Case Study 1: Bogdan et al. 2020

Two versions of a simple model

The first case study in this manuscript creates a model for *Carpobrotus spp.* It is a simple IPM (i.e. no discrete states, one continuous state variable). The data that the regressions are fit to are included in the `ipmr` package, and can be accessed with `data(iceplant_ex)`.

The IPM can be written on paper as follows:

1. $n(z', t+1) = \int_L^U K(z', z) n(z, t) dz$
2. $K(z', z) = P(z', z) + F(z', z)$
3. $P(z', z) = S(z) * G(z', z)$
4. $F(z', z) = p_f(z) * f_s(z) * p_r * f_d(z')$

The components of each sub-kernel are either regression models or constants. Their functional forms are given below:

5. $Logit^{-1}(S(z)) = \alpha_s + \beta_s * z$
6. $G(z', z) \sim Norm(\mu_g, \sigma_g)$
7. $\mu_g = \alpha_g + \beta_g * z$
8. $Logit^{-1}(p_f(z)) = \alpha_{p_f} + \beta_{p_f} * z$
9. $Log(f_s(z)) = \alpha_{f_s} + \beta_{f_s} * z$
10. $f_d(z') \sim Norm(\mu_{f_d}, \sigma_{f_d})$

αs and βs correspond to intercepts and slopes from regression models, respectively. The other parameters are constants derived directly from the data itself.

```
library(ipmr)

data(iceplant_ex)

# growth model.

grow_mod <- lm(log_size_next ~ log_size, data = iceplant_ex)
grow_sd  <- sd(resid(grow_mod))

# survival model

surv_mod <- glm(survival ~ log_size, data = iceplant_ex, family = binomial())

# Pr(flowering) model

repr_mod <- glm(repro ~ log_size, data = iceplant_ex, family = binomial())

# Number of flowers per plant model
```

```r
flow_mod <- glm(flower_n ~ log_size, data = iceplant_ex, family = poisson())

# New recruits have no size(t), but do have size(t + 1)

recr_data <- subset(iceplant_ex, is.na(log_size))

recr_mu  <- mean(recr_data$log_size_next)
recr_sd  <- sd(recr_data$log_size_next)

# This data set doesn't include information on germination and establishment.
# Thus, we'll compute the realized recruitment parameter as the number
# of observed recruits divided by the number of flowers produced in the prior
# year.

recr_n   <- length(recr_data$log_size_next)

flow_n   <- sum(iceplant_ex$flower_n, na.rm = TRUE)

recr_pr  <- recr_n / flow_n


# Now, we put all parameters into a list. This case study shows how to use
# the mathematical notation, as well as how to use predict() methods

all_params <- list(
  surv_int = coef(surv_mod)[1],
  surv_slo = coef(surv_mod)[2],
  repr_int = coef(repr_mod)[1],
  grow_int = coef(grow_mod)[1],
  grow_slo = coef(grow_mod)[2],
  grow_sdv = grow_sd,
  repr_slo = coef(repr_mod)[2],
  flow_int = coef(flow_mod)[1],
  flow_slo = coef(flow_mod)[2],
  recr_n   = recr_n,
  flow_n   = flow_n,
  recr_mu  = recr_mu,
  recr_sd  = recr_sd,
  recr_pr  = recr_pr
)
```

The next chunk generates a couple constants used to implement the model. We add 20% to the smallest and largest observed sizes to minimize eviction, and will implement the model with 100 meshpoints.

```r
L <- min(c(iceplant_ex$log_size,
           iceplant_ex$log_size_next),
         na.rm = TRUE) * 1.2

U <- max(c(iceplant_ex$log_size,
           iceplant_ex$log_size_next),
         na.rm = TRUE) * 1.2

n_mesh_p <- 100
```

We now have the parameter set prepared, and have the boundaries for our domains set up. We are ready to implement the model. We'll specify `return_main_env = TRUE` in `make_ipm()` because we'll need the mesh points and bin width for some subsequent analyses.

```
carpobrotus_ipm <- init_ipm('simple_di_det') %>%
  define_kernel(
    name      = "P",
    formula   = S * G,
    family    = "CC",
    G         = dnorm(sa_2, mu_g, grow_sdv),
    mu_g      = grow_int + grow_slo * sa_1,
    S         = plogis(surv_int + surv_slo * sa_1),
    data_list = all_params,
    states    = list(c('sa')),
    evict_cor = TRUE,
    evict_fun = truncated_distributions("norm", "G")
  ) %>%
  define_kernel(
    name      = "F",
    formula   = recr_pr * f_s * f_d * p_f,
    family    = "CC",
    f_s       = exp(flow_int + flow_slo * sa_1),
    f_d       = dnorm(sa_2, recr_mu, recr_sd),
    p_f       = plogis(repr_int + repr_slo * sa_1),
    data_list = all_params,
    states    = list(c("sa")),
    evict_cor = TRUE,
    evict_fun = truncated_distributions("norm", "f_d")
  ) %>%
  define_k(
    name      = "K",
    family    = "IPM",
    K         = P + F,
    n_sa_t_1  = K %*% n_sa_t,
    data_list = all_params,
    states    = list(c('sa')),
    evict_cor = FALSE
  ) %>%
  define_impl(
    make_impl_args_list(
      kernel_names = c("K", "P", "F"),
      int_rule     = rep('midpoint', 3),
      dom_start    = rep('sa', 3),
      dom_end      = rep('sa', 3)
    )
  ) %>%
  define_domains(
    sa = c(L, U, n_mesh_p)
  )  %>%
  define_pop_state(
    n_sa = rep(1/100, n_mesh_p)
  ) %>%
  make_ipm(iterate         = TRUE,
           iterations      = 100,
```

```
            return_main_env = TRUE)


asymp_grow_rate <- lambda(carpobrotus_ipm)
asymp_grow_rate
```

```
## [1] 0.9759257
```

Using predict methods instead

We can simplify the code a bit more and get rid of the mathematical expressions for each regression model's link function by using `predict()` methods instead. The next chunk shows how to do this. Instead of extracting parameter values, we put the model objects themselves into the `data_list`. Next, we specify the `newdata` object where the name corresponds to the variable name(s) used in the model in question, and the values are the domain you want to evaluate the model on.

```
pred_par_list <- list(
  grow_mod = grow_mod,
  grow_sdv = grow_sd,
  surv_mod = surv_mod,
  repr_mod = repr_mod,
  flow_mod = flow_mod,
  recr_n   = recr_n,
  flow_n   = flow_n,
  recr_mu  = recr_mu,
  recr_sd  = recr_sd,
  recr_pr  = recr_pr
)

predict_method_carpobrotus <- init_ipm('simple_di_det') %>%
  define_kernel(
    name      = "P",
    formula   = S * G,
    family    = "CC",
    G         = dnorm(sa_2, mu_g, grow_sdv),
    mu_g      = predict(grow_mod,
                        newdata = data.frame(log_size = sa_1),
                        type = 'response'),
    S         = predict(surv_mod,
                        newdata = data.frame(log_size = sa_1),
                        type = "response"),
    data_list = pred_par_list,
    states    = list(c('sa')),
    evict_cor = TRUE,
    evict_fun = truncated_distributions("norm", "G")
  ) %>%
  define_kernel(
    name      = "F",
    formula   = recr_pr * f_s * f_d * p_f,
    family    = "CC",
    f_s       = predict(flow_mod,
                        newdata = data.frame(log_size = sa_1),
                        type = "response"),
```

```
    f_d       = dnorm(sa_2, recr_mu, recr_sd),
    p_f       = predict(repr_mod,
                        newdata = data.frame(log_size = sa_1),
                        type = "response"),
    data_list = pred_par_list,
    states    = list(c("sa")),
    evict_cor = TRUE,
    evict_fun = truncated_distributions("norm", "f_d")
  ) %>%
  define_k(
    name      = "K",
    family    = "IPM",
    K         = P + F,
    n_sa_t_1  = K %*% n_sa_t,
    data_list = list(),
    states    = list(c('sa')),
    evict_cor = FALSE
  ) %>%
  define_impl(
    make_impl_args_list(
      kernel_names = c("K", "P", "F"),
      int_rule     = rep('midpoint', 3),
      dom_start    = rep('sa', 3),
      dom_end      = rep('sa', 3)
    )
  ) %>%
  define_domains(
    sa = c(L, U, n_mesh_p)
  )  %>%
  define_pop_state(
    n_sa = rep(1/100, n_mesh_p)
  ) %>%
  make_ipm(iterate    = TRUE,
           iterations = 100)
```

Further analyses

Many analyses require a bit more than just computing asymptotic lambda. Below, we will compute the kernel sensitivity, elasticity, R0, and generation time. First, we will define a couple helper functions. These are not included in `ipmr`, but will eventually be implemented in a separate package that can handle the various classes that `ipmr` works with.

```
sens <- function(ipm_obj, d_z) {

  K <- ipm_obj$iterators$K

  w <- Re(eigen(K)$vectors[ , 1])
  v <- Re(eigen(t(K))$vectors[ , 1])

  return(
    outer(v, w) / sum(v * w * d_z)
  )
}
```

```r
elas <- function(ipm_obj, d_z) {

  K           <- ipm_obj$iterators$K

  sensitivity <- sens(ipm_obj, d_z)

  lamb        <- lambda(ipm_obj, comp_method = "eigen")

  out         <- sensitivity * (K / d_z) / lamb

  return(out)

}

R_nought <- function(ipm_obj) {

  Pm <- ipm_obj$sub_kernels$P
  Fm <- ipm_obj$sub_kernels$F

  I  <- diag(dim(Pm)[1])

  N  <- solve(I - Pm)

  R  <- Fm %*% N

  return(
    Re(eigen(R)$values)[1]
  )

}

gen_time <- function(ipm_obj) {

  lamb     <- unname(lambda(ipm_obj, comp_method = "eigen"))

  r_nought <- R_nought(ipm_obj)

  return(log(r_nought) / log(lamb))
}
```

We just need to extract the `d_z` value and meshpoints. We can extract this information in a list form using the `int_mesh()` function on our IPM object. The `d_z` in this case will be called `d_sa` because we named our domain `"sa"` when we implemented the model. Once we have that, we can begin computing the life history traits of interest.

```r
mesh_info <- int_mesh(carpobrotus_ipm)

sens_mat <- sens(carpobrotus_ipm, mesh_info$d_sa)
elas_mat <- elas(carpobrotus_ipm, mesh_info$d_sa)

R0    <- R_nought(carpobrotus_ipm)
gen_T <- gen_time(carpobrotus_ipm)

R0
```

```
## [1] 0.5079748
```

```
gen_T
```

```
## [1] 27.79469
```

The next step of the sensitivity and elasticity analysis is usually to visualize the result. We'll go through two options: one using the `graphics` package and one using the `ggplot2` package. The `ggplot2` method will require us to define a function that transforms our matrix into a dataframe with 3 columns. The first two columns contain matrix indices, and the third column contains the actual values.

First, the `graphics` package.

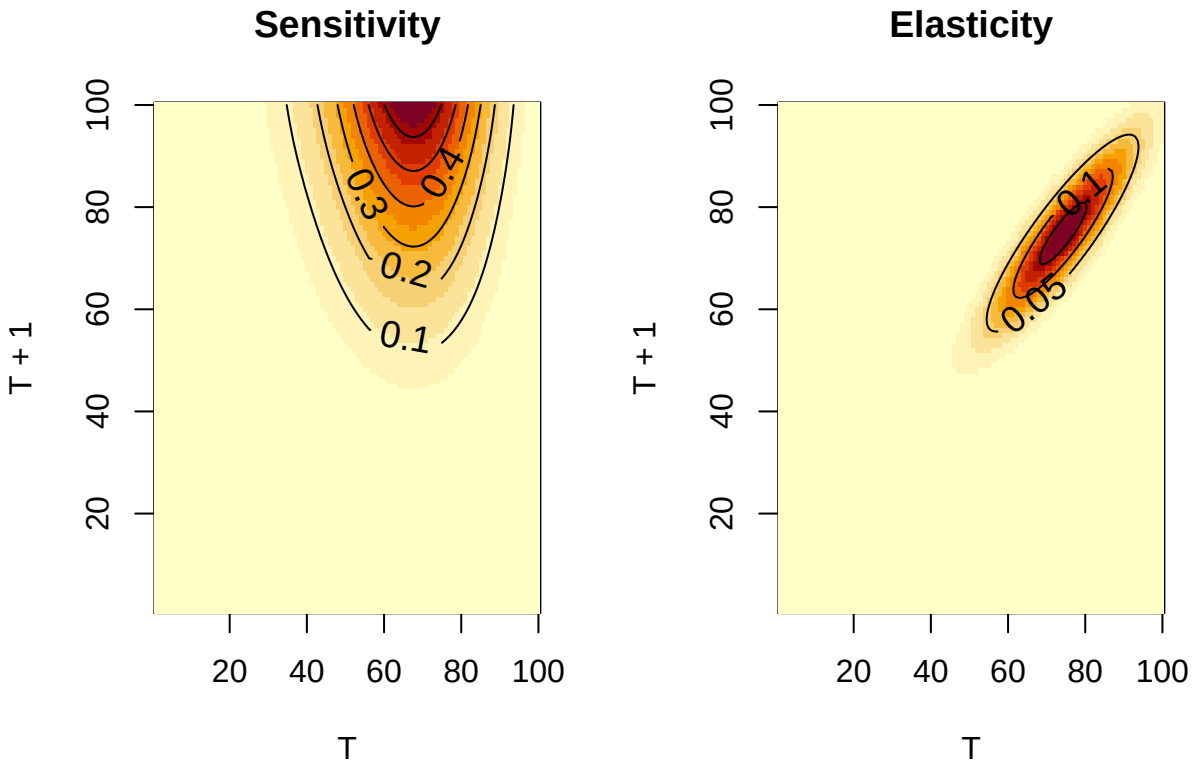
```
par(mfrow = c(1, 2))

x <- y <- seq_len(ncol(sens_mat))

image(x = x,
      y = y,
      z = t(sens_mat),
      main = "Sensitivity", xlab = "T", ylab = "T + 1")

contour(x = x,
        y = y,
        t(sens_mat),
        nlevels = 5,
        labcex = 1.2,
        add = TRUE)

image(x = x,
      y = y,
      z = t(elas_mat),
      main = "Elasticity", xlab = "T", ylab = "T + 1",
      add = FALSE)
contour(x = x,
        y = y,
        t(elas_mat),
        nlevels = 5,
        labcex = 1.2,
        add = TRUE)
```



Now, for the ggplot2 version. First, create a function to create long format data frames for each matrix. Then, get the meshpoints for the model out of the main_env. Once we have those, we can use geom_tile and geom_contour to generate the ggplots, and grid.arrange from the gridExtra package to put them side by side.

```
library(ggplot2)
library(gridExtra)

mat_to_df <- function(mat, meshp) {

  meshp$value <- NA_real_

  it <- 1

  for(i in seq_len(dim(mat)[1])) {
    for(j in seq_len(dim(mat)[2])) {

      meshp[it, 3] <- mat[i, j]
      it <- it + 1

    }
  }

  return(meshp)
}

mesh_p <- data.frame(x = mesh_info$sa_1,
```

```
                    y = mesh_info$sa_2)

sens_df <- mat_to_df(sens_mat, mesh_p)
elas_df <- mat_to_df(elas_mat, mesh_p)

def_theme <- theme(
  panel.background = element_blank(),
  axis.text        = element_blank(),
  axis.title.x     = element_text(size   = 16,
                                  margin = margin(
                                    t = 20,
                                    r = 0,
                                    l = 0,
                                    b = 2
                                  )
  ),
  axis.title.y     = element_text(size   = 16,
                                  margin = margin(
                                    t = 0,
                                    r = 20,
                                    l = 2,
                                    b = 0
                                  )
  ),
  axis.ticks = element_blank(),
  legend.title = element_text(size = 16)
)

sens_plt <- ggplot(sens_df) +
  geom_tile(aes(x    = x,
                y    = y,
                fill = value)) +
  geom_contour(aes(x = x,
                   y = y,
                   z = value),
               color = "black",
               size = 0.7) +
  scale_fill_gradient("Value",
                      low = "red",
                      high = "yellow") +
  scale_x_continuous(name = "T") +
  scale_y_continuous(name = "T + 1") +
  def_theme +
  ggtitle("Sensitivity")

elas_plt <- ggplot(elas_df) +
  geom_tile(aes(x    = x,
                y    = y,
                fill = value)) +
  geom_contour(aes(x = x,
                   y = y,
                   z = value),
               color = "black",
```

```
                size = 0.7) +
  scale_fill_gradient("Value",
                      low = "red",
                      high = "yellow") +
  scale_x_continuous(name = "T") +
  scale_y_continuous(name = "T + 1") +
  def_theme +
  ggtitle("Elasticity")

grid.arrange(sens_plt, elas_plt,
             layout_matrix = matrix(c(1, 2,
                                      1, 2),
                                    nrow = 2,
                                    byrow = TRUE))
```