Lab 6: Policy Search

CEVE 421/521

Fri., Mar. 1

In this lab, we will implement single-objective policy search for our house elevation problem. These methods can also be used for multi-objective policy search, but this does increase computational complexity.

0.1 Setup

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.

```
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.

```
using Revise
using HouseElevation
```

1 Problem description

In lab 5, you used exploratory modeling to consider how different plausible scenarios might affect the flood risk for a house. In this lab, we will use policy search to find the "optimal" elevation for the house.

1.1 Decision variables

We're going to focus on a single decision variable: how high to elevate a house. Of course, running a full optimization here is probably overkill, as Zarekarizi et al. (2020) showed that a brute force search over all possible elevations is sufficient to find a good solution. However, we want to build up some optimization expertise to help us with more complex problems in the future.

We will use a continuous decision variable, the height of the house above the ground. We limit it between 0 and 14 feet.

1.2 Objective function

For now, we'll keep the same objective function that we've been using: net present value, considering both the cost of heightening the house and the discounted expected costs of future flood damage.

As you know, it's not enough to state the objective function, however. We also need to consider the state(s) of the world over which we will optimize.

```
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

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

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

function draw_surge_distribution()
      = rand(Normal(5, 1))
      = rand(Exponential(1.5))
      = rand(Normal(0.1, 0.05))
    return GeneralizedExtremeValue( ,  ,  )
end
function draw_discount_rate()
    return 0.0
end

```

```julia
N_SOW = 100
sows = [
    SOW(rand(slr_scenarios), draw_surge_distribution(), draw_discount_rate()) for
    _ in 1:N_SOW
] # for 10 SOWs
```

1.2.1 More-efficient storm surge sampling

Thus far, we have estimated annual expected losses in a particular year using a simple Monte Carlo estimate:

$$\mathbb{E}[f(x)] = \int p(x)f(x)\,dx \quad \approx \frac{1}{N}\sum_{i=1}^{N} f(x_i) \quad \text{where } x_i \sim p(x)$$

where, in our context, x is the flood depth and $f(x)$ is the depth-damage function. However, this is not the most efficient way to estimate the expected value of a function, because it requires a lot of samples from $p(x)$ to get a good estimate. This will invariably result in a lot of samples very near each other in the middle of the distribution, where the function is relatively flat, and few samples in the tails, where the function is steep.

A more efficient way to estimate the expected value of a function is to use trapezoidal integration. This is a simple idea: we divide the domain of the function into n intervals, and then estimate the area under the curve in each interval using the trapezoidal rule:

$$\int_a^b f(x)\,dx \approx \sum_{i=1}^{n-1} \frac{f(x_i) + f(x_{i+1})}{2}(x_{i+1} - x_i)$$

where x_i are the n points at which we evaluate the function between a and b. If we want to estimate an expectation, we need to multiply this by the probability of each interval, which we can estimate using the empirical distribution of our samples.

$$\mathbb{E}[f(x)] \approx \sum_{i=1}^{n-1} \frac{p(x_i)f(x_i) + p(x_{i+1})f(x_{i+1})}{2}(x_{i+1} - x_i).$$

One important note here is that an expectation is infinite, but we truncate the integral at some point. We will use the 0.0005 and 0.9995 quantiles of the distribution of flood depths to define the bounds of the integral, which introduces some (here very small) error.

1.2.2 Validation

We can make sure that we get the same thing by comparing our result to a simple Monte Carlo estimate using 25,000 samples. We can also time how long they take using the `@time` macro (which is actually not a great way to evaluate code – a better approach uses the BenchmarkTools package but we won't go into this here – instead we run use `@time` the *second* time we call each function, which gives a better estimate than timing it the first time).

```julia
a = Action(3.0u"ft")
sow = first(sows)
_ = run_sim(a, sow, p)
@time run_sim(a, sow, p)
```

```
  0.000756 seconds (1.48 k allocations: 577.188 KiB)
```

```
-283854.1700907072
```

This is much faster than the old method, but gives us the same result

```
_ = HouseElevation.run_sim_old(a, sow, p)
@time HouseElevation.run_sim_old(a, sow, p)
```

```
  0.089079 seconds (1.17 k allocations: 45.819 MiB, 6.96% gc time)
```

```
-291265.3844527373
```

This makes our function a lot faster to run! We only have to evaluate our function about 150 times per year, rather than 10000.

1.3 Metaheuristics.jl

We are attempting to solve a single-objective optimization problem that is likely nonlinear and nonconvex. We will use the Metaheuristics.jl package to do this. This package implements a number of optimization algorithms, including genetic algorithms, that are well-suited to this type of problem. Let's follow a quick overview from the docs.

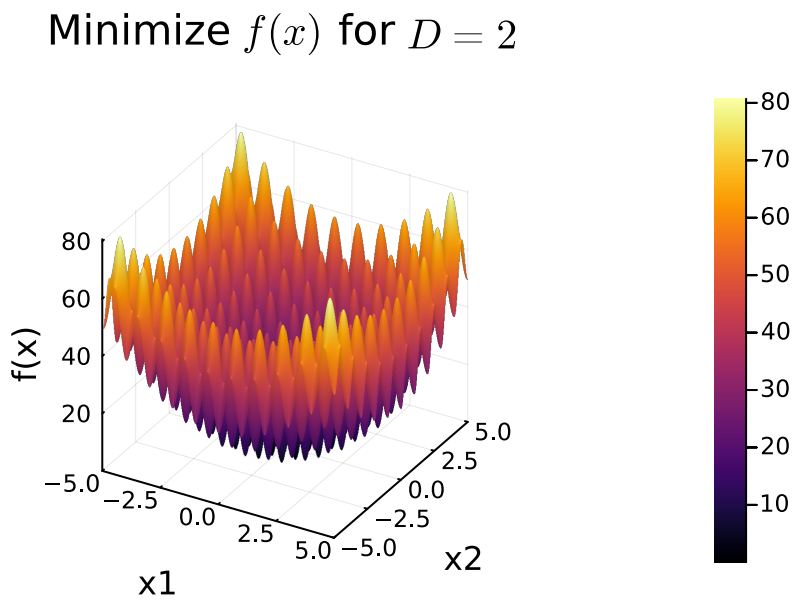
Let's say we want to minimize the following function:

$$f(\mathbf{x}) = 10D + \sum_{i=1}^{D} \left(x_i^2 - 10\cos(2\pi x_i)\right)$$

where $\mathbf{x} \in [-5, 5]^D$, i.e., $-5 \leq x_i \leq 5$ for $i = 1, \ldots, D$. We can plot it for $D = 2$:

```
f(x) = 10length(x) + sum(x .^ 2 - 10cos.(2 * x))
let
    # Generate a grid of points for the surface plot
    x = range(-5; stop=5, length=1000)
    y = range(-5; stop=5, length=1000)
    z = [f([i, j]) for i in x, j in y]

    # Create the surface plot
    surface(
        x, y, z; xlabel="x1", ylabel="x2", zlabel="f(x)", title=L"Minimize $f(x)$ for $D=2$"
    )
end
```



Let's minimize it with $D = 10$. We need to define bounds of the optimization, which constrains the search space for the decision variable.

```
D = 10
bounds = boxconstraints(; lb=-5ones(D), ub=5ones(D))
```

```
BoxConstrainedSpace{Float64}([-5.0, -5.0, -5.0, -5.0, -5.0, -5.0, -5.0, -5.0, -5.0, -5.0], [5.0
```

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

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

```
Optimization Result
===================
  Iteration:       668
  Minimum:         1.98992
  Minimizer:       [-6.36036e-09, -1.0928e-09, -0.994959, …, -0.994959]
  Function calls:  46760
  Total time:      0.0552 s
  Stop reason:     Due to Convergence Termination criterion.
```

We can view the minimum of the objective function with

```
minimum(result)
```

```
1.9899181141865938
```

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

```
minimizer(result)
```

```
10-element Vector{Float64}:
 -6.36036467785244e-9
```

```
-1.0927951567001593e-9
-0.994958636641635
-5.614051171308327e-9
 1.1078054493982587e-10
-5.348024268333651e-10
 7.174152931733373e-10
 1.1646748887131931e-9
 1.5329953879830772e-9
-0.9949586342945572
```

Often, however, some additional specifications are needed. In our case, we'll want to add some specifications for the optimizer. For example, we can set a time limit (in seconds) for the optimization – this may cause it to terminate before convergence! Another useful option is `f_tol_rel`, which sets the relative tolerance for the objective function value. For our case study, we'll want a very loose tolerance (high value of `f_tol_rel`) because the function we are minimizing has results on the order of tens of thousands of dollars. For this tutorial, we'll use the default value of `f_tol_rel`.

```
options = Options(; time_limit=10.0)
```

```
Options
=======
  rng:              TaskLocalRNG()
  seed:             1705272695
  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:        2.220446049250313e-16
  time_limit:       10.0
  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( _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!(918)
result = optimize(f, bounds, algorithm)
```

```
Optimization Result
===================
  Iteration:       603
  Minimum:         0
  Minimizer:       [-2.00019e-09, -1.2566e-09, 9.04557e-10, …, -2.38253e-09]
  Function calls:  42210
  Total time:      0.0570 s
  Stop reason:     Due to Convergence Termination criterion.
```

and we can check if we get a different result in the next iteration

```julia
Random.seed!(952)
result = optimize(f, bounds, algorithm)
```

```
Optimization Result
===================
  Iteration:       1
  Minimum:         0
  Minimizer:       [-2.27445e-09, 1.29854e-09, 1.28329e-09, …, -1.72435e-09]
  Function calls:  70
  Total time:      0.0409 s
  Stop reason:     Due to Convergence Termination criterion.
```

2 Your tasks

2.1 Set up caching

You'll want to `pip install jupyter-cache` and then make sure you have caching enabled for this one, since some of the computations are a bit slow and you don't want to have to re-run them every time you open VS Code.

2.2 Explore

Before digging too deep into the case study, play around with some of the parameters in this optimization tutorial. Vary D, the bounds of the optimization problem, the stopping criteria, or the algorithm. Get some intuition and **ask any questions** you have about the optimization process.

2.3 Optional: improved estimation of flood PDF

If you'd like, you can use Extremes.jl (or another method you're familiar with) and the gauge data for the nearest station. See previous instructions for tips on the data. You can use this to make your case study more realistic. If you don't want to do this, you can continue to use your hypothetical distribution.

2.4 Optimization

In order to use this optimization package on our problem, we need to define an objective function. This includes not only the objective, but also the SOW(s) over which we will optimize. This also introduces a trade-off: using a few SOWs will make the optimization faster, but may result in a suboptimal solution. Using many SOWs will make the optimization slower, but may result in a more robust solution.

We'll keep the number of SOWs used for optimization relatively small, and then we'll evaluate performance (of our "optimal" solution) using a larger number of SOWs.

1. Set your random seed to 2024 so that you always get the same answers when you re-run your code.
2. Generate `N_SOW = 100_000` sows at random as in the previous lab and/or as in the template code provided above.
3. Pick the first `N_SOW_opt = 10` of these sows to use for optimization. You can (and should!!) increase this number once you have a working solution, but we'll use just a few to make sure everything is working.
4. Define an objective function that takes in a single number as an input (the elevation of the house in feet) and returns one objective function (the net present value of the house for that elevation).
    1. Convert the input scalar to an `Action`
    2. Call `run_sim` on each of the `N_SOW_opt` sows and the elevation to get the expected value of the objective function.
    3. Return the negative of the sum of these expected values (since we are minimizing).
5. Test your objective function with a few different elevations to make sure it's working.
6. Run the optimization with the objective function and see what elevation it recommends.
7. Validate the result by plotting the objective function for a range of elevations (from 0 to 14 ft) using all your SOWs. Is the recommended elevation the minimum? (We're lucky that in this problem we can compare our optimization solution to a brute-force approach!) If it doesn't seem to be the minimum:
    1. try increasing `N_SOW_opt` and see if the result changes.
    2. check whether the optimization algorithm is converging

2.5 Reflection

Conclude your analysis by reflecting on the following questions

1. How are we framing this problem? What are the decision variables, the objective function, and the states of the world over which we optimize?
2. Digging deeper, we are averaging the objective function computed over a finite number of states of the world. This assumes that they are all drawn from a distribution representing the "true" distribution of states of the world. Is this a good assumption?
3. What's not being considered in this analysis that might be important?

Zarekarizi, M., Srikrishnan, V., & Keller, K. (2020). Neglecting uncertainties biases house-elevation decisions to manage riverine flood risks. *Nature Communications*, *11*(1, 1), 5361. https://doi.org/10.1038/s41467-020-19188-9