5. Variability and custom types

Last time

- Random motion: Random walks
- Buiding up a simulation: Many trajectories
- `Vector` of `Vector`s vs. `Matrix`

Goals for today

- How to summarise a probability distribution
- Different kinds of random walkers

- Defining custom types in Julia
- Writing generic code

Variability

- We have **finite sample** from ideal **population**
- If we repeat experiment, get different sample with different counts.
- How characterize this *variability*?

Shape of distribution

- See that position “clusters around” central value: **expected value**
- Values near extremes “never” occur

Shape of distribution

- See that position “clusters around” central value: **expected value**
- Values near extremes “never” occur
- Characterise using **summary statistics**: numbers that summarise aspects of **distribution**
- Simplest: **sample mean** = average value

Shape of distribution

- See that position “clusters around” central value: **expected value**
- Values near extremes “never” occur
- Characterise using **summary statistics**: numbers that summarise aspects of **distribution**
- Simplest: **sample mean** = average value

- Shape is “bell curve”: **Gaussian** or **normal** distribution

Mean

- 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:

```
mean(data) = sum(data) / length(data)


m = sum(n1_data) / length(n1_data)
```

- NB: `mean` is in `Statistics` standard library (no need to install)
- Add to plot using `vline!([m])`

Centre the data

- Distribution “spreads out” from mean

Centre the data

- Distribution “spreads out” from mean
- Want to know **how far** it spreads

Centre the data

- Distribution “spreads out” from mean
- Want to know **how far** it spreads

  ,

- **Centre** data by subtracting mean:

```
centered_data = data .- m
```

Spread

- Measure spread as average “*distance from mean*”

Spread

- Measure spread as average "*distance from mean*"
- If just take `mean` of new data, get tiny result near 0:

```
mean(data)
```

- (`1e-14` means 1×10^{-14}, i.e. a value that is effectively 0.)

Spread

- Measure spread as average "*distance from mean*"
- If just take `mean` of new data, get tiny result near 0:

```
mean(data)
```

- (`1e-14` means 1×10^{-14}, i.e. a value that is effectively 0.)
- Why? Negative values *cancel out* positive values
- Need to *avoid cancellation*. How?

Spread II

- Options to avoid cancellation of displacements from mean:
  - take **absolute value**

Spread II

- Options to avoid cancellation of displacements from mean:
  - take **absolute value**

```
spread = mean(abs.(centered_data))    # no standard name?
```

Spread III

- Another option: **square** distance from mean:

```
variance = mean(centered_data .^ 2)
```

Spread III

- Another option: **square** distance from mean:

```
variance = mean(centered_data .^ 2)
```

- Must take $\sqrt{}$ for "correct units" (metres vs. metres^2):

```
σ = √(variance)
```

Spread III

- Another option: **square** distance from mean:

```
variance = mean(centered_data .^ 2)
```

- Must take $\sqrt{}$ for “correct units” (metres vs. metres^2):

```
σ = √(variance)
```

- σ is called **standard deviation**
- For this distribution, both measures of spread give similar result

Spread III

- Most data is concentrated “near the mean”

Spread III

- Most data is concentrated “near the mean”
- How near?

Spread III

- Most data is concentrated “near the mean”
- How near?
- Most data is in interval $[\mu - 2\sigma, \mu + 2\sigma]$
- How much? Calculate!:

Spread III

- Most data is concentrated “near the mean”
- How near?
- Most data is in interval $[\mu - 2\sigma, \mu + 2\sigma]$
- How much? Calculate!:

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

Spread III

- Most data is concentrated “near the mean”
- How near?
- Most data is in interval $[\mu - 2\sigma, \mu + 2\sigma]$
- How much? Calculate!:

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

- Approx. 95%: “universal” in many (but *not all* situations)

Many walkers

- Now can collect data on many walkers:

```
using StatsBase

T = 10
N = 100

data = [walk_position(T) for i in 1:N]

counts = countmap(data)

bar(counts)
```

Time evolution of statistics

- How calculate time evolution of mean & variance as function of time n?

Time evolution of statistics

- How calculate time evolution of mean & variance as function of time n?
- Need access to all walker positions at all times

Time evolution of statistics

- How calculate time evolution of mean & variance as function of time n?
- Need access to all walker positions at all times
- Store whole history of each walker – lots of memory
- Or evolve walkers for m steps, calculate, then evolve futher

Simulate many walkers

- How make several walkers?

```julia
num_walkers = 100
walkers = zeros(Int, num_walkers)
```

- Need function that *modifies* its argument
- Julia convention: `!` at end of function name:

```julia
function move!(walkers, i)
    walkers[i] += jump()
end
```

- Now move *all* walkers
- Use another method with *same name* since common functionality

```
function move!(walkers)
    for i in 1:length(walkers)
        move!(walkers, i)
    end
end
```

- Make *interactive visualization*: pre-generate data

Different types of walkers

- So far restricted to walker on integers
- Generalize
- E.g. steps uniformly distributed on interval $[-0.5, 0.5]$
- How generate?
- `rand()`: uniform random number in interval $[0, 1)$

- Make function:

```
continuous_jump() = rand() - 0.5
```

- Different "type of jump"
- Make new **abstraction**: random walker defined by given `jump` function
- Makes previous code more **generic**

Make code more generic – abstraction

- *Pass in jump function as argument* to previous function – *code is the same as before*!

```julia
function walk(jump, N)
    x = 0
    positions = [x]

    for i in 1:N
        x += jump()    # now calls custom jump function
        push!(positions, x)
    end

    return positions
end
```

Difficulties

- Walkers have position with different *types* and different *jump functions*
- `x = 0` defines `x` as *integer*
- In problem set 3 will have an internal state too
- Need a better solution

User-defined types

- Collect information for each walker in a **new type**

User-defined types

- Collect information for each walker in a **new type**
- Define using a `mutable struct` in Julia:

"‘‘jl

mutable struct MyWalker x::Int end

w = MyWalker(3) # constructor function

w.x += 1

Abstract types

- `DiscreteWalker` and `ContinuousWalker` are kinds of a supertype `Walker`:

```
abstract type Walker end

mutable struct DiscreteWalker <: Walker   # subtype
    x::Int
end

mutable struct ContinuousWalker <: Walker
    x::Float64
end

position(w::Walker) = w.x
setposition!(w::Walker, x) = w.x = x
```

Jump functions

- Rewrite `jump` functions:

```
jump(w::DiscreteWalker) = rand( (-1, +1) )
jump(w::ContinuousWalker) = rand() - 0.5
```

- Define `initialize!` function:

Walk function

- Rewrite `walk` function:

```
function walk!(w::Walker, N)   # modifies its argument
    positions = [position(w)]

    for i in 1:N
        x = position(w)
        new_x = x + jump(w)

        set_position!(w, new_x)   # now calls custom jump fu
        push!(positions, new_x)
    end

    return positions
end
```

- Make walkers:

```
d = DiscreteWalker(0)
c = ContinuousWalker(0.0)

pos1 = walk(d, 10)
pos2 = walk(c, 10)
```

- Julia generates **specialized code** for each version

Many walkers

- Now can collect data on many walkers:

```
using StatsBase

T = 10
N = 100

data = [walk_position(T) for i in 1:N]

counts = countmap(data)
ks = sort(collect(keys(counts)))

bar(ks, [counts[k] for k in ks])
```

Time evolution of statistics

- How calculate time evolution of mean & variance as function of time n?
- Need access to all walker positions at all times
- Store whole history of each walker – $$$ in memory
- Or evolve walkers for m steps, calculate, then evolve futher.

Simulate many walkers

- How make several walkers?

```julia
num_walkers = 100
walkers = zeros(Int, num_walkers)
```

- Need function that *modifies* its argument
- Julia convention: `!` at end of function name:

```julia
function move!(walkers, i)
    walkers[i] += jump()
end
```

- Now move *all* walkers
- Use another method with *same name* since common functionality

```
function move!(walkers)
    for i in 1:length(walkers)
        move!(walkers, i)
    end
end
```

- Make *interactive visualization*: pre-generate data

Different types of walkers

- So far restricted to walker on integers
- Generalize
- E.g. steps uniformly distributed on interval $[-0.5, 0.5]$
- How generate?
- `rand()`: uniform random number in interval $[0, 1)$

- Make function:

```
continuous_jump() = rand() - 0.5
```

- Different “type of jump”
- Make new **abstraction**: random walker defined by given `jump` function
- Makes previous code more **generic**

Make code more generic – abstraction

- *Pass in jump function as argument* to previous function – *code is the same as before*!

```
function walk(jump, N)
    x = 0
    positions = [x]

    for i in 1:N
        x += jump()    # now calls custom jump function
        push!(positions, x)
    end

    return positions
end
```

Difficulties

- Walkers have position with different *types* and different *jump functions*
- `x = 0` defines `x` as *integer*
- In problem set 3 will have an internal state too
- Need a better solution

User-defined types

- Collect information for each walker in **new type**
- `DiscreteWalker` and `ContinuousWalker` are kinds of a supertype `Walker`:

```
abstract type Walker end

mutable struct DiscreteWalker <: Walker   # subtype
    x::Int
end

mutable struct ContinuousWalker <: Walker
    x::Float64
end

position(w::Walker) = w.x
```

Jump functions

- Rewrite `jump` functions:

```
jump(w::DiscreteWalker) = rand( (-1, +1) )
jump(w::ContinuousWalker) = rand() - 0.5
```

- Define `initialize!` function:

Walk function

- Rewrite `walk` function:

```
function walk!(w::Walker, N)    # modifies its argument
    positions = [position(w)]

    for i in 1:N
        x = position(w)
        new_x = x + jump(w)

        set_position!(w, new_x)    # now calls custom jump fu
        push!(positions, new_x)
    end

    return positions
end
```

- Make walkers:

```
d = DiscreteWalker(0)
c = ContinuousWalker(0.0)

pos1 = walk(d, 10)
pos2 = walk(c, 10)
```

- Julia generates **specialized code** for each version

Julia objects in detail

- Simplest discrete random walker as a Julia object / type:

```
mutable struct SimpleWalker
    x::Int
end
```

- This defines a *new type* called `SimpleWalker`
- Type definition species structure consisting of one or several **fields** / **attributes** that live inside it
- Think of a box containing data
- No objects have been created; only a possible object “shape” has been defined

Constructors

- Julia creates default **constructor** functions with same name as type:

```
methods(SimpleWalker)
```

- Create objects by calling these functions:

```
d = SimpleWalker(0)

typeof(d)
```

- Automatically fills in field values in this new object from function arguments (in order of arguments)

Field access

- Access fields of object with `.`:

`d.x`

d

- Returns value of variable `x` *belonging to* `d`, i.e. the value of the field `x` that “lives inside” the object `d`

Functions acting on objects

- Julian style: Define functions that act on objects:

```julia
function pos(d::SimpleWalker)
    return d.x
end

pos(d)
```

- Short form of function definition:

```julia
pos(d::SimpleWalker) = d.x
```

Mutating functions

- If function *mutates* (modifies) object internals, add `!` to function name:

```
function jump!(w::SimpleWalker)
    w.x += rand( (-1, +1) )   # modifies w.x
end

jump!(d)

@show d
```

Walking a walker

- Use above functions to write random walk
- Note that the function does mutate the object, so called `walk!`:

```
function walk!(w::SimpleWalker, N)
    positions = [pos(w)]

    for i in 1:N
        jump!(w)
        push!(positions, pos(w))
    end

    return positions
end
```

Continuous walker

- Define a new walker type `AnotherWalker`
- Problem: `walk!` function will not work, since its argument is restricted to `SimpleWalker` type
- Need to be able to tell Julia that two different types should **share common behaviour**
- Solution: common **abstract supertype** `Walker`

Abstract common type

- Common abstract supertype:

```
abstract type end Walker
```

- Define types to be subtypes of `Walker` using `<:` ("subtype of")

```
mutable struct DiscreteWalker <: Walker
    x::Int
end

mutable struct ContinuousWalker <: Walker
    x::Float64
end
```

Checking type of objects

- Create objects:

```
d = DiscreteWalker(0)
c = ContinuousWalker(0.0)
```

- Check types: `julia   d isa DiscreteWalker   d isa Walker     # also true`

Common functionality: Single method

- When functionality is common, define function acting on *supertype*:

```
pos(w::Walker) = w.x    # works for *any* Walker!
```

- It works on any object whose type is a subtype of `Walker`:

Distinct functionality

- If distinct functionality for different types, define *different methods* of *same* function:

```
jump!(w::DiscreteWalker) = w.x += rand( (-1, +1) )
jump!(w::ContinuousWalker) = w.x += rand() - 0.5

jump!(c)
pos(c)

jump!(d)
pos(d)
```

Walking any walker

- Define `walk!` for *any* walker by just changing allowed input type
- Uses functions `pos` and `jump!` that must work for any type of `Walker`:

```
function walk!(w::Walker, N)
    positions = [pos(w)]

    for i in 1:N
        jump!(w)
        push!(positions, pos(w))
    end

    return positions
end
```

New walker type

- To define a new walker, just need `jump!` for that new type
- Then `walk!` will already *just work*
- e.g. 2D walker – problem set 3
- If define new subtype of `Walker` whose position is not `x`, define method of `pos` for that type:

```
mutable struct NewWalker
    y::Int
end

pos(w::NewWalker) = w.y

jump!(w::NewWalker) = w += 1
```

Summary of objects

- Objects / user-defined types / custom types wrap up several pieces of data that belong to same object that is being modelled: (type of) **encapsulation**
- Object in computer world corresponds more closely to our mental picture of the object in real world
- Abstraction that allows us to *reuse code*

Summary

. . .

- Characterise variability using **mean** and **variance** or **standard deviation**

Summary

. . .

- Characterise variability using **mean** and **variance** or **standard deviation**

- Most data within 2 standard deviations of mean
- When distribution is result of adding up many effects