The limits of computers

## Machine epsilon changes depending on the amount of storage allocated

```
println("32 bit machine epsilon is $(eps(Float32))")
```

```
## 32 bit machine epsilon is 1.1920929e-7
```

```
println("Is 1 + ϵ/2 > 1? $(Float32(1) + eps(Float32)/2 > 1)")
```

```
## Is 1 + ϵ/2 > 1? false
```

```
println("Is 1 - ϵ/2 < 1? $(Float32(1) - eps(Float32)/2 < 1)")
```

```
## Is 1 - ϵ/2 < 1? true
```

# The limits of computers

## Machine epsilon changes depending on the amount of storage allocated

```
println("32 bit machine epsilon is $(eps(Float32))")
```

```
## 32 bit machine epsilon is 1.1920929e-7
```

```
println("Is 1 + ϵ/2 > 1? $(Float32(1) + eps(Float32)/2 > 1)")
```

```
## Is 1 + ϵ/2 > 1? false
```

```
println("Is 1 - ϵ/2 < 1? $(Float32(1) - eps(Float32)/2 < 1)")
```

```
## Is 1 - ϵ/2 < 1? true
```

## This means theres a tradeoff between precision and storage requirements, this matters for low-memory systems like GPUs

# The limits of computers

## 2. There is a smallest representable number

```
println("64 bit smallest float is $(floatmin(Float64))")
```

```
## 64 bit smallest float is 2.2250738585072014e-308
```

```
println("32 bit smallest float is $(floatmin(Float32))")
```

```
## 32 bit smallest float is 1.1754944e-38
```

```
println("16 bit smallest float is $(floatmin(Float16))")
```

```
## 16 bit smallest float is 6.104e-5
```

# The limits of computers

## 3. There is a largest representable number

```
println("64 bit largest float is $(floatmax(Float64))")
```

```
## 64 bit largest float is 1.79769313486231e308
```

```
println("32 bit largest float is $(floatmax(Float32))")
```

```
## 32 bit largest float is 3.4028235e38
```

```
println("16 bit largest float is $(floatmax(Float16))")
```

```
## 16 bit largest float is 6.55e4
```

# The limits of computers: time is a flat circle

```
println("The largest 64 bit integer is $(typemax(Int64))")
```

```
## The largest 64 bit integer is 9223372036854775807
```

```
println("Add one to it and we get: $(typemax(Int64)+1)")
```

```
## Add one to it and we get: -9223372036854775808
```

```
println("It loops us around the number line: $(typemin(Int64))")
```

```
## It loops us around the number line: -9223372036854775808
```

# The limits of computers

**The scale of your problem matters**

# The limits of computers

**The scale of your problem matters**

If a parameter or variable is > floatmax or < floatmin, you will have a very bad time

# The limits of computers

**The scale of your problem matters**

If a parameter or variable is > floatmax or < floatmin, you will have a very bad time

Scale numbers appropriately (e.g. millions of dollars, not millionths of cents)

# Computer arithmetic: Error

We can only represent a finite number of numbers

# Computer arithmetic: Error

We can only represent a finite number of numbers

This means we will have error in our computations

# Computer arithmetic: Error

We can only represent a finite number of numbers

This means we will have error in our computations

Error comes in two major forms:

1. Rounding
2. Truncation

# Rounding

We will always need to round numbers to the nearest computer representable number, this introduces error

```
println("Half of π is: $(π/2)")
```

```
## Half of π is: 1.5707963267948966
```

# Rounding

We will always need to round numbers to the nearest computer representable number, this introduces error

```
println("Half of π is: $(π/2)")
```

```
## Half of π is: 1.5707963267948966
```

The computer gave us a rational number, but $\pi/2$ should be irrational

# Truncation

Lots of important numbers are defined by infinite sums $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$

# Truncation

Lots of important numbers are defined by infinite sums $e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$

It turns out that computers can't add up infinitely many terms because there is finite space

$\rightarrow$ we need to truncate the sum

# Why does this matter?

Errors are small, who cares?

# Why does this matter?

Errors are small, who cares?

You should!

Because errors can propagate and grow as you keep applying an algorithm (e.g. function iteration)

# Error example 1

Consider a simple quadratic: $x^2 - 26x + 1 = 0$ with solution $x = 13 - \sqrt{168}$

# Error example 1

Consider a simple quadratic: $x^2 - 26x + 1 = 0$ with solution $x = 13 - \sqrt{168}$

```
println("64 bit: 13 - √168 = $(13-sqrt(168))")
```

```
## 64 bit: 13 - √168 = 0.038518603184279 24
```

```
println("32 bit: 13 - √168 = $(convert(Float32,13-sqrt(168)))")
```

```
## 32 bit: 13 - √168 = 0.038518604
```

```
println("16 bit: 13 - √168 = $(convert(Float16,13-sqrt(168)))")
```

```
## 16 bit: 13 - √168 = 0.0385
```

# Error example 2

Lets add and subtract some numbers and play around with the associative property of real numbers:

$$x = (10^{-20} + 1) - 1$$
$$y = 10^{-20} + (1 - 1)$$

# Error example 2

Lets add and subtract some numbers and play around with the associative property of real numbers:

$$x = (10^{-20} + 1) - 1$$
$$y = 10^{-20} + (1 - 1)$$

Very clearly we should get $x = y$, but do we? Let's find out

# Error example 2

```
x = (1e-20 + 1) - 1   # initialize x
y = 1e-20 + (1 - 1)   # initialize y
x_equals_y = (x == y) # store boolean of whether x == y
```

# Error example 2

```
x = (1e-20 + 1) - 1   # initialize x
y = 1e-20 + (1 - 1)   # initialize y
x_equals_y = (x == y) # store boolean of whether x == y
```

```
if x_equals_y
    println("X equals Y!")
else
    println("X does not equal Y!")
    println("The difference is: $(x-y).")
end
```

```
## X does not equal Y!
## The difference is: -1.0e-20.
```

# Error example 2

The two numbers were not equal, we got $y > x$

Why?

# Error example 2

The two numbers were not equal, we got $y > x$

Why?

Adding numbers of greatly different magnitudes does not always work like you would want

# Error example 2

```
x = (1e-20 + 1) - 1   # initialize x
y = 1e-20 + (1 - 1)   # initialize y
```

```
println("x is $x")
```

```
## x is 0.0
```

```
println("y is $y")
```

```
## y is 1.0e-20
```

# Error example 2

```
x = (1e-20 + 1) - 1   # initialize x
y = 1e-20 + (1 - 1)   # initialize y
```

```
println("x is $x")
```

```
## x is 0.0
```

```
println("y is $y")
```

```
## y is 1.0e-20
```

When we added $10^{-20}$ to 1, it got rounded away

# Error example 3

Lets just subtract two numbers: 100000.2 - 100000.1

We know the answer is: 0.1

# Error example 3

Lets just subtract two numbers: 100000.2 - 100000.1

We know the answer is: 0.1

```
println("100000.2 - 100000.1 is: $(100000.2 - 100000.1)")
```

```
## 100000.2 - 100000.1 is: 0.09999999999126885
```

```
if (100000.2 - 100000.1) == 0.1
    println("and it is equal to 0.1")
else
    println("and it is not equal to 0.1")
end
```

```
## and it is not equal to 0.1
```

# Error example 3

Why do we get this error?

# Error example 3

Why do we get this error?

Neither of the two numbers can be precisely represented by the machine!

$$100000.1 \approx 8589935450993459 \times 2^{-33} = 100000.09999999997671693563461 30$$
$$100000.2 \approx 8589936309986918 \times 2^{-33} = 100000.19999999995343387126922 61$$

So their difference won't necessarily be 0.1

There are tools for approximate equality

```
isapprox(100000.2 - 100000.1, 0.1)
```

```
## true
```

# Rounding and truncation recap

This matters, particularly when you're trying to evaluate logical expressions of equality

# Calculus on a machine

# Differentiation

Derivatives are obviously important in economics for finding optimal allocations, etc

The formal definition of a derivative is:

# Differentiation

Derivatives are obviously important in economics for finding optimal allocations, etc

The formal definition of a derivative is:

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

# Differentiation

Derivatives are obviously important in economics for finding optimal allocations, etc

The formal definition of a derivative is:

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

But we can let $t = 1/h$ and reframe this as an infinite limit

# Differentiation

Derivatives are obviously important in economics for finding optimal allocations, etc

The formal definition of a derivative is:

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

But we can let $t = 1/h$ and reframe this as an infinite limit

$$\frac{df(x)}{dx} = \lim_{t \to \infty} \frac{f(x+1/t) - f(x)}{1/t}$$

which we know a computer can't handle because of finite space to store $t$

# Computer differentiation

How do we perform derivatives on computers if we can't take the limit?

# Computer differentiation

How do we perform derivatives on computers if we can't take the limit?

**Finite difference methods**

# Computer differentiation

How do we perform derivatives on computers if we can't take the limit?

**Finite difference methods**

# Computer differentiation

How do we perform derivatives on computers if we can't take the limit?

## Finite difference methods

What does a finite difference approximation look like?

# Forward difference

The forward difference looks exactly like the formal definition without the limit:

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

# Forward difference

The forward difference looks exactly like the formal definition without the limit:

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

Works the same for partial derivatives:

$$\frac{\partial g(x,y)}{\partial x} \approx \frac{g(x+h,y) - g(x,y)}{h}$$

Let's see how it works in practice by calculating derivatives of $x^2$ at $x = 2$

# Forward difference

```
deriv_x_squared(h,x) = ((x+h)^2 - x^2)/h # derivative function
```

# Forward difference

```
deriv_x_squared(h,x) = ((x+h)^2 - x^2)/h # derivative function
```

```
println("
        The deriviative with h=1e-8 is: $(deriv_x_squared(1e-8,2.))
        The deriviative with h=1e-12 is: $(deriv_x_squared(1e-12,2.))
        The deriviative with h=1e-30 is: $(deriv_x_squared(1e-30,2.))
        The deriviative with h=1e-1 is: $(deriv_x_squared(1e-1,2.))")
```

```
##
##         The deriviative with h=1e-8 is: 3.999999975690116
##         The deriviative with h=1e-12 is: 4.000355602329364
##         The deriviative with h=1e-30 is: 0.0
##         The deriviative with h=1e-1 is: 4.100000000000001
```

# Error, it's there

None of the values we chose for $h$ were perfect,

but clearly some were better than others

# Error, it's there

None of the values we chose for $h$ were perfect,

but clearly some were better than others

**Why?**

# Error, it's there

None of the values we chose for $h$ were perfect,
but clearly some were better than others

**Why?**

We face two opposing forces:

# Error, it's there

None of the values we chose for $h$ were perfect,
but clearly some were better than others

**Why?**

We face two opposing forces:

- We want $h$ to be as small as possible so that
  we can approximate the limit as well as we possibly can, BUT

# Error, it's there

None of the values we chose for $h$ were perfect,
but clearly some were better than others

**Why?**

We face two opposing forces:

- We want $h$ to be as small as possible so that
  we can approximate the limit as well as we possibly can, BUT
- If $h$ is small then $f(x+h)$ is close to $f(x)$,
  we can run into rounding issues like we saw for $h = 10^{-30}$

# Error, it's there

We can select $h$ in an optimal fashion: $h = \max\{|x|, 1\}\sqrt{\epsilon}$

# Error, it's there

We can select $h$ in an optimal fashion: $h = \max\{|x|, 1\}\sqrt{\epsilon}$

There's proofs for why this is the case but generally testing out different $h$'s works fine

# How much error is in a finite difference?

Can we measure the error growth rate in $h$ (i.e. Big O notation)?

# How much error is in a finite difference?

Can we measure the error growth rate in $h$ (i.e. Big O notation)?

Perform a first-order taylor expansion of $f(x)$ around $x$:

# How much error is in a finite difference?

Can we measure the error growth rate in $h$ (i.e. Big O notation)?

Perform a first-order taylor expansion of $f(x)$ around $x$:

$$f(x+h) = f(x) + f'(x)h + O(h^2)$$

Recall $O(h^2)$ means the error in our approximation grows quadratically in $h$, we only did a linear approximation

# How much error is in a finite difference?

Can we measure the error growth rate in $h$ (i.e. Big O notation)?

Perform a first-order taylor expansion of $f(x)$ around $x$:

$$f(x+h) = f(x) + f'(x)h + O(h^2)$$

Recall $O(h^2)$ means the error in our approximation grows quadratically in $h$, we only did a linear approximation

How can we use this to understand the error in our finite difference approximation?

# How much error is in a finite difference?

Rearrange to obtain: $f'(x) = \frac{f(x+h)-f(x)}{h} + O(h^2)/h$

# How much error is in a finite difference?

Rearrange to obtain: $f'(x) = \frac{f(x+h)-f(x)}{h} + O(h^2)/h$

$f'(x) = \frac{f(x+h)-f(x)}{h} + O(h)$ because $O(h^2)/h = O(h)$

# How much error is in a finite difference?

Rearrange to obtain: $f'(x) = \frac{f(x+h)-f(x)}{h} + O(h^2)/h$

$f'(x) = \frac{f(x+h)-f(x)}{h} + O(h)$ because $O(h^2)/h = O(h)$

**Forward differences have linearly growing errors**

# How much error is in a finite difference?

Rearrange to obtain: $f'(x) = \frac{f(x+h)-f(x)}{h} + O(h^2)/h$

$f'(x) = \frac{f(x+h)-f(x)}{h} + O(h)$ because $O(h^2)/h = O(h)$

**Forward differences have linearly growing errors**

If we halve $h$, we halve the error in our approximation (ignoring rounding/truncation issues)

# Improvements on the forward difference

How can we improve the accuracy of the forward difference?

# Improvements on the forward difference

How can we improve the accuracy of the forward difference?

First, **why** do we have error?

# Improvements on the forward difference

How can we improve the accuracy of the forward difference?

First, **why** do we have error?

Because we are approximating the slope of a tangent curve at $x$ by a secant curve passing through $(x, x+h)$

# Improvements on the forward difference

How can we improve the accuracy of the forward difference?

First, **why** do we have error?

Because we are approximating the slope of a tangent curve at $x$ by a secant curve passing through $(x, x+h)$

The secant curve has the average slope of $f(x)$ on $[x, x+h]$

# Improvements on the forward difference

How can we improve the accuracy of the forward difference?

First, **why** do we have error?

Because we are approximating the slope of a tangent curve at $x$ by a secant curve passing through $(x, x+h)$

The secant curve has the average slope of $f(x)$ on $[x, x+h]$

We want the derivative at $x$, which is on the edge of $[x, x+h]$, how about we **center** $x$?

# Central differences

We can approximate $f'(x)$ in a slightly different way:

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

# Central differences

We can approximate $f'(x)$ in a slightly different way:

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

This leaves $x$ in the middle of the interval
over which we are averaging the slope of $f(x)$

# Central differences

We can approximate $f'(x)$ in a slightly different way:

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

This leaves $x$ in the middle of the interval
over which we are averaging the slope of $f(x)$

**Is this an improvement on forward differences?**

# How much error is in a central finite difference?

Lets do two second-order Taylor expansions:

- $f(x+h) = f(x) + f'(x)h + f''(x)h^2/2! + O(h^3)$
- $f(x-h) = f(x) + f'(x)(-h) + f''(x)(-h)^2/2! + O(h^3)$

# How much error is in a central finite difference?

Lets do two second-order Taylor expansions:

- $f(x+h) = f(x) + f'(x)h + f''(x)h^2/2! + O(h^3)$
- $f(x-h) = f(x) + f'(x)(-h) + f''(x)(-h)^2/2! + O(h^3)$

Subtract the two expressions (note that $O(h^3) - O(h^3) = O(h^3)$)
and then divide by $2h$ to get

# How much error is in a central finite difference?

Lets do two second-order Taylor expansions:

- $f(x+h) = f(x) + f'(x)h + f''(x)h^2/2! + O(h^3)$
- $f(x-h) = f(x) + f'(x)(-h) + f''(x)(-h)^2/2! + O(h^3)$

Subtract the two expressions (note that $O(h^3) - O(h^3) = O(h^3)$) and then divide by $2h$ to get

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

# How much error is in a central finite difference?

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

# How much error is in a central finite difference?

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

Error falls quadratically in $h$, if we halve $h$ we reduce error by 75%

# How much error is in a central finite difference?

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

Error falls quadratically in $h$, if we halve $h$ we reduce error by 75%

Optimal selection of $h$ for central differences is $h = \max\{|x|, 1\}\epsilon^{1/3}$

# Why use anything but central differences?

Suppose we're computing a Jacobian of a multidimensional function,
why would we ever use forward differences instead of central differences?

# Why use anything but central differences?

Suppose we're computing a Jacobian of a multidimensional function,
why would we ever use forward differences instead of central differences?

For each central difference:

# Why use anything but central differences?

Suppose we're computing a Jacobian of a multidimensional function,
why would we ever use forward differences instead of central differences?

For each central difference:

We need to compute $g(x_1 - h, x_2, \dots)$ and $g(x_1 + h, x_2, \dots)$ for each $x_i$

# Why use anything but central differences?

Suppose we're computing a Jacobian of a multidimensional function, why would we ever use forward differences instead of central differences?

For each central difference:

We need to compute $g(x_1 - h, x_2, \dots)$ and $g(x_1 + h, x_2, \dots)$ for each $x_i$

But for a forward difference we only need to compute $g(x_1, x_2, \dots)$ once and then $g(x_1 + h, x_2, \dots)$ for each $x_i$

# Why use anything but central differences?

Suppose we're computing a Jacobian of a multidimensional function, why would we ever use forward differences instead of central differences?

For each central difference:

We need to compute $g(x_1 - h, x_2, \dots)$ and $g(x_1 + h, x_2, \dots)$ for each $x_i$

But for a forward difference we only need to compute $g(x_1, x_2, \dots)$ once and then $g(x_1 + h, x_2, \dots)$ for each $x_i$

Forward differences saves on # of operations at the expense of accuracy

For high dimensional functions it may be worth the tradeoff

# Higher order finite differences

We can use these techniques to approximate higher order derivatives

# Higher order finite differences

We can use these techniques to approximate higher order derivatives

For example, take two third order Taylor expansions

# Higher order finite differences

We can use these techniques to approximate higher order derivatives

For example, take two third order Taylor expansions

- $f(x+h) = f(x) + f'(x)h + f''(x)h^2/2! + f'''(x)h^3/3! + O(h^4)$
- $f(x-h) = f(x) + f'(x)(-h) + f''(x)(-h)^2/2! + f'''(x)(-h)^3/3! + O(h^4)$

# Higher order finite differences

We can use these techniques to approximate higher order derivatives

For example, take two third order Taylor expansions

- $f(x+h) = f(x) + f'(x)h + f''(x)h^2/2! + f'''(x)h^3/3! + O(h^4)$
- $f(x-h) = f(x) + f'(x)(-h) + f''(x)(-h)^2/2! + f'''(x)(-h)^3/3! + O(h^4)$

Add the two expressions and then divide by $h^2$ to get

# Higher order finite differences

We can use these techniques to approximate higher order derivatives

For example, take two third order Taylor expansions

- $f(x+h) = f(x) + f'(x)h + f''(x)h^2/2! + f'''(x)h^3/3! + O(h^4)$
- $f(x-h) = f(x) + f'(x)(-h) + f''(x)(-h)^2/2! + f'''(x)(-h)^3/3! + O(h^4)$

Add the two expressions and then divide by $h^2$ to get

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2)$$

# Higher order finite differences

We can use these techniques to approximate higher order derivatives

For example, take two third order Taylor expansions

- $f(x+h) = f(x) + f'(x)h + f''(x)h^2/2! + f'''(x)h^3/3! + O(h^4)$
- $f(x-h) = f(x) + f'(x)(-h) + f''(x)(-h)^2/2! + f'''(x)(-h)^3/3! + O(h^4)$

Add the two expressions and then divide by $h^2$ to get

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2)$$

Second derivatives are important for calculating Hessians
and checking maxima or minima

# Differentiation without error?

Finite differences put us in between two opposing forces on the size of $h$

# Differentiation without error?

Finite differences put us in between two opposing forces on the size of $h$

Can we improve upon finite differences?

# Differentiation without error?

Finite differences put us in between two opposing forces on the size of $h$

Can we improve upon finite differences?

**Analytic derivatives**

One way is to code up the actual derivative

# Differentiation without error?

Finite differences put us in between two opposing forces on the size of $h$

Can we improve upon finite differences?

## Analytic derivatives

One way is to code up the actual derivative

```
deriv_x_squared(x) = 2x
```

```
## The deriviative is: 4.0
```

# Differentiation without error?

Finite differences put us in between two opposing forces on the size of $h$

Can we improve upon finite differences?

## Analytic derivatives

One way is to code up the actual derivative

```
deriv_x_squared(x) = 2x
```

```
## The deriviative is: 4.0
```

Exact solution!

# Automatic differentiation

Coding up analytic derivatives by hand for complex problems is not always great because

# Automatic differentiation

Coding up analytic derivatives by hand for
complex problems is not always great because

It can take A LOT of programmer time, more than it is worth

# Automatic differentiation

Coding up analytic derivatives by hand for
complex problems is not always great because

It can take A LOT of programmer time, more than it is worth

Humans are suseptible to error in coding or calculating the derivative mathematically

# Autodiff: let the computer do it

Think about this: your code is **always** made up of simple arithmetic operations

- add, subtract, divide, multiply
- trig functions
- exponentials/logs
- etc

# Autodiff: let the computer do it

Think about this: your code is **always** made up of simple arithmetic operations

- add, subtract, divide, multiply
- trig functions
- exponentials/logs
- etc

The closed form derivatives of these operations is not hard, it turns out your computer can do it and yield exact solutions

# Autodiff: let the computer do it

How?

# Autodiff: let the computer do it

How?

There are methods that basically apply a giant chain rule to your whole program, and break down the derivative into the (easy) component parts that another package knows how to handle

# Autodiff: let the computer do it

How?

There are methods that basically apply a giant chain rule to your whole program, and break down the derivative into the (easy) component parts that another package knows how to handle

```
# the function, it needs to be written in a particular way because the autodiff package is dumb
# *(x,y) is the julia function way to do x*y,
# autodiff needs to be in terms of julia functions to work correctly ¯\_(ツ)_/¯
ff(x) = *(x[1],x[1]) # x^2
x = [2.]; # location to evaluate: ff(x) = 2^2
```

# Autodiff: let the computer do it

```
using ForwardDiff
g(f,x) = ForwardDiff.gradient(f,x); # g = ∇f
```

```
## g (generic function with 1 method)
```

```
println("ff'(x) at $(x[1]) is: $(g(ff,x)[1])") # display gradient value
```

```
## ff'(x) at 2.0 is: 4.0
```

# Autodiff: let the computer do it

```
using ForwardDiff
g(f,x) = ForwardDiff.gradient(f,x); # g = ∇f
```

```
## g (generic function with 1 method)
```

```
println("ff'(x) at $(x[1]) is: $(g(ff,x)[1])") # display gradient value
```

```
## ff'(x) at 2.0 is: 4.0
```

## Exact solution!

# Autodiff: let the computer do it

Once you get the hang of coding up function for autodiff it's not that hard

```
fff(x) = sin(*(x[1],x[1])) # x^2
```

# Autodiff: let the computer do it

## Once you get the hang of coding up function for autodiff it's not that hard

```
fff(x) = sin(*(x[1],x[1])) # x^2
```

```
x = [2.]                                    # location to evaluate: ff(x) = 2^2
```

```
## 1-element Array{Float64,1}:
##  2.0
```

```
g(f,x) = ForwardDiff.gradient(f,x)        # g = ∇f
```

```
## g (generic function with 1 method)
```

```
println("fff'(x) at $(x[1]) is: $(g(fff,x)[1])") # display gradient value
```

```
## fff'(x) at 2.0 is: -2.6145744834544478
```

# Calculus operations

Integration, trickier than differentiation

# Calculus operations

Integration, trickier than differentiation

We integrate to do a lot of stuff in economics

# Calculus operations

Integration, trickier than differentiation

We integrate to do a lot of stuff in economics

- Expectations
- Add up a continuous measure of things

# Calculus operations

Integration, trickier than differentiation

We integrate to do a lot of stuff in economics

- Expectations
- Add up a continuous measure of things

$\int_D f(x)dx$, $f : \mathcal{R}^n - \mathcal{R}$, $D \subset \mathcal{R}^n$

# How to think about integrals

Integrals are effectively infinite sums

# How to think about integrals

Integrals are effectively infinite sums

1 dimensional example:

$\lim_{dx_i \to 0} \sum_{i=0}^{(a-b)/dx_i} f(x_i) dx_i$

where $dx_i$ is some subset of $[a, b]$ and $x_i$ is some evaluation point (e.g. midpoint of $dx_i$)

# Infinite limits strike again

Just like derivatives, we face an infinite limit as $(a - b)/dx_i \to \infty$

We avoid this issue in the same way as derivatives, we replace the infinite sum with something we can handle

# Monte Carlo integration

Probably the most commonly used form in empirical econ

# Monte Carlo integration

Probably the most commonly used form in empirical econ

Approximate an integral by relying on LLN
and "randomly" sampling the integration domain

# Monte Carlo integration

Probably the most commonly used form in empirical econ

Approximate an integral by relying on LLN
and "randomly" sampling the integration domain

Can be effective for very high dimensional integrals

Very simple and intuitive

But, produces a random approximation

# Monte Carlo integration

Integrate $\xi = \int_0^1 f(x)dx$ by drawing $N$ uniform samples, $x_1, \ldots, x_N$ over interval $[0, 1]$

# Monte Carlo integration

Integrate $\xi = \int_0^1 f(x)dx$ by drawing $N$ uniform samples, $x_1, \ldots, x_N$ over interval $[0, 1]$

$\xi$ is equivalent to $E[f(x)]$ with respect to a uniform distribution, so estimating the integral is the same as estimating the expected value of $f(x)$

# Monte Carlo integration

Integrate $\xi = \int_0^1 f(x)dx$ by drawing $N$ uniform samples, $x_1, \ldots, x_N$ over interval $[0, 1]$

$\xi$ is equivalent to $E[f(x)]$ with respect to a uniform distribution, so estimating the integral is the same as estimating the expected value of $f(x)$

In general we have that $\hat{\xi} = V \frac{1}{N} \sum_{i=1}^{N} f(x_i)$

where $V$ is the volume over which we are integrating

# Monte Carlo integration

Integrate $\xi = \int_0^1 f(x)dx$ by drawing $N$ uniform samples, $x_1, \ldots, x_N$ over interval $[0, 1]$

$\xi$ is equivalent to $E[f(x)]$ with respect to a uniform distribution, so estimating the integral is the same as estimating the expected value of $f(x)$

In general we have that $\hat{\xi} = V \frac{1}{N} \sum_{i=1}^{N} f(x_i)$

where $V$ is the volume over which we are integrating

LLN gives us that the $plim_{N \to \infty} \hat{\xi} = \xi$

# Monte Carlo integration

The variance of $\hat{\xi}$ is

$$\sigma^2_{\hat{\xi}} = var(\frac{V}{N}\sum_{i=1}^{N} f(x_i)) = \frac{V^2}{N^2}\sum_{i=1}^{N} var(f(X)) = \frac{V^2}{N}\sigma^2_{f(X)}$$

# Monte Carlo integration

The variance of $\hat{\xi}$ is

$$\sigma^2_{\hat{\xi}} = var(\frac{V}{N}\sum_{i=1}^{N} f(x_i)) = \frac{V^2}{N^2}\sum_{i=1}^{N} var(f(X)) = \frac{V^2}{N}\sigma^2_{f(X)}$$

So average error is $\frac{V}{\sqrt{N}}\sigma_{f(X)}$, this gives us its rate of convergence: $O(\sqrt{N})$

Note:

1. The rate of convergence is independent of the dimension of x
2. Quasi-Monte Carlo methods can get you $O(1/N)$

# Monte Carlo integration

Suppose we want to integrate $x^2$ from 0 to 10, we know this is $10^3/3 = 333.333$

```
# Package for drawing random numbers
using Distributions

# Define a function to do the integration for an arbitrary function
function integrate_function(f, lower, upper, num_draws)

  # Draw from a uniform distribution
  xs = rand(Uniform(lower, upper), num_draws)

  # Expectation = mean(x)*volume
  expectation = mean(f(xs))*(upper - lower)

end
```

# Monte Carlo integration

Suppose we want to integrate $x^2$ from 0 to 10, we know this is $10^3/3 = 333.333$

```
# Integrate
f(x) = x.^2;
integrate_function(f, 0, 10, 1000)
```

```
## 341.6592011016046
```

Pretty close!

# Quadrature rules

We can also approximate integrals using a technique called **quadrature**

# Quadrature rules

We can also approximate integrals using a technique called **quadrature**

With quadrature we effectively take weighted sums to approximate integrals

# Quadrature rules

We can also approximate integrals using a technique called **quadrature**

With quadrature we effectively take weighted sums to approximate integrals

We will focus on two classes of quadrature for now:

1. Newton-Cotes (the kind you've seen before)
2. Gaussian (probably new)

# Newton-Cotes quadrature rules

Suppose we want to integrate a one dimensional function $f(x)$ over $[a, b]$

How would you do it?

# Newton-Cotes quadrature rules

Suppose we want to integrate a one dimensional function $f(x)$ over $[a, b]$

How would you do it?

One answer is to replace the function with
something easy to integrate: **a piecewise polynomial**

# Newton-Cotes quadrature rules

Suppose we want to integrate a one dimensional function $f(x)$ over $[a, b]$

How would you do it?

One answer is to replace the function with
something easy to integrate: **a piecewise polynomial**

Key things to define up front:

- $x_i = a + (i-1)/h$ for $i = 1, 2, \ldots, n$ where $h = \frac{b-a}{n-1}$

$x_i$s are the **quadrature nodes** of the approximation scheme and divide the interval into $n-1$ equally spaced subintervals of length $h$

# Midpoint rule

Most basic Newton-Cotes method:

1. Split $[a, b]$ into intervals
2. Approximate the function in each subinterval by a constant equal to the function at the midpoint of the subinterval

# Midpoint rule

Most basic Newton-Cotes method:

1. Split $[a, b]$ into intervals
2. Approximate the function in each subinterval by a constant equal to the function at the midpoint of the subinterval

$\int_{x_i}^{x_{i+1}} f(x)dx \approx hf(\frac{1}{2}(x_{i+1} + x_i))$

# Midpoint rule

Most basic Newton-Cotes method:

1. Split $[a, b]$ into intervals
2. Approximate the function in each subinterval by a constant equal to the function at the midpoint of the subinterval

$\int_{x_i}^{x_{i+1}} f(x)dx \approx hf(\frac{1}{2}(x_{i+1} + x_i))$

Approximates $f$ by a step function

# Trapezoid rule

Increase complexity by 1 degree:

1. Split $[a, b]$ into intervals
2. Approximate the function in each subinterval by a linear interpolation passing through $(x_i, f(x_i))$ and $(x_{i+1}, f(x_{i+1}))$

# Trapezoid rule

Increase complexity by 1 degree:

1. Split $[a, b]$ into intervals
2. Approximate the function in each subinterval by a linear interpolation passing through $(x_i, f(x_i))$ and $(x_{i+1}, f(x_{i+1}))$

$\int_{x_i}^{x_{i+1}} f(x)dx \approx \frac{h}{2}[f(x_i) + f(x_{i+1})]$

# Trapezoid rule

Increase complexity by 1 degree:

1. Split $[a, b]$ into intervals
2. Approximate the function in each subinterval by a linear interpolation passing through $(x_i, f(x_i))$ and $(x_{i+1}, f(x_{i+1}))$

$\int_{x_i}^{x_{i+1}} f(x)dx \approx \frac{h}{2}[f(x_i) + f(x_{i+1})]$

We can aggregate this up to: $\int_a^b f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i)$
where $w_1 = w_n = h/2$ and $w_i = h$ otherwise

# How accurate is this rule?

Trapezoid rule is $O(h^2)$ / first-order exact: it can integrate any linear function exactly

# Simpsons rule

Increase complexity by 1 degree:

1. Let $n$ be odd, then approximate the function across a **pair** of subintervals by a quadratic interpolation passing through $(x_{2i-1}, f(x_{2i-i}))$, $(x_{2i}, f(x_{2i}))$, and $(x_{2i+1}, f(x_{2i+1}))$

# Simpsons rule

Increase complexity by 1 degree:

1. Let $n$ be odd, then approximate the function across a **pair** of subintervals by a quadratic interpolation passing through $(x_{2i-1}, f(x_{2i-i}))$, $(x_{2i}, f(x_{2i}))$, and $(x_{2i+1}, f(x_{2i+1}))$

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx \frac{h}{3}[f(x_{2i-1}) + 4f(x_{2i}) + f(x_{2i+1})]$$

# Simpsons rule

Increase complexity by 1 degree:

1. Let $n$ be odd, then approximate the function across a **pair** of subintervals by a quadratic interpolation passing through $(x_{2i-1}, f(x_{2i-i}))$, $(x_{2i}, f(x_{2i}))$, and $(x_{2i+1}, f(x_{2i+1}))$

$$\int_{x_i}^{x_{i+1}} f(x)dx \approx \frac{h}{3}[f(x_{2i-1}) + 4f(x_{2i}) + f(x_{2i+1})]$$

We can aggregate this up to: $\int_a^b f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i)$
where $w_1 = w_n = h/3$, otherwise and $w_i = 4h/3$ if $i$ is even and $w_i = 2h/3$ if $i$ is odd

# How accurate is this rule?

Simpson's rule is $O(h^4)$ / third-order exact: it can integrate any cubic function exactly

# How accurate is this rule?

Simpson's rule is $O(h^4)$ / third-order exact: it can integrate any cubic function exactly

That's weird! Why do we gain 2 orders of accuracy when increasing one order of approximation complexity?

# How accurate is this rule?

Simpson's rule is $O(h^4)$ / third-order exact: it can integrate any cubic function exactly

That's weird! Why do we gain 2 orders of accuracy when increasing one order of approximation complexity?

1. The approximating piecewise quadratic is exact at the end points and midpoint of the conjoined two subintervals

# How accurate is this rule?

Simpson's rule is $O(h^4)$ / third-order exact: it can integrate any cubic function exactly

That's weird! Why do we gain 2 orders of accuracy when increasing one order of approximation complexity?

1. The approximating piecewise quadratic is exact at the end points and midpoint of the conjoined two subintervals
2. Clearly the difference between a cubic $f(x)$ and the quadratic approximation in $[x_{2i-1}, x_{2i+1}]$ is another cubic function

# How accurate is this rule?

Simpson's rule is $O(h^4)$ / third-order exact: it can integrate any cubic function exactly

That's weird! Why do we gain 2 orders of accuracy when increasing one order of approximation complexity?

1. The approximating piecewise quadratic is exact at the end points and midpoint of the conjoined two subintervals
2. Clearly the difference between a cubic $f(x)$ and the quadratic approximation in $[x_{2i-1}, x_{2i+1}]$ is another cubic function
3. This cubic function is **odd** with respect to the midpoint $\rightarrow$ integrating over the first subinterval cancels integrating over the second subinterval

# Gaussian quadrature rules

How did we pick the $x_i$ quadrature nodes for Newton-Cotes rules?

# Gaussian quadrature rules

How did we pick the $x_i$ quadrature nodes for Newton-Cotes rules?

Evenly spaced, but no particular reason for doing so...

# Gaussian quadrature rules

How did we pick the $x_i$ quadrature nodes for Newton-Cotes rules?

Evenly spaced, but no particular reason for doing so...

Gaussian quadrature selects these nodes more efficiently
and relies on **weight functions** $w(x)$

# Gaussian quadrature rules

Gaussian rules try to **exactly integrate** some finite dimensional collection of functions (i.e. polynomials up to some degree)

# Gaussian quadrature rules

Gaussian rules try to **exactly integrate** some finite dimensional collection of functions (i.e. polynomials up to some degree)

For a given order of approximation $n$, the weights $w_1, \ldots, w_n$ and nodes $x_1, \ldots, x_n$ are chosen to satisfy $2n$ **moment matching conditions**:

# Gaussian quadrature rules

Gaussian rules try to **exactly integrate** some finite dimensional collection of functions (i.e. polynomials up to some degree)

For a given order of approximation $n$, the weights $w_1, \ldots, w_n$ and nodes $x_1, \ldots, x_n$ are chosen to satisfy $2n$ **moment matching conditions**:

$\int_I x^k w(x) dx = \sum_{i=1}^n w_i x_i^k$, for $k = 0, \ldots, 2n-1$

where $I$ is the interval over which we are integrating
and $w(x)$ is a given weight function

# Gaussian quadrature improves accuracy

The moment matching conditions pin down $w_i$s and $x_i$s so we can approximate an integral by a weighted sum of the function at the prescribed nodes

# Gaussian quadrature improves accuracy

The moment matching conditions pin down $w_i$s and $x_i$s so we can approximate an integral by a weighted sum of the function at the prescribed nodes

$$\int_i f(x)w(x)dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

# Gaussian quadrature improves accuracy

The moment matching conditions pin down $w_i$s and $x_i$s so we can approximate an integral by a weighted sum of the function at the prescribed nodes

$$\int_i f(x)w(x)dx \approx \sum_{i=1}^{n} w_i f(x_i)$$

Gaussian rules are $2n-1$ order exact,
we can exactly compute the integral of any polynomial order $2n-1$

# Gaussian quadrature takeaways

Gaussian quadrature effectively discretizes some distribution $p(x)$ into mass points (nodes) and probabilities (weights) for some other discrete distribution $\bar{p}(x)$

# Gaussian quadrature takeaways

Gaussian quadrature effectively discretizes some distribution $p(x)$ into mass points (nodes) and probabilities (weights) for some other discrete distribution $\bar{p}(x)$

Given an approximation with $n$ mass points, $X$ and $\bar{X}$ have identical moments up to order $2n$, and as $n \to \infty$ we have a continuum of mass points and recover the continuous pdf

# Gaussian quadrature takeaways

Gaussian quadrature effectively discretizes some distribution $p(x)$ into mass points (nodes) and probabilities (weights) for some other discrete distribution $\bar{p}(x)$

Given an approximation with $n$ mass points, $X$ and $\bar{X}$ have identical moments up to order $2n$, and as $n \to \infty$ we have a continuum of mass points and recover the continuous pdf

But what do we pick for the weighting function $w(x)$?

# Gauss-Legendre

We can start out with a simple $w(x) = 1$, this gives us **Gauss-Legendre** quadrature

This can approximate the integral of any function arbitrarily well by increasing $n$

# Gauss-Laguerre

Sometimes we want to compute exponentially discounted sums like:
$\int_I f(x)e^{-x}dx$

The weighting function $e^{-x}$ is Gauss-Laguerre quadrature

# Gauss-Hermite

Sometimes we want to take expectations of normally distributed variables: $\int_I f(x)e^{-x^2}dx$

There exist packages or look-up tables to get the prescribed weights and nodes for each of these schemes

# Linear Algebra

Lots of computational problems break down into linear systems

Many non-linear models are linearized

How do we **actually** solve these systems inside the machine?

# L-U Factorization

If $A$ in $Ax = b$ is upper or lower triangular,

we can solve for $x$ recursively via forward/backward substitution

# L-U Factorization

If $A$ in $Ax = b$ is upper or lower triangular,
we can solve for $x$ recursively via forward/backward substitution

Consider a lower triangular matrix

# L-U Factorization

If $A$ in $Ax = b$ is upper or lower triangular,
we can solve for $x$ recursively via forward/backward substitution

Consider a lower triangular matrix

The first element is the only non-zero value in the first row so $x_1$ is easy to solve for

# L-U Factorization

If $A$ in $Ax = b$ is upper or lower triangular,
we can solve for $x$ recursively via forward/backward substitution

Consider a lower triangular matrix

The first element is the only non-zero value in the first row so $x_1$ is easy to solve for

The equation in row 2 contains $x_2$ and the already solved for $x_1$
so we can easily solve for $x_2$ and then continue until we solve for all $x$s

# Forward substitution

Forward substitution gives us solutions

$x_i = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j \right)$, for all $i$

# Forward substitution

Forward substitution gives us solutions

$x_i = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j \right)$, for all $i$

# Forward substitution

Forward substitution gives us solutions

$x_i = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j \right)$, for all $i$

L-U factorization is an algorithm that decomposes $A$
into the product of lower and upper triangular matrices

# L-U Factorization has two steps

1. Factor $A$ into lower $L$ and upper $U$ triangular matrices using Gaussian elimination
   - We can do this for any non-singular square matrix

# L-U Factorization has two steps

1. Factor $A$ into lower $L$ and upper $U$ triangular matrices using Gaussian elimination
    - We can do this for any non-singular square matrix
2. Solve for $x$
    1. $(LU)x = b$
    2. Solve for $y$ : $Ly = b$ using forward substitution
    3. Using the solved $y$, we know $Ux = y$ and can solve with backward substitution

# Why bother with this scheme?

Why not just use another method like Cramer's rule?

# Why bother with this scheme?

Why not just use another method like Cramer's rule?

**Speed**

# Why bother with this scheme?

Why not just use another method like Cramer's rule?

**Speed**

LU is less than $O(n^3)$

# Why bother with this scheme?

Why not just use another method like Cramer's rule?

**Speed**

LU is less than $O(n^3)$

Cramer's rule is $O(n! \times n)$

# Why bother with this scheme?

Why not just use another method like Cramer's rule?

**Speed**

LU is less than $O(n^3)$

Cramer's rule is $O(n! \times n)$

For a 10x10 system this can really matter

# Example: LU vs Cramer

Julia description of the division operator `\`: *If A is upper or lower triangular (or diagonal), no factorization of A is required and the system is solved with either forward or backward substitution. For non-triangular square matrices, an LU factorization is used.*

So we can do LU factorization approaches to solutions by just doing `x = A\b`, but we can write it ourselves as well

# Example: LU vs Cramer

## Cramer's Rule can be written as a simple loop:

```
function solve_cramer(A, b)

    dets = Vector(undef, length(b))

    for index in eachindex(b)
        B = copy(A)
        B[:, index] = b
        dets[index] = det(B)
    end

    return dets ./ det(A)

end
```

```
n = 100
A = rand(n, n)
b = rand(n)
```

# Example: LU vs Cramer

## Let's see the full results of the competition for a 10x10:

```
using BenchmarkTools
cramer_time = @elapsed solve_cramer(A, b);
cramer_allocation = @allocated solve_cramer(A, b);
lu_time = @elapsed A\b;
lu_allocation = @allocated A\b;

println("Cramer's rule solved in $cramer_time seconds and used $cramer_allocation kilobytes of me
LU solved in $(lu_time) seconds and used $(lu_allocation) kilobytes of memory.
LU is $(round(cramer_time/lu_time, digits = 0)) times faster and uses $(round(cramer_allocation/
```

```
## Cramer's rule solved in 0.036732678 seconds and used 16193280 kilobytes of memory.
## LU solved in 0.000147926 seconds and used 81872 kilobytes of memory.
## LU is 248.0 times faster and uses 198.0 times less memory.
```

# Mechanics of factorizing

Gaussian elimination is where we use row operations

1. swapping rows
2. multiplying by non-zero scalars
3. add a scalar multiple of one row to another

# Mechanics of factorizing

Gaussian elimination is where we use row operations

1. swapping rows
2. multiplying by non-zero scalars
3. add a scalar multiple of one row to another

to turn a matrix $(I\,A)$ into $(L\,U)$

# Numerical error blow up

Small errors can have big effects, for example:

$$\begin{bmatrix} -M^{-1} & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

where $M$ is big

# Numerical error blow up

Small errors can have big effects, for example:

$$\begin{bmatrix} -M^{-1} & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

where $M$ is big

Lets use L-U Factorization to solve it:

$$\begin{bmatrix} -M^{-1} & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} -M^{-1} & 1 \\ 1 & 1 \end{bmatrix}$$

# Numerical error blow up

Subtract $-M$ times the first row from the second to get the L-U factorization

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} -M^{-1} & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -M & 1 \end{bmatrix} \begin{bmatrix} -M^{-1} & 1 \\ 0 & M+1 \end{bmatrix}$$

# Numerical error blow up

Subtract $-M$ times the first row from the second to get the L-U factorization

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} -M^{-1} & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ -M & 1 \end{bmatrix} \begin{bmatrix} -M^{-1} & 1 \\ 0 & M+1 \end{bmatrix}$$

We can get closed-form solutions by applying forward substitution:

$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} M/(M+1) \\ (M+2)/(M+1) \end{bmatrix}$$

# Numerical issues

Both variables are approximately 1 for large $M$,

but remember adding small numbers to big numbers causes problems numerically

# Numerical issues

Both variables are approximately 1 for large $M$,
but remember adding small numbers to big numbers causes problems numerically

If $M = 10000000000000000000$, the computer will return $x_2$
is equal to precisely 1, this isn't terribly wrong

# Numerical issues

Both variables are approximately 1 for large $M$,
but remember adding small numbers to big numbers causes problems numerically

If $M = 10000000000000000000$, the computer will return $x_2$
is equal to precisely 1, this isn't terribly wrong

When we then perform the second step of backwards substitution, we solve for $x_1 = -M(1 - x_2) = 0$, this is **very** wrong

Large errors like this often occur because diagonal elements are very small

# Julia example

```
function solve_lu(M)
    b = [1, 2]
    U = [-M^-1 1; 0 M+1]
    L = [1. 0; -M 1.]
    y = L\b
    # Round element-wise to 3 digits
    x = round.(U\y, digits = 5)
end;

true_solution(M) = round.([M/(M+1), (M+2)/(M+1)], digits = 5);
```

# Julia example

```
## True solution for M=10 is approximately [0.90909, 1.09091], computed solution is [0.90909, 1.09091]

## True solution for M=1e10 is approximately [1.0, 1.0], computed solution is [1.0, 1.0]

## True solution for M=1e15 is approximately [1.0, 1.0], computed solution is [1.11022, 1.0]

## True solution for M=1e20 is approximately [1.0, 1.0], computed solution is [-0.0, 1.0]

## Julia's division operator is actually pretty smart though, true solution for M=1e20 is A\b = [1.0, 1
```

# Ill-conditioning

A matrix $A$ is said to be ill-conditioned if

a small perturbation in $b$ yields a large change in $x$

# Ill-conditioning

A matrix $A$ is said to be ill-conditioned if
a small perturbation in $b$ yields a large change in $x$

One way to measure ill-conditioning in a matrix is the elasticity of the solution with respect to $b$,

$$\sup_{||\delta b||>0} \frac{||\delta x||/||x||}{||\delta b||/||b||}$$

which yields the percent change in $x$ given
a percentage point change in the magnitude of $b$

# Ill-conditioning

If this elasticity is large, then computer representations of the system of equations can lead to large errors due to rounding

# Ill-conditioning

If this elasticity is large, then computer representations of the system of equations can lead to large errors due to rounding

Approximate the elasticity by computing the **condition number**

$$\kappa = ||A|| \cdot ||A^{-1}||$$

# Ill-conditioning

If this elasticity is large, then computer representations of the system of equations can lead to large errors due to rounding

Approximate the elasticity by computing the **condition number**

$$\kappa = ||A|| \cdot ||A^{-1}||$$

$\kappa$ gives the least upper bound of the elasticity

$\kappa$ is always larger than one and a rule of thumb is that for every order of magnitude, a significant digit is lost in the computation of $x$

```
cond([1. 1.; 1. 1.00000001])
```

# Next week

Coding, generic coding, reproducibility, the shell