3. Randomness and probability: Stochastic thinking

Last time

- Dynamics of systems with *deterministic* rules (no noise / randomness / stochasticity)
- $x_{n+1} = f(x_n)$
- `for` loops, `Vector`s
- Anonymous functions
- Plotting, interactive visualization

Goals for today

- Example of nonlinear dynamics
- Fundamentals of **probability**
- Discrete **random variables** and distributions
- Understanding via calculation + visualization
- Characterise **variability** of random variable: probability distribution + summary statistics

Nonlinear dynamics

- General dynamics with singe-step function f:

$$x_{n+1} = f(x_n)$$

- **Discrete-time dynamical system**
- Apply f to previous output at each step.
- Example: $f = \cos$.
- "Repeatedly press the cos key on your calculator". What happens?

- Similar code to before

```
function iterate_cos(x0, N)
    xs = [x0]
    x = x0

    for n = 1:N
        x = cos(x)
        push!(xs, x)
    end

    return xs
end
```

- Simple so update value directly: `x = cos(x)`

Fixed points

- Behaviour is *completely different*
- Iterates **converge**:

$$x_n \to x^* \text{ as } n \to \infty$$

- **Fixed point**: *does not change* when f is applied:

$$x^* = f(x^*)$$

- **Transcendental equation**: *impossible* to find explicit form for solution.
- But iterative method successfully solves this equation (with certain precision).

Rate of convergence

- How “good” is this method?
- Measure *distance* of x_n from x^*, i.e. $\delta_n := |x_n - x^*|$
- **Exercise**: Implement this.
- Find that it converges “quickly”.
- How characterize *rate* of convergence, i.e. how *fast* does δ_n decrease as function of n?
- Plot data differently: **log scale**
- `yscale=:log` in `Plots.jl`

Stability analysis

- What can we say about $\delta_n := |x_n - x^*|$ analytically?
- For large n, know x_n *close to* x^*, so

$$x_{n+1} = x^* + \delta_{n+1} = f(x_n) = f(x^* + \delta_n)$$

$$\simeq f(x^*) + \delta_n \, f'(x^*)$$

- So δ_n approximately satisfies *linear* dynamics:

$$\delta_{n+1} = \lambda \, \delta_n,$$

with $\lambda = f'(x^*)$ (constant).

- *We already understand this!*
- **Computing / mathematics: often try to reduce new problem to problem you can already solve!**
- Behaviour of nonlinear system near fixed points often reduces to analyzing *linearized* system.

Randomness and probability

Why randomness / stochasticity?

- Many things in world behave predictably
- E.g. Newtonian mechanics at scale of galaxies
- Model with **deterministic** model
- Others are **random** (or *seem* so), e.g. rolling a die
- Is a coin toss *really* random? Diaconis et al, “Dynamical bias in the coin toss”; and this non-technical note
- Quantum mechanics: Microscopic world *is* random

Randomness as uncertainty

- Even deterministic systems can behave “randomly”:
  - logistic map (Pset 1)
  - Lorenz system – model of weather
- **Brownian motion** (1827): particle immersed in water
- **Model** as bouncing balls
- Deterministic, chaotic many-body dynamical system
- Simulation
- Randomness $\equiv$ *unknown information* in dynamical processes.

Computing using randomness

- How generate randomness on computer?
- Answer 1: Computers are deterministic, so *we can't*!
- Answer 2: Start-up sequence of computer generates "entropy" = unpredictable bits
- Answer 3: Use real physical process, e.g. noise from electronic circuit or atmospheric noise: www.random.org

Pseudo-random numbers

- Answer 4: Generate “random-looking” sequences
- Use sufficiently complex deterministic process – iterated function
- Bad “random number” generators invalidated results of many Monte Carlo simulations of phase transitions in statistical physics from 1970s
- Explore different random number generators: RandomNumbers.jl package
- Make sure they pass randomness tests

Simple example:

- "Linear congruential generator"
- $x_{n+1} = (a x_n + b) \bmod m$

```
const a = UInt(6364136223846793005)  # unsigned integer
const b = UInt(1442695040888963407)

my_rand_int(x) = a*x + b

x = UInt(3)
for i in 1:10
    global x = myrandint(x)
    y = x / typemax(UInt)  # convert to interval [0, 1)
    @show y
end
```

Throwing a die

- Simplest random processes:
  - toss a coin
  - roll a die
- Let's *simulate* rolling a die *on the computer*:
  - generate integer between 1 and 6
  - each number should be "equally likely"

- What does “equally likely” mean?
- Another word: **uniform** (“all look the same”)
- Each should be produced “with the same probability”
- Interpretation: After a large number N of rolls, the proportion of 1s should be “the same” as the proportion of 2s etc.

Rolling a die in Julia

- Main function: `rand`
- `rand(X)` **samples** objects “randomly”– i.e. **uniformly** – from set `X`:

```julia
X = 1:6
typeof(X)
```

```julia
Array(X)    # make into array to see what's inside

collect(X)  # alternative
```

- Sample:

```julia
rand(X)
```

- `rand` is unusual function: result returned changes each time it's called
- [In fact, it is silently modifying a **global state variable**:

```
import Random
global_rng = copy(Random.GLOBAL_RNG)

@show rand()
@show rand()

@show rand(global_rng)
@show rand(global_rng)
```

]

Random variables

- Call X the result of the action “roll a die”.
- What kind of object is X?
- Every time we ask it for its value it gives a different **outcome**.
- Name: **random variable**.
- Need to know **how frequently** it produces each outcome: **probability distribution**.

Discrete probability = counting

- We want to say Prob$(X = 1) = \frac{1}{6}$
- What does this mean?
- If roll die "large number" N of times, **count** n_1, the number of 1s.
- Expect **proportion** p_1/N to be "close to" $\frac{1}{6}$.

Computational thinking: Do the experiment!

- Computers are good at counting!:
  - Generate data
  - Count how many times each possible outcome occurs
- Need 6 numbers during the count, so need *mutable* data structure.

Computational experiment

- `Vector` is faster than `Dict`. (Really? Benchmark! – **exercise**)

```
roll_die(n) = rand(1:n)  # roll an n-sided die

N = 100

sides = 6
data = [roll_die(sides) for i in 1:N]

counts = zeros(Int, sides)

for result in data
    counts[result] += 1
end
```

Use `Dict` instead

- For general data, cannot do this – don't know set of possible outcomes.
- Then should use a dictionary.
- Of course, we should *put this useful functionality into a function*
- **Exercise**: Implement this
- Provided by `countmap` in `StatsBase.jl` package.

Plotting the data

- We have *categorical data*: outcomes are discrete categories (values cannot be compared)
- E.g. no sense in saying that $1 < 2$ in context of die roll.
- Plot categorical data using *bar chart*:

```
using Plots

bar(counts, leg=false)
```

Make it interactive?

- Suppose we roll one die at a time and want to update the statistics
- We can do this using `Interact`, but...
- We don't want to regenerate new data each time, but rather use the same data
- So pre-generate data *before* plotting
- Plot only relevant *portion* of data

Frequencies

- Instead of counts, plot **proportion** or **frequency**
- Compare to expected result:

```
bar(counts ./ N, leg=false)
hline!( [1/6], ls=:dash, lw=2)    # horizontal line
```

- Here the `.` means **broadcast**: apply the operation element by element

Probability distribution

- Heights of bars are **probability** that each value occured in the **sample**.
- Collection of all probabilities (proportions / frequencies) is called the **probability distribution**.
- Gives Prob$(X = i)$ for each possible outcome i in *discrete* set

Variability

- We have **finite sample** from ideal **population**
- If repeat experiment, get different sample with different counts
- Plot implies die is *biased* (non-uniform) – one bar taller than others.
- But repeating calculation gives *different* results each time
- How characterize this *variability*?

Characterising variability

- Focus on n_1 := total number of 1s out of N
- Also a random variable; ask same questions:
  - what are possible outcomes?
  - how often does each outcome occur?
- I.e. want **probability distribution** of n_1
- Expect: p_1 "close" to $1/6$, so n_1 "close to" $N/6$.
- **Variability**: how *far away* from $N/6$ can n_1 be?
- How count number of rolls that give 1?

Probability distribution of n_1

- Use `for` loop – **exercise**
- Julia has tools to simplify (but probably not quicker – **benchmark!**): `julia  n1(N) = count( rand(1:6) == 1 for i in 1:N )  # or count`
- *Generator expression* $\equiv$ array comprehension *without creating array*

- What is **support** of n_1: set of possible outcomes?
- Minimum possible value is 0, maximum is N; all values in between are possible.
- Intuitively those extreme values are *very* unlikely (improbable = low probability).
- **Exercise**: How improbable?
- Can calculate mathematically and/or do *computer experiment*.

n_1 experiment

- Run experiment for n_1:

```
N = 1000  # number of die rolls
num_experiments = 10000

# repeat experiment:
n1_data = [n1(N) for i in 1:num_experiments]

using StatsBase
counts = countmap(n1_data)

bar(counts)
```

- See that n_1 “clusters around” the **expected value**.
- Values near extremes “never” occur.
- Characterise using **summary statistics**: numbers that summarise aspects of **distribution**.
- Simplest: **sample mean** = average value
- Given outcomes x_i for $i = 1, \ldots, N$, (arithmetic) mean is

$$\bar{x} := \frac{1}{N} \sum_{i=1}^{N} x_i$$

- Calculate in Julia:

```
m = sum(n1_data) / length(n1_data)    # or mean(n1_data)
```

- Add to plot:

Centering

- Distribution “spreads out” away from mean – how far?
- How can we measure this?
- First **centre** the data by subtracting the mean:

```
n1_centered = n1_data .- m

bar(countmap(n1_centered))
vline!([0], c=:red, lw=2, ls=:dash)
```

Spread

- Want to measure spread as some kind of “average spread from mean” = “average *distance* from mean”
- If just take `mean` of new data, get tiny result near 0:

```
mean(n1_centered)
```

- (`1e-14` means 1×10^{-14}, i.e. a value that is effectively 0.)
- Why? Problem is that negative values cancel out positive values.
- Need to be more clever by removing this cancellation.

Spread II

- (At least) 2 possible solutions: take **absolute value** of displacements from mean, or **square them**:

```
spread = mean(abs.(n1_centered))


variance = mean(n1_centered.^2 )
σ = √variance


@show spread, variance
```

- `Variance` defined by squaring, so must take $\sqrt{}$ for "correct units" (metres vs. metres^2)
- σ is called **standard deviation**
- For this distribution, both measures of spread give approx. same result

Spread III

- Let's plot these:

```
bar(countmap(n1_centered))
vline!([-σ, σ], c=:red, lw=2, ls=:dash)
vline!([-2σ, 2σ], c=:green, lw=2, ls=:dash)
```

- Most data is in interval $[\mu - 2\sigma, \mu + 2\sigma]$.
- How much? Calculate!

```
count(-2σ .< n1_centered .< 2σ) / length(n1_centered)
```

- Approx. 95%: "universal" in many (but *not all* situations) – see later

Review

- Random variables have random outcomes
- Probability distribution measures how frequently different outcomes occur
- Variability between different experiments measured by mean and variance