Final Project Code

Lucia Romero-Alston (lmr12)

Thu., Apr. 4

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
using KernelDensity
using Statistics
using StatsBase
```

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 = "Cafeteria Restaurant, structure"
    row = @rsubset(haz_fl_dept, :Description == desc)[1, :] # select the row I
want
    area = 4004u"ft^2"
    height_above_gauge = 4u"ft"
              House(row;    area=area,    height_above_gauge=height_above_gauge,
value_usd=400_000)
end

p = ModelParams(; house=house, years=2024:2100)
```

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

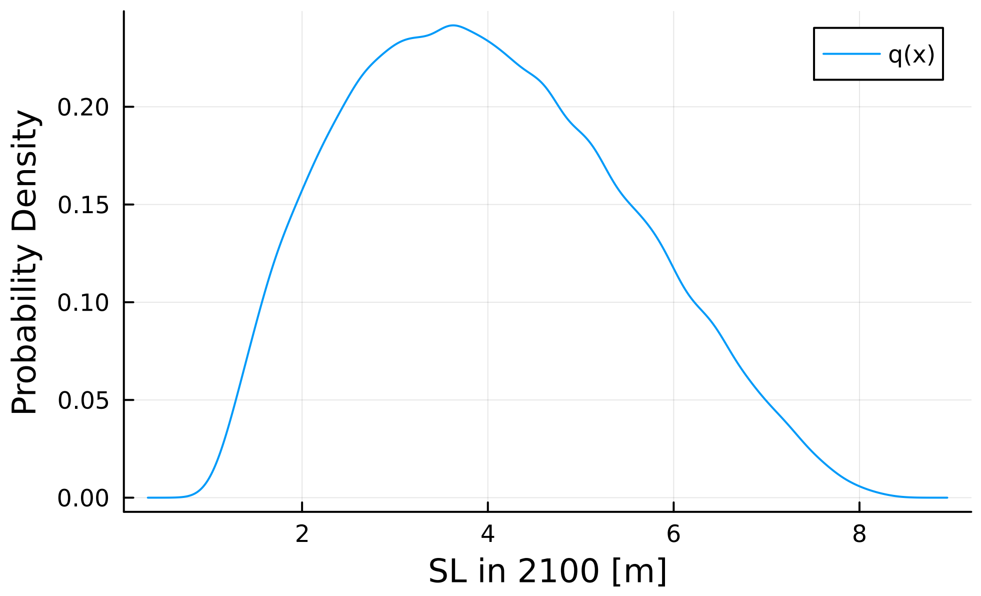
The goal of this project is to more accurately represent sea-level rise in our model. To do this we will be employing rejection sampling to compare the sea-level rise at 2100 based on the parameter values from the Oddo et al. (2017) paper to that coming from probability distributions produced using expert judgement.

```
# create the distribution of sea-level rise values in 2100, x, that is simpler
to follow and sample (sample distribution)
xq = let
    dfq = CSV.read("data/slr_oddo.csv", DataFrame)
   [Oddo17SLR(a, b, c, tstar, cstar) for (a, b, c, tstar, cstar) in eachrow(dfq)]
end

s = []
for i in 1:length(xq)
    slr_param = getindex(xq,i)
    slr_x = slr_param(2100)
    push!(s, slr_x)
end

q = kde(float.(s))

plot(q.x, q.density, xlabel="SL in 2100 [m]", ylabel="Probability Density",
label="q(x)")
```



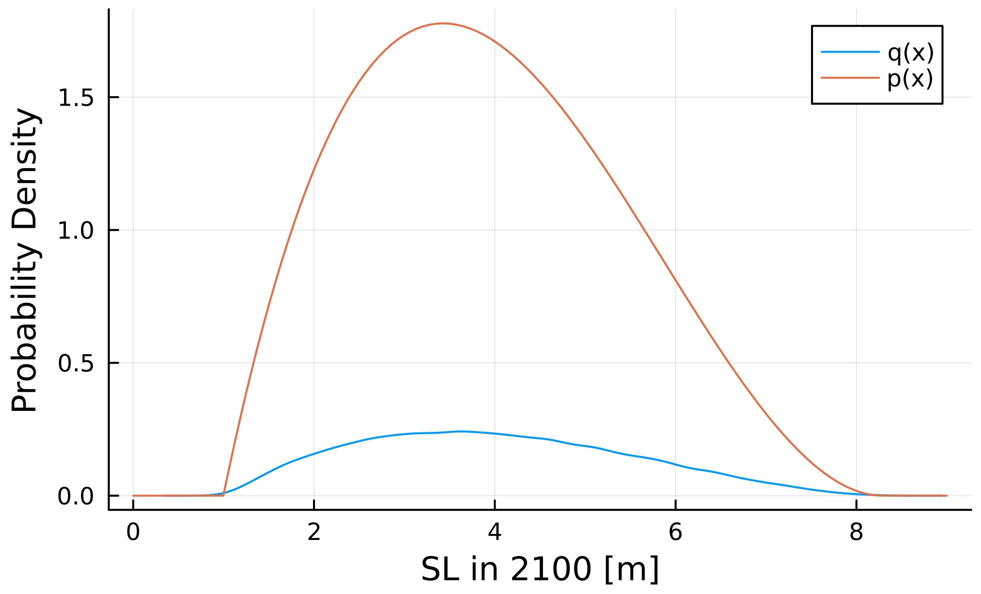
```
# create the distribution of sea-level rise values in 2100, x, that is based on
expert judgement, which we can use to compare our values to and accept or reject
values based on their likelihood in this function and in reality (this is the
beta distribution).

lb = 0.9973656603828569     # from q
ub = 8.28765880297302       # from q

function target_pdf(x)
    x_rescaled = (x-lb)/(ub-lb)
    return pdf(Beta(2,3),x_rescaled)
end
```

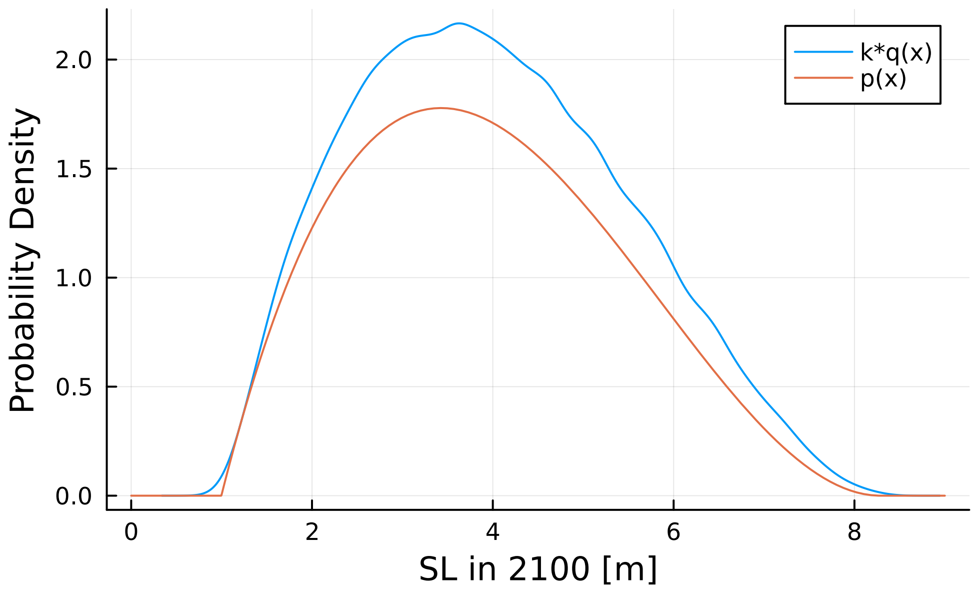
```
target_pdf (generic function with 1 method)
```

```
plot(q.x, q.density, xlabel="SL in 2100 [m]", ylabel="Probability Density",
label="q(x)")
plot!(target_pdf, 0, 9, label="p(x)")
```



```
# we want to modify the sample distribution so that it follows the same domain
and is always greater than or equal to the beta distribution. We will do this
using a scaling constant k.
k_array = []
#x = collect(lb:0.01:ub)
for i in 1:length(s)
    ratios = target_pdf(getindex(s,i)) ./ pdf(q, getindex(s,i))
    push!(k_array, ratios)
end
k = maximum(k_array)

plot(q.x, k.*q.density, xlabel="SL in 2100 [m]", ylabel="Probability Density",
label="k*q(x)")
plot!(target_pdf, 0, 9, label="p(x)")
```



```julia
function sample(size)
    xsample = []
    n = Normal(0.0, 1.0)
    for i in 1:size
        xs = rand(s)
        cs = rand(n)
        if (target_pdf(xs) / (k .* pdf(q,xs))) >= cs
            push!(xsample,xs)
        end
    end
    return xsample
end
```

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

```julia
slr_predictions = sample(10000)
slr_scenarios = []
for i in 1:length(slr_predictions)
    c = getindex(slr_predictions,i)
    for l in 1:length(s)
        m = getindex(s,l)
        if c == m
        push!(slr_scenarios, getindex(xq,l))
        end
    end
```

```
end
return slr_scenarios
```

```
19901-element Vector{Any}:
 Oddo17SLR{Float64}(27.69257787, 2.143358669, 0.002199997, 2083.243249,
15.27748429)
  Oddo17SLR{Float64}(35.95109589, 2.42558408, 0.003855458, 2077.824951,
31.95755043)
  Oddo17SLR{Float64}(35.95109589, 2.42558408, 0.003855458, 2077.824951,
31.95755043)
 Oddo17SLR{Float64}(53.74950965, 2.685557655, 0.005079925, 2063.795657,
27.49672245)
 Oddo17SLR{Float64}(53.74950965, 2.685557655, 0.005079925, 2063.795657,
27.49672245)
 Oddo17SLR{Float64}(6.096911357, 1.578988778, -0.00095835, 2067.601599,
31.78058585)
  Oddo17SLR{Float64}(33.62602049, 2.140382638, 0.001804258, 2048.34855,
25.37248392)
  Oddo17SLR{Float64}(33.62602049, 2.140382638, 0.001804258, 2048.34855,
25.37248392)
 Oddo17SLR{Float64}(32.94573361, 2.347509418, 0.003205018, 2059.523037,
32.02504491)
 Oddo17SLR{Float64}(32.94573361, 2.347509418, 0.003205018, 2059.523037,
32.02504491)
 Oddo17SLR{Float64}(32.94573361, 2.347509418, 0.003205018, 2059.523037,
32.02504491)
 Oddo17SLR{Float64}(32.94573361, 2.347509418, 0.003205018, 2059.523037,
32.02504491)
 Oddo17SLR{Float64}(32.94573361, 2.347509418, 0.003205018, 2059.523037,
32.02504491)
 ⋮
  Oddo17SLR{Float64}(38.53489108, 2.210176804, 0.00148769, 2018.197144,
14.77578359)
  Oddo17SLR{Float64}(38.53489108, 2.210176804, 0.00148769, 2018.197144,
14.77578359)
  Oddo17SLR{Float64}(38.53489108, 2.210176804, 0.00148769, 2018.197144,
14.77578359)
 Oddo17SLR{Float64}(54.08134042, 2.772611982, 0.005506049, 2017.341108,
16.43723051)
 Oddo17SLR{Float64}(50.84392605, 2.581534108, 0.003348909, 2052.590385,
17.58170243)
 Oddo17SLR{Float64}(50.84392605, 2.581534108, 0.003348909, 2052.590385,
17.58170243)
 Oddo17SLR{Float64}(50.84392605, 2.581534108, 0.003348909, 2052.590385,
17.58170243)
 Oddo17SLR{Float64}(50.84392605, 2.581534108, 0.003348909, 2052.590385,
17.58170243)
```

```
 Oddo17SLR{Float64}(50.84392605, 2.581534108, 0.003348909, 2052.590385,
17.58170243)
 Oddo17SLR{Float64}(50.84392605, 2.581534108, 0.003348909, 2052.590385,
17.58170243)
 Oddo17SLR{Float64}(23.22492942, 1.824699877, -0.000509546, 2057.334926,
23.79367908)
 Oddo17SLR{Float64}(28.90572688, 1.725206709, -0.000744653, 2020.898468,
19.91245549)
```

```
function draw_surge_distribution()
    μ = rand(Normal(5, 1))
    σ = rand(Exponential(1.25))
    ξ = rand(Normal(0.1, 0.05))
    return GeneralizedExtremeValue(μ, σ, ξ)
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

```
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)
```

Optimization

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.

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

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

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)
```

Running

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

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

```
Optimization Result
===================
  Iteration:       75
  Minimum:         454808
  Minimizer:       [9.54111]
  Function calls:  525
  Total time:      3.7566 s
  Stop reason:     Due to Convergence Termination criterion.
```

We can view the minimum of the objective function with

```julia
minimum(result)
```

```
454808.01732672227
```

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

```julia
minimizer(result)
```

```
1-element Vector{Float64}:
 9.54111013773624
```

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}(22.61426175,    1.978239504,    0.001067023,
2034.73004,            20.51759182),            GeneralizedExtremeValue{Float64}
(μ=5.1592657251467795,       σ=1.786085589594579,       ξ=0.04909541822954309),
0.043912364344874985)
    SOW{Float64}(Oddo17SLR{Float64}(47.60422624,    2.764449412,    0.006843915,
2074.999255,            21.43938634),            GeneralizedExtremeValue{Float64}
(μ=4.801196796558416,       σ=0.6923008755715017,       ξ=0.23052884287490827),
0.07869168998582142)
     SOW{Float64}(Oddo17SLR{Float64}(45.4413483,     2.590082671,     0.00477723,
2060.129678, 18.2646484), GeneralizedExtremeValue{Float64}(μ=5.840579770203203,
σ=3.5558601124004285, ξ=0.08723607897000375), 0.06488106252216932)
    SOW{Float64}(Oddo17SLR{Float64}(47.87553469,    2.319340975,    0.002182193,
2034.434172,            20.6749975),            GeneralizedExtremeValue{Float64}
(μ=3.8315083460098016,      σ=3.554939606231098,      ξ=-0.0028238758770310346),
0.04051324148798102)
    SOW{Float64}(Oddo17SLR{Float64}(35.85946597,    2.452775818,    0.004918914,
2044.977533, 31.0743158), GeneralizedExtremeValue{Float64}(μ=5.503318936597412,
σ=0.6057691505230971, ξ=0.14893794145853154), 0.07318252811671196)
    SOW{Float64}(Oddo17SLR{Float64}(26.38092309,    2.30793719,    0.003462224,
2016.172383,            8.368073592),            GeneralizedExtremeValue{Float64}
(μ=4.0973209937081165,       σ=0.5519168910230567,       ξ=0.07677316316267815),
0.022963396163057218)
    SOW{Float64}(Oddo17SLR{Float64}(36.87831619,    2.669125054,    0.005855494,
2073.553442,            15.94656013),            GeneralizedExtremeValue{Float64}
(μ=4.976640818675022,        σ=2.929128322094571,        ξ=0.0748038905239377),
0.07234831771584435)
    SOW{Float64}(Oddo17SLR{Float64}(27.60569459,    2.484090062,    0.004874684,
2058.32466, 22.26280587), GeneralizedExtremeValue{Float64}(μ=6.304505229243999,
σ=0.6796814459767018, ξ=0.13313994855425473), 0.04084383078870752)
    SOW{Float64}(Oddo17SLR{Float64}(40.91826533,    2.609482382,    0.006679396,
2031.148616,            17.77504924),            GeneralizedExtremeValue{Float64}
(μ=5.612056817161585,       σ=2.81935964343091,       ξ=0.026574484379920854),
0.05338896379200811)
    SOW{Float64}(Oddo17SLR{Float64}(54.03561279,    2.652309904,    0.004821768,
2065.631127,            28.62358697),            GeneralizedExtremeValue{Float64}
(μ=5.974257340053487,      σ=1.3745730137960008,      ξ=-0.004411514151707152),
0.02982395279108502)
    SOW{Float64}(Oddo17SLR{Float64}(42.35319498,    2.107910979,    0.000589705,
2062.750402,            17.97846928),            GeneralizedExtremeValue{Float64}
(μ=5.706722362384169,       σ=0.4475118422886198,       ξ=0.08349999654312897),
0.025689250596865044)
    SOW{Float64}(Oddo17SLR{Float64}(30.86820292,    2.101634955,    0.002403907,
2052.995763,            31.46292524),            GeneralizedExtremeValue{Float64}
(μ=4.307007231689129, σ=1.9945414455727353, ξ=0.20881284794970267),
0.07615161997445703)
    SOW{Float64}(Oddo17SLR{Float64}(20.6055642, 1.802857256, 0.000652724,
2020.972713, 10.02678381), GeneralizedExtremeValue{Float64}
(μ=6.433525656652894, σ=0.28491381216716927, ξ=0.042331800536263224),
0.05989456228664741)
 ⋮
   SOW{Float64}(Oddo17SLR{Float64}(29.96314366, 2.200314792, 0.002979216,
2086.059146, 29.47874344), GeneralizedExtremeValue{Float64}
(μ=5.852048346873611, σ=0.46933006204308647, ξ=0.08830977838825098),
0.020802045320114662)
    SOW{Float64}(Oddo17SLR{Float64}(51.97159988, 2.2802137, 0.001601966,
2019.34052, 16.23208714), GeneralizedExtremeValue{Float64}(μ=4.936344528332246,
σ=0.29964843733494845, ξ=0.061382696771434375), 0.05374843832753328)
   SOW{Float64}(Oddo17SLR{Float64}(20.76603107, 1.880122683, 0.000362745,
2041.750543, 19.34488305), GeneralizedExtremeValue{Float64}
(μ=3.7820151093398553, σ=0.1403473868398733, ξ=0.06665005213494601),
0.012138111249493516)
   SOW{Float64}(Oddo17SLR{Float64}(28.5307798, 1.661047776, -0.002325799,
2032.097324, 10.47266447), GeneralizedExtremeValue{Float64}
(μ=5.591290057672752, σ=0.022244130648271264, ξ=0.055152538774899895),
0.03510151584563126)
   SOW{Float64}(Oddo17SLR{Float64}(23.39940423, 1.960294658, 0.000984744,
2029.868744, 26.96131761), GeneralizedExtremeValue{Float64}
(μ=5.484879083835489, σ=3.0099405201654132, ξ=0.08636514474849032),
0.04019400052648379)
     SOW{Float64}(Oddo17SLR{Float64}(44.48535161, 2.11901817, -6.64e-5,
2040.261081, 27.03670059), GeneralizedExtremeValue{Float64}
(μ=4.098889452454249, σ=0.11111117305426037, ξ=0.022236381784040596),
0.06672330608824631)
   SOW{Float64}(Oddo17SLR{Float64}(45.41345516, 2.884825094, 0.006460763,
2031.411928, 11.36815248), GeneralizedExtremeValue{Float64}
(μ=5.6632834428719585, σ=0.30204647767371096, ξ=0.14034773727908867),
0.02816427502575243)
   SOW{Float64}(Oddo17SLR{Float64}(45.39672029, 2.412330673, 0.002759026,
2089.14643, 15.76434648), GeneralizedExtremeValue{Float64}(μ=5.644533129673197,
σ=0.7939976999990541, ξ=0.09857385622856223), 0.04469711813736867)
   SOW{Float64}(Oddo17SLR{Float64}(42.67263735, 2.712022566, 0.005831119,
2041.69063, 28.35441963), GeneralizedExtremeValue{Float64}(μ=4.243520195003441,
σ=3.900339636627553, ξ=0.15984690402869664), 0.0977582976083996)
   SOW{Float64}(Oddo17SLR{Float64}(10.0018396, 1.303071469, -0.003457793,
2018.694655, 18.20921485), GeneralizedExtremeValue{Float64}
(μ=4.740965199481158, σ=0.49909292679914224, ξ=0.0061093216506191705),
0.09350454479095838)
    SOW{Float64}(Oddo17SLR{Float64}(41.0258394, 2.42594007, 0.004442993,
2078.753658, 29.99330153), GeneralizedExtremeValue{Float64}
(μ=5.046108868294086, σ=0.22009332585830527, ξ=0.13780160468736796),
0.04567843528244755)
```

```
   SOW{Float64}(Oddo17SLR{Float64}(29.63894656,   2.036171717,   0.000939955,
2077.471886, 20.46651097), GeneralizedExtremeValue{Float64}(μ=5.31457800663226,
σ=4.7149979394504316, ξ=0.055416193099212314), 0.05945120410893797)
```

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

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

```
Options
=======
  rng:               TaskLocalRNG()
  seed:              501990258
  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.

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

```
Algorithm Parameters
====================
  ECA(η_max=2.0, K=7, N=0, N_init=0, p_exploit=0.95, p_bin=0.02, ε=0.0,
adaptive=false, resize_population=false)

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.

```
```

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

```
Optimization Result
===================
  Iteration:       102
  Minimum:         448188
  Minimizer:       [8.85572]
  Function calls:  714
  Total time:      50.3050 s
  Stop reason:     Due to Convergence Termination criterion.
```

We can view our result with

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

```
448188.3801104225
```

```
1-element Vector{Float64}:
 8.855719847465929
```

Validation

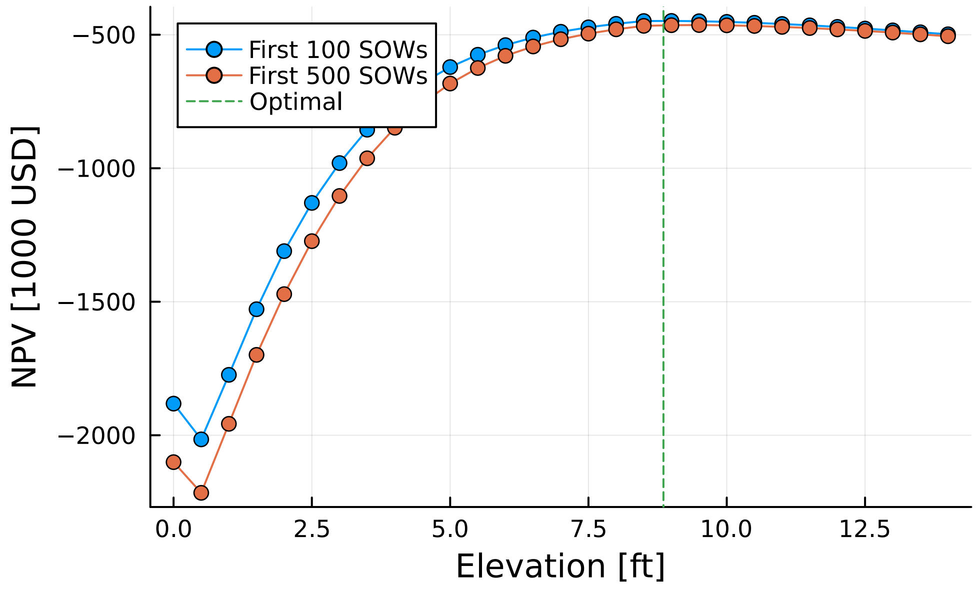
In this case, we don't really *need* optimization – we can use brute force. We can compare by plotting the objective function for a range of elevations (from 0 to 14 ft) using all SOWs.

```
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

```
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