# AGEC 652 - Lecture 2.2

## Numerical Calculus

Diego S. Cardoso

Spring 2022

# Course roadmap

1. Intro to Scientific Computing
2. **Numerical operations and representations**
    1. Numerical arithmetic
    2. **Numerical differentiation and integration** ← You are here
    3. Function approximation
3. Systems of equations
4. Optimization
5. Structural estimation

*These slides are based on Miranda & Fackler (2002), Judd (1998), Nocedal & Wright (2006), and course materials by Ivan Rudik.

# Big O notation

A useful way to represent limiting behavior

# Big O notation

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

The usual way is to use **Big O (or order) notation**

- General mathematical definition: describes the *limiting behavior of a function* when the argument tends towards a particular value or infinity
  - You've seen this before in the expression of Taylor series' errors
- Programming context: describes the limiting behavior of algorithms in terms of run time/memory/accuracy as input size grows

# Big O notation

Written as: $O(f(x))$

Formally, for two real-valued functions $f(x)$ and $g(x)$ we say

$$g(x) = O(f(x))$$

if there $\exists$ a positive $C$ such that

$$|g(x)| \leq C|f(x)|, \text{as } x \to a$$

So $g$ is bounded by $f$ up to a scalar. Another way of saying it (if $g$ is non-zero): $\lim_{x \to a} \frac{|g(x)|}{|f(x)|} < \infty$

Intuitively, $g$ never gets "too far away" from $f$ as $x \to a$

# Big O notation

Here is how to think about it:

## $\mathbf{O(x)}$: linear

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

Examples?

- Time to find a particular value in an unsorted array
  - For each element, check whether it is the value we want

# Big O notation

## $\mathbf{O(c^x)}$ : exponential

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

Examples?

- Time to solve a standard dynamic program, e.g. traveling salesman
  - For each city $i = 1, \ldots, n$, solve a Bellman equation 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

Examples?

- Solving traveling salesman by brute force
  - 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:

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

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

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

# Big O notation: accuracy example

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

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 $x^7$

If we are approximating about zero so $x$ is small: $x^n$ 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)
```

```
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: -1.734723475976807e-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: speed examples

$\mathbf{O(1)}$: algorithm executes in **constant time**

The size of the input does not affect execution speed

- Accessing a specific location in an array: `x[10]`

# Big O notation: speed examples

$\mathbf{O(x)}$: 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: speed examples

**$O(x^2)$ :** algorithm executes in **quadratic time**

- More generally called **polynomial time** for $O(x^n)$

Execution speed grows quadratically in input size

Example:

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

# Numerical differentiation

# 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?

## Finite difference methods

Idea: Approximate the limit by letting $h$ be a small number

What does a finite difference approximation look like?

# Forward difference

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

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

It works the same for partial derivatives:

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

# Forward difference

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

1. Write a function `deriv_x_squared(h,x)` that returns the forward difference approximation to the derivative of $x^2$ around with step size $h$
2. Evaluate the the function for `x=2` and 3 values of `h`
    1. `h = 1e-1`
    2. `h = 1e-12`
    3. `h = 1e-30`

# Forward difference

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

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

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

# 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: $h = \max\{|x|, 1\}\sqrt{\epsilon}$

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)?

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

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

Recall that $O(h^2)$ means the error in our approximation grows quadratically in $h$, though 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: $f'(x) = \frac{f(x+h)-f(x)}{h} + O(h^2)/h$

$\Rightarrow f'(x) = \frac{f(x+h)-f(x)}{h} + O(h)$ 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?

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:

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

- 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:

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

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?

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

Error is quadratic in $h$: if we halve $h$ we reduce error by 75%

# 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 the *number of operations at the expense of accuracy*

- For high dimensional functions it may be worth the trade-off

# 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 $h$

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! *(But up to machine precision, of course)*

# Analytic derivatives

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 susceptible to error in coding or calculating the derivative mathematically

There is another option...

# 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

That's called **automatic differentiation**, or **autodiff** for short

# 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 details of decomposing calculations with computer instructions get pretty complicated pretty fast. We're not going to code it by hand. Instead, we're going to use a package for that: `ForwardDiff`

- The name follows from *forward mode*, which is one way of doing autodiff
- Check out Nocedal & Wright Ch.8 for more details

# Autodiff: let the computer do it

```
# Your function 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 = [2.0]; # location to evaluate: ff(x) = 2^2
```

```
using ForwardDiff # This is the package that has autodiff
g(f,x) = ForwardDiff.gradient(f,x); # Define g = ∇f for a generic function
```

```
## 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])) # write it in the autodiff way
x = [2.0]  # location to evaluate: ff(x) = 2^2
g(f,x) = ForwardDiff.gradient(f,x)  # g = ∇f
```

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

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

# Autodiff: let the computer do it

Try it out! Code up the derivative of $log(10 + x)$ at `x=5.5` using automatic differentiation

1. Make sure you have the `ForwardDiff` package
    1. If not type `using Pkg` then `Pkg.add("ForwardDiff")`
2. Define your function `my_fun`
    1. Pay attention to how you write the sum!
3. Define `x` as a 1-element array containing `5.5`
4. Define the gradient operator `g` using `ForwardDiff.gradient(f,x)`
5. Use `g` to evaluate the derivative
6. Derive the analytic solution and compare to your numeric one

# Autodiff: let the computer do it

```
my_fun(x) = log(+(x[1], 10.0));
x = [5.5];
g(f,x) = ForwardDiff.gradient(f,x);
g(my_fun,x)[1]
```

```
## 0.06451612903225806
```

```
1/(10.0 + x[1])
```

```
## 0.06451612903225806
```

# Calculus operations

First thing: integration is trickier than differentiation

We need to do integration for many things in economics

- Calculating expectations
  - $E[g(x)] = \int_S g(x)dP(x),\ g : S \to \mathbb{R},\ P$ is the probability measure on measurable space $S$
- Adding up a continuous measure of things
  - $\int_D f(x)dx,\ f : \mathbb{R}^n \to \mathbb{R},\ D \subset \mathcal{R}^n$

# 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 the lenght of some subset of $[a, b]$
- $x_i$ is some evaluation point (e.g. midpoint of $dx_i$)

# Infinite limits strike again

Just like derivatives, we face an infinite limit because as $dx_i \to 0$, $\frac{(a-b)}{dx_i} \to \infty$

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

We will see two types of methods:

1. Stochastic approximation with Monte Carlo
2. Quadrature

# Monte Carlo integration

It's probably the most commonly used form in empirical economics

**Key idea**: *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

We 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

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

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

Notes:

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_mc(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_mc(f, 0, 10, 1000)
```

```
## 334.6998692605923
```

Pretty close!

# 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?

One answer is to replace the function with something easier to integrate: **a piece-wise 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 $n-1$ evenly spaced subintervals of length $h$

# Midpoint rule

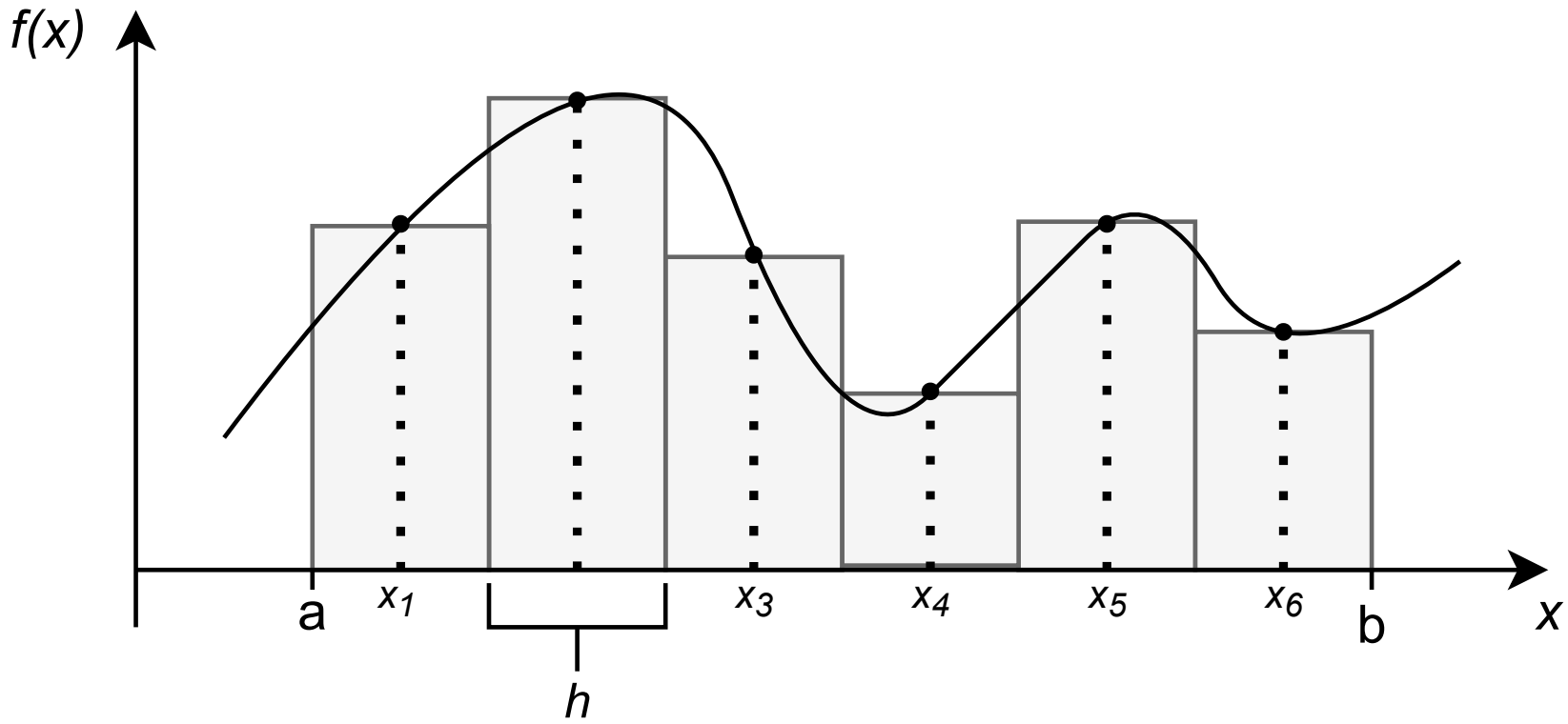
Most basic Newton-Cotes method:

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

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

Approximates $f$ by a step function

# Midpoint rule



# Midpoint rule: let's code it up

Let's integrate $x^2$ again from $0$ to $100$ (the answer is 333.33...)

```
# Generic function to integrate with midpoint
function integrate_midpoint(f, a, b, N)
    # Calculate h given the interval [a,b] and N nodes
    h = (b-a)/(N-1)

    # Calculate the nodes starting from a+h/2 til b-h/2
    x = collect(a+h/2:h:b-h/2)

    # Calculate the expectation
    expectation = sum(h*f(x))
end;
```

# Midpoint rule: let's code it up

```
f(x) = x.^2;

println("Integrating with 5 nodes:$(integrate_midpoint(f, 0, 10, 5))")
```

```
## Integrating with 5 nodes:328.125
```

```
println("Integrating with 50 nodes:$(integrate_midpoint(f, 0, 10, 50))")
```

```
## Integrating with 50 nodes:333.29862557267813
```

```
println("Integrating with 100 nodes:$(integrate_midpoint(f, 0, 10, 100))")
```

```
## Integrating with 100 nodes:333.3248307995783
```

Decent approximation with only 5 nodes. We get pretty close to the answer once we move up to 100 nodes.

# 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 $(x_{i+1}, f(x_{i+1}))$

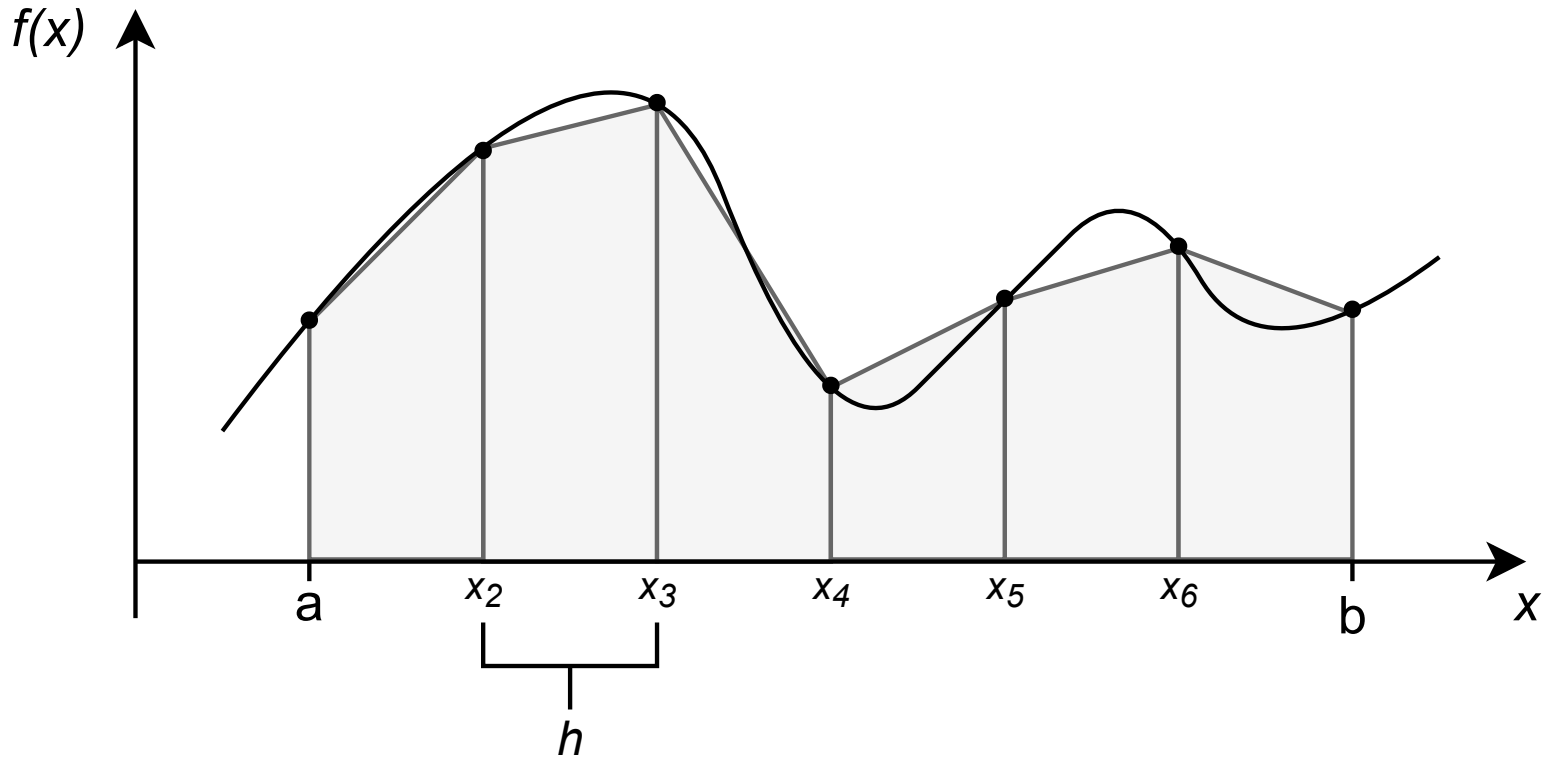
$\int_{x_i}^{x_{i+1}} f(x)dx \approx \frac{h}{2}[f(x_i) + f(x_{i+1})]$

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

# Trapezoid rule



# How accurate is this rule?

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

# Simpson's rule

Increase complexity by 1 degree:

Let $n$ be odd then approximate the function across a *pair of subintervals* by a quadratic interpolation passing through

- $(x_{2i-1}, f(x_{2i-1}))$
- $(x_{2i}, f(x_{2i}))$ and
- $(x_{2i+1}, f(x_{2i+1}))$

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

# Simpson's rule

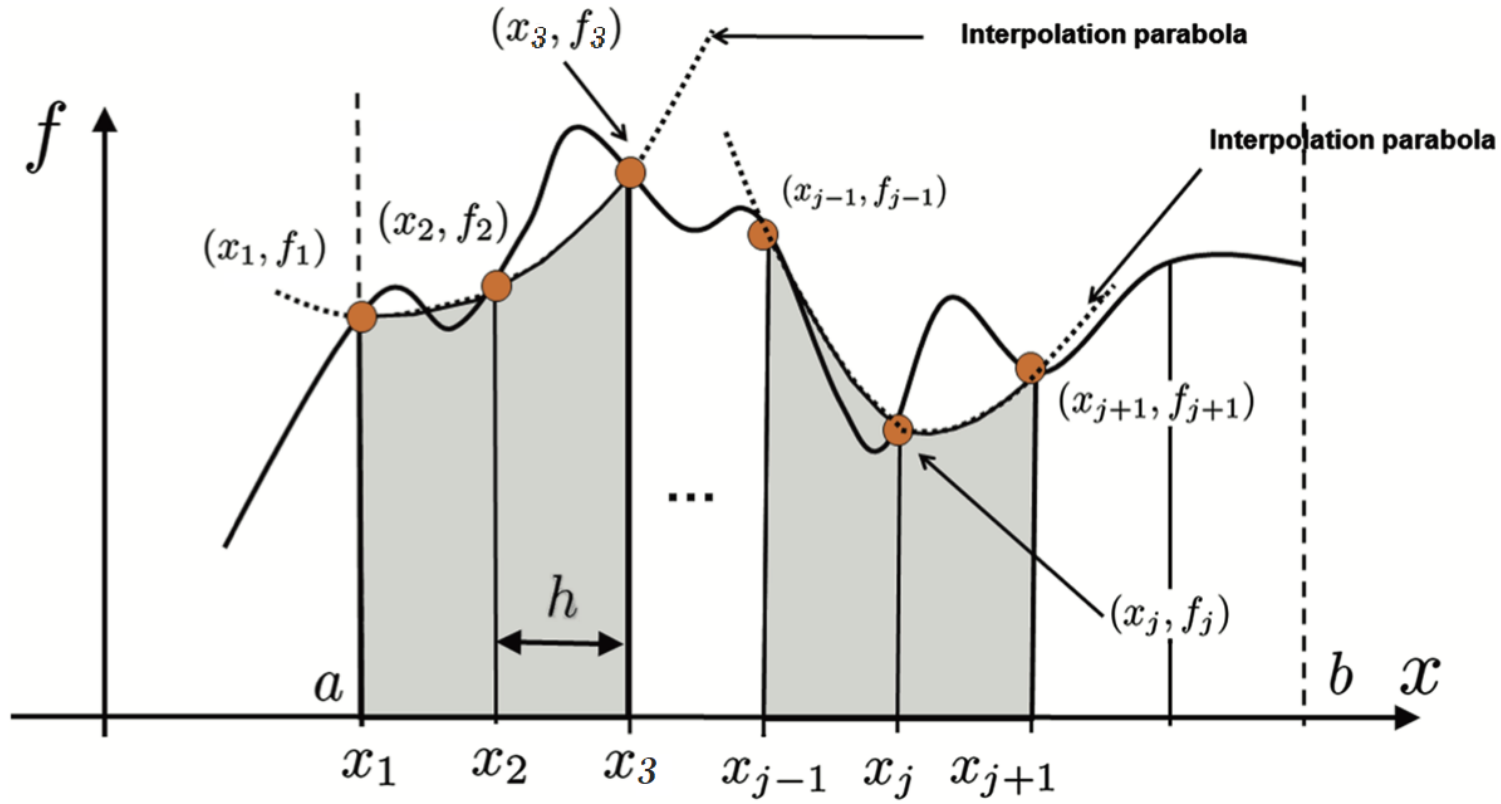
- $\int_{x_i}^{x_{i+1}} f(x)dx \approx \frac{h}{3}[f(x_{2i-1}) + 4f(x_{2i}) + f(x_{2i+1})]$

We can aggregate this by summing over *every two subintervals*

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

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

# Simpson's rule



# How accurate is this rule?

Simpson's rule is $O(h^4)$ and **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. 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

# Note on accuracy

The derivation of the order of error terms is similar to what we did for forward differences: we use Taylor expansions

But there are some extra steps because we have to bring the integral in. We're going to skip these derivations

**BUT** it is important to know that these derivations *assume smooth functions*. That might not be the case in economic models because corner solutions might exist!

- Simpson's is more accurate than the trapezoid rule for smooth functions
- But for functions with kinks/discontinuities in the 1st derivative, the trapezoid rule is often more accurate than Simpson's

# Higher order integration

We can generalize the unidimensional Newton-Cotes quadrature with tensor products

Suppose we want to integrate over a rectangle $\{(x_1, x_2)|a_1 \leq x_1 \leq b_1, a_2 \leq x_2 \leq b_2\} \in \mathbb{R}^2$

We compute separately the $n_1$ nodes and weights $(x_{1i}, w_{1i})$ and $n_2$ for $(x_{2j}, w_{2j})$. The tensor product will give us a grid with $n = n_1 n_2$ points

$$\int_{a_1}^{b_1} \int_{a_2}^{b_2} f(x_1, x_2) dx_2 dx_1 \approx \sum_{i=1}^{n_1} \sum_{j=1}^{n_2} w_{1i} w_{2j} f(x_{1i}, x_{2j})$$

- We can use the same idea for 3+ dimensions. But the number of points increases very quickly: **curse of dimensionality**

# 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)

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**:

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

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

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

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 $\bar{p}(x)$

- Given an approximation with $n$ mass points, $X$ and $\bar{X}$ have identical moments up to order $2n$
- 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**

- In economics, this method is particularly suited for applications with continuous-time discounting

# Gauss-Hermite

Sometimes we want to take expectations of functions of normally distributed variables

$$\int_I f(x)e^{-x^2}dx$$

- Other methods exist for specialized functional forms/distributions: gamma, beta, chi-square, etc

# Gaussian quadrature nodes and weights

*OK, Gaussian quadrature sounds nice. But how do we actually find the nodes and weights?*

It's a non-trivial task: we have to solve $2n$ nonlinear equations to get $n$ pairs $(x_i, w_n)$

We'll use a package for that: `QuantEcon`

- Another package `FastGaussQuadrature` does a better job with high-order integrals or weights with singularities
- `QuadGK` has methods to improve precision further with adaptive subdomains

# Gaussian quadrature with QuantEcon

Let's go back to our previous example and integrate $x^2$ from $0$ to $10$ again (the answer is 333.333...)

```
using QuantEcon;
# Our generic function to integrate with Gaussian quadrature
function integrate_gauss(f, a, b, N)
    # This function get nodes and weights for Gauss-Legendre quadrature
    x, w = qnwlege(N, a, b)

    # Calculate expectation
    expectation = sum(w .* f(x))
end;
```

# Gaussian quadrature with QuantEcon

```
f(x) = x.^2;
println("Integrating with 1 node:$(integrate_gauss(f, 0, 10, 1))")
```

```
## Integrating with 1 node:250.0
```

```
println("Integrating with 2 nodes:$(integrate_gauss(f, 0, 10, 2))")
```

```
## Integrating with 2 nodes:333.3333333333334
```

All we needed was 2 nodes!

- We need 1000 draws with Monte Carlo and 100 nodes with midpoint to get limited approximations

# Other quadratures with `QuantEcon`

`QuantEcon.jl` has a lot more to offer[1]

- Trapezoid (`qnwtrap`) and Simpson's (`qnwsimp`) rule
- Gauss-Hermite (`qnwnorm`) or Gaussian with any continuous distribution (`qnwdist`)
- Generic quadrature integration (`quadrect`)

Check out the documentation at http://quantecon.github.io/QuantEcon.jl/latest/api/QuantEcon.html

# Practice integration with `QuantEcon`

Let's integrate $f(x) = \frac{1}{x^2+1}$ from 0 to 1

Tip: Type `?` in the REPL and the name of the function to read its documentation and learn how to call it

1. Use `quadrect` to integrate using Monte Carlo
    1. You will need to code your function to accept an array (`.` syntax)
2. Use `qnwtrap` to integrate using the Trapezoid rule quadrature
3. Use `qnwlege` to integrate using Gaussian quadrature

Do we know what is the analytic solution to this integral?

# Course roadmap

Up next:

1. Intro to Scientific Computing
2. **Numerical operations and representations**
    1. Numerical arithmetic
    2. Numerical differentiation and integration
    3. Function approximation: (postponed)
3. **Systems of equations** ←
4. Optimization
5. Structural estimation