# Lab 6 Solutions

CEVE 421-521

Thu., Apr. 4

## 1 Setup

```julia
using CSV
using DataFrames
using DataFramesMeta
using Distributions
using LaTeXStrings
using Metaheuristics
using Plots
using Random
using Unitful

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

We also load our local package as in lab 5.

```julia
using Revise
using HouseElevation
```

## 2 States of the world

We begin by defining the variables that don't change from one SOW to the next. We load these into the `ModelParams`.

```julia
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 = 12u"ft"
    House(row; area=area, height_above_gauge=height_above_gauge, value_usd=250_000)
end

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

Next we define how we will sample the states of the world.

```julia
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

function draw_surge_distribution()
      = rand(Normal(5, 1))
      = rand(Exponential(1.25))
      = rand(Normal(0.1, 0.05))
    return GeneralizedExtremeValue( ,  ,  )
end

function draw_discount_rate()
    return rand(Normal(0.05, 0.03))
end

function draw_sow()
    slr = rand(slr_scenarios)
    surge_params = draw_surge_distribution()
    discount = draw_discount_rate()
    return SOW(slr, surge_params, discount)
end
```

Finally we can sample the SOWs

```julia
Random.seed!(421521)
N_SOW = 10_000
N_SOW_opt = 10 # to start
sows = [draw_sow() for _ in 1:N_SOW]
sows_opt = first(sows, N_SOW_opt)
```

# 3 Optimization

## 3.1 Bounds

We have a single decision variable, the height of the house above the ground. This can be any real number between 0 and 14 feet.

```julia
bounds = boxconstraints(; lb=[0.0], ub=[14.0])
```

```
BoxConstrainedSpace{Float64}([0.0], [14.0], [14.0], 1, true)
```

## 3.2 Objective function

We next need an objective function. Recall that we want to *maximize* NPV, but the optimization package we are using is set up to *minimize*.

```julia
function objective_function(Δh::Vector{Float64})
    a = Action(Δh[1])
    npvs = [run_sim(a, sow, p) for sow in sows_opt]
```

```
    return -mean(npvs)
end
```

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

### 3.3 Running

We can throw this straight into the `optimize` function:

```
result = optimize(objective_function, bounds)
```

```
Optimization Result
===================
  Iteration:       57
  Minimum:         80168.3
  Minimizer:       [10.9924]
  Function calls:  399
  Total time:      3.0392 s
  Stop reason:     Due to Convergence Termination criterion.
```

We can view the minimum of the objective function with

```
minimum(result)
```

```
80168.32817561974
```

and the value of the decision variable that achieves that minimum with:

```
minimizer(result)
```

```
1-element Vector{Float64}:
 10.99236059868491
```

This seems like it's working plausibly. Let's try now with more SOWs.

```
N_SOW_opt = 100
sows_opt = first(sows, N_SOW_opt)
```

```
100-element Vector{SOW{Float64}}:
 SOW{Float64}(Oddo17SLR{Float64}(58.66628415, 2.709557857, 0.003600726, 2061.196955, 24.7855783
 SOW{Float64}(Oddo17SLR{Float64}(18.98238754, 2.076689405, 0.002459112, 2017.216294, 15.5840157
 SOW{Float64}(Oddo17SLR{Float64}(13.54334493, 1.576769557, -0.001892038, 2019.630551, 13.5630666
 SOW{Float64}(Oddo17SLR{Float64}(38.14589083, 2.623948719, 0.005085519, 2016.508542, 8.13462905
 SOW{Float64}(Oddo17SLR{Float64}(40.9256403, 2.858600773, 0.008671168, 2024.544892, 18.75223102
 SOW{Float64}(Oddo17SLR{Float64}(36.12774556, 2.258399655, 0.002399946, 2066.78925, 19.7923147)
 SOW{Float64}(Oddo17SLR{Float64}(54.44686357, 2.440676913, 0.003209864, 2039.323894, 22.8970514
 SOW{Float64}(Oddo17SLR{Float64}(42.54886307, 2.374354031, 0.003786949, 2018.717192, 9.58709084
 SOW{Float64}(Oddo17SLR{Float64}(5.951503134, 1.501882544, -0.002460307, 2022.058777, 10.7895300
 SOW{Float64}(Oddo17SLR{Float64}(17.66182918, 1.54415968, -0.002055399, 2058.500619, 31.6412588
 SOW{Float64}(Oddo17SLR{Float64}(39.36100099, 2.426238607, 0.00403237, 2027.055898, 10.85150744
 SOW{Float64}(Oddo17SLR{Float64}(51.98651863, 2.760247424, 0.00577138, 2031.033814, 27.0621288)
 SOW{Float64}(Oddo17SLR{Float64}(31.20844945, 2.11554997, 0.001930065, 2027.473128, 21.54471462
```

```
 SOW{Float64}(Oddo17SLR{Float64}(47.00832611, 2.647493943, 0.004804853, 2057.476657, 22.223900
 SOW{Float64}(Oddo17SLR{Float64}(36.73834466, 2.205458479, 0.002122161, 2034.741468, 20.800943
 SOW{Float64}(Oddo17SLR{Float64}(44.05127787, 2.338748756, 0.002183976, 2057.606653, 15.064978
 SOW{Float64}(Oddo17SLR{Float64}(28.75735666, 2.444711243, 0.004489676, 2073.837867, 23.352426
 SOW{Float64}(Oddo17SLR{Float64}(35.84037864, 2.395483919, 0.003342545, 2020.868925, 20.065402
 SOW{Float64}(Oddo17SLR{Float64}(46.47782198, 2.652879425, 0.004551166, 2083.94697, 31.7156261
 SOW{Float64}(Oddo17SLR{Float64}(47.06379093, 2.686402018, 0.005309616, 2030.347586, 20.327023
 SOW{Float64}(Oddo17SLR{Float64}(20.60575154, 1.814858117, 0.001384537, 2096.479489, 33.518256
 SOW{Float64}(Oddo17SLR{Float64}(31.47268467, 2.161480684, 0.001951459, 2041.194249, 17.510204
 SOW{Float64}(Oddo17SLR{Float64}(33.88063067, 2.470390408, 0.003485804, 2063.105751, 24.994383
 SOW{Float64}(Oddo17SLR{Float64}(14.57458321, 1.356668977, -0.004109447, 2041.776196, 22.79443
 SOW{Float64}(Oddo17SLR{Float64}(52.97055604, 2.796456852, 0.006349309, 2096.336577, 22.775082
```

Since I'm using more SOWs here, I'll also increase the time limit for the optimization to three minutes.

```julia
options = Options(; time_limit=180.0, f_tol_rel=10.0)
```

```
Options
=======
  rng:             TaskLocalRNG()
  seed:            4263123106
  x_tol:           1.0e-8
  f_tol:           1.0e-12
  g_tol:           0.0
  h_tol:           0.0
  debug:           false
  verbose:         false
  f_tol_rel:       10.0
  time_limit:      180.0
  iterations:      0
  f_calls_limit:   0.0
  store_convergence: false
  parallel_evaluation: false
```

To use options, we have to choose an algorithm. See list of algorithms here. The `ECA` algorithm is suggested as a default, so we'll use that.

```julia
algorithm = ECA(; options=options)
```

```
Algorithm Parameters
====================
  ECA( _max=2.0, K=7, N=0, N_init=0, p_exploit=0.95, p_bin=0.02,  =0.0, adaptive=false, resize_

Optimization Result
===================
  Empty status.
```

Before we run the optimization, let's set a random seed. This will make our results more reproducible. We can then vary the seed to see how sensitive our results are to the random seed.

```julia
Random.seed!(421521)
result = optimize(objective_function, bounds, algorithm)
```

```
Optimization Result
===================
  Iteration:       99
  Minimum:         80637.5
  Minimizer:       [10.1205]
  Function calls:  693
  Total time:      55.6784 s
  Stop reason:     Due to Convergence Termination criterion.
```

We can view our result with

```julia
display(minimum(result))
display(minimizer(result))
```

```
80637.52834867894
```

```
1-element Vector{Float64}:
 10.120502588290023
```

# 4 Validation

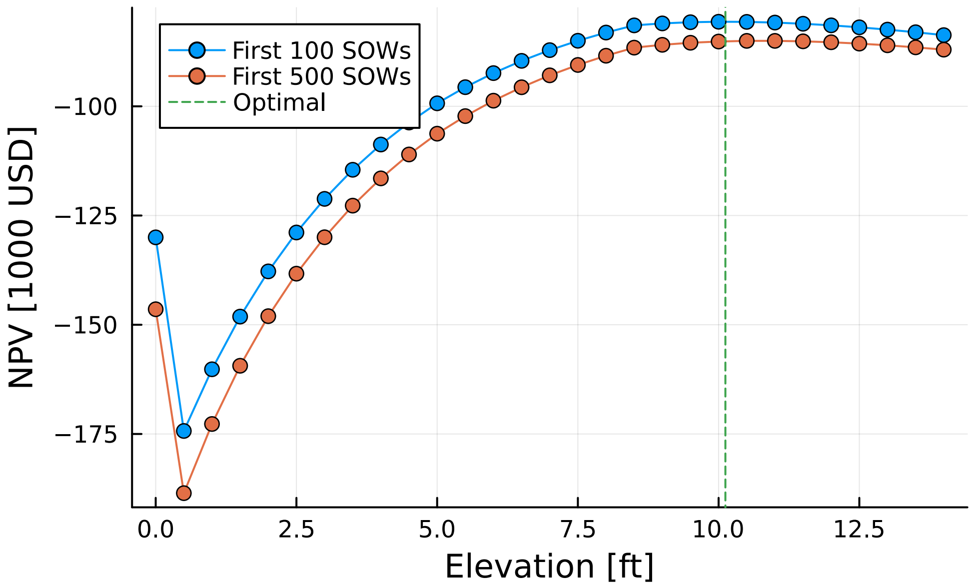
In this case, we don't really *need* optimization – we can use brute force. We can compare by plotting the objective function for a range of elevations (from 0 to 14 ft) using all SOWs.

```julia
elevations_try = 0:0.5:14
actions_try = Action.(elevations_try)

N_more = 500
npvs_opt = [mean([run_sim(a, sow, p) for sow in sows_opt]) for a in actions_try]
npvs_moore = [
    mean([run_sim(a, sow, p) for sow in first(sows, N_more)]) for a in actions_try
]
```

and plot

```julia
plot(
    elevations_try,
    npvs_opt ./ 1000;
    xlabel="Elevation [ft]",
    ylabel="NPV [1000 USD]",
    label="First $(N_SOW_opt) SOWs",
    marker=:circle,
)
plot!(elevations_try, npvs_moore ./ 1000; label="First $(N_more) SOWs", marker=:circle)
vline!([minimizer(result)]; label="Optimal", linestyle=:dash)
```




Key insights:

1. Our optimization appears to be working well, and maximizes the blue curve as it should
2. There is a substantial difference between the blue and red lines, indicating that using different SOWs (from the same distribution!) can make a big difference
3. Going from zero (don't elevate) to a small elevation is always bad, as you gain little flood protection but have to pay the fixed costs of elevation
4. The optimal elevation is highly sensitive to assumptions about the SOWs