Solution 3: Inference on continuous spaces

Set the seed for reproducibility

```
set.seed(1)
```

Q.1: functions on the unit interval

Monte Carlo estimate

```
mc_estimate = function(f){
  mean(f(runif(10000)))
}
```

Estimate the value of the Gaussian integral

```
my_complicated_function = function(x){
  exp(-x^2)
}
cat(paste0("estimate = ", round(mc_estimate(my_complicated_function),3)))
```

```
estimate = 0.746
```

```
cat(paste0("true = ", 0.7468241328124270253994674361319))
```

```
true = 0.746824132812427
```

Estimate the value of the sin(cos(sin(x))) integral

```
another_complicated_function = function(x){
  sin(cos(sin(x)))
}
cat(paste0("estimate = ", round(mc_estimate(another_complicated_function),3)))
```

```
estimate = 0.759
```

Compare to the output given by `R`'s (deterministic) numerical integration routine (**note: this is not required for marking**)

```
integrate(another_complicated_function, 0, 1)
```

```
0.7593064 with absolute error < 8.4e-15
```

Q.2: implementing SNIS for simPPLe

Note: We provide Julia and Python/JAX implementations of the following answers, as well. The alternative solutions are attached to the end of this PDF.

Start by loading `distr` and the scaffolding code

```
suppressPackageStartupMessages(library(distr))

## Utilities to make the distr library a bit nicer to use

p <- function(distribution, realization) {
  d(distribution)(realization) # return the PMF or density
}

Bern = function(probability_to_get_one) {
  DiscreteDistribution(
    supp = 0:1, prob = c(1-probability_to_get_one, probability_to_get_one)
  )
}

## Key functions called by simPPLe programs
```

```
# Use simulate(distribution) for unobserved random variables
simulate <- function(distribution) {
  r(distribution)(1) # sample once from the given distribution
}

# Use observe(realization, distribution) for observed random variables
observe = function(realization, distribution) {
  # `<<-` lets us modify variables that live in the global scope from inside a function
  weight <<- weight * p(distribution, realization)
}
```

Q.2.3: coin example

```
fair_coin = Bern(0.5)
coin_toss = simulate(fair_coin)
outcome_probability = p(fair_coin, coin_toss)
cat(paste("The outcome is", coin_toss, "and its probability is", outcome_probability))
```

```
The outcome is 1 and its probability is 0.5
```

Q.2.4: implement posterior

We use these constructs to write the rest of the PPL. The first step is to define a global variable that will accumulate the likelihood of a given program

```
weight = 1.0
```

Now we complete the skeleton for the `posterior` function

```
posterior = function(ppl_function, number_of_iterations) {
  numerator = 0.0
  denominator = 0.0
  for (i in 1:number_of_iterations) {
    weight <<- 1.0       # reset the weight accumulator
    val = ppl_function() # run the forward simulator and store the query value
    numerator   = numerator + weight*val
    denominator = denominator + weight
  }
  return(numerator/denominator)
}
```

Q.2.5: Test the PPL

Load the code provided for the coin bag example

```
coin_flips = rep(0, 4) # "dataset" of four identical coin flips = (0, 0, 0, 0)

# simPPLe's description of our "bag of coins" example
my_first_probabilistic_program = function() {

  # Similar to forward sampling, but use 'observe' when the variable is observed
  coin_index = simulate(DiscreteDistribution(supp = 0:2))
  for (i in seq_along(coin_flips)) {
    prob_heads = coin_index/2
    observe(coin_flips[i], Bern(1 - prob_heads))
  }

  # return the test function g(x, y)
  return(ifelse(coin_index == 1, 1, 0))
}
```

Run the PPL code

```
posterior(my_first_probabilistic_program, 10000)
```

```
[1] 0.05979932
```

Compare to the true value: 0.05882353.

Alternative solution to Q2 with Julia

Let's first define our main functions, including `observe`, and `posterior`.

```
using Pkg
Pkg.activate("solutions/")
using Distributions
using SplittableRandoms

const current_log_likelihood = Ref(0.0)
```

```
function observe(observation, distribution)
    current_log_likelihood[] += logpdf(distribution, observation)
end


function posterior(rng, probabilistic_program, n_particles)
    samples = Float64[]
    log_weights = Float64[]

    for i in 1:n_particles
        current_log_likelihood[] = 0.0
        push!(samples, probabilistic_program(rng))
        push!(log_weights, current_log_likelihood[])
    end

    return sum(samples .* exponentiate_normalize(log_weights))
end


### Utils
function exponentiate_normalize(vector)
    exponentiated = exp.(vector .- maximum(vector))
    return exponentiated / sum(exponentiated)
end
```

Next, we define our probabilistic program:

```
const coin_flips = [0, 0, 0, 0]

function my_first_probabilistic_program(rng)
    coin_index = rand(rng, DiscreteUniform(0, 2))
    for i in 1:length(coin_flips)
        observe(coin_flips[i], Bernoulli(coin_index / 2))
    end
    return coin_index == 1 ? 1 : 0
end
```

Finally, we can verify the quality of the approximation

```
rng = SplittableRandom(1)
println("        MC: ", posterior(rng, my_first_probabilistic_program, 1_000_000))
```

Compare to the true value: 0.05882353.

Alternative solution to Q2 with Python/JAX

```python
import jax
import jax.numpy as jnp
from jax.random import PRNGKey, categorical, uniform
from jax.scipy.stats import bernoulli
```

```python
# Global variable to accumulate the log weight
log_weight = 0.0

def reset_log_weight():
    global log_weight
    log_weight = 0.0

def update_log_weight(delta):
    global log_weight
    log_weight += delta

def simulate(distribution_fn, key):
    """Simulate a value from a given distribution function."""
    return distribution_fn(key)

def observe(obs, log_prob_fn, distribution_params):
    """Update the (log) weight based on the observed value."""
    delta = log_prob_fn(obs, **distribution_params)
    update_log_weight(delta)

def posterior(probabilistic_program, n_particles, key):
    """Compute the posterior using importance sampling."""
    keys = jax.random.split(key, n_particles)

    def single_run(key):
        reset_log_weight()
        result = probabilistic_program(key)
        return result, log_weight

    results, log_weights = jax.vmap(single_run)(keys) # Vectorize particles to make it effic
    log_weights = log_weights - jnp.max(log_weights)  # Normalize for numerical stability
    weights = jnp.exp(log_weights)
    normalized_weights = weights / jnp.sum(weights)

    return jnp.sum(results * normalized_weights)
```

```python
def my_first_probabilistic_program(key):
    """Example probabilistic program for the coin bag problem."""
    subkey, key = jax.random.split(key)
    coin_index = categorical(subkey, jnp.array([1/3, 1/3, 1/3]))  # Uniform over {0, 1, 2}

    coin_flips = jnp.array([0, 0, 0, 0])  # Dataset of observed flips

    for flip in coin_flips:
        subkey, key = jax.random.split(key)
        prob_heads = coin_index / 2.0
        observe(flip, bernoulli.logpmf, {'p': prob_heads})

    return jnp.where(coin_index == 1, 1.0, 0.0)
```

```python
# Run the posterior calculation
key = PRNGKey(42)
n_particles = 10000
result = posterior(my_first_probabilistic_program, n_particles, key)


print(f"Estimated posterior: {result}")
```

```
Estimated posterior: 0.059422388672828674
```

```python
# Compare to the true value: 0.05882353
```