Lab 5: Sea-Level Rise

CEVE 421/521

Fri., Feb. 16

There are two objectives of this lab:

1. To familiarize ourselves with an increasingly complex model of our house-elevation problem
2. To conduct exploratory modeling to understand the implications of different parameter values and how they affect our decision-making

1 Setup

1.1 The usual

As always:

1. Clone the lab repository to your computer
2. Open the lab repository in VS Code
3. Open the Julia REPL and activate, then instantiate, the lab environment
4. Make sure you can render: `quarto render template.qmd` in the terminal.
    - If you run into issues, try running `] build IJulia` in the Julia REPL (`]` enters the package manager).
    - If you still have issues, try opening up `blankfile.py`. That should trigger VS Code to give you the option to install the Python extension, which you should do. Then you should be able to open a menu in the bottom right of your screen to select which Python installation you want VS Code to use.

1.2 Load packages

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

Plots.default(; margin=5Plots.mm)
```

1.3 Local package

We're starting to accumulate a lot of code describing our model. A good way to store this model is by creating a local package. I have created a package called `HouseElevation` that contains the model code. You don't need to do anything special to install it, and you don't need to edit the code, though I'd encourage you to have a look around!

When we work with local packages, it's common to use another package called `Revise`. This is a cool package that will automatically propagate any changes you make to the package to any code that uses the package. You don't need to worry about this for now – just load them.

```
using Revise
using HouseElevation
```

2 Building the model

We've added a bit of complexity to our model. In this section, we walk through each of the sections of the model.

2.1 House

> **!** Important
>
> We will consider a single house, and will ignore uncertainty in the depth-damage function or other house parameters

- Neglect uncertainty in depth-damage function
- Consider a single building
- We're going to put all relevant information into a `House` object:
  - Depth-damage function
  - Area
  - Cost (USD)
  - Elevation relative to gauge
  - Metadata

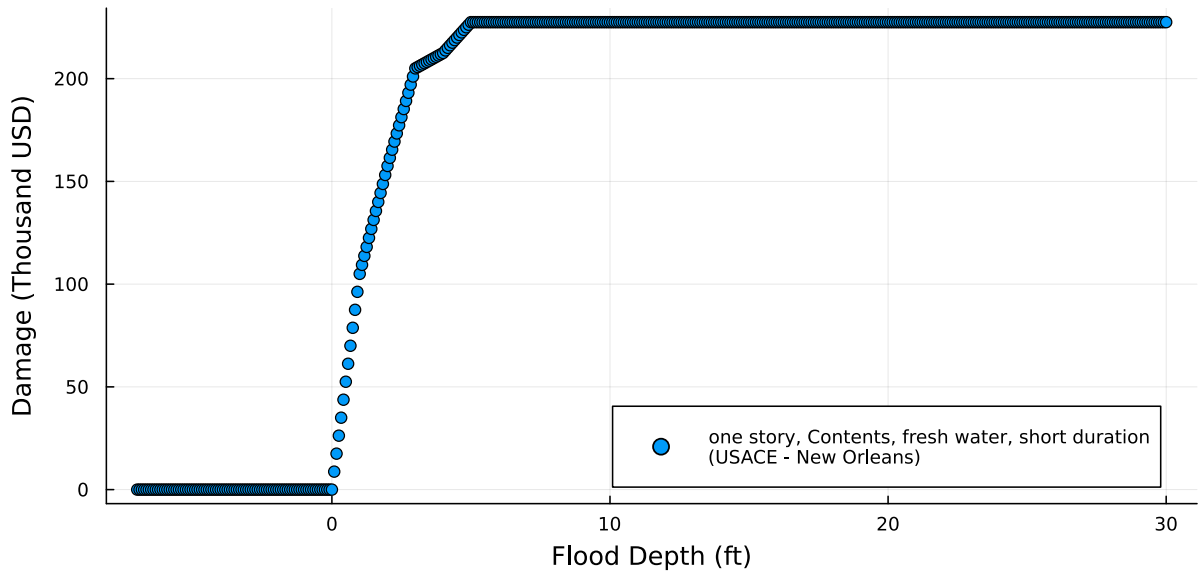
We can create a `House` as follows – note that we're using a `let...end` block to create the `House` object. This means that any variables defined inside the block are not available outside the block, which is a good way to avoid "polluting the global namespace."

```
house = let
    haz_fl_dept = CSV.read("data/haz_fl_dept.csv", DataFrame) # read in the file
    desc = "one story, Contents, fresh water, short duration"
    row = @rsubset(haz_fl_dept, :Description == desc)[1, :] # select the row I want
    area = 500u"ft^2"
    height_above_gauge = 10u"ft"
    House(
        row;
        area=area,
        height_above_gauge=height_above_gauge,
        value_usd=250_000,
```

```
    )
end
```

We can then use the `House` object to calculate the damage to the house for a given flood depth. Let's convert the damage to dollars by multiplying the fraction (given by our depth-damage function) by the value of the house. For example:

```
let
    depths = uconvert.(u"ft", (-7.0u"ft"):(1.0u"inch"):(30.0u"ft"))
    damages = house.ddf.(depths) ./ 100
    damages_1000_usd = damages .* house.value_usd ./ 1000
    scatter(
        depths,
        damages_1000_usd;
        xlabel="Flood Depth",
        ylabel="Damage (Thousand USD)",
        label="$(house.description)\n($(house.source))",
        legend=:bottomright,
        size=(800, 400),
        yformatter=:plain, # prevents scientific notation
    )
end
```



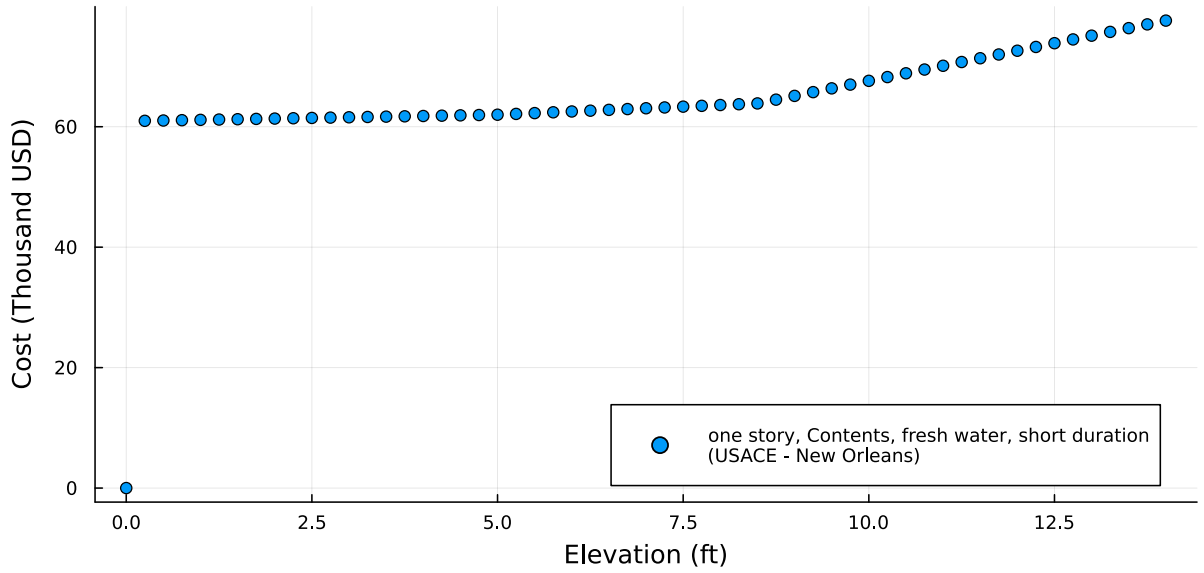
We can also use the `House` object to calculate the cost of raising the house to a given elevation. We use the `elevation_cost` function like this:

```
elevation_cost(house, 10u"ft")
```

```
67620.0
```

and again we can plot this.

```julia
let
    elevations = 0u"ft":0.25u"ft":14u"ft"
    costs = [elevation_cost(house, e ) for e  in elevations]
    scatter(
        elevations,
        costs ./ 1_000;
        xlabel="Elevation",
        ylabel="Cost (Thousand USD)",
        label="$(house.description)\n($(house.source))",
        legend=:bottomright,
        size=(800, 400),
        yformatter=:plain, # prevents scientific notation
    )
end
```



2.2 Sea-level rise

> **!** Important
>
> We will sample many different scenarios of sea-level rise

We're modeling sea-level rise following the approach of Oddo et al. (2017). Essentially, we use five parameters: a, b, c, t^*, and c^*. The local sea-level in year t is given by equation 6 of Oddo et al. (2017):

$$\text{SLR} = a + b(t - 2000) + c(t - 2000)^2 + c^* \mathbb{I}(t > t^*)(t - t^*)$$

The authors note:

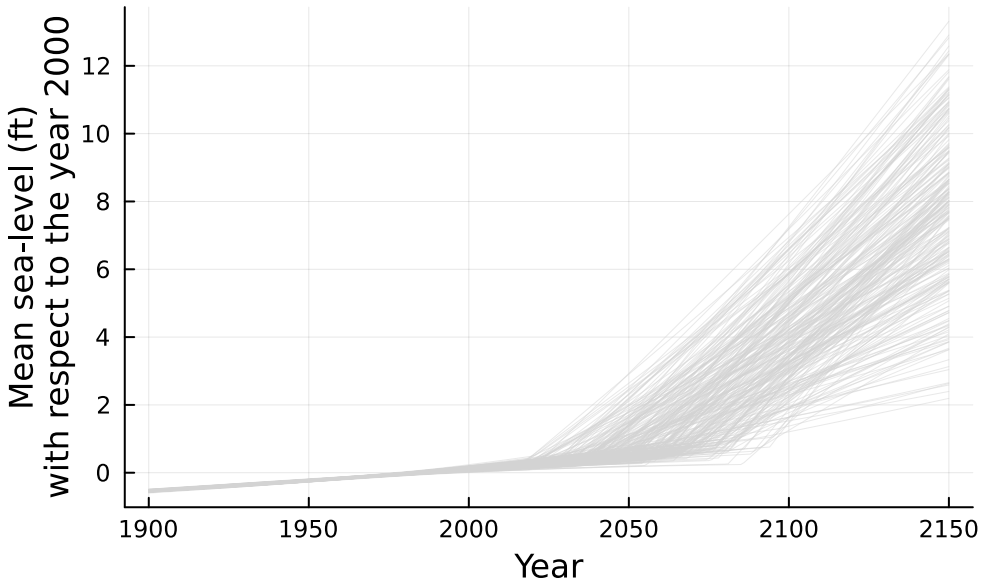
In this model, the parameters a, b, and c represent the reasonably well-characterized process of thermosteric expansion as a second-order polynomial. It also accounts for more poorly understood processes, including potential abrupt sealevel rise consistent with sudden changes in ice flow dynamics.Here, c^* represents an increase in the rate of sea-level rise that takes place at some uncertain time, t^*, in the future.

This is, of course, a highly simplified model. However, the parameters can be calibrated to match historical sea-level rise (i.e., throwing out any parameter values that don't match the historical record) and use a statistical inversion method to estimate the parameters. One could also calibrate the parameters to match other, more complex, physics-based models. We'll use Monte Carlo simulations from Oddo et al. (2017), available on GitHub. These were actually calibrated for the Netherlands, but we'll pretend that sea-level rise in your location matches (which – as we know – it doesn't).

```
slr_scenarios = let
    df = CSV.read("data/slr_oddo.csv", DataFrame)
    [Oddo17SLR(a, b, c, tstar, cstar) for (a, b, c, tstar, cstar) in eachrow(df)]
end
println("There are $(length(slr_scenarios)) parameter sets")
```

We can plot these scenarios to get a sense of the range of sea-level rise we might expect.

```
let
    years = 1900:2150
    p = plot(;
        xlabel="Year",
        ylabel="Mean sea-level (ft)\nwith respect to the year 2000",
        label="Oddo et al. (2017)",
        legend=false
    )
    for s in rand(slr_scenarios, 250)
        plot!(p, years, s.(years); color=:lightgrey, alpha=0.5, linewidth=0.5)
    end
    p
end
```



The key insight you should take from this plot is that uncertainty in future sea level increases over time!

2.3 Storm surge

> **!** Important
>
> We will consider parametric uncertainty in the storm surge

The next component of the model is the storm surge (i.e., the height of the flood above mean sea-level). We can model the water level *at the gauge* as the sum of the local sea-level and the storm surge. We can then model the water level *at the house* as the water level at the gauge minus the elevation of the house above the gauge.

We will consider parametric uncertainty in the storm surge. From lab 3, you should have a `GeneralizedExtremeValue` distribution for the storm surge. We can then sample parameters from a range centered on this distribution. For example, in the example for lab 3 we had `GeneralizedExtremeValue(5, 1.5, 0.1)`. We can use this function to create a distribution for the storm surge.

```
function draw_surge_distribution()
      = rand(Normal(5, 1))
      = rand(Exponential(1.5))
      = rand(Normal(0.1, 0.05))
    GeneralizedExtremeValue( ,  ,  )
end
```

```
draw_surge_distribution (generic function with 1 method)
```

We can then call this function many times to get many different distributions for the storm surge.

For example,

```
[draw_surge_distribution() for _ in 1:1000]
```

> **!** Important
>
> This is NOT statistical estimation. We are not saying anything at all about whether these parameters are consistent with observations. In fact, even when parameters are uncertain, sampling around a point estimate in this manner usually produces lots of parameter values that are highly implausible. Here, we are just exploring the implications of different parameter values. Building a better model for storm surge is a great idea for your final project!

2.4 Discount rate

> **!** Important
>
> We will consider parametric uncertainty in the discount rate.

The discount rate is an important economic parameter in our NPV analysis. There are elements of discounting that are perhaps not random (e.g., how much do you value the future versus the present?) while there are other elements that are very much random (what is the opportunity cost of spending money now?) We will model this by treating the discount rate as a random variable, but more sophisticated analyses are possible. We can use the following function

```
function draw_discount_rate()
    return rand(Normal(0.04, 0.02))
end
```

Note that we are now defining the discount rate as a proportion (from 0 to 1) rather than a percentage (from 0 to 100).

2.5 Running a simulation

In the notation we've seen in class, we have a system model f that takes in a state of the world $\mathbf{s}$, an action a, and outputs some metric or metrics. I've reproduced this in our model, adding one extra piece: a `ModelParams` object that contains all the parameters of the model that *don't change from one simulation to the next.*

In our model, the `ModelParams` are the house characteristics (area, value, and depth-damage curve) and the years we're considering. You should consider different time horizons!

```
p = ModelParams(
    house=house,
    years=2024:2083
)
```

The next step is to create an object to hold our state of the world (SOW). We can create one like this. In the next step, we'll want to sample a large ensemble of SOWs.

```
sow = SOW(
    rand(slr_scenarios),
```

```
    draw_surge_distribution(),
    draw_discount_rate()
)
```

Last, we need to define our action. For now, our action is very simple: we're going to raise the house to a fixed elevation. However, in the future we might have a more complex action (e.g., when the sea level exceeds some threshold $t1$, raise the house by some fixed amount $t2$, which has two parameters). We define our action as follows:

```
a = Action(3.0u"ft")
```

Finally, we have a function to run the simulation. This function takes in the model parameters, the state of the world, and the action, and returns the NPV of the action. Please have a look at `run_sim.jl` to see how this is implemented!

```
res = run_sim(a, sow, p)
```

```
-67601.11999736077
```

3 Exploratory modeling

Now that you've figured out how this model works, it's your turn to conduct some exploratory modeling. In `template.qmd`, I've provided only the code required to load packages.

3.1 Apply the model to your site

1. Build your own house object, based on the house you've been using (or you can switch if you'd like)
    a. Briefly explain where you got the area, value, and depth-damage curve from
    b. Plot the depth-damage curve
    c. Plot the cost of raising the house to different elevations from 0 to 14 ft
2. Read in the sea-level rise data
3. Modify my code to create a function to draw samples of storm surge and the discount rate. Explain your modeling choices!
4. Define an illustrative action, SOW, and model parameters, and run a simulation.

3.2 Large ensemble

Now that you've got the model working for your site, you should run a large ensemble of simulations (explain how you interpret "large").

1. Sample many SOWs (see below)
2. Sample a range of actions. You can do this randomly, or you can look at just a couple of actions (e.g., 0, 3, 6, 9, 12 ft) – explain your choice.
3. Run the simulations for each SOW and action. You can use a for loop for this.
4. Create a DataFrame of your key inputs and results (see below)

Here's how you can create a few SOWs and actions and run the simulations for each:

```julia
sows = [SOW(rand(slr_scenarios), draw_surge_distribution(), draw_discount_rate()) for _ in 1:10
actions = [Action(3.0u"ft") for _ in 1:10] # these are all the same
results = [run_sim(a, s, p) for (a, s) in zip(actions, sows)]
```

```
10-element Vector{Float64}:
      -1.5076983845316784e6
  -72707.9086785899
  -67993.58770275544
 -381916.92636288557
  -90155.19691535276
  -92163.87465134572
  -77660.25514005122
 -208089.6406442849
 -137963.75396342622
  -61593.131393393625
```

Here's how you can create a dataframe of your results. Each row corresponds to one simulation, and the columns are the inputs and outputs of the simulation.

```julia
df = DataFrame(
    npv=results,
    Δh_ft=[a.Δh_ft for a in actions],
    slr_a=[s.slr.a for s in sows],
    slr_b=[s.slr.b for s in sows],
    slr_c=[s.slr.c for s in sows],
    slr_tstar=[s.slr.tstar for s in sows],
    slr_cstar=[s.slr.cstar for s in sows],
    surge_ =[s.surge_dist.  for s in sows],
    surge_ =[s.surge_dist.  for s in sows],
    surge_ =[s.surge_dist.  for s in sows],
    discount_rate=[s.discount_rate for s in sows],
)
```

	npv	Δh_ft	slr_a	slr_b	slr_c	slr_tstar	slr_cstar	surge_	
	Float64	Float64	Float64	Float64	Float64	Float64	Float64	Float64	
1	-1.5077e6	3.0	39.3458	2.64922	0.00554768	2061.69	29.1087	5.94461	...
2	-72707.9	3.0	34.9213	2.38842	0.00438375	2083.67	17.1654	5.07671	...
3	-67993.6	3.0	40.6015	2.19189	0.0018416	2081.03	15.1822	6.95566	...
4	-3.81917e5	3.0	37.7635	2.27993	0.00176545	2085.8	30.718	4.35868	...
5	-90155.2	3.0	36.5181	2.17943	0.0010171	2066.95	28.7953	4.19493	...
6	-92163.9	3.0	37.2432	2.15507	0.00151998	2076.1	20.0193	5.20923	...
7	-77660.3	3.0	34.9104	2.40555	0.0047068	2024.08	5.38038	5.56816	...
8	-2.0809e5	3.0	48.5117	2.72021	0.00569688	2081.13	20.2882	5.57976	...
9	-1.37964e5	3.0	44.4039	2.48205	0.00385763	2037.89	12.6648	4.96039	...
10	-61593.1	3.0	20.7712	2.16755	0.00328647	2064.41	19.1239	4.18841	...

3.3 Analysis

Now, analyze your results. You can use scatterplots and other visualizations, or any other statistical analyses that you think may be helpful. Remember that the goal is to understand how different parameter values affect the success or failure of different actions.

Some questions to consider:

- When do you get the best results?
- When do you get the worst results?
- What are the most important parameters?
- If you had unlimited computing power, would you run more simulations? How many?
- What are the implications of your results for decision-making?

Oddo, P. C., Lee, B. S., Garner, G. G., Srikrishnan, V., Reed, P. M., Forest, C. E., & Keller, K. (2017). Deep uncertainties in sea-level rise and storm surge projections: Implications for coastal flood risk management. *Risk Analysis*, *0*(0). https://doi.org/ghkp82