Lab 3: Depth-Damage Models

DataFrames and Distributions

Leanh Nguyen

Fri., Jan. 26

Table of contents

using CSV  
using DataFrames  
using DataFramesMeta  
using Distributions  
using Interpolations  
using Plots  
using StatsPlots  
using Unitful  
  
Plots.default(; margin=6Plots.mm)

# 1. Site Information

## 1.1 Pick a site

Galveston Pier 21, TX

## 1.2 Pick a building

Building: Ocean Star Offshore Drilling Rig and Museum Gift Store

Location: 29.310430907645863, -94.79173653184932

Address: 2002 Wharf Rd, Galveston, TX 77550

Reason: This building is very close to the gauge as the gauge is located right outside the building. The gauge is located right between the coast and the building.

## 1.3 Find the building’s elevation

Elevation: roof = 5 meters above sea level, base = 2 meters above sea level

# 2. Depth-Damage

Find the building’s depth-damage data. Find a depth-damage function for your building. Use one of the depth-damage functions in haz\_fl\_dept.csv. Read the documemtation and make sure you understand what it means. Explain why this is an appropriate depth-damage function to use.

Build the depth-damage curve. Use the tools we have built in this workshop to build a depth-damage curve for your building. Plot the curve in 1 inch increments from -10 to 30 feet, as above. Explain what it means.

Deterministic depth-damage functions from the HAZUS model developed by the US Army Corps of Engineers: [the data source](https://zenodo.org/records/10027236) Work with haz\_fl\_dept file

haz\_fl\_dept = CSV.read("data/haz\_fl\_dept.csv", DataFrame)  
first(haz\_fl\_dept, 3)

Line 2

This lets us view the first three rows

## 2.1 Parsing

Use the depth-damage functions in this file. However, the depths are stored in a somewhat annoying format (e.g., “ft04m” means -4 feet). Create functionality in the depthdamage.jl file and load it as follows:

include("depthdamage.jl")

DepthDamageData

Use DepthDamageData. This is a data structure or type that stores the depth-damage data, as well as any relevant metadata. Defined a constructor that takes in the row of a DataFrame and creates a DepthDamageData object.

demo\_row = @rsubset(  
 haz\_fl\_dept, :Description == "Gift Shop, structure" # 1   
)[  
 1, :,  
]  
dd = DepthDamageData(demo\_row)

DepthDamageData(Quantity{Float64, 𝐋, Unitful.FreeUnits{(ft,), 𝐋, nothing}}[-4.0 ft, -3.0 ft, -2.0 ft, -1.0 ft, 0.0 ft, 1.0 ft, 2.0 ft, 3.0 ft, 4.0 ft, 5.0 ft … 15.0 ft, 16.0 ft, 17.0 ft, 18.0 ft, 19.0 ft, 20.0 ft, 21.0 ft, 22.0 ft, 23.0 ft, 24.0 ft], [0.0, 0.0, 0.0, 0.0, 0.0, 5.0, 8.0, 9.0, 9.0, 9.0 … 69.0, 73.0, 77.0, 80.0, 83.0, 86.0, 89.0, 92.0, 95.0, 98.0], String7("COM1"), "236", String31("USACE - Galveston"), "Gift Shop, structure", "missing")

This prints out a bunch of data. It has the following fields, which should broadly match with our DataFrame: (:depths, :damages, :occupancy, :dmg\_fn\_id, :source, :description, :comment) 1. 271 COM1 236 USACE - Galveston Gift Shop, structure. I chose this structure type as the building is a gift shop for a museum. The description from the haz\_fl\_dept perfectly matched the structure’s description.

demo\_row = @rsubset(  
 haz\_fl\_dept, :Description == "one story, no basement, Contents", :Source == "USACE - Galveston" # 1   
)[  
 1, :,  
]  
ee = DepthDamageData(demo\_row)

DepthDamageData(Quantity{Float64, 𝐋, Unitful.FreeUnits{(ft,), 𝐋, nothing}}[-4.0 ft, -3.0 ft, -2.0 ft, -1.0 ft, 0.0 ft, 1.0 ft, 2.0 ft, 3.0 ft, 4.0 ft, 5.0 ft … 15.0 ft, 16.0 ft, 17.0 ft, 18.0 ft, 19.0 ft, 20.0 ft, 21.0 ft, 22.0 ft, 23.0 ft, 24.0 ft], [0.0, 0.0, 0.0, 0.0, 8.0, 42.0, 60.0, 71.0, 77.0, 82.0 … 90.0, 90.0, 90.0, 90.0, 90.0, 90.0, 90.0, 90.0, 90.0, 90.0], String7("RES1"), "55", String31("USACE - Galveston"), "one story, no basement, Contents", "missing")

1. To test the sensitivity of my results to my choice of depth-damage function. I used “one story, no basement, Contents” 79 RES1 55 USACE - Galveston one story, no basement, Contents.

fieldnames(typeof(dd))

(:depths, :damages, :occupancy, :dmg\_fn\_id, :source, :description, :comment)

## 2.2 Plotting

Now that we’ve created a DepthDamageData object, we can plot it. When we plot things with units, the Unitful package (as long as we are using it) knows how to handle them.

scatter(  
 dd.depths,  
 dd.damages;  
 xlabel="Flood Depth at House",  
 ylabel="Damage (%)",  
 label="$(dd.description) ($(dd.source))",  
 legend=:bottomright,  
 size=(700, 500),  
)

scatter(  
 ee.depths,  
 ee.damages;  
 xlabel="Flood Depth at House",  
 ylabel="Damage (%)",  
 label="$(ee.description) ($(ee.source))",  
 legend=:bottomright,  
 size=(700, 500),  
)

1. There is a significant difference when using a different depth-damage function. The gift shop is less vulnerable to flood depth, with damage increasing the most when depth > 5 ft. But when using the one-story depth-damage function, the structure is more vulnerable to flood depth, with damage increasing the most when depth > 0 (instant exponential increase in damage).

## 2.3 Interpolating

This is great. However, what if we want to estimate damage between the points? We need a way to **interpolate**. We can do this using the Interpolations package!

itp = let  
 depth\_ft = ustrip.(u"ft", dd.depths)  
 damage\_frac = dd.damages  
 Interpolations.LinearInterpolation(  
 depth\_ft,  
 damage\_frac;  
 extrapolation\_bc=Interpolations.Flat(),  
 )  
end

Line 1

I really like these let...end blocks and use them quite a bit. The main thing to know is that all the variables defined inside the let block are only available inside the let block. Once we get to the end of the block, they vanish! This keeps us from defining tons of variables that get in each others’ way.

Line 2

The Interpolations package doesn’t take units on its input, so we convert the input (which can be of any length unit) to feet before passing it in. If our depths are in meters or millimeters, it won’t be a problem – the ustrip function will convert to feet and then turn them into scalars.

Line 7

Interpolations requires us to specify how to extrapolate. We choose Flat(), meaning that anything below the lowest value in the table will be assumed to have the same damage as the lowest value in the table and anything above the highest value in the table will be assumed to have the same damage as the highest value in the table.

Now we can use this interpolation function to estimate damage at any depth.

let  
 dmg\_fn(x) = itp(ustrip.(u"ft", x))  
 dmg\_fn.([3.1u"ft", 2.2u"m", 91.4u"inch"])  
end

Line 2

Convert the input to feet

Line 3

Estimate damage at 3.1 feet, 2.2 meters, and 91.4 inches

3-element Vector{Float64}:  
 9.0  
 14.87139107611549  
 16.466666666666665

## 2.4 Packaging

To make life simple, we can define a function that takes in some depths and some damages and *returns a function* that can be used to estimate damage at any depth.

function get\_depth\_damage\_function(  
 depth\_train::Vector{<:T}, dmg\_train::Vector{<:AbstractFloat}  
) where {T<:Unitful.Length}  
  
 # interpolate  
 depth\_ft = ustrip.(u"ft", depth\_train)  
 interp\_fn = Interpolations.LinearInterpolation(  
 depth\_ft,  
 dmg\_train;  
 extrapolation\_bc=Interpolations.Flat(),  
 )  
  
 damage\_fn = function (depth::T2) where {T2<:Unitful.Length}  
 return interp\_fn(ustrip.(u"ft", depth))  
 end  
 return damage\_fn  
end

Line 8

The Interpolations package doesn’t take units on its input, so we convert the input (which can be of any length) to feet before passing it in. If our depths are in meters or millimeters, it won’t be a problem – the ustrip function will convert to feet and then turn them into scalars.

Line 10

Interpolations requires us to specify how to extrapolate. We choose Flat(), meaning that anything below the lowest value in the table will be assumed to have the same damage as the lowest value in the table and anything above the highest value in the table will be assumed to have the same damage as the highest value in the table.

Line 14

This is a bit confusing. We are defining a function, inside of a function.

Line 16

We return the function that we just defined. So when we call this function, we get a function – we in turn need to call that function on something else.

get\_depth\_damage\_function (generic function with 1 method)

damage\_fn = get\_depth\_damage\_function(dd.depths, dd.damages)

#20 (generic function with 1 method)

Now damage\_fn is a function. It takes in a depth, with some type of length unit defined using Unitful, and returns the damage in percent. We can use this to plot a depth-damage curve:

p = let  
 depths = uconvert.(u"ft", (-10.0u"ft"):(1.0u"inch"):(30.0u"ft"))  
 damages = damage\_fn.(depths)  
 scatter(  
 depths,  
 damages;  
 xlabel="Flood Depth",  
 ylabel="Damage (%)",  
 label="$(dd.description) ($(dd.source))",  
 legend=:bottomright,  
 size=(800, 400),  
 linewidth=2,  
 )  
end  
p

Line 2

We create a vector of depths from -7 feet to 30 feet, in 1 inch increments. We use uconvert to convert the units to feet (by default, Unitful converts to meters when we add together length units).

Line 3

Our damage\_fn is defined to take in a single scalar. To make predictions about a Vector of depths, we use . to broadcast the function over the vector.

Of course, if we use plot instead of scatter, then we get a line plot which is automatically smooth.

1. From the graph, we can see from our depth-damage curve that there is a direct relationship between damage (%) and flood depth (ft) (i.e., as depth increases, damage increases). However, damage does not increase linearly with flood depth. Here, for the gift shop, damage increases the most when flood depth exceeds ~5 feet until flood depth exceeds 20 feet. Around 25 feet, the gift store is inundated (100% damage).

## 2.5 Expected damages

Now that we have a depth-damage function, we can combine it with a probability distribution of hazard to assess the annual expected damages. First, we need to come up with a distribution of hazard! We’re not going to go into extreme value statistics today. Instead, we’re going to consider a simple distribution that is often used to model extreme events: the generalized extreme value distribution. We’ll consider **hypothetical**parameter values as an illustration. If you want to adjust them, go for it! This is the distribution of the maximum water level in a given year, in feet, at our gauge, for a single year (i.e., we’re not considering sea-level rise).

gauge\_dist = GeneralizedExtremeValue(5, 1.5, 0.1)

GeneralizedExtremeValue{Float64}(μ=5.0, σ=1.5, ξ=0.1)

We can see some quantiles of the distribution:

quantile.(gauge\_dist, [0.5, 0.9, 0.99, 0.999])

4-element Vector{Float64}:  
 5.559968481853559  
 8.785530774057731  
 13.761464356944844  
 19.927437691344867

and we can plot it using StatsPlots:

p1 = plot(  
 gauge\_dist;  
 label="Gauge Distribution",  
 xlabel="Water Level (ft)",  
 ylabel="Probability Density",  
 legend=:topright,  
 linewidth=2,  
)

Our building might be above the gauge. We can correct for this by adding an offset to the **location parameter** of the gauge. Again, we are taking hypothetical values here.

offset = 6.56168 # building is 2 meters (6.56168 ft) above gauge  
house\_dist = GeneralizedExtremeValue(gauge\_dist.μ - offset, gauge\_dist.σ, gauge\_dist.ξ)

GeneralizedExtremeValue{Float64}(μ=-1.56168, σ=1.5, ξ=0.1)

We can plot this

plot!(p1, house\_dist; label="House Distribution", linewidth=2)

Line 1

This adds to the existing plot

We can see that the distribution of hazard at the house is shifted left. That is, the house is less likely to experience extreme water levels than the gauge. This makes sense – it’s higher up!

**Important:** this is a very simple approach and only makes sense if the house is near the gauge.

Now that we have a distribution of hazard, we can combine it with our depth-damage function to estimate the expected annual damages. A very simple Monte Carlo algorithm is

1. Sample values from the distribution of hazard
2. For each value, estimate the damage using the depth-damage function
3. Average the damages

# 3. Expected annual damages

Implement the Monte Carlo algorithm described above to estimate the expected annual damages. Use 1,000,000 samples. Explain what this means.

# Monte Carlo algorithm to estimate expected annual damages  
function monte\_carlo\_simulation(N, hazard\_distribution)  
 # Initialize array to store damages  
 damages = zeros(N)  
   
 # Sample N values from the distribution of hazard  
 hazard\_samples = rand(hazard\_distribution, N)  
   
 # Calculate damages for each hazard sample  
 for i in 1:N  
 damages[i] = damage\_fn.(hazard\_samples[i]u"ft")  
 end  
   
 # Average the damages  
 annual\_damage = sum(damages) / N  
 return annual\_damage  
end  
  
# set hazard distribution  
hazard\_distribution = house\_dist  
  
# Number of samples  
N = 1000000  
  
# Perform Monte Carlo algorithm  
annual\_damage = monte\_carlo\_simulation(N, house\_dist)  
println("Average annual damages estimated using Monte Carlo simulation: ", annual\_damage) # (1)

Average annual damages estimated using Monte Carlo simulation: 1.9954170381464178

1. This means that annual damages due to flooding is low for this structure (1.9% damage average). This is most likely due to the raised platform that the gift store is on, allowing it to be above flood waters during storms.

# 4. Discussion

Finally, add any plots or discussion that you think are relevant! For example, consider looking at plausible alternative depth-damage functions and the sensitivity of your results to the choice of depth-damage function. - Done above