Appendix A: Code and supplementary figures for simulation and policy analyis

2019-06-05

Contents

Setup

The POMDP calculations presented here are performed using the `sarsop` R package, which we have written to provide a general purpose implementation of the SARSOP algorithm of Kurniawati, Hsu, and Lee (2008). The version used in this analysis can be installed from GitHub using the command: `devtools::install_github("boettiger-lab/sarsop@0.5.0")`.

```
#devtools::install_github("boettiger-lab/sarsop@0.5.0")
library(sarsop)        # Our main POMDP package
library(tidyverse)     # for munging and plotting
library(parallel)
library(gridExtra)
```

```
tic()
options(mc.cores=parallel::detectCores())
log_dir <- "../data/appendixA" # Store the computed solution files here
```

Definitions: Population Model and Utility function

```
r <- 0.75
K <- 1

## Unlike classic Gordon-Schaefer, this assumes harvest
## occurs right after assessment, before recruitment
f <- function(x, h){
    s <- pmax(x-h, 0)
    s + s * r * (1 - s / K)
}
```

A simply utility (reward) function: a fixed price per fish with no cost to fishing. Alternate models can easily be considered, we focus on this simple utility throughout as this is the choice typically focused on in the optimal control literature, implicitly assumed by the basic MSY model, and does not change the general results. Note that setting a smaller discount will take longer to converge to smooth POMDP solution, but result in `S_star` closer to the simple `B_MSY`.

```
reward_fn <- function(x,h) pmin(x,h)
discount <- 0.95
```

Calculating MSY, TAC, and CE

The approach below uses a generic optimization routine to find the stock size at which the maximum growth rate is achieved. (Note that this also depends on the discount rate since future profits are worth proportionally less than current profits).

```
## A generic routine to find stock size (x) which maximizes
## growth rate (f(x,0) - x, where x_t+1 = f(x_t))
S_star <- optimize(function(x) -f(x,0) + x / discount,
                   c(0, 2*K))$minimum
```

Since we `observe` -> `harvest` -> `recruit`, we would observe the stock at its pre-harvest size, $X_t \sim B_{MSY} + H_{MSY}$.

```
#B_MSY <- S_star     # recruit first, as in classic Graham-Schaefer
B_MSY <- f(S_star,0) # harvest first, we observe the population at B_MSY + h
```

```
#MSY <- f(B_MSY,0) - B_MSY  # recruit first
MSY <- B_MSY - S_star  # harvest first

F_MSY <- MSY / B_MSY
F_TAC = 0.8 * F_MSY
```

As a basic reference point, simulate these three policies in a purely deterministic world. Unlike later simulations, here we consider all states an actions exactly (that is, within floating point precision). Later, states and actions are limited to a discrete set, so solutions can depend on resolution and extent of that discretization.

```
MSY_policy <- function(x) F_MSY * x
TAC_policy <- function(x) F_TAC * x

## ASSUMES harvest takes place *before* recruitment, f(x-h), not after.
escapement_policy <- function(x) pmax(x - S_star,0)

x0 <- K/6
Tmax = 100
do_det_sim <- function(policy, f, x0, Tmax){
    action <- state <- obs <- as.numeric(rep(NA,Tmax))
    state[1] <- x0

    for(t in 1:(Tmax-1)){
      action[t] <- policy(state[t])
      obs[t] <- state[t] - action[t]  # we observe->harvest->recruitment
      state[t+1] <- f(state[t], action[t])
    }
    data.frame(time = 1:Tmax, state, action, obs)
  }

det_sims <-
  list(MSY = MSY_policy,
       TAC = TAC_policy,
       CE = escapement_policy) %>%
  map_df(do_det_sim,
         f, x0, Tmax,
         .id = "method")

write_csv(det_sims, "../data/appendixA/det_sims.csv")
```

With no stochasticity, MSY leads to the same long-term stock size as under the constant escapement rule, but takes longer to get there. This level is essentially B_{MSY}, though because the model considered here implements events in the order: *observe, harvest, recruit*; rather than *observe, recruit, harvest*, we see the stock at the pre-harvest size of $B_{MSY} + H_{MSY}$ (that is, $K/2 + rK/4$). More conservative rules, such as a harvest set to 80% of MSY result in faster recovery of the stock than under MSY, but slower than under constant escapement. Due to the reduced maximum harvest rate, such rules lead to stock returning to a value higher than B_{MSY}.

```
det_sims %>%
  ggplot(aes(time, state, col=method)) +
  geom_line(lwd=1) +
  coord_cartesian(ylim = c(0, 1)) +
  theme(legend.position = "bottom") +
  ylab("Mean biomass")
```

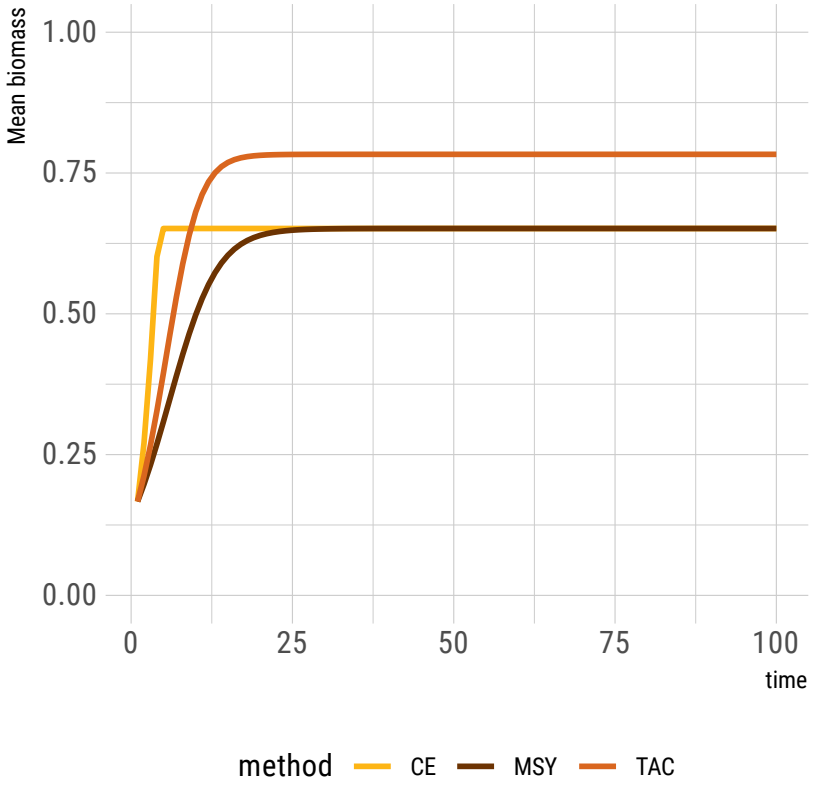

Figure S1: MSY, TAC, and CE policies under deterministic simulation. MSY and CE converge to the same standing biomass long term, but CE reaches this level more quickly. TAC always harvests less than MSY, resulting in a higher standing biomass but lower long-term yeild. Note that POMDP is identical to CE when no uncertainty is present.

Solving the POMDP problem

Introduce a discrete grid

```
## Discretize space
states <- seq(0,2, length=100)
actions <- states
observations <- states
```

We compute the above policies on this grid for later comparison.

```
index <- function(x, grid) map_int(x, ~ which.min(abs(.x - grid)))


policies <- data.frame(
#  CE = index(pmax(f(states,0) - S_star,0), actions), # obs,recruit,harv, f(x_t) - h_t
  CE = index(escapement_policy(states),     actions), # obs,harv,recruit, f(x_t - h_t)
  MSY = index(MSY_policy(states),               actions),
  TAC = index(TAC_policy(states),               actions))
```

Define the POMDP matrices

We compute POMDP matrices for a range of `sigma_g` and `sigma_m` values. (We consider `sigma_g = 0.02` rather than precisely zero here to avoid a degeneracy in the discretized transition matrix. Calculations for the precise deterministic solution can be solved without discretization as shown above, and match the results of this nearly-zero noise level in the discrete simulation, as expected.)

```
meta <- expand.grid(sigma_g = c(0.02, 0.1, 0.15),
                    sigma_m = c(0, 0.1, 0.15),
                    stringsAsFactors = FALSE) %>%
        mutate(scenario  = as.character(1:length(sigma_m)))
meta
```

```
  sigma_g sigma_m scenario
1    0.02    0.00        1
2    0.10    0.00        2
3    0.15    0.00        3
4    0.02    0.10        4
5    0.10    0.10        5
6    0.15    0.10        6
7    0.02    0.15        7
8    0.10    0.15        8
9    0.15    0.15        9
```

```
models <-
  parallel::mclapply(1:dim(meta)[1],
           function(i){
  fisheries_matrices(
  states = states,
  actions = actions,
  observed_states = observations,
  reward_fn = reward_fn,
  f = f,
```

```
  sigma_g = meta[i,"sigma_g"][[1]],
  sigma_m = meta[i,"sigma_m"][[1]],
  noise = "normal")
})
```

Compute the POMDP solution

The POMDP solution is represented by a collection of alpha-vectors and values, returned in a `*.policyx` file. Each scenario (parameter combination of `sigma_g`, `sigma_m`, and so forth) results in a separate solution file.

Because this solution is computationally somewhat intensive, be sure to have ~ 4 GB RAM per core if running the 9 models in parallel. Alternately, readers can skip the evaluation of this code chunk and read the cached solution from the `policyx` file using the `*_from_log` functions that follow:

```
dir.create(log_dir, FALSE)
system.time(
  alphas <-
    parallel::mclapply(1:length(models),
    function(i){
      log_data <- data.frame(model = "gs",
                             r = r,
                             K = K,
                             sigma_g = meta[i,"sigma_g"][[1]],
                             sigma_m = meta[i,"sigma_m"][[1]],
                             noise = "normal",
                             scenario = meta[i, "scenario"][[1]])

      sarsop(models[[i]]$transition,
             models[[i]]$observation,
             models[[i]]$reward,
             discount = discount,
             precision = 1e-8,
             timeout = 15000,
             log_dir = log_dir,
             log_data = log_data)
    })
)
```

```
     user    system   elapsed
32319.426   112.756 16559.557
```

Simulations

Having calculated the POMDP solution in terms of these `alpha` vectors, we can easily compute the POMDP policy for any given prior and update the prior given further observations according to Bayes rule. We assume a uniform prior belief at the start of the POMDP simulation. These tasks are performed by the `sim_pomdp` function. Simulating the static strategies (CE, MSY, TAC) is easier, since the policy is uniquely determined by the observed state; we simply need to simulate the stochastic growth and process and measurement with error at each time step an then compute the corresponding policy. The `sim_pomdp` function performs such static simulations when given a pre-specified policy instead of a set of `alpha` vectors.

Simulating the static policies under uncertainty

```
set.seed(12345)

Tmax <- 100
x0 <- which.min(abs(K/6 - states))
reps <- 100
static_sims <-
 map_dfr(models, function(m){
            do_sim <- function(policy) sim_pomdp(
                        m$transition, m$observation, m$reward, discount,
                        x0 = x0, Tmax = Tmax, policy = policy, reps = reps)$df
            map_dfr(policies, do_sim, .id = "method")
          }, .id = "scenario")
```

Simulating the POMDP policies under uncertainty

```
set.seed(12345)

unif_prior <- rep(1, length(states)) / length(states)
pomdp_sims <-
  map2_dfr(models, alphas, function(.x, .y){
             sim_pomdp(.x$transition, .x$observation, .x$reward, discount,
                       unif_prior, x0 = x0, Tmax = Tmax, alpha = .y,
                       reps = reps)$df %>%
              mutate(method = "POMDP") # include a column labeling method
           },
           .id = "scenario")

write_csv(pomdp_sims, file.path(log_dir, "pomdp_sims.csv"))
```

We then combine the resulting data frames, transition (mutate) the units from grid indices (`state`, `action` $\in 1...N$) to continuous values (`states`, `actions` $\in [0, 2]$), and select necessary columns before writing the data out to a file.

```
pomdp_sims <- read_csv(file.path(log_dir, "pomdp_sims.csv")) %>%
  mutate(scenario = as.character(scenario))


sims <- bind_rows(static_sims, pomdp_sims) %>%
    left_join(meta) %>%  ## include scenario information (sigmas; etc)
    mutate(state = states[state], action = actions[action]) %>%
    select(time, state, rep, method, sigma_m, sigma_g, value)

write_csv(sims, file.path(log_dir, "sims.csv"))
```

Figure S2: Full simulation plots

We the results varying over different noise intensities, `sigma_g`, and `sigma_m`. Note that Figure 1 of the main text shows only the case of `sigma_g = 0.15`, `sigma_m = 0.1` and omits MSY to simplify the presentation.

```
sims <- read_csv(file.path(log_dir, "sims.csv"), col_types = "inicnnn")

sims  %>%
```

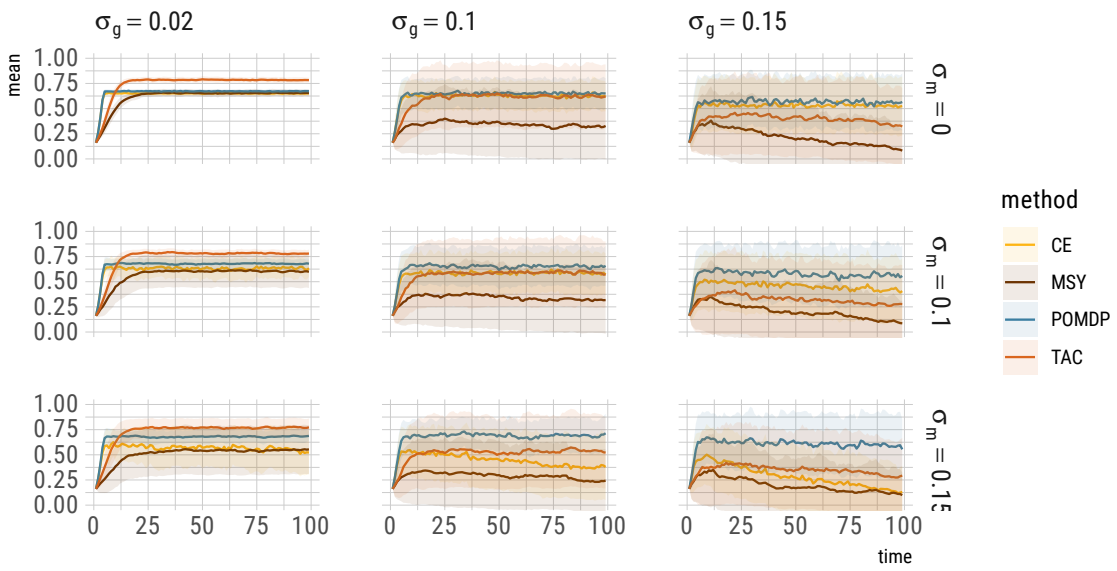

Figure S2: Simulations as in the main text, Figure 1, shown across varying growth noise levels and including MSY strategy for comparison.

```
  select(time, state, rep, method, sigma_m, sigma_g) %>%
  group_by(time, method, sigma_m, sigma_g) %>%
  summarise(mean = mean(state), sd = sd(state)) %>%
  ggplot(aes(time, mean, col=method, fill=method)) +
  geom_line() +
  geom_ribbon(aes(ymax = mean + sd, ymin = mean-sd), col = NA, alpha = 0.1) +
  facet_grid(sigma_m ~ sigma_g,
             labeller = label_bquote(sigma[m] == .(sigma_m),
                                     sigma[g] == .(sigma_g))) +
  coord_cartesian(ylim = c(0, 1))
```

Figure S3: Economic value

Economic value is shown under varying noise levels under each strategy.

```
sims %>%
  select(time, value, rep, method, sigma_m, sigma_g) %>%
  filter(sigma_g %in% c("0.1", "0.15")) %>%
  group_by(rep, method, sigma_m, sigma_g) %>%
  summarise(npv = sum(value)) %>%
  group_by(method, sigma_m, sigma_g) %>%
  summarise(net_value = mean(npv), se = sd(npv) / mean(npv)) %>%
  ungroup() %>%

  ggplot(aes(method, net_value, ymin=net_value-se, ymax=net_value+se, fill=method)) +
  geom_bar(position=position_dodge(), stat="identity") +
  geom_errorbar(size=.3,width=.2,
```

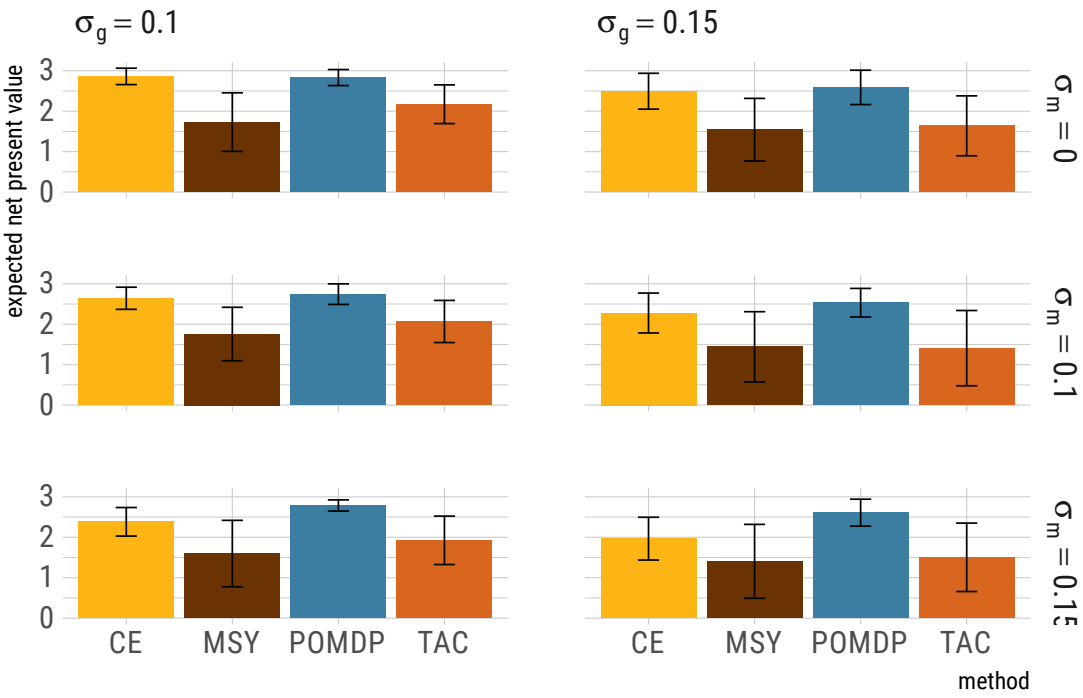

Figure S3: Economic value as in Fig 2 main text, shown across different levels of environmental noise, including MSY strategy.

```
              position=position_dodge(.9)) +
  facet_grid(sigma_m~sigma_g,
             labeller = label_bquote(sigma[m] == .(sigma_m),
                                     sigma[g] == .(sigma_g))) +
  theme(legend.position = "none") +
  ylab("expected net present value")
```

POMDP simulations overestimating measurement uncertainty

While the POMDP approach requires an estimate of the measurement error, the precise distribution of measurement errors will itself be unknown in most cases. However, the POMDP approach is quite robust to overestimation of the measurement error. As an extreme example of this, we consider the case where the POMDP solution assumes the largest measurement error level considered here, $\sigma_m = 0.15$, while performing simulations in which measurements occur without error:

```
set.seed(12345)
true <- 3 # sigma_g = 0.15, sigma_m = 0
source(system.file("examples/pomdp_overestimates.R", package="sarsop")) # portable example code
pomdp_overest_sims <-
  map2_dfr(models, alphas, function(.x, .y){
    pomdp_overestimates(transition = .x$transition,
            model_observation = .x$observation,
```

```
            reward = .x$reward,
            discount = discount,
            true_observation = models[[true]]$observation,
            x0 = x0,
            Tmax = Tmax,
            alpha = .y,
            reps = reps)$df %>%
              mutate(method = "POMDP") # include a column labeling method
          },
          .id = "scenario"
  )
write_csv(pomdp_overest_sims, file.path(log_dir, "pomdp_overest_sims.csv"))
```

Combine the POMDP simulations results from over-estimating measurement error with the previous results from the static policies for comparison:

```
pomdp_overest_sims <- read_csv(file.path(log_dir, "pomdp_overest_sims.csv")) %>%
  mutate(scenario = as.character(scenario))

overest <-
bind_rows(static_sims, pomdp_overest_sims) %>%
  left_join(meta, by = "scenario") %>%  ## include scenario information (sigmas; etc)
  mutate(state = states[state], action = actions[action]) %>%
  select(time, state, rep, method, sigma_m, sigma_g, value) %>%
  group_by(time, method, sigma_m, sigma_g) %>%
  summarise(mean = mean(state), sd = sd(state)) %>%
  filter(sigma_g == "0.15") %>%
  ungroup()

overest %>% filter(method != "POMDP", sigma_m == "0")  %>%
  bind_rows(
  overest %>% filter(method == "POMDP", sigma_m == "0.15")) %>%
  select(-sigma_m, -sigma_g) %>%
  write_csv(file.path(log_dir, "overest_sims.csv"))
```

```
read_csv(file.path(log_dir, "overest_sims.csv")) %>%
  ggplot(aes(time, mean, col=method, fill=method)) +
  geom_line(lwd = 1) +
  geom_ribbon(aes(ymax = mean + sd, ymin = mean-sd), col = NA, alpha = 0.1) +
  coord_cartesian(ylim = c(0, 1))
```

Policy plots

Static policies

First we gather the static policies (CE, TAC, MSY) into a table and convert into terms of harvest and escapement.

```
policy_table <- tibble(state = 1:length(states)) %>%
  bind_cols(policies) %>%
  gather(policy, harvest, -state) %>%
  mutate(harvest = actions[harvest], state = states[state]) %>%
  mutate(escapement = state - harvest)
```

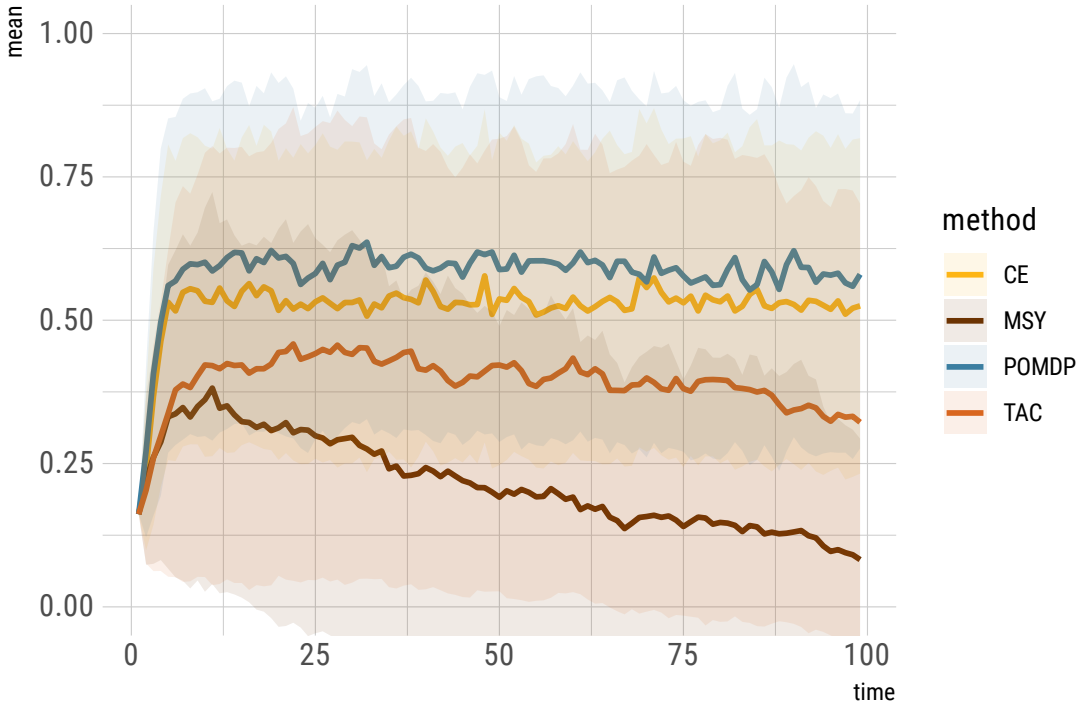

Figure S4: over-estimating measurement uncertainty using POMDP, as in Fig 3 main text.

TAC harvests are always smaller than MSY harvests, but unlike the deterministic optimal solution, TAC and MSY solutions never go to zero. The discrete grid makes these appear slightly stepped.

Policies under uncertainty: S = D

Because Reed (1979) has proven that the optimal escapement in the stochastic case is equivalent to the deterministic case, $S = D$, we have simply relied on the deterministic calculation for constant escapement. We can instead calculate the optimally policy for such a stochastic but fully observed Markov Decision Process (MDP) directly given the discritized model transition matrix using Stochastic Dynamic Programming (SDP). Reed (1979) essentially tells us that for small growth noise (satisfying or approximately satisfying Reed's self-sustaining condition) that the stochastic optimal policy is equal to the deterministic optimal policy. We can confirm this numerically as follows.

First we grab the transition matrix we have already defined when `sigma_g = 0.1`, and then solve the SDP using the `MDPtoolbox` package:

```
i <- meta %>% filter(sigma_g == 0.1, sigma_m ==0) %>%
  pull(scenario) %>% as.integer()
m <- models[[i]]
mdp <- MDPtoolbox::mdp_policy_iteration(m$transition, m$reward, discount)
```

We plot the resulting policy in comparison with the other static policies, which shows that even for such a large noise we get a stochastic constant escapement equal to that of of the deterministic CE calculation:

```
bind_rows(policy_table,
  data_frame(state = states,
             policy = "SDP",
             harvest = actions[mdp$policy],
             escapement = states - actions[mdp$policy])) %>%
  ggplot(aes(state, escapement, col=policy)) +
  geom_line(lwd=1)  +
  coord_cartesian(xlim = c(0,K), ylim = c(0, .8)) +
  labs(caption = "SDP solution when sigma_g = 0.1")
```

Repeating this larger stochastic growth `sigma_g = 0.15`, shows only slight deviation from CE ($S > D$, as Reed (1979) proves for sufficiently large noise to violate the self-sustaining criterion. Technically since our noise model in normally distributed none of our populations are self-sustaining for infinite time, but that deviation is quite small, as seen here. Note also the solution is still a constant-escapement type.)

```
i <- meta %>% filter(sigma_g == 0.15, sigma_m ==0) %>%
  pull(scenario) %>% as.integer()
m <- models[[i]]
mdp <- MDPtoolbox::mdp_policy_iteration(m$transition, m$reward, discount)

bind_rows(policy_table,
  data_frame(state = states,
             policy = "SDP",
             harvest = actions[mdp$policy],
             escapement = states - actions[mdp$policy])) %>%
  ggplot(aes(state, escapement, col=policy)) +
  geom_line(lwd=1)  +
  coord_cartesian(xlim = c(0,K), ylim = c(0, .8)) +
  labs(caption = "SDP solution when sigma_g = 0.15")
```

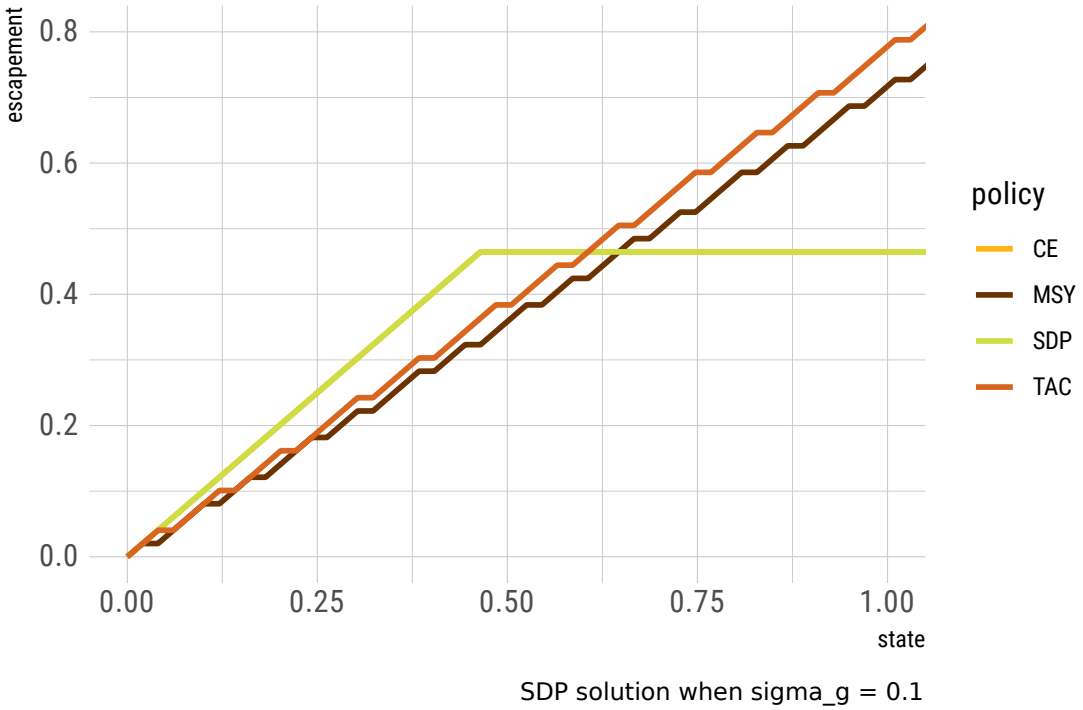

Figure S5: The CE policy is identical to the solution found by SDP under very small noise. MSY and TAC can be seen to correspond to lower escapements at small stock sizes, since they continue to harvest stocks that are below B_{MSY}.

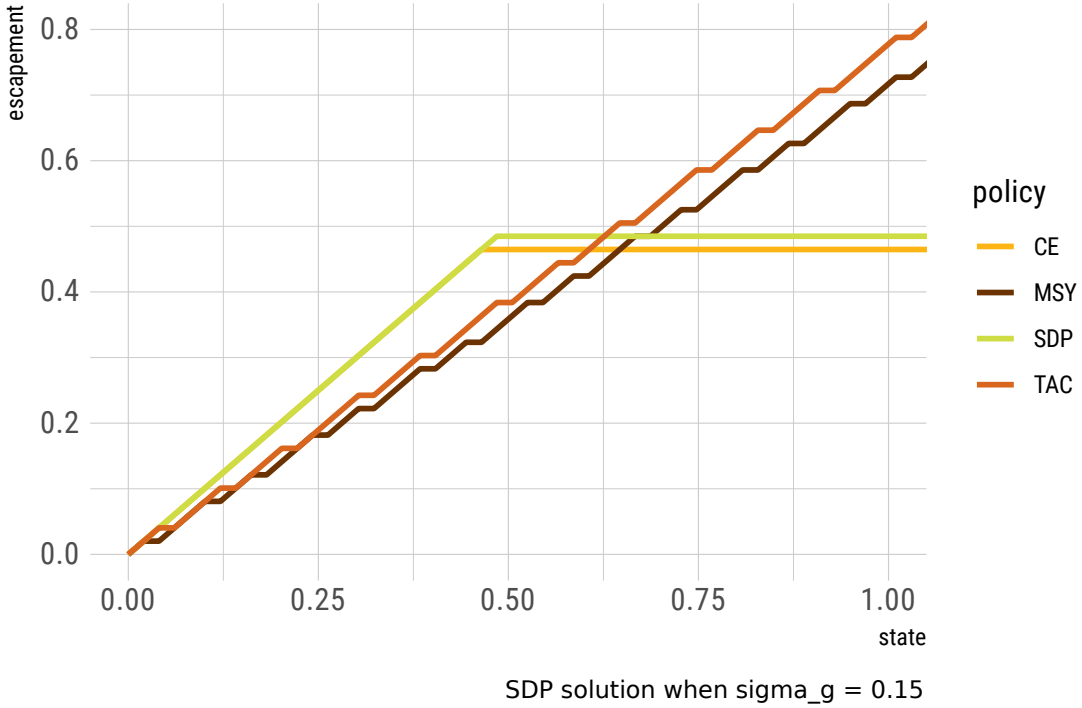

Figure S6: For large growth noise, the escapement under SDP is still constant but very slightly higher than the (deterministic) CE solution, $S > D$.

Comparing POMDP Policies

The comparison of POMDP policy is yet more complicated, but the POMDP policy cannot be expressed merely in terms of a target harvest (or escapement) level given an estimation of the stock size (state). The optimal solution for the partially observed system must also reflect all prior observations of the system, not merely the most recent observation, as the system is not Markovian in the observed state variable. We summarize this history as a prior "belief" about the state, which is updated according to Bayes rule after each observation. Note that Sethi et al. (2005) plots solutions with measurement uncertainty without any reference to the prior. Bayes Law tells us that without a statement of prior belief over the true state, $P(x)$ we cannot go from the measurement model that defines the probability of observing y given a true state$x$; $P(y|x)$, into the probability of the true state given the measurement, $P(x|y)$. Further, this prior $P(x)$ should be updated with respect to each subsequent observation, we cannot simply assume it is a fixed value as Sethi et al. (2005). This explains their counter-intuitive finding that increased uncertainty should result in increased harvest rates with increased uncertainty.

Let us look at the POMDP solutions under various priors focusing on the case of moderate uncertainty, $\sigma_g = 0.15$ and $\sigma_m = 0.1$. (Recall we have already solved the POMDP solution for this model in the simulations above, as defined by the `alpha` vectors, so we can quickly load that solution now.)

```
i <- meta %>%
  filter(sigma_g == 0.15, sigma_m == 0.1) %>%
  pull(scenario) %>% as.integer()
m <- models[[i]]
alpha <- alphas[[i]] # we need the corresponding alpha vectors
```

We will consider what the POMDP solution looks like under a few different prior beliefs. A uniform prior sounds like a conservative assumption, but it is not: it puts significantly more weight on improbably large stock values than other priors. (Loading the α vectors from our POMDP solution computed earlier, we can then compute a POMDP given these α, the matrices for transition, observation, and reward, and the prior we are using)

```
unif_prior = rep(1, length(states)) / length(states) # initial belief
unif <- compute_policy(alpha, m$transition, m$observation, m$reward,  unif_prior)
```

For a representative set of priors, we will consider priors centered at the target B_{MSY} size (or S^* in the language of Reed), at half B_{MSY} (i.e. $K/4$), and at $\frac{3}{4}K$, each with a standard deviation of $\sigma_m = 0.1$ (i.e. the uncertainty around a single observation of a stock at that size.)

```
prior_star <- m$observation[, which.min(abs(states - S_star)),1]
prior_low <- m$observation[,which.min(abs(states - 0.25 * K)),1]
prior_high <- m$observation[,which.min(abs(states - 0.75 * K)),1]

star <- compute_policy(alpha, m$transition, m$observation, m$reward,  prior_star)
low <- compute_policy(alpha, m$transition, m$observation, m$reward,  prior_low)
high <- compute_policy(alpha, m$transition, m$observation, m$reward,  prior_high)
```

We gather these solutions into a single data frame and convert from grid indices to continuous values

```
df <- unif
df$medium <- star$policy
df$low <- low$policy
df$high <- high$policy

pomdp_policies <- df %>%
  select(state, uniform = policy, low, medium, high) %>%
  gather(policy, harvest, -state) %>%
  mutate(state = states[state],
         harvest = actions[harvest]) %>%
```

```
  mutate(escapement = state-harvest) %>%
  select(state, policy, harvest, escapement)

all_policies <- bind_rows(policy_table, pomdp_policies)
write_csv(all_policies, file.path(log_dir, "all_policies.csv"))
```

Plot these policies in terms of both Harvest and Escapement:

```
recode_policies <-
  all_policies %>%
  filter(policy %in% c("CE", "MSY", "TAC", "low", "high")) %>%
  mutate(policy =
    fct_recode(policy,
               "POMDP: low prior" = "low",
               "POMDP: high prior" = "high"))
```

```
p1 <- recode_policies %>%
  ggplot(aes(state, escapement, col = policy)) +
  geom_line(lwd=1) +
  coord_cartesian(xlim = c(0,1), ylim=c(0,.8)) +
  theme(legend.position = "none")

p2 <- recode_policies %>%
  ggplot(aes(state, harvest, col = policy)) +
  geom_line(lwd=1) +
  coord_cartesian(xlim = c(0,1), ylim=c(0,.6)) +
  theme(legend.position = "bottom")
```

```
p3 <- read_csv(file.path(log_dir, "priors.csv")) %>%
ggplot(aes(state, prior, col = policy)) +
  geom_line(lwd=1) +
  coord_cartesian(xlim = c(0,1)) +
  theme(legend.position = "none")

gridExtra::grid.arrange(p1, p2, p3, ncol=1)
```

Evolution of the POMDP prior belief

To get a sense of how the prior belief (and consequently the POMDP policy) is continually adapting over the course of a simulation (unlike the static policies), it is instructive to examine a single simulation rather than the ensemble of replicates considered above.

```
set.seed(12345)
Tmax <- 20
sim <- sim_pomdp(m$transition, m$observation, m$reward, discount,
                 unif_prior, x0 = 10, Tmax = Tmax, alpha = alpha)
```

```
sim$df %>%
  select(-value) %>%
  mutate(state = states[state], action = actions[action], obs = observations[obs]) %>%
  gather(variable, stock, -time) %>%
  ggplot(aes(time, stock, color = variable)) +
  geom_line(lwd=1)  + scale_color_manual(values = unname(colors))
```

Here we plot the evolution of the prior belief in state over the course of the above simulation in increasingly darker curves. Note that the the beginning belief is uniform, but quickly tightens into a distribution centered

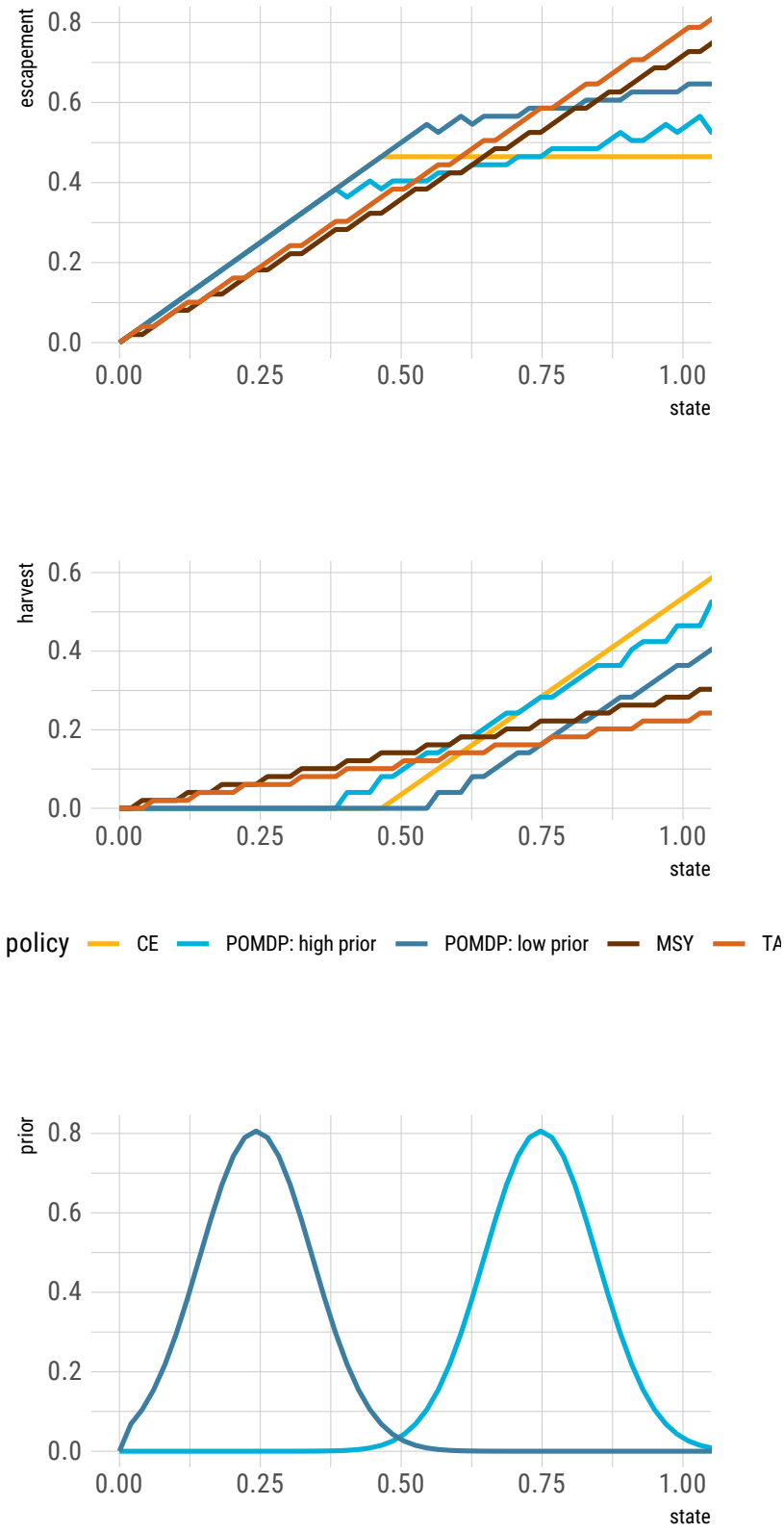

Figure S7: policy plots, as in manuscript, figure 4. Here we include a comparison to MSY as well.

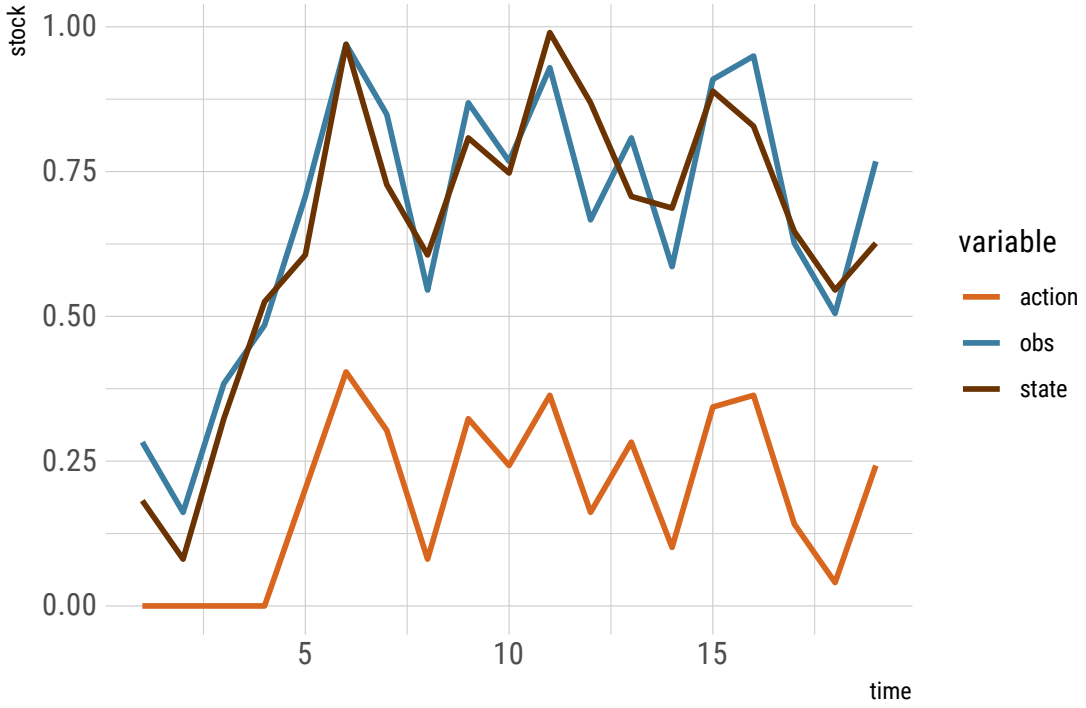

Figure S8: Observation, True state, and Harvest quota over the course of a single simulation under POMDP management.

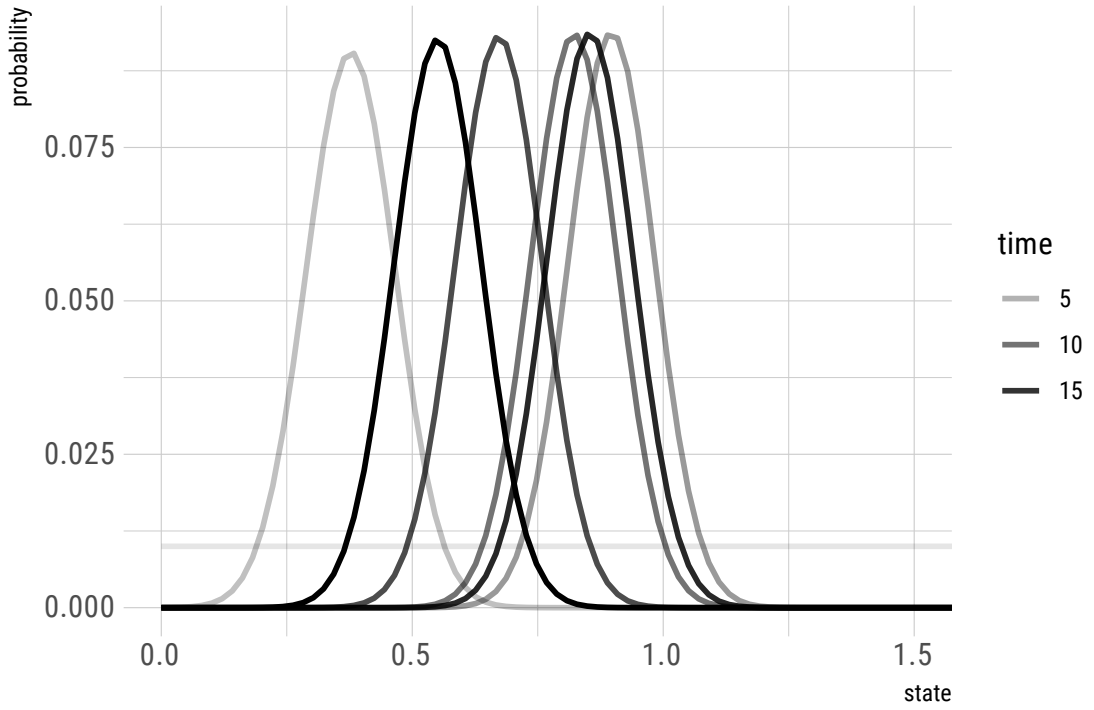

Figure S9: The evolution of the POMDP prior belief distribution over the course of the simulation.

around the region of the most recent observation. Note that this belief distribution never converges: as the true state constantly changes in due to stochasticity in population recruitment, so too does the belief distribution constantly change in response to new observations of that true state.

```
sim$state_posterior %>%
  data.frame(time = 1:Tmax) %>%
  filter(time %in% seq(1,Tmax, by = 3)) %>%
  gather(state, probability, -time, factor_key =TRUE) %>%
  mutate(state = states[as.numeric(state)]) %>%
  ggplot(aes(state, probability, group = time, alpha = time)) +
  geom_line(lwd = 1) +
  coord_cartesian(xlim = c(0,1.5))
```

System Information

Total runtime:

```
toc()
```

```
16597.136 sec elapsed
```

Hardware:

```
system2("grep", c("MemTotal", "/proc/meminfo"), stdout = TRUE)
```

```
[1] "MemTotal:       32938360 kB"
```

```
system2('grep', '"model name" /proc/cpuinfo', stdout = TRUE)
```

```
 [1] "model name\t: Intel Xeon E312xx (Sandy Bridge)"
 [2] "model name\t: Intel Xeon E312xx (Sandy Bridge)"
 [3] "model name\t: Intel Xeon E312xx (Sandy Bridge)"
 [4] "model name\t: Intel Xeon E312xx (Sandy Bridge)"
 [5] "model name\t: Intel Xeon E312xx (Sandy Bridge)"
 [6] "model name\t: Intel Xeon E312xx (Sandy Bridge)"
 [7] "model name\t: Intel Xeon E312xx (Sandy Bridge)"
 [8] "model name\t: Intel Xeon E312xx (Sandy Bridge)"
 [9] "model name\t: Intel Xeon E312xx (Sandy Bridge)"
[10] "model name\t: Intel Xeon E312xx (Sandy Bridge)"
[11] "model name\t: Intel Xeon E312xx (Sandy Bridge)"
[12] "model name\t: Intel Xeon E312xx (Sandy Bridge)"
[13] "model name\t: Intel Xeon E312xx (Sandy Bridge)"
[14] "model name\t: Intel Xeon E312xx (Sandy Bridge)"
[15] "model name\t: Intel Xeon E312xx (Sandy Bridge)"
[16] "model name\t: Intel Xeon E312xx (Sandy Bridge)"
```

Software:

```
devtools::session_info()
```

```
- Session info -------------------------------------------------------------------
 setting  value
 version  R version 3.6.0 (2019-04-26)
 os       Debian GNU/Linux 9 (stretch)
 system   x86_64, linux-gnu
 ui       RStudio
 language (EN)
 collate  en_US.UTF-8
 ctype    en_US.UTF-8
 tz       Etc/UTC
 date     2019-06-05

- Packages -----------------------------------------------------------------------
 package     * version date       lib source
 assertthat    0.2.1   2019-03-21 [1] CRAN (R 3.6.0)
 backports     1.1.4   2019-04-10 [1] CRAN (R 3.6.0)
 broom         0.5.2   2019-04-07 [1] CRAN (R 3.6.0)
 Cairo       * 1.5-10  2019-03-28 [1] CRAN (R 3.6.0)
 callr         3.2.0   2019-03-15 [1] CRAN (R 3.6.0)
```

```
cellranger    1.1.0   2016-07-27 [1] CRAN (R 3.6.0)
cli           1.1.0   2019-03-19 [1] CRAN (R 3.6.0)
colorspace    1.4-1   2019-03-18 [1] CRAN (R 3.6.0)
crayon        1.3.4   2017-09-16 [1] CRAN (R 3.6.0)
desc          1.2.0   2018-05-01 [1] CRAN (R 3.6.0)
devtools      2.0.2   2019-04-08 [1] CRAN (R 3.6.0)
digest        0.6.19  2019-05-20 [1] CRAN (R 3.6.0)
dplyr       * 0.8.1   2019-05-14 [1] CRAN (R 3.6.0)
evaluate      0.13    2019-02-12 [1] CRAN (R 3.6.0)
extrafont   * 0.17    2014-12-08 [1] CRAN (R 3.6.0)
extrafontdb   1.0     2012-06-11 [1] CRAN (R 3.6.0)
forcats     * 0.4.0   2019-02-17 [1] CRAN (R 3.6.0)
fs            1.3.1   2019-05-06 [1] CRAN (R 3.6.0)
gdtools       0.1.8   2019-04-02 [1] CRAN (R 3.6.0)
generics      0.0.2   2018-11-29 [1] CRAN (R 3.6.0)
ggplot2     * 3.1.1   2019-04-07 [1] CRAN (R 3.6.0)
ggthemes    * 4.2.0   2019-05-13 [1] CRAN (R 3.6.0)
glue          1.3.1   2019-03-12 [1] CRAN (R 3.6.0)
gridExtra   * 2.3     2017-09-09 [1] CRAN (R 3.6.0)
gtable        0.3.0   2019-03-25 [1] CRAN (R 3.6.0)
haven         2.1.0   2019-02-19 [1] CRAN (R 3.6.0)
highr         0.8     2019-03-20 [1] CRAN (R 3.6.0)
hms           0.4.2   2018-03-10 [1] CRAN (R 3.6.0)
hrbrthemes  * 0.6.0   2019-01-21 [1] CRAN (R 3.6.0)
htmltools     0.3.6   2017-04-28 [1] CRAN (R 3.6.0)
httr          1.4.0   2018-12-11 [1] CRAN (R 3.6.0)
jsonlite      1.6     2018-12-07 [1] CRAN (R 3.6.0)
knitr         1.23    2019-05-18 [1] CRAN (R 3.6.0)
labeling      0.3     2014-08-23 [1] CRAN (R 3.6.0)
lattice       0.20-38 2018-11-04 [2] CRAN (R 3.6.0)
lazyeval      0.2.2   2019-03-15 [1] CRAN (R 3.6.0)
linprog       0.9-2   2012-10-17 [1] CRAN (R 3.6.0)
lubridate     1.7.4   2018-04-11 [1] CRAN (R 3.6.0)
magrittr      1.5     2014-11-22 [1] CRAN (R 3.6.0)
Matrix        1.2-17  2019-03-22 [2] CRAN (R 3.6.0)
MDPtoolbox    4.0.3   2017-03-03 [1] CRAN (R 3.6.0)
memoise       1.1.0   2017-04-21 [1] CRAN (R 3.6.0)
modelr        0.1.4   2019-02-18 [1] CRAN (R 3.6.0)
munsell       0.5.0   2018-06-12 [1] CRAN (R 3.6.0)
nlme          3.1-139 2019-04-09 [2] CRAN (R 3.6.0)
packrat       0.5.0   2018-11-14 [1] CRAN (R 3.6.0)
pillar        1.4.0   2019-05-11 [1] CRAN (R 3.6.0)
pkgbuild      1.0.3   2019-03-20 [1] CRAN (R 3.6.0)
pkgconfig     2.0.2   2018-08-16 [1] CRAN (R 3.6.0)
pkgload       1.0.2   2018-10-29 [1] CRAN (R 3.6.0)
plyr          1.8.4   2016-06-08 [1] CRAN (R 3.6.0)
prettyunits   1.0.2   2015-07-13 [1] CRAN (R 3.6.0)
processx      3.3.1   2019-05-08 [1] CRAN (R 3.6.0)
ps            1.3.0   2018-12-21 [1] CRAN (R 3.6.0)
purrr       * 0.3.2   2019-03-15 [1] CRAN (R 3.6.0)
R6            2.4.0   2019-02-14 [1] CRAN (R 3.6.0)
Rcpp          1.0.1   2019-03-17 [1] CRAN (R 3.6.0)
readr       * 1.3.1   2018-12-21 [1] CRAN (R 3.6.0)
readxl        1.3.1   2019-03-13 [1] CRAN (R 3.6.0)
```

```
 remotes       2.0.4   2019-04-10 [1] CRAN (R 3.6.0)
 reshape2      1.4.3   2017-12-11 [1] CRAN (R 3.6.0)
 rlang         0.3.4   2019-04-07 [1] CRAN (R 3.6.0)
 rmarkdown     1.12    2019-03-14 [1] CRAN (R 3.6.0)
 rprojroot     1.3-2   2018-01-03 [1] CRAN (R 3.6.0)
 rstudioapi    0.10    2019-03-19 [1] CRAN (R 3.6.0)
 Rttf2pt1      1.3.7   2018-06-29 [1] CRAN (R 3.6.0)
 rvest         0.3.4   2019-05-15 [1] CRAN (R 3.6.0)
 sarsop      * 0.6.0   2019-06-04 [1] Github (boettiger-lab/sarsop@6d6d2f8)
 scales        1.0.0   2018-08-09 [1] CRAN (R 3.6.0)
 sessioninfo   1.1.1   2018-11-05 [1] CRAN (R 3.6.0)
 stringi       1.4.3   2019-03-12 [1] CRAN (R 3.6.0)
 stringr     * 1.4.0   2019-02-10 [1] CRAN (R 3.6.0)
 testthat      2.1.1   2019-04-23 [1] CRAN (R 3.6.0)
 tibble      * 2.1.1   2019-03-16 [1] CRAN (R 3.6.0)
 tictoc      * 1.0     2014-06-17 [1] CRAN (R 3.6.0)
 tidyr       * 0.8.3   2019-03-01 [1] CRAN (R 3.6.0)
 tidyselect    0.2.5   2018-10-11 [1] CRAN (R 3.6.0)
 tidyverse   * 1.2.1   2017-11-14 [1] CRAN (R 3.6.0)
 usethis       1.5.0   2019-04-07 [1] CRAN (R 3.6.0)
 withr         2.1.2   2018-03-15 [1] CRAN (R 3.6.0)
 xfun          0.7     2019-05-14 [1] CRAN (R 3.6.0)
 xml2          1.2.0   2018-01-24 [1] CRAN (R 3.6.0)
 yaml          2.2.0   2018-07-25 [1] CRAN (R 3.6.0)

[1] /usr/local/lib/R/site-library
[2] /usr/local/lib/R/library
```

References

Kurniawati, Hanna, David Hsu, and Wee Sun Lee. 2008. "SARSOP : Efficient Point-Based POMDP Planning by Approximating Optimally Reachable Belief Spaces." *Proceedings of Robotics: Science and Systems IV*.

Reed, William J. 1979. "Optimal escapement levels in stochastic and deterministic harvesting models." *Journal of Environmental Economics and Management* 6 (4). Elsevier: 350–63. https://doi.org/10.1016/0095-0696(79)90014-7.

Sethi, Gautam, Christopher Costello, Anthony Fisher, Michael Hanemann, and Larry Karp. 2005. "Fishery management under multiple uncertainty." *Journal of Environmental Economics and Management* 50 (2): 300–318. https://doi.org/10.1016/j.jeem.2004.11.005.