6. Random walks II & Monte Carlo methods

Last time

- Introduced random walks
- Random step generation
- Different types of random walks
- Introduction to objects / types

Goals for today

- Objects / types in detail: different kinds of random walkers
- Monte Carlo methods
- Non-random calculations with random processes: calculating areas

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:

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

pos(d)
```

- Short form of function definition:

```
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!`:

```julia
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:

```julia
abstract type end Walker
```

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

```julia
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*

Monte Carlo methods

What are Monte Carlo methods?

- Monte Carlo: City where there are many casinos
- **Monte Carlo method**: Algorithm that uses random numbers to generate a probability distribution that solves a problem
- Will see that can sometimes use *random* processes to answer *non-random* questions
- Result will be **approximation** to true value
- Expect approximation to improve if use more randomness

Example: Monty Hall goat problem

- Game show (originally hosted by Monty Hall):
  You have a choice of 3 doors: Behind one is a car (which you want) Behind the other 2 doors are goats (which you don't want). You pick a door, say door 1, which remains closed. The game show host opens another door, say door 3, which has a goat. She asks you if you would like to switch to the other closed door.
- The host knows which door has the car.
- Should you switch? Vote

Monte Carlo simulation of the Monty Hall

- Many hours of controversy on internet forums and in classrooms
- Problem in **conditional probability**: probability that something is true, *given* that something else is known.
- We can find the correct answer using a Monte Carlo simulation.
- In this case: “run the experiment lots of times”!
- Won’t necessarily help understand *why* that’s the correct answer

Monty Hall algorithm

- Algorithm:
  - Fix location of car
  - Choose random door
  - Find which door(s) host could open
  - Open (remove) host's choice
  - Find possible new choice
  - Switch if desired
  - Check if car is found
- Convert algorithm into code

Code

```julia
function monty_hall(switch::Bool)
    car_location = rand(1:3)
    my_choice = rand(1:3)

    if switch
        host_choices = setdiff(1:3, [car_location, my_choice])
        host_opens = rand(host_choices)

        possible_doors = setdiff(1:3, [my_choice, host_opens])
        my_choice = rand(possible_doors)  # modifies my_choice
    end

    return my_choice == car_location
end
```

Using randomness for non-random calculations

- Until now: Used randomness to model probabilistic situations
- Now: use randomness in a surprising way: to calculate non-random quantities!
- E.g.: What is value of π? Certainly non-random.
- Could use e.g. infinite series – some converge amazingly fast.
- Instead, calculate π to low precision using a general method
- **Monte Carlo integration**: use randomness to calculate **area** of complicated shape
- NB: Unlike differentiation, can prove that no general

π as an area

- Monte Carlo integration calculates volumes (areas in 2D)
- How relate π to an area?
- Area of disc with radius r is $A(r) = \pi r^2$, so calculate $A(1)$
- Monte Carlo methods are only way to integrate in high-dimensional spaces.
- Applications in high-energy physics, Bayesian statistics, statistical mechanics, etc.
- Idea: Count fast to find the probability of complicated events

Idea: Shooting darts

- Idea: Given region with unknown area, enclose in region whose area A we *already know*
- Examples: rectangles, area under polynomial
- Rectangle: Area under very simple polynomial!

$$A = \text{base} \times \text{height}$$

Shooting darts at a pie

- Center unit disc at origin. Enclose by square $[-1,1] \times [-1,1]$
- **Exercise**: Draw the square and the circle.
- *Throw darts* at square, i.e. generate random points
- Some will land inside circle, some outside.
- **Exercise**: Throw $N = 1000$ “darts” at the square. Colour the ones inside the circle differently.
- Gives *idea* to approximate area of circle or other region
- Called **rejection sampling**: we *reject* points outside desired region.

Implementation

- `rand()` generates uniform random number in $[0, 1)$.
- How make uniform random number in $[a, b)$?

```
uniform(a, b) = a + rand() * (b - a)
```

- Code:

```
function area_circle(N)

    num_inside = 0

    for i in 1:N

        x = uniform(-1.0, 1.0)
        y = uniform(-1.0, 1.0)

        if x^2 + y^2 <= 1
            num_inside += 1
        end
    end
```

Variability

- This calculation gives equivalent of a mean.
- If repeat calculation, will get different floating-point result. How different?

```
N = 1000
data = [area_circle(N) for i in 1:1000]

using Plots
scatter(data, leg=false)
```

- Results are centered around **mean** value:

```
mean(data)
```

- This value is close to true value of π
- But as before, there is *variability* in the data:

```
using StatsBase

scatter(data, alpha=0.5)
hline!([mean(data)], lw=3, ls=:dash)
ylims!(0, 4, ms=1, leg=false)
```

Floating-point data

- Note that data now consists of floating-point numbers, instead of integers
- But still can measure variability as before, using

```
m = mean(data)
σ = sqrt(mean((data .- m).^2))
```

- Again can ask what fraction of the data lies within some range:

```
count(m - 2σ .< data .< m + 2σ) / length(data)
```

- Get very similar answer TO discrete case.
- How study *probability distribution* of this random variable?
- $\longrightarrow$ Next class

Generalize to higher dimensions

- Can we use same method to calculate volume of unit ball in 3 dimensions?
- Defined by $x^2 + y^2 + z^2 \leq 1$
- What about in n dimensions?

$$B_n := \{\mathbf{x} \in \mathbb{R}^n : \sum_i x_i^2 \leq 1\}$$

Review

- Julia objects
- Monte Carlo methods
- Solve non-random problems with randomness