# Lab 3: Depth-Damage Models

DataFrames and Distributions

CEVE 421/521

Fri., Jan. 26

## 1 Overview

Today, we're going to be working with depth-damage functions. This will give us practice:

- working with and manipulating tabular data
- writing functions

In addition, the depth-damage function you choose / build will be a building block for your final project.

### 1.1 Setup

As before:

1. Clone the repository for this lab to your computer and open it in VS Code.
2. In the Julia REPL, `activate` and then `instantiate` the project environment.
3. Check that you can preview the project by running `quarto preview template.qmd` in the terminal (not Julia REPL). If that doesn't work, open the Julia REPL, enter package mode with `]`, and run `build IJulia`.
4. If that doesn't work, ask for help! The way VS Code looks for Python on your computer can be weird and counterintuitive.

### 1.2 Load packages

As usual, we load all required packages at the top of the notebook, in one place.

```
using CSV
using DataFrames
using DataFramesMeta
using Distributions
using Interpolations
using Plots
using StatsPlots
using Unitful

Plots.default(; margin=6Plots.mm)                                    ①
```

① This updates the default margin in our plots so that axis labels don't get cut off.

## 2 Depth-damage data

Today we're going to work with deterministic depth-damage functions from the HAZUS model developed by the US Army Corps of Engineers. Please see the data source for more information and links. We're going to work with the `haz_fl_dept` file today. We can read it in as before:

```
haz_fl_dept = CSV.read("data/haz_fl_dept.csv", DataFrame)
first(haz_fl_dept, 3)                                                    ①
```

① This lets us view the first three rows

| | Column1 | Occupancy | DmgFnId | Source | Description | |
|---|---|---|---|---|---|---|
| | Int64 | String7 | Int64 | String31 | String? | |
| 1 | 1 | RES1 | 183 | USACE - Wilmington | two story, Pile foundation, structure | … |
| 2 | 2 | RES1 | 184 | USACE - Wilmington | two story, Structure | … |
| 3 | 3 | RES1 | 105 | FIA | one floor, no basement, Structure, A-Zone | … |

We can see that there are a lot of columns. Each depth-damage curve is a row, and each depth (or metadata) is a column.

There's a lot to explore with these dataframes. One thing we can do in VS Code is to click on the Julia tab in VS Code (three circles on the far left of your window). This will show you a list of all the variables in your workspace. Click on `haz_fl_dept` and then click on the "preview" icon that pops up (looks like a newspaper emoji). This lets you interactively explore the DataFrame.

### 2.1 Parsing

We'd like to be able to use the depth-damage functions in this file. However, the depths are stored in a somewhat annoying format (e.g., "ft04m" means -4 feet). To make life simple, I've created some functionality in the `depthdamage.jl` file that you can use. We can load it as follows:

```
include("depthdamage.jl")
```

`DepthDamageData`

The main thing that we'll use is called `DepthDamageData`. This is a `data structure` or `type` that stores the depth-damage data, as well as any relevant metadata. If you've created a class in a language like C++ or Python, it's the same idea. I've also defined a `constructor` that takes in the row of a DataFrame and creates a `DepthDamageData` object, to make life easy.

I'll show you how to do this for an illustrative depth-damage function from the New Orleans USACE.

```
demo_row = @rsubset(
    haz_fl_dept, :Description == "one story, Contents, fresh water, short duration"
)[
    1, :,
]
dd = DepthDamageData(demo_row)
```

```
DepthDamageData(Quantity{Float64,  , Unitful.FreeUnits{(ft,),  , nothing}}[-4.0 ft, -3.0 ft, -2
```

This prints out a bunch of data. We can see that it has the following fields, which should broadly match with our DataFrame:

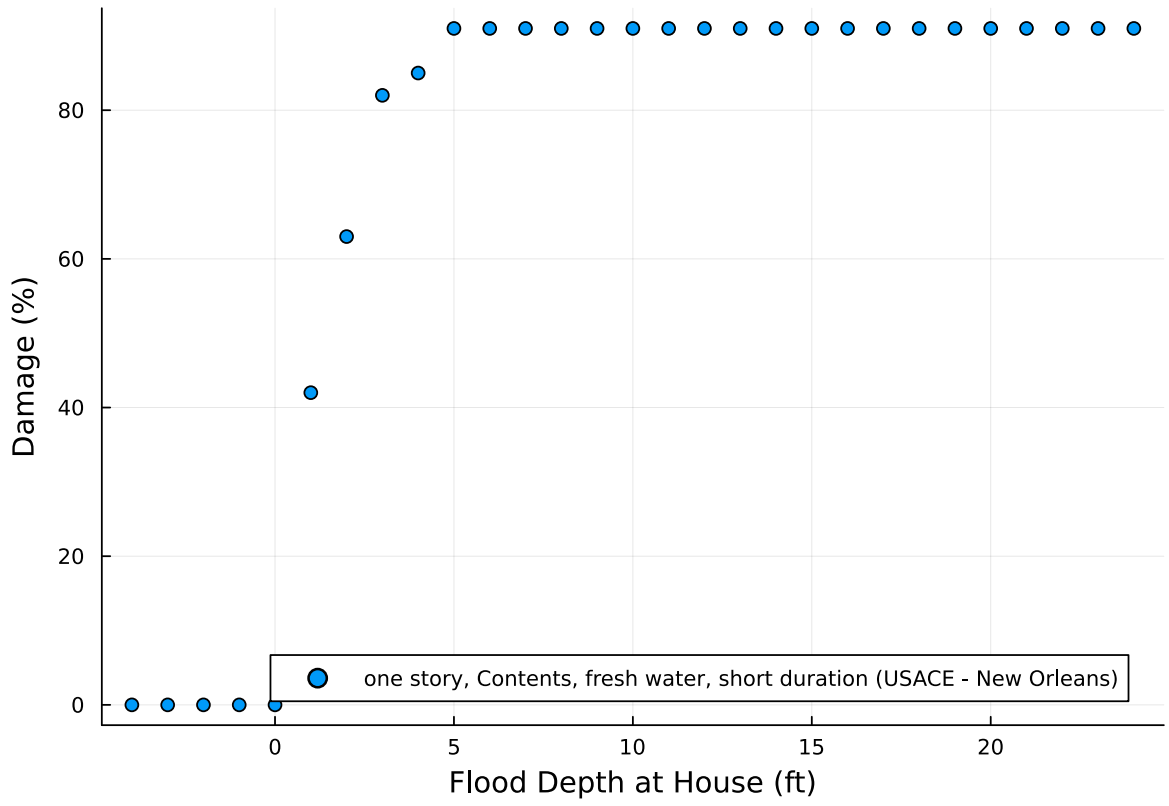
```
fieldnames(typeof(dd))
```

```
(: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),
)
```



## 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                                                                    ①
    depth_ft = ustrip.(u"ft", dd.depths)                                     ②
    damage_frac = dd.damages
    Interpolations.LinearInterpolation(
        depth_ft,
        damage_frac;
        extrapolation_bc=Interpolations.Flat(),                              ③
    )
end
```

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.

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.

③ `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))                                       ①
    dmg_fn.([3.1u"ft", 2.2u"m", 91.4u"inch"])                                ②
end
```

① Convert the input to feet

② Estimate damage at 3.1 feet, 2.2 meters, and 91.4 inches

```
3-element Vector{Float64}:
 82.30000000000001
 91.0
 91.0
```

## 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,                                                    ①
        dmg_train;
        extrapolation_bc=Interpolations.Flat(),                      ②
    )

    damage_fn = function (depth::T2) where {T2<:Unitful.Length}
        return interp_fn(ustrip.(u"ft", depth))                      ③
    end
    return damage_fn                                                 ④
end
```

① 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.

② `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.

③ This is a bit confusing. We are defining a function, inside of a function.

④ 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)
```

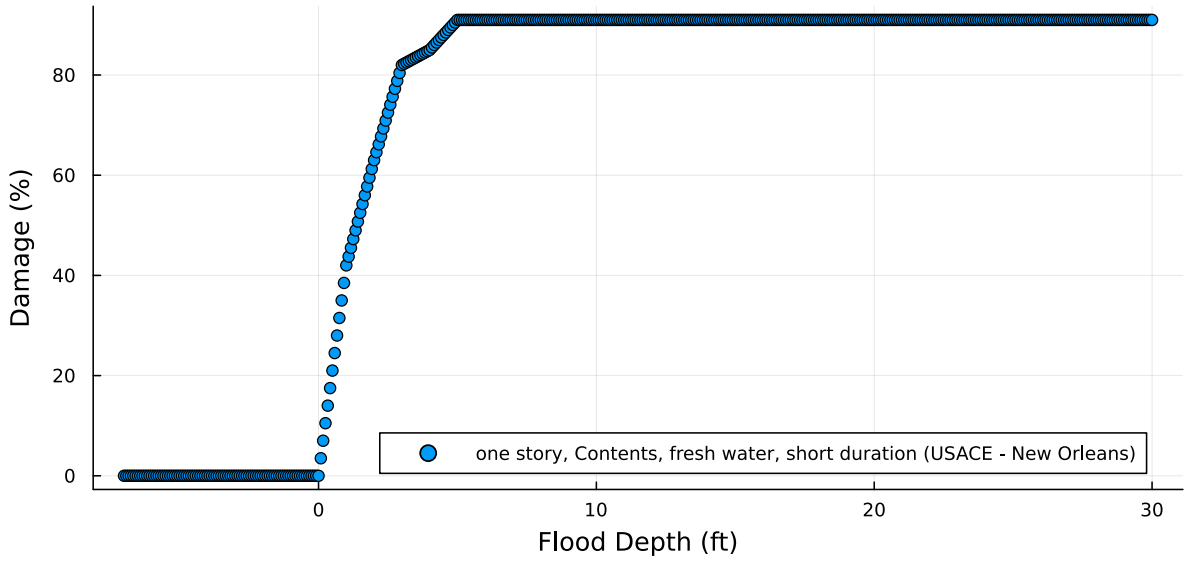
```
#16 (generic function with 1 method)
```

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", (-7.0u"ft"):(1.0u"inch"):(30.0u"ft"))  ①
    damages = damage_fn.(depths)                                     ②
    scatter(
        depths,
        damages;
        xlabel="Flood Depth",
        ylabel="Damage (%)",
        label="$(dd.description) ($(dd.source))",
        legend=:bottomright,
        size=(800, 400),
        linewidth=2,
    )
end
p
```

① 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).

② 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.

## 3 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}( =5.0,  =1.5,  =0.1)
```

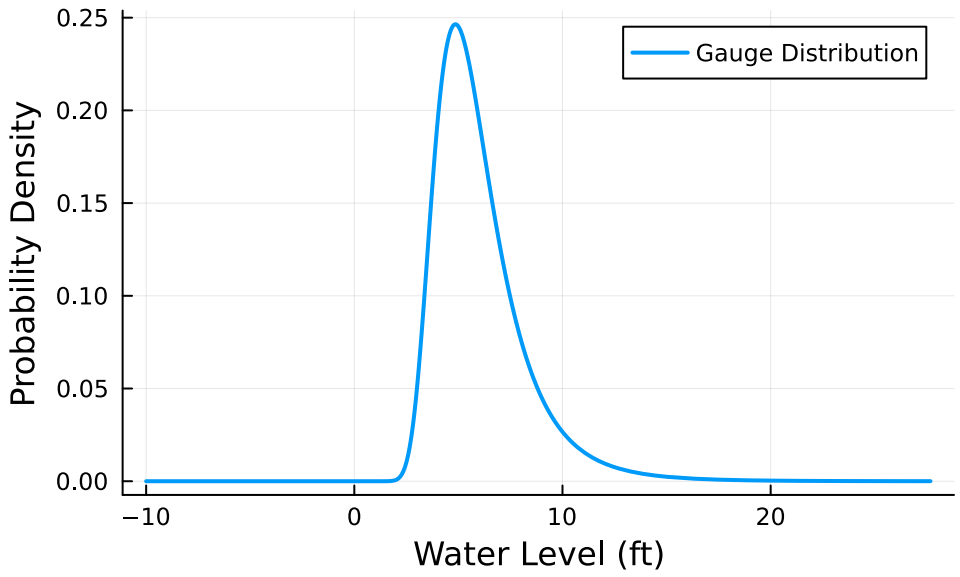
We can see some quantiles of the distribution:

```
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!

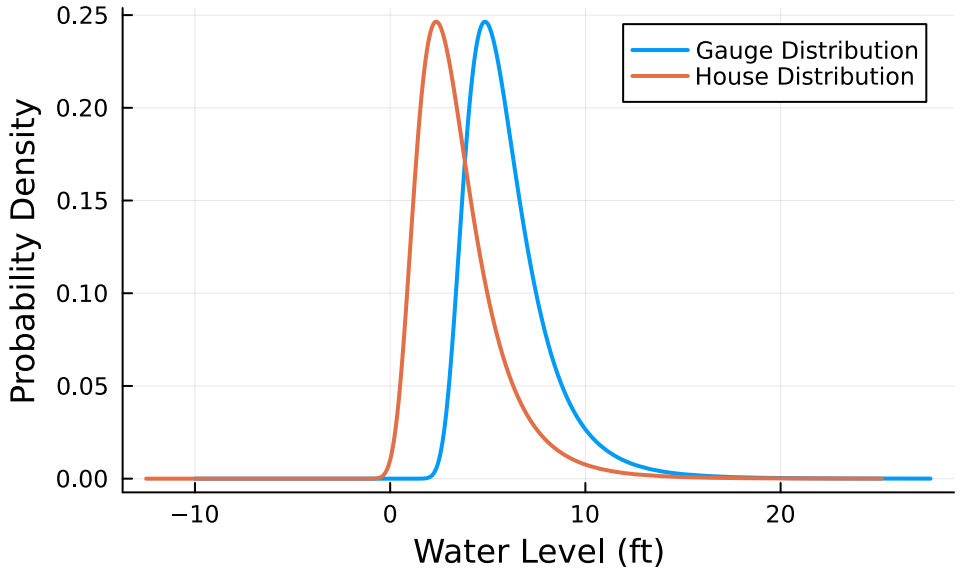
```
offset = 2.5 # house is 2.5 feet above gauge
house_dist = GeneralizedExtremeValue(gauge_dist.  - offset, gauge_dist. , gauge_dist. )
```

```
GeneralizedExtremeValue{Float64}( =2.5,  =1.5,  =0.1)
```

We can plot this

```
plot!(p1, house_dist; label="House Distribution", linewidth=2)                    ①
```

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

## 4 Instructions

Edit the `template.qmd` file to complete the following tasks.

1. **Pick a site.** For your final project, we will develop a decision support tool around the question of whether a building subject to coastal flooding should be elevated. For future analyses to run smoothly, you should pick a building that is near a water gauge with a long record – I suggest Sewells Point, VA or Galveston Pier 21, TX, but feel free to find another gauge. You can change sites later, but you'll need to re-do this analysis. Once you have chosen your site, make sure there's a long record of extreme water levels (not many gaps) by clicking `Tides/Water Levels` and then `Extreme Water Levels`. You should get something that looks like this. Indicate which site you have chosen.
2. **Pick a building.** Find the gauge on Google Maps, and then find a building that is near the gauge. Define what you mean by "near", but the building should not be too far inland. Indicate which building you have chosen.
3. **Find the building's elevation.** Find the elevation of the building. You can do this with USGS data following these instructions. Record your estimate.

4. **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.
5. **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.
6. Implement the Monte Carlo algorithm described above to estimate the expected annual damages. Use 1,000,000 samples. Explain what this means.

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.