The conditionals are distributed as Bernoulli with parameter $p = P(O_i | S_i)$ and $p = P(S_{j+1} | S_j)$.

%%
%% Score the probability of a set of observations given an assignment of the latent states
%%
%function log_prob = log_score_observation(observations, states, coin_weights)
log_prob = [];

for i=1:length(observations);
    if observations(i) == 0;
        if states(i) == 0;
            log_prob = [ log_prob log(1-coin_weights(1)) ];
        else
            log_prob = [ log_prob log(1-coin_weights(2)) ];
        end
    else
        if states(i) == 0;
            log_prob = [ log_prob log(coin_weights(1)) ];
        else
            log_prob = [ log_prob log(coin_weights(2)) ];
        end
    end
end
end
assert(log_score_observation(1,1,[.3 .7]) == log(.7))
assert(log_score_observation(1,0,[.3 .7]) == log(.3))
assert(log_score_observation(0,0,[.3 .7]) == log(.7))

1B-2

%%
%% Score the probability of a state given the previous state
%%
function log_prob = log_score_state_pair(prev_state, state, prob_switch)
    if prev_state == state;
        log_prob = log(1-prob_switch);
    else
        log_prob = log(prob_switch);
    end
end

assert(log_score_state_pair(1,1,.15)==log(.85))
assert(log_score_state_pair(1,0,.15)==log(.15))

1B-3

function log_prob = log_score_joint(state, state_obs, prev_state, next_state, ...
    prob_switch, coin_weights)
    if state == 1;
        not_state = 0;
    else
        not_state = 1;
    end

    if state == prev_state;
        p_state_given_prev_state = 1-prob_switch;
    end

else
    p_state_given_prev_state = prob_switch;
end

if state == next_state;
    p_next_state_given_state = 1-prob_switch;
else
    p_next_state_given_state = prob_switch;
end

if state_obs == 0;
    if state == 0;
        p_state_obs_given_state = 1-coin_weights(1);
    else
        p_state_obs_given_state = 1-coin_weights(2);
    end
else
    if state == 0;
        p_state_obs_given_state = coin_weights(1);
    else
        p_state_obs_given_state = coin_weights(2);
    end
end

if not_state == prev_state;
    not_p_state_given_prev_state = 1-prob_switch;
else
    not_p_state_given_prev_state = prob_switch;
end

if not_state == next_state;
    not_p_next_state_given_state = 1-prob_switch;
else
    not_p_next_state_given_state = prob_switch;
end

if state_obs == 0;
    if not_state == 0;
        not_p_state_obs_given_state = 1-coin_weights(1);
    else
        not_p_state_obs_given_state = 1-coin_weights(2);
    end
else
    if not_state == 0;
        not_p_state_obs_given_state = coin_weights(1);
    else
        not_p_state_obs_given_state = coin_weights(2);
    end
end
if not_state == 0;
    not_p_state_obs_given_state = coin_weights(1);
else
    not_p_state_obs_given_state = coin_weights(2);
end
end

not_log_prob = not_p_state_given_prev_state*not_p_next_state_given_state...
   *not_p_state_obs_given_state;

numerator = p_state_given_prev_state*p_next_state_given_state...
   *p_state_obs_given_state;

log_prob = log(numerator/(numerator+not_log_prob));
end

assert(log_score_joint(1,1,1,1,0.15,[0.3 0.7])==log(0.9868))
assert(log_score_joint(0,1,1,1,0.15,[0.3 0.7])==log(0.0132))

1C-i

%%
%% Sample assignment of initial state
%%

function reassignment = sample_init_assignment(state, state_obs, next_state, ...
    prob_switch, coin_weights);

if next_state == 1;
    p_S1_is1_given_S2 = 1-prob_switch;
    p_S1_is0_given_S2 = prob_switch;
else
    p_S1_is1_given_S2 = prob_switch;
    p_S1_is0_given_S2 = 1-prob_switch;
end
if state_obs == 0;
function reassignment = sample_last_assignment(state, state_obs, prev_state, prob_switch, coin_weights)

if prev_state == 1;
    p_SN_given_prev_state = (1-prob_switch);
    p_SN_is0_given_prev_state = (prob_switch);
else
    p_SN_given_prev_state = (prob_switch);
    p_SN_is0_given_prev_state = (1 - prob_switch);
end

if state_obs == 0
    p_ON_given_SN = (1-coin_weights(2));
    p_ON_given_SN_is0 = (1-coin_weights(1));
else
    p_ON_given_SN = (coin_weights(2));
    p_ON_given_SN_is0 = (coin_weights(1));
end
p_SN_equals_1 = (p_SN_given_prev_state * p_ON_given_SN)/(p_SN_given_prev_state * p_ON_given_SN+ p_SN_is0_given_prev_state*p_ON_given_SN_is0);

reassignment = binornd(1,p_SN_equals_1);

end

1C-iii

%% Takes a state, its emission, previous state and next state, and
%% returns a reassignment.
%%
function reassignment = sample_reassignment(state, state_obs, prev_state, ...
    next_state, prob_switch, coin_weights)
if state_obs == 1;
    is1_p_ON_given_SN = coin_weights(2);
else
    is1_p_ON_given_SN = 1-coin_weights(2);
end

if next_state == 1;
    is1_p_SNplus1_given_SN = 1-prob_switch;
else
    is1_p_SNplus1_given_SN = prob_switch;
end

if prev_state == 1;
    is1_p_SN_given_SNminus1 = 1-prob_switch;
else
    is1_p_SN_given_SNminus1 = prob_switch;
end

if state_obs == 1;
    is0_p_ON_given_SN = coin_weights(1);
else
    is0_p_ON_given_SN = 1-coin_weights(1);
end

if next_state == 1;
    is0_p_SNplus1_given_SN = prob_switch;
else
    is0_p_SNplus1_given_SN = 1-prob_switch;
end

if prev_state == 1;
    is0_p_SN_given_SNminus1 = prob_switch;
else
    is0_p_SN_given_SNminus1 = 1-prob_switch;
end

is1_p_SN_given_SNplus1_SNminus1_ON = (is1_p_ON_given_SN*is1_p_SNplus1_given_SN*...
    is1_p_SN_given_SNminus1)/(is1_p_ON_given_SN*is1_p_SNplus1_given_SN*...
    is1_p_SN_given_SNminus1)+(is0_p_ON_given_SN*is0_p_SNplus1_given_SN*...
    is0_p_SN_given_SNminus1));

reassignment = binornd(1,is1_p_SN_given_SNplus1_SNminus1_ON);
end

1D

%% A sampler for solving the dealer die switching problem.

function samples = hmm_sampler(observations, num_iters, coin_weights, prob_switch)
    num_states = length(observations);
    init_states = zeros(1,num_states);
    states = init_states;
    samples = []; 

    samples(1,1) = sample_init_assignment(states(1),...
        observations(1),states(2),prob_switch,coin_weights);
    for j=2:(num_states-1);
        samples(j,1) = sample_reassignment(states(j),...
observations(j), states(j-1), states(j+1),...
prob_switch, coin_weights);
end
samples(num_states,1) = sample_last_assignment(
states(num_states), observations(num_states), states(num_states-1),...
prob_switch, coin_weights);

for i=2:num_iters;
samples(1,i) = sample_init_assignment(samples(1,(i-1)),...
observations(1), samples(2,(i-1)), prob_switch, coin_weights);
for j=2:(num_states-1);
samples(j,i) = sample_reassignment(samples(j,(i-1)),...
observations(j), samples(j-1,(i-1)), samples(j+1,(i-1)),...
prob_switch, coin_weights);
end
samples(num_states,i) = sample_last_assignment(
samples(num_states,(i-1)), observations(num_states), samples(num_states-1,(i-1)),...
prob_switch, coin_weights);
end

samples = samples';

% For each iteration... draw samples, store them in 'samples' variable
end
Coin weights = [0.30, 0.70], $p_{\text{switch}} = 0.15$, number of samples = 5000
In the case of the more severely biased coins \((p(H)=[0.1, 0.9])\) The predictions maintain the same shape but become more categorical. The model is more certain about which coins is generating the flip at any given time. In the case of the higher switching probability \((p_{\text{switch}}=0.65)\), the models predictions are compressed to a range closer to 0.5 but amplified within that range, in that flips that have previously been attributed to the same coin are now attributed to different coins. Generally certainty for any given flip decreases. It is hard to assess the errors made by the model without knowing which coins actually generated the flips in the observed sequences. However, assuming that the sequence was generated in a \((p(H)=[0.3, 0.7], p_{\text{switch}}=0.15)\) scenario, it seems like the model with high \(p_{\text{switch}}\) might be more error prone, while the model assuming a stronger bias might still do fairly well albeit being overly confident.
2A-iii

Random Sequence 1

- Coin weights = [0.30, 0.70], \( p_{\text{switch}} = 0.15 \), number of samples = 5000
- Coin weights = [0.10, 0.90], \( p_{\text{switch}} = 0.15 \), number of samples = 5000
- Coin weights = [0.30, 0.70], \( p_{\text{switch}} = 0.65 \), number of samples = 5000

Random Sequence 2

- Coin weights = [0.30, 0.70], \( p_{\text{switch}} = 0.15 \), number of samples = 5000
- Coin weights = [0.10, 0.90], \( p_{\text{switch}} = 0.15 \), number of samples = 5000
- Coin weights = [0.30, 0.70], \( p_{\text{switch}} = 0.65 \), number of samples = 5000

Random Sequence 3

- Coin weights = [0.30, 0.70], \( p_{\text{switch}} = 0.15 \), number of samples = 5000
- Coin weights = [0.10, 0.90], \( p_{\text{switch}} = 0.15 \), number of samples = 5000
- Coin weights = [0.30, 0.70], \( p_{\text{switch}} = 0.65 \), number of samples = 5000
Because the generating model assumes fairly biased coins it will always be prone to overfit when randomly generated sequences contain a large number of adjacent flips with identical outcomes. In Random sequence 1, for example, coins 23 to 29 all came up heads. The models best guess here is to assume a head biased coin. However, when outcomes alternate, the generating model predicts neither coin with great confidence (Random sequence 2). Since the model is looking for patterns, it will find them if they happen to exist in the randomly generated sequence, but if the surface pattern is approximately random the model will not wrongly predict one of the coins with great confidence. When the switching probability is high, however, random patterns are fitted as originating from different coins. A model that assumes more extreme coin weights assigns its guesses higher confidence and thus also overfits more than the generating model.

2B

Sequence 1 – Individual Subjects

1 All graphs plotting subject responses/author’s intuitions have “number of samples = 1” in the header, even though all models were sampled 5000 times. This is because the intuitions are only a single sample and has no consequences for the results reported here.
Sequence 2 – Individual Subjects

Sequence 3 – Individual Subjects
Sequence 4 – Individual Subjects

If I hadn’t been Bayesian before, by the time MATLAB has generated these plots I would have been completely convinced. We asked 6 subjects, all philosophers and thus extremely error prone due to their
ability to overlogicize simple tasks. Nonetheless the subjects’ average ratings are fit extremely well by the models predictions.

2C

pSWITCH=.15, coin_weights=[.3 .7]

pSWITCH=.15, coin_weights=[.1 .9]
pSWITCH=.65, coin_weights=[.3 .7]

The model with high p_switch is extremely error prone (15/32 correct). It therefore seems that this parameter setting is the intrinsically hardest. The strongly biased coin model (27/32 correct) and the generating model (28/32 correct) both do very well. Judging on these test runs, it seems that there is no intrinsic difference between in difficulty for this model to accommodate more extreme coin biases.
My Intuitions.

My own intuitions concur with the models prediction in the case of the generating model and the parsing model with the more biased coins. I am also horribly off for the scenario with high switching probability. It seems that the models predictive accuracy is constrained to scenarios with low switching probability, which makes intuitive sense as coin-flip patterns are generally less informative under these conditions.
The HMM approach seems to capture human behavior extremely well. The generating models predictions are uncertain when coin flips alternate, which also matches human intuitions well. The model may make errors when confronted with an almost random sequence, with a slight bias for e.g. heads, if this sequence indeed originated from a tail-biased coin. This again seems to be an error humans would make as well. One problem of the model that could potentially be improved is its strictly local attention span. In sequence 3 that we tested on human subjects flips 25-31 come up heads, while flip 32 alone comes up tails. No human decided that there was a coin switch involved here, potentially because humans evaluated the entire sequence from 25-32 as a single chunk. The model could then probably be improved by conditioning not solely on the immediately following and preceding coins but a number of neighboring coins to either side. For this the structure of the model would have to be changed such that for each state there exist connections between a number of preceding and following states.