PS2

March 26, 2024

1 Problem Set 2 – Sampling-based algorithms in Gen

Instructions: Finish the code following the instructions. Execute all blocks of your Jupyter notebook, write analyses in English in the corresponding Markdown blocks ("marked as YOUR ANSWER HERE"), save the final notebook as a pdf file, and submit it on Canvas.

Acknowledge the contributions if you collaborated with someone else. You must write your own code and analyses.

First, let's install neccessary packages.

Contributors: TJ Bai

[1]:
```
import Pkg
Pkg.activate("CogAI")
Pkg.add(["CSV", "DataFrames"])
# load necessary packages for this problem set
# Note that running this for the first time might take a good 15 mins &ndash;␣
 ↪plan ahead
using Gen
using Plots
using CSV
using DataFrames

include("utils/draw.jl")
```

```
  Activating project at `~/Library/CloudStorage/OneDrive-
JohnsHopkins/Documents/JHU/jhu-2024-spring/cog-ai/ps2/CogAI`
   Resolving package versions…
  No Changes to `~/Library/CloudStorage/OneDrive-
JohnsHopkins/Documents/JHU/jhu-2024-spring/cog-ai/ps2/CogAI/Project.toml`
  No Changes to `~/Library/CloudStorage/OneDrive-
JohnsHopkins/Documents/JHU/jhu-2024-spring/cog-ai/ps2/CogAI/Manifest.toml`
```

```
has_argument_grads (generic function with 33 methods)
```

1.1 Question 1

Nimet has been inspecting 3 colonies of leaf bugs; the average size of the members of each colony is different and denoted as $\theta_1, \theta_2, \theta_3$.

Nimet knows that these 3 colonies are related, sprung off from a common colony last year: That is, she believes the average size of leafbugs in each colony should come from a shared underlying distribution.

She measures the size of the 40 members of each colony, denoted $\vec{x}_1, \vec{x}_2, \vec{x}_3$.

Nimet intuitively arrives at an inference about the average size of the members of each colony, as well as the shared parameter determining the average size of leafbugs in each colony, based on these observations from each colony.

Your task is to formalize Nimet's inference process. You can assume that the average size of leafbugs in a colony (i.e., θs) can be modeled as coming from a Gamma distribution, $Gamma(\alpha, scale = 2)$, shared across the three colonies. Furthermore, you can assume a prior over the `shape` parameter of the Gamma distribution, denoted α, to be a uniform over the range of 1.0 to 15.0. (We assume that we know the scale parameter of the Gamma distribution `scale=2`).

Finally assume that the size measurements come from a normal distribution with known standard deviation $\vec{x}_{i,k} \sim Normal(\theta_i, \sigma = 0.1)$, where $i \in [1, 3]$ indexes the colony and $k \in [1, 40]$ indexes members in that colony.

The following shows a graphical model of the generative model you can use to formalize Nimet's thought process.

1.1.1 Q 1A [3 pts]

Fill in the following to write a Gen generative function of the prior illustrated in the graphical model based on the parametric distributions you decided appropriate.

[2]:
```
@gen function leafbug_colonies()
    scale = 2
     = 0.1

     = ({: } ~ uniform(1, 15))

    theta_one = ({:theta_one} ~ gamma( , scale))
    theta_two = ({:theta_two} ~ gamma( , scale))
    theta_three = ({:theta_three} ~ gamma( , scale))
    thetas = (theta_one, theta_two, theta_three)

    for colony_id in 1:3
        for leafbug_id in 1:40
            {:data => colony_id => leafbug_id => :x} ~␣
 ↪normal(thetas[colony_id],  )
        end
    end
end
```

```
DynamicDSLFunction{Any}(Dict{Symbol, Any}(), Dict{Symbol, Any}(), Type[], false,␣
 ↪Union{Nothing, Some{Any}}[], var"##leafbug_colonies#225", Bool[], false)
```

Now simulate your generative function to ensure that its outputs make sense

[3]:
```
# Run your model `leafbug_colonies` forward
trace = simulate(leafbug_colonies, ())
get_choices(trace)
```

```
: : 6.386116539910839

:theta_three : 15.27273386851144

:theta_two : 7.9929045427673335

:theta_one : 9.355240602906914

:data

    2

        5

            :x : 8.180896127749454

        16

            :x : 7.9105838826210055

        20

            :x : 8.018554038873079

        35

            :x : 8.10318315733256

        12

            :x : 7.873158302696361

        24

            :x : 8.110541453802895

        28

            :x : 7.938759706829921

        8

            :x : 7.958813554981893
```

```
17
    :x : 8.019167773967068
30
    :x : 8.10815457457524
1
    :x : 8.039853623189849
19
    :x : 7.904183099666843
22
    :x : 7.851456670087554
23
    :x : 8.001927802622937
6
    :x : 7.957564989202683
32
    :x : 8.055819467115578
11
    :x : 8.11575021415647
36
    :x : 7.9438564714814905
37
    :x : 7.753842221899983
9
    :x : 7.9172812932673695
```

```
31
    :x : 7.849560762648136
14
    :x : 8.146279921775596
3
    :x : 7.989281635220254
39
    :x : 8.05430702059285
29
    :x : 8.035212835007707
7
    :x : 8.031450907015351
25
    :x : 7.974439929935403
33
    :x : 7.840774152967088
40
    :x : 8.07512649619624
34
    :x : 7.845912669477415
4
    :x : 8.071744051674456
13
    :x : 7.92315521411545
```

```
    15
        :x : 8.03343506285452
    2
        :x : 7.861193282303918
    10
        :x : 7.852478800653663
    18
        :x : 8.047747731202293
    21
        :x : 7.90118162104995
    26
        :x : 8.089197579433824
    27
        :x : 7.761259633177567
    38
         :x : 7.872801802638366
3
    5
        :x : 15.41572927084297
    16
        :x : 15.405684526619899
    20
        :x : 15.163984537906037
    35
```

```
    :x : 15.211067284528518

12

    :x : 15.219562643307624

24

    :x : 15.19413623182105

28

    :x : 15.25102837858265

8

    :x : 15.157852641460394

17

    :x : 15.27188611459831

30

    :x : 15.163445308026038

1

    :x : 15.20967401541675

19

    :x : 15.323906931025155

22

    :x : 15.322033205336087

23

    :x : 15.447379296152416

6

    :x : 15.06552824661105

32
```

```
    :x : 15.378245674227994

11

    :x : 15.216104264445644

36

    :x : 15.188719243990555

37

    :x : 15.168207166689182

9

    :x : 15.386381181601232

31

    :x : 15.21664941739539

14

    :x : 15.267531154919316

3

    :x : 15.238529284500274

39

    :x : 15.326650403477977

29

    :x : 15.23706385156473

7

    :x : 15.212947324426253

25

    :x : 15.206741696497225

33
```

```
    :x : 15.384924558146748

40

    :x : 15.172964143463062

34

    :x : 15.282051303482039

4

    :x : 15.260923296853386

13

    :x : 15.292978419694869

15

    :x : 15.233000747809289

2

    :x : 15.140670060748187

10

    :x : 15.264210341960984

18

    :x : 15.200932195645311

21

    :x : 15.426228340211042

26

    :x : 15.335542270137868

27

    :x : 15.151289970419752

38
```

```
        :x : 15.265381707821028
1
    5
        :x : 9.41490135421852
    16
        :x : 9.337718243771125
    20
        :x : 9.282740152799779
    35
        :x : 9.269251224057939
    12
        :x : 9.281661932473334
    24
        :x : 9.283729254471721
    28
        :x : 9.476553116440614
    8
        :x : 9.251167130481019
    17
        :x : 9.299190737460993
    30
        :x : 9.342509590496647
    1
        :x : 9.366186196878404
```

```
19
    :x : 9.550841491635282
22
    :x : 9.442561283739062
23
    :x : 9.227103393832525
6
    :x : 9.440621317455294
32
    :x : 9.37757360358538
11
    :x : 9.273969946882364
36
    :x : 9.427405204365646
37
    :x : 9.308017845268704
9
    :x : 9.203958831234287
31
    :x : 9.498722761444217
14
    :x : 9.43753991590068
3
    :x : 9.277763586895766
```

```
39
    :x : 9.229611522196373
29
    :x : 9.401201655777097
7
    :x : 9.51460145365914
25
    :x : 9.385671288119815
33
    :x : 9.567578788176712
40
    :x : 9.433943035230802
34
    :x : 9.189215429532421
4
    :x : 9.17057086033496
13
    :x : 9.429107931151178
15
    :x : 9.35930474294365
2
    :x : 9.195629233666253
10
    :x : 9.339973026722618
```

```
        18
            :x : 9.209712108111098
        21
            :x : 9.56612966867396
        26
            :x : 9.571170293430423
        27
            :x : 9.181477963288913
        38
             :x : 9.255988434125612
```

Fill in the following code to load your data from `./data/leafbug_sizes.csv`. Each column is measurements in a colony.

[4]:
```
function make_observations(df::DataFrame)
    constraints = choicemap()

    for colony_id in 1:3
        column = df[!, "colony$colony_id"]
        for leafbug_id in 1:40
            constraints[:data => colony_id => leafbug_id => :x] =␣
 ↪column[leafbug_id]
        end
    end
    constraints
end

df = DataFrame(CSV.File("./data/leafbug_sizes.csv"))
observations = make_observations(df)
```

```
  :data
      2
          5
```

```
    :x : 21.256299943303237

16

    :x : 21.369553855546922

20

    :x : 21.251178256760287

35

    :x : 21.13898163338359

12

    :x : 21.206757582334074

24

    :x : 21.305648418588138

28

    :x : 21.124249614774968

8

    :x : 21.270571265316352

17

    :x : 21.15103714447646

30

    :x : 21.242026934969317

1

    :x : 21.280550797233133

19

    :x : 21.167870648339825

22
```

```
    :x : 21.415392567275152
23
    :x : 21.31267170351833
6
    :x : 21.444309279181613
32
    :x : 21.16340763307204
11
    :x : 21.17076575140692
36
    :x : 21.20782303072503
37
    :x : 21.490522248225464
9
    :x : 21.34635239160401
31
    :x : 21.366652578233328
14
    :x : 21.33728428942978
3
    :x : 21.315046898168635
39
    :x : 21.11654080954066
29
```

```
    :x : 21.202155179705287
7
    :x : 21.33692390293639
25
    :x : 21.259610875884228
33
    :x : 21.334142625932714
40
    :x : 21.44068600948441
34
    :x : 21.256062428338073
4
    :x : 21.17471117540589
13
    :x : 21.292904457362326
15
    :x : 21.30743877622591
2
    :x : 21.279811863483275
10
    :x : 21.354177849946023
18
    :x : 21.248554836646427
21
```

```
        :x : 21.384415530067873

    26

        :x : 21.20760698324941

    27

        :x : 21.30200422876948

    38

        :x : 21.139677503159035

3

    5

        :x : 14.462428425653458

    16

        :x : 14.376158344054238

    20

        :x : 14.527898470606846

    35

        :x : 14.301969003110443

    12

        :x : 14.357463633990962

    24

        :x : 14.327457135313393

    28

        :x : 14.436631820228438

    8

        :x : 14.433220799417642
```

```
17
    :x : 14.474220549312056
30
    :x : 14.522274559246911
1
    :x : 14.520778909998096
19
    :x : 14.347168853120486
22
    :x : 14.347697616630622
23
    :x : 14.632410529563295
6
    :x : 14.5198607727158
32
    :x : 14.228578117995914
11
    :x : 14.34338675096528
36
    :x : 14.493876080765476
37
    :x : 14.545930198954435
9
    :x : 14.079934578903943
```

```
31
    :x : 14.294365880199793
14
    :x : 14.455310782632042
3
    :x : 14.411234338468233
39
    :x : 14.541478055374826
29
    :x : 14.473955022855868
7
    :x : 14.566324422227305
25
    :x : 14.260755967740058
33
    :x : 14.442256693705398
40
    :x : 14.537655808668882
34
    :x : 14.321300534164756
4
    :x : 14.399783971696532
13
    :x : 14.253174152370581
```

```
15
    :x : 14.402998608222044
2
    :x : 14.346464131874358
10
    :x : 14.623867230122473
18
    :x : 14.37531269647273
21
    :x : 14.327172463483583
26
    :x : 14.590292339097198
27
    :x : 14.293602715412835
38
    :x : 14.3408765397632
1
5
    :x : 7.004529706339084
16
    :x : 6.906500744186133
20
    :x : 6.947830966187184
35
```

```
    :x : 7.072430146294981

12

    :x : 6.916783987654109

24

    :x : 6.903614526696025

28

    :x : 6.957121102965965

8

    :x : 6.8632343680017796

17

    :x : 6.773745783969346

30

    :x : 7.1045795611321285

1

    :x : 6.966861067205217

19

    :x : 6.949279547613951

22

    :x : 7.033690808080646

23

    :x : 6.825823355933302

6

    :x : 6.883530496900299

32
```

```
    :x : 6.946071380710548

11

    :x : 6.922722731237116

36

    :x : 6.817557924317697

37

    :x : 6.930337602890122

9

    :x : 6.887814580914907

31

    :x : 6.885623498780866

14

    :x : 7.118939864075353

3

    :x : 7.073226774371389

39

    :x : 6.914983198951852

29

    :x : 7.153966031688359

7

    :x : 7.005738978789077

25

    :x : 6.7806894682886885

33
```

```
    :x : 6.959374747809114

40

    :x : 6.917775226514681

34

    :x : 7.060009839348497

4

    :x : 7.048382758703765

13

    :x : 7.015821666349473

15

    :x : 6.884558781830699

2

    :x : 6.845316851700036

10

    :x : 6.987098072194721

18

    :x : 6.8774875279024545

21

    :x : 6.890629875243033

26

    :x : 6.891840232791859

27

    :x : 6.803498148984123

38
```

```
:x : 6.838459493864463
```

1.1.2 Q 1B [1 pt]

Given your generative model and observations, we will perform importance resampling for posterior inference. Fill in the code to obtain 100 posterior samples each using 1000 resampling steps.

[5]:
```
traces = Vector()
for _ in 1:100
    (trace, _) = importance_resampling(leafbug_colonies, (), observations, 1000)
    push!(traces, trace)
end
```

1.1.3 Q 1C [1 pts]

Fill in the following codeblock to visualize your posterior samples.

First, show the mean log probability of all traces

Then make a plot with 2 subplots

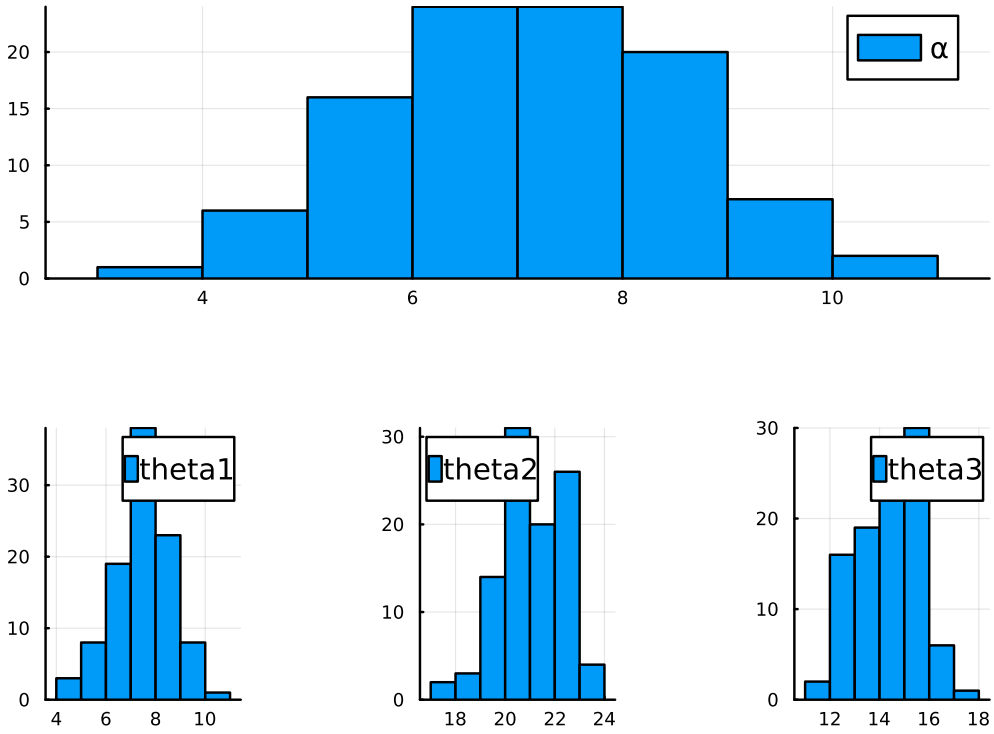
- Subplot 1: Make a histogram plot showing α.
- Subplot 2: Make another histogram plot showing the three θs on the same plot.

[6]:
```
# function for compute the average log probs of all traces
function logmeanexp(scores)
    logsumexp(scores) - log(length(scores))
end;

# function for display all traces
function display_results(traces)
    # compute the average log prob of all traces
    log_probs = [get_score(t) for t in traces]
    println("Average log probability: $(logmeanexp(log_probs))")

    # collect the inferred  s and  s across the chains and plot them.
     s = [t[: ] for t in traces]
    theta_ones = [t[:theta_one] for t in traces]
    theta_twos = [t[:theta_two] for t in traces]
    theta_threes = [t[:theta_three] for t in traces]

    # plot the movements
     _plot = histogram( s, thickness_scaling=3.5, size=(1000, 800), label=" ",␣
↪xtickfontsize=5, ytickfontsize=5)
    theta_ones_plot = histogram(theta_ones, thickness_scaling=3.5, size=(300,␣
↪800), label="theta1", xtickfontsize=5, ytickfontsize=5)
    theta_twos_plot = histogram(theta_twos, thickness_scaling=3.5, size=(300,␣
↪800), label="theta2", xtickfontsize=5, ytickfontsize=5)
```

```
    theta_threes_plot = histogram(theta_threes, thickness_scaling=3.5,␣
 ↪size=(300, 800), label="theta3", xtickfontsize=5, ytickfontsize=5)
    _plot = plot(theta_ones_plot, theta_twos_plot, theta_threes_plot,␣
 ↪layout=(1, 3), size=(1000, 800), xtickfontsize=5, ytickfontsize=5)

    plot( _plot,  _plot, layout=(2, 1), size=(1600, 1200))
end;
```

[7]:
```
# run the display_results function to show the average log prob and inferred␣
 ↪variables
# your code here
display_results(traces)
```

```
Average log probability: -5.428575652848323
```



1.1.4 Q 1D [3 pts]

Now let's implement a MCMC algorithm with a cutomized proposal. In particular,

1. Random work proposal for
2. Then block resimulation for s

Expected result: with 1000 updates (similar amount of compute compared to importance resampling), the mean log prob should be at least around 100

You need to tune the max step size in random walk proposal to reach good inference results.

[8]:
```
@gen function random_walk_proposal(current_trace)
    step_size = 3
      ~ uniform(max(current_trace[:] - step_size, 1), min(current_trace[:] +␣
 ↪step_size, 15))
end;
```

[9]:
```
function MCMC_inference(observations, num_updates)
    (tr, _) = generate(leafbug_colonies, (), observations)

    for iter_id = 1:num_updates
        # random walk MH update for
        (tr, _) = mh(tr, random_walk_proposal, ())

        # block resimulation MN update for  s
        (tr, _) = mh(tr, Gen.select(:theta_one))
        (tr, _) = mh(tr, Gen.select(:theta_two))
        (tr, _) = mh(tr, Gen.select(:theta_three))
    end
    tr
end;
```

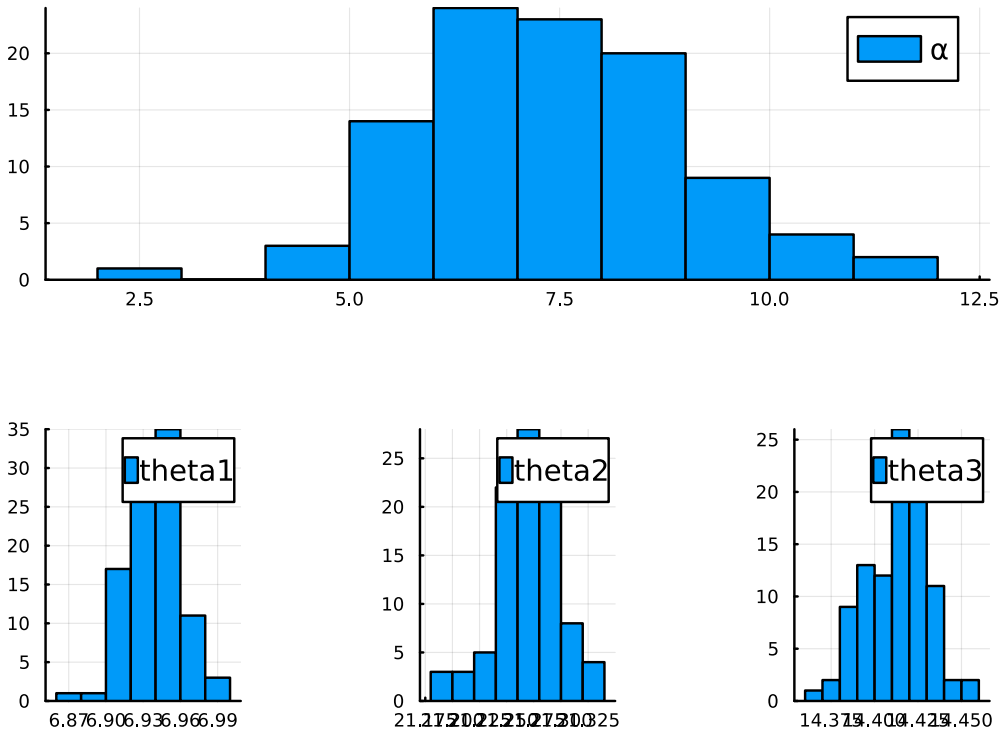
1.1.5 Q 1E [1 pt]

Fill in the code to obtain 100 posterior samples each using 1000 MH updates; then display the results using the display_results() function.

[10]:
```
# Note: this codeblock might take a while to run
traces = Vector()
for _ in 1:100
    push!(traces, MCMC_inference(observations, 1000))
end
```

[11]:
```
display_results(traces)
```

```
Average log probability: 89.68495605512602
```



1.1.6 Q 1F [1 pt]

Compare the results of MCMC and importance resampling. Briefly explain the difference. Write 1 - 2 sentences in the Markdown block below.

Based on the importance sampling histogram, the distributions of theta and alpha have more variance and potential outliers. MCMC sampling is signifcantly better because of the random-walk proposal, which is more informed than random sampling, and block resimulation, which improves resimulation since we only sample a subset of the distribution.

1.2 Question 2

The iron core of our planet conducts electricity, which creates a magnetic field around Earth. This magnetic field not only provides a protective shield against sun's unwanted rays, but also creates a kind of a map that remains relatively constant across time. Many animal species are thought to rely on this magnetic field for wayfinding, from migratory birds across the open skies to fish, reptiles and crustaceans in the deep ocean. For example, sea turtles can use the magnetic field to make plans between points A and B on the globe, and to orient themselves, i.e., to know where they are on the Earth.

Let's assume that the planet's magnetic field can be described using a two-dimensional grid. The sea turtle moves in this gridworld one step at a time, in one of the four cardinal direction: north (n),

south (s), east (e), and west (w). The following struct encodes these mechanics of grid movements and it will come handy as we go along.

[12]:
```
struct Movement
    dx::Real
    dy::Real
end

const N = Movement( 0,  1)
const E = Movement( 1,  0)
const S = Movement( 0, -1)
const W = Movement(-1,  0)
const DIRECTIONS = [N, S, E, W]
```

```
4-element Vector{Movement}:
 Movement(0, 1)
 Movement(0, -1)
 Movement(1, 0)
 Movement(-1, 0)
```

Each cell in this grid emits the *intensity* and *direction* of the magnetic field at that cell. The `intensity` and `direction` of a coordinate at `x` and `y` are defined as:

Your overall goal in this question will be to infer a posterior over the sequence of movements of the sea turtle from a sequence of intensity and direction observations. We will build up to that.

1.2.1 Q 2A [4 pts]

Imagine that suddenly, this sea turtle finds itself in the middle of an oceanic storm. The storm is such that:

- The sea turtle knows where it is at the beginning of the storm (time step `k=0`), including its `x` and `y`
- At each time step, its movement – a single step in a cardinal direction: north `n`, south `s`, east `e` or west `w` – are dictated by the waves and turbulence of the ocean (not controlled by the sea turtle). Assume that these dynamics are random – a multinomial distribution with equal weight on each direction:

$$p(m_t|m_{t-1}) = p(m_t) = Multinomial([n, s, e, w])$$

- At each time step, the sea turtle observes noisy magnetic field measurements (because of the storm, it cannot additionally rely on vision or smell)

$$p(intensity_t) \sim Normal(x + y, \sigma)$$

$$p(direction_t) \sim Normal(|x - y|, \sigma)$$

  where $\sigma = 0.1$

(You might be able to relate to the experience of our sea turtle friend if you can remember the last time you were on a Ferris Wheel with your eyes closed. In such a scenario, when you rely

just on your vestibular system to tell your pose in space, you are likely to experience all sorts of hallucinated backward flips.)

Your task is write a generative model of this process using Gen's generative functions and the generative function combinator `Unfold`. You will write a generative function (a temporal kernel) called `ferriswheel_kernel`, and input it to the `Unfold` combinator to create a temporal generative model called `ferriswheel`.

Assume that the initial state of the sea turtle is provided to `ferriswheel` as a global variable (not modeled as a random variable).

Start with implementing a Julia `struct` to represent and modify the state information at each time step; the struct should include the following three entities. (For each variable, you must indicate its type.)

- `movement`: current movement
- `x`: current coordinate in the east-west axis
- `y`: current coordinate in the north-south axis

[13]:
```
struct Field
    movement::Movement
    x::Real
    y::Real
end
```

Using the definitions above, fill in the following `intensity` function to compute the intensity of a `Field`.

[14]:
```
function intensity(field::Field)
    field.x + field.y
end
```

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

Using the definitions above, fill in the following `direction` function to compute the direction of a `Field`.

[15]:
```
function direction(field::Field)
   abs(field.x - field.y)
end
```

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

We need a way to update the `x` and `y` entries of a `Field` based on a `Movement`. Using Julia's support for multiple dispatch and overriding functions and primitives, we provide a redefined addition, `Base.:+`, which applies a `Movement` to the `x` and `y` entries of a Field, returning a new `Field` with the addition as well as the input `Movement`. This will come in handy as we go along.

[16]:
```
function Base.:+(field::Field, movement::Movement)
    return Field(
        movement,
        field.x + movement.dx,
```

```
        field.y + movement.dy,
    )
end
```

Fill in the following code block to complete the definition of ferrishweel_kernel and create a function called chain using Gen's Unfold combinator and this kernel.

[17]:
```
@gen function ferriswheel_kernel(k::Int, curr_field::Field)
    # observation noise of the mangetic field (intensity and direction)
     = 0.1

    # Draw a movement
    draw_move = {:draw_move} ~ categorical([0.25, 0.25, 0.25, 0.25])
    movement = DIRECTIONS[draw_move]

    next_field = curr_field + movement

    # observe noisy intensity/direction measurements
    {:obs_intensity} ~ normal(intensity(next_field),  )
    {:obs_direction} ~ normal(direction(next_field),  )

    # Return the updated field
    next_field
end
```

```
DynamicDSLFunction{Any}(Dict{Symbol, Any}(), Dict{Symbol, Any}(), Type[Int64,␣
↪Field], false, Union{Nothing, Some{Any}}[nothing, nothing],␣
↪var"##ferriswheel_kernel#235", Bool[0, 0], false)
```

Fill in the following codeblock to create the temporal generative model `ferrishweel`

[26]:
```
@gen function ferriswheel(K::Int)
    global init_field
    trajectory ~ Unfold(ferriswheel_kernel)(K, init_field)
end
```

```
DynamicDSLFunction{Any}(Dict{Symbol, Any}(), Dict{Symbol, Any}(), Type[Int64],␣
↪false, Union{Nothing, Some{Any}}[nothing], var"##ferriswheel#240", Bool[0],␣
↪false)
```

1.2.2 Q 1B [1 pt]

Draw a sequence of 10 movements from your generative model. The initial state is provided (`init_field`).

Use the `get_choices` and `get_retval` functions to display the random choices and return values associated with the trace you simulated of the generative function.

[27]:
```
# Start at coordinates (N, 3, 3)
init_field = Field(N, 3, 3)
```

```
# Run your model forward with 10 movements, name your variable `trace`
# your code here
trace = simulate(ferriswheel, (10,))

# Display the choices from `trace`
# your code here
get_choices(trace)
```

```
  :trajectory

      1

          :obs_direction : 0.9961282863727762

          :obs_intensity : 5.190624166012839

          :draw_move : 2

      2

          :obs_direction : 1.9838404282239697

          :obs_intensity : 5.795375316313746

          :draw_move : 3

      3

          :obs_direction : 1.0773478469578643

          :obs_intensity : 6.979425356421778

          :draw_move : 1

      4

          :obs_direction : -0.027733728666021435

          :obs_intensity : 7.941599992449401

          :draw_move : 1

      5

          :obs_direction : 1.1839059027905325
```

```
    :obs_intensity : 8.974514568849838
    :draw_move : 3
6
    :obs_direction : -0.04260893163201411
    :obs_intensity : 7.971313862182246
    :draw_move : 4
7
    :obs_direction : 1.074379192516852
    :obs_intensity : 9.014130541251724
    :draw_move : 1
8
    :obs_direction : 0.01155586763250342
    :obs_intensity : 9.864977701696361
    :draw_move : 3
9
    :obs_direction : 1.031312503151113
    :obs_intensity : 8.98502441306235
    :draw_move : 2
10
    :obs_direction : 0.10589806106153582
    :obs_intensity : 10.167826717444362
    :draw_move : 1
```

[28]:
```
# Get the return values for your `trace
# your code here
```

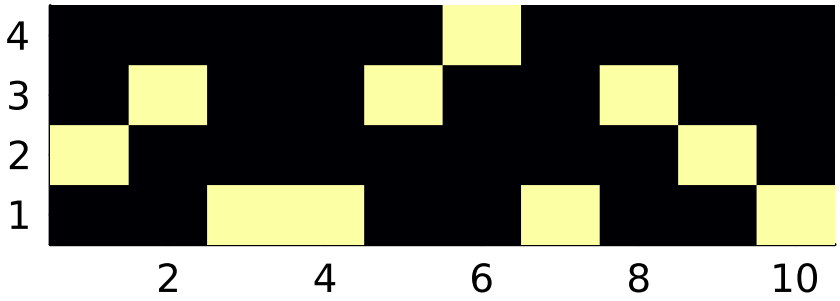
```
get_retval(trace)
```

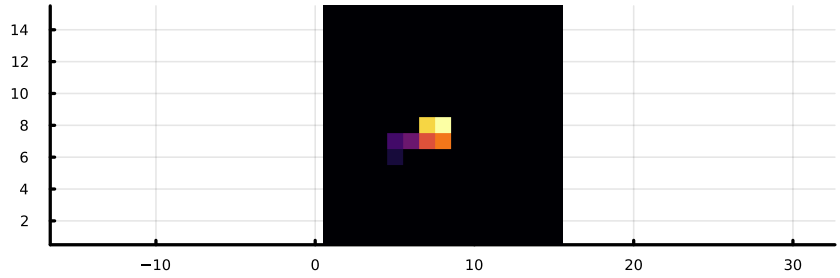
```
Persistent{Any}[Field(Movement(0, -1), 3, 2), Field(Movement(1, 0), 4, 2),␣
↪Field(Movement(0, 1), 4, 3), Field(Movement(0, 1), 4, 4), Field(Movement(1,␣
↪0), 5, 4), Field(Movement(-1, 0), 4, 4), Field(Movement(0, 1), 4, 5),␣
↪Field(Movement(1, 0), 5, 5), Field(Movement(0, -1), 5, 4), Field(Movement(0,␣
↪1), 5, 5)]
```

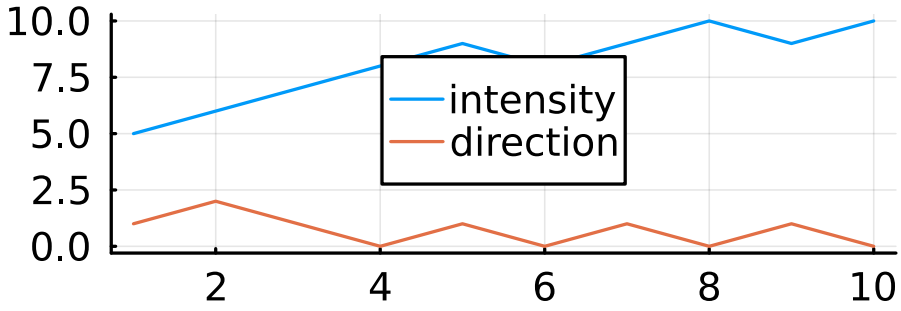
Execute the next code block to visualize your sample. (Review this visualization code, but nothing to fill in.) You will see that it plots a binary heatmap of which movements occured (n, s, e, w), a gray scale heatmap showing the trajectory, and a line plot of how intensities and directions changed throughout the sequence.

[29]:
```
# a helper function to visualize things
# we visualize the movements and
#   the predicted intensity and direction
#   values according to the coordinates of
#   visited cells
function visualize(trace; title="")
    choices = get_choices(trace)
    fields  = get_retval(trace)

    # get the movements and coordinates
    ms = [field.movement for field in fields]
    xs = [field.x for field in fields]
    ys = [field.y for field in fields]

    # predicted intensities
    intensities = intensity.(fields)

    # predicted directions
    directions = direction.(fields)

    # create a binary matrix of movements (4 x length(ms))
    binary_movements = falses(4, length(ms))
    for (index, movement) in enumerate(ms)
        movement_order = findfirst(isequal(movement), DIRECTIONS)
        binary_movements[movement_order, index] = true
    end

    # create a gray scale matrix of coordinates, [-2, 12] x [-2, 12]
    # the brighter the color, the more recent the step is
    binary_coordinates = zeros(15, 15)
    n = length(xs)
    for t = 1:n
        binary_coordinates[xs[t] + 3, ys[t] + 3] = t / n
    end
```

```
    # plot the movements
    p1 = plot(
        binary_movements,
        seriestype=:heatmap,
        legend=false,
        thickness_scaling=3.5,
        title=title,
        titlefont=5,
    )
    p2 = plot(
        binary_coordinates,
        seriestype=:heatmap,
        legend=false,
        thickness_scaling=3.5,
        aspect_ratio = :equal,
        size=(800, 800),
        xtickfontsize=3,
        ytickfontsize=3,
    )
    # plot intensities and directions
    p3 = plot(
        collect(1:length(ms)),
        [intensities, directions],
        thickness_scaling=3.5,
        labels=["intensity" "direction"]
    )
    plot(p1, p2, p3, layout=(3,1), legend=:inside, size=(1200, 1600))
end

visualize(trace)
```

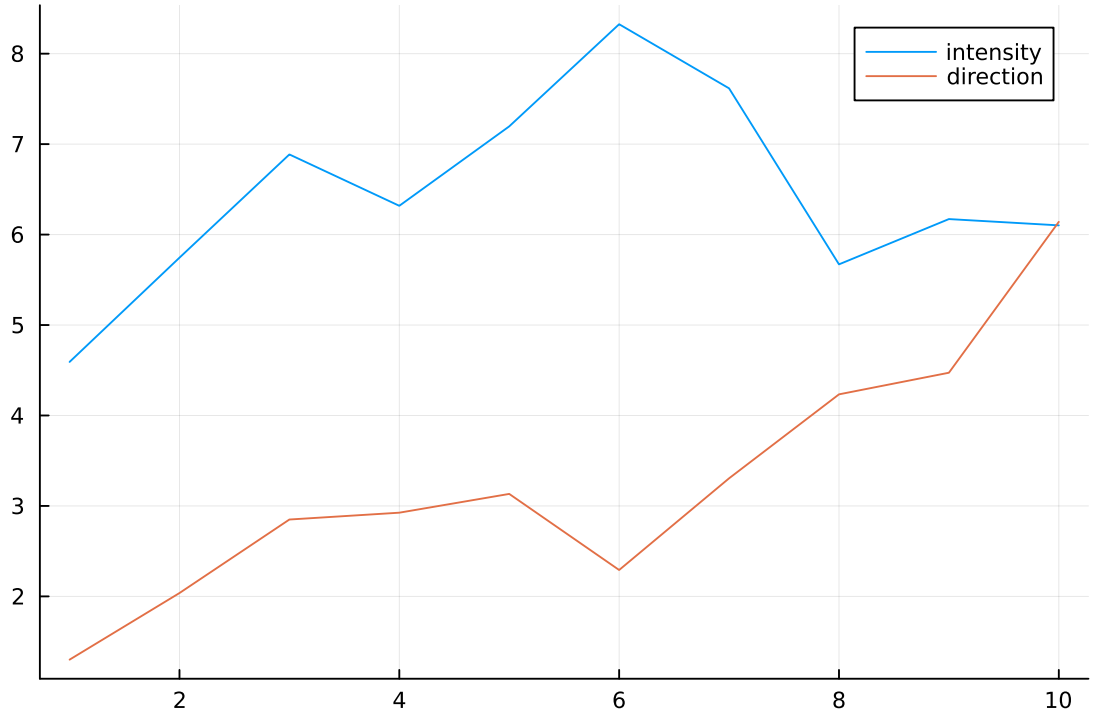






Execute the following codeblock to load and visualize the observed intensity and directions

[30]:
```
# load observations (sensory features)
obs_fields = DataFrame(CSV.File("./data/observed_fields.csv"))

# visualize (just run the following code)
plot(
    collect(1:size(obs_fields, 1)),
    [obs_fields[!, :intensity], obs_fields[!, :direction]],
    labels=["intensity" "direction"]
)
```



1.2.3 Q 1C [4 pts]

In Gen, write a particle filtering algorithm to infer a posterior distribution over movements given the intensity and direction measurements in `observations`.

[31]:
```
function particle_filter(num_particles::Int, obs_fields, num_samples::Int)
    #inititial observation
    init_obs = choicemap(
        (:trajectory => 0 => :obs_intensity, obs_fields[1, :intensity]),
        (:trajectory => 0 => :obs_direction, obs_fields[1, :direction]),
    )
```

```
    state = initialize_particle_filter(ferriswheel, (0,), init_obs,␣
 ↪num_particles)

    for (idx, obs_field) in enumerate(eachrow(obs_fields))
        # Resample
        maybe_resample!(state, ess_threshold=num_particles/2)

        # load observations of this time step
        obs = choicemap(
            (:trajectory => idx => :obs_intensity, obs_field[:intensity]),
            (:trajectory => idx => :obs_direction, obs_field[:direction]),
        )

        # Re-weight by the likelihood
        particle_filter_step!(state, (idx,), (UnknownChange(),), obs)
    end

    return sample_unweighted_traces(state, num_samples)
end
```

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

Now call this particle filter inference procedure with 1000 particles and return 100 samples

[32]:
```
pf_traces = particle_filter(1000, obs_fields, 100);
```

1.2.4 Q 1D [1 pts]

The following codeblock visualizes these 100 posterior samples you just computer, one after the other; each frame shows the inferred sequence of movements and trajectory according to the posterior sample (top) and the predicted intensities and directions. View this animation and explain what it reveals about the posterior distribution. Write 1-2 sentences in the Markdown block after the animation.

[33]:
```
viz = Plots.@animate for (trace_id, trace) in enumerate(pf_traces)
    visualize(trace;title="Sample $trace_id / 100")
end
gif(viz, fps=1)
```

```
 Info: Saved animation to
/var/folders/kl/wbnrhqz91z3ccc0kctsjcbvh0000gn/T/jl_xAPoDo7IJ0.gif
 @ Plots /Users/avnukala/.julia/packages/Plots/a3u1v/src/animation.jl:156
```

```
Plots.AnimatedGif("/var/folders/kl/wbnrhqz91z3ccc0kctsjcbvh0000gn/T/
 ↪jl_xAPoDo7IJ0.gif")
```

From the animation, we notice the posterior distribution is mostly composed of two trajectories that appear as mirrors of each other. From this observation, we draw the conclusion that there are two probable trajectories of the turtle. In our computation of intensity and direction, the return values are not dependent on x and y individually, meaning that if we were to swap the values of

x and y, we could get the same intensity and direction. This explains why we see two probable trajectories, which, based on the "swap theory", are equally likely.