Advancing Integral Projection Models with New Computational Tools and Sources of Data

Contents

1 Introduction

blah blah blah

2 Chapter 1: ipmr: an R package to flexibly implement Integral Projection Models

2.1 Abstract

1. Integral projection models (IPMs) are an important tool for studying the dynamics of populations structured by one or more continuous traits (*e.g.* size, height, body mass). Researchers use IPMs to investigate questions ranging from linking drivers to population dynamics, planning conservation and management strategies, and quantifying selective pressures in natural populations. The popularity of

stage-structured population models has been supported by *R* scripts and packages (*e.g.* `IPMpack`, `popbio`, `popdemo`, `lefko3`) aimed at ecologists, which have introduced a broad repertoire of functionality and outputs. However, pressing ecological, evolutionary, and conservation biology topics require developing more complex IPMs, and considerably more expertise to implement them. Here, we introduce `ipmr`, a flexible *R* package for building, analyzing, and interpreting IPMs.

2. The `ipmr` framework relies on the mathematical notation of the models to express them in code format. Additionally, this package decouples the model parameterization step from the model implementation step. The latter point substantially increases `ipmr`'s flexibility to model complex life cycles and demographic processes.

3. `ipmr` can handle a wide variety of models, including those that incorporate density dependence, discretely and continuously varying stochastic environments, and multiple continuous and/or discrete traits. `ipmr` can accommodate models with individuals cross-classified by age and size. Furthermore, the package provides methods for demographic analyses (*e.g.* asymptotic and stochastic growth rates) and visualization (*e.g.* kernel plotting).

4. `ipmr` is a flexible *R* package for integral projection models. The package substantially reduces the amount of time required to implement general IPMs. We also provide extensive documentation with six vignettes and help files, accessible from an R session and online.

Keywords: elasticity, integral projection model, life history, population dynamics, population growth rate, sensitivity, structured populations

2.2 Introduction

Integral projection models (IPMs) are an important and widely used tool for ecologists studying structured population dynamics in discrete time. Since the paper introducing IPMs was published over two decades ago (Easterling et al., 2000), at least 255 peer-reviewed publications on at least 250 plant species and 60 animal species have used IPMs (Figure S1.XXX). These models have addressed questions ranging from invasive species population dynamics (*e.g.* Crandall & Knight, 2017), effect of climate drivers on population persistence (*e.g.* Compagnoni et al., 2021), evolutionary stable strategies (*e.g.* Childs et al., 2004), and rare/endangered species conservation (*e.g.* Ferrer-Cervantes et al., 2012).

The IPM was introduced as alternative to matrix population models, which model populations structured by discrete traits (Caswell, 2001). Some of the advantages of using an IPM include (i) the ability to model populations structured by continuously distributed traits, (ii) the ability to flexibly incorporate discrete and continuous traits in the same model (*e.g.* seeds in a seedbank and a height structured plant population (Crandall & Knight, 2017), or number of females, males, and age-1 recruits for fish species (Erickson et al., 2017)), (iii) efficient parameterization of demographic processes with familiar regression methods (Coulson, 2012), and (iv) the numerical discretization of continuous kernels (see below) means that the tools available for matrix population models are usually also applicable for IPMs. Furthermore, researchers have developed methods to incorporate spatial dynamics (Jongejans et al., 2011), environmental stochasticity (Rees & Ellner, 2009), and density/frequency dependence into IPMs (Adler et al., 2010, Ellner et al., 2016). These developments were accompanied by the creation of software tools and guides to assist with IPM parameterization, implementation, and analysis. These tools range from *R* scripts with detailed annotations (Coulson, 2012, Merow et al., 2014, Ellner et al., 2016) to *R* packages (Metcalf et al., 2013, Shefferson et al., 2020).

Despite the array of resources available to researchers, implementing an IPM is still not a straightforward exercise. For example, an IPM that simulates a population for 100 time steps requires the user to either write or adapt from published guides multiple functions (*e.g.* to summarize demographic functions into the proper format), implement the numerical approximations of the model's integrals, ensure that individuals are not accidentally sent beyond the integration bounds ("unintentional eviction", *sensu* Williams et al., 2012), and track how the population state changes over the course of a simulation. Stochastic IPMs present further implementation challenges. In addition to the aforementioned elements, users must generate the sequence of environments that the population experiences. There are multiple ways of simulating environmental stochasticity, each with their own strengths and weaknesses (Metcalf et al. 2015).

`ipmr` manages these key details while providing the user flexibility in their models. `ipmr` uses the `rlang` package for metaprogramming (Henry & Wickham, 2019), which enables `ipmr` to provide a miniature domain specific language for implementing IPMs. `ipmr` aims to mimic the mathematical syntax that describes IPMs as closely as possible (Fig. 1.1, Box 1.1, Tables 1.1 and 1.2). This *R* package can handle models with individuals classified by a mixture of any number of continuously and discretely distributed traits. Furthermore, `ipmr` introduces specific classes and methods to deal with both discretely and continuously varying stochastic environments, density-independent and -dependent models, as well as age structured populations (Case Study 2). `ipmr` decouples the parameterization (*i.e.* regression model fitting) and implementation steps (*i.e.* converting the regression parameters into a full IPM), and does not attempt to help users with the parameterization task. This provides greater flexibility in modeling trait-demography relationships, and enables users to specify IPMs of any functional form that they desire.

2.3 Terminology and IPM construction

An IPM describes how the abundance and distribution of trait values (also called *state variables/states*, denoted z and z') for a population changes in discrete time. The distribution of trait values in a population at time t is given by the function $n(z,t)$. A simple IPM for the trait distribution z' at time $t+1$ is then

$$n(z', t+1) = \int_L^U K(z', z) n(z, t) dz. \tag{1.1}$$

$K(z', z)$, known as the *projection kernel*, describes all possible transitions of existing individuals and recruitment of new individuals from t to $t+1$, generating a new trait distribution $n(z', t+1)$. L, U are the lower and upper bounds for values that the trait z can have, which defines the *domain* over which the integration is performed. The integral $\int_L^U n(z,t)dz$ gives the total population size at time t.

To make the model more biologically interpretable, the projection kernel $K(z', z)$ is usually split into *sub-kernels* (Eq 1.2). For example, a projection kernel to describe a lifecycle where individuals can survive, transition to different state values, and reproduce via sexual and asexual pathways, can be split as follows

$$K(z', z) = P(z', z) + F(z', z) + C(z', z), \tag{1.2}$$

where $P(z', z)$ is a sub-kernel describing transitions due to survival and trait changes of existing individuals, $F(z', z)$ is a sub-kernel describing per-capita sexual contributions of existing individuals to recruitment, and $C(z', z)$ is a sub-kernel describing per-capita asexual contributions of existing individuals to recruitment. The sub-kernels are typically comprised of functions derived from regression models that relate an individual's trait value z at time t to a new trait value z' at $t+1$. For example, the P kernel for Soay sheep (*Ovis aries*) on St. Kilda (Eq 1.3) may contain two regression models: (i) a logistic regression of survival on log body mass (Eq 1.4), and (ii) a linear regression of log body mass at $t+1$ on log body mass at t (Eq 1.5-1.6). In this example, f_G is a normal probability density function with μ_G given by the linear predictor of the mean, and with σ_G computed from the standard deviation of the residuals from the linear regression model.

$$P(z', z) = s(z) * G(z', z), \tag{1.3}$$

$$Logit(s(z)) = \alpha_s + \beta_s * z, \tag{1.4}$$

$$G(z', z) = f_G(z', \mu_G, \sigma_G), \tag{1.5}$$

$$\mu_G = \alpha_G + \beta_G * z. \tag{1.6}$$

Analytical solutions to the integral in Eq 1.1 are usually not possible (Ellner & Rees, 2006). However, numerical approximations of these integrals can be constructed using a numerical integration rule. A commonly used rule is the midpoint rule (more complicated and precise methods are possible and will be implemented, though are not yet, see Ellner et al., 2016, Chapter 6). The midpoint rule divides the domain $[L, U]$ into m artifical size bins centered at z_i with width $h = (U - L)/m$. The midpoints $z_i = L + (i - 0.5) * h$ for $i = 1, 2, ..., m$. The midpoint rule approximation for Eq 1.1 then becomes:

$$n(z_j, t+1) = h \sum_{i=1}^{m} K(z_j, z_i) n(z_i, t) \tag{1.7}$$

In practice, the numerical approximation of the integral converts the continuous projection kernel into a (large) discretized matrix. A matrix multiplication of the discretized projection kernel and the discretized trait distribution then generates a new trait distribution, a process referred to as *model iteration* (*sensu* Easterling et al., 2000).

Equations 1.1 and 1.2 are an example of a *simple IPM*. A critical aspect of `ipmr`'s functionality is the distinction between *simple IPMs* and *general IPMs*. A simple IPM incorporates a single continuous state variable. Equations 1.1 and 1.2 represent a simple IPM because there is only one continuous state, z, and no additional discrete states. A general IPM models one or more continuous state variables, and/or discrete states. General IPMs are useful for modelling species with more complex life cycles. Many species' life cycles contain multiple life stages that are not readily described by a single state variable. Similarly, individuals with similar trait values may behave differently depending on environmental context. For example, Bruno et al. (2011) modeled aspergillosis impacts on sea fan coral (*Gorgonia ventalina*) population dynamics by creating a model where colonies were cross classified by tissue area (continuously distributed) and infection status (a discrete state with two levels - infected and uninfected). Coulson, Tuljapurkar & Childs (2010) constructed a model for Soay sheep where the population was structured by body weight (continuously distributed) and age (discrete state). Mixtures of multiple continuous and discrete states are also possible. Indeed, the vital rates of many species with complex life cycles are often best described with multivariate state distributions (Caswell & Salguero-Gómez, 2013). A complete definition of the simple/general distinction is given in Ellner et al. (2016, Chapter 6).

2.4 A brief worked example of a simple IPM with `ipmr`

Box 1.1 shows a brief example of how `ipmr` converts parameter estimates into an IPM. Perhaps the most frequently used metric derived from IPMs is the asymptotic per-capita population growth rate (λ, Caswell 2001). When $\lambda > 1$, the population is growing, while $\lambda < 1$ indicates population decline. `ipmr` makes deriving estimates of λ straightforward. Box 1.1 demonstrates how to parameterize a simple, deterministic IPM and estimate λ. The example uses a hypothetical species that can survive and grow, and reproduce sexually (but not asexually, so $C(z', z) = 0$ in Equation 2). The population is structured by size, denoted z and z', and there is no seedbank.

The $P(z', z)$ kernel is given by Eq 1.3, and the vital rates therein by Eq 1.4-1.6. The $F(z', z)$ kernel is given Eq 1.8:

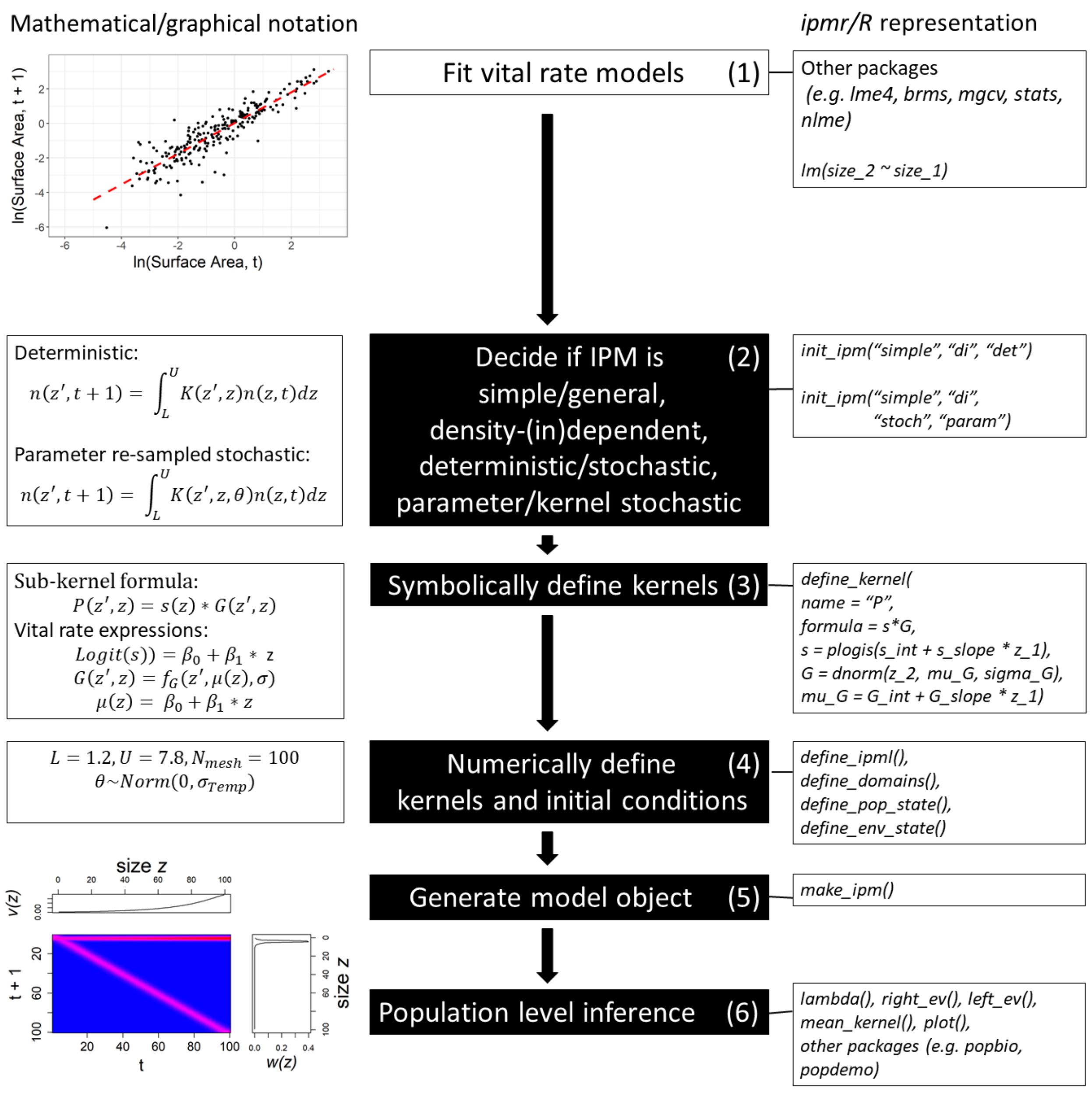
$$F(z', z) = r_d(z') * r_n(z), \tag{1.8}$$

$$r_d(z') = f_{r_d}(z', \mu_{r_d}, \sigma_{r_d}), \tag{1.9}$$

$$Log(r_n(z)) = \alpha_{r_n} + \beta_{r_n} * z. \tag{1.10}$$

Eq 1.9 is a recruit size distribution (where f_{r_d} denotes a normal probability density function), and Eq 1.10 describes the number of new recruits produced by plants as a function of size z.

```r
library(ipmr)

# This section produces the result of Step 1 in Figure 1.1.

data_list <- list(
  s_i  = -0.65, # Intercept of the survival model (Logistic regression)
  s_z  = 0.75,  # Slope of the survival model
  G_i  = 0.96,  # Intercept of the growth model (Gaussian regression)
  G_z  = 0.66,  # Slope of the growth model
  sd_G = 0.67,  # Standard deviation of residuals of growth model
  mu_r = -0.08, # Mean of the recruit size distribution
  sd_r = 0.76,  # Standard deviation of the recruit size distribution
  r_n_i = -1,   # Intercept of recruit production model (Poisson regression)
  r_n_z = 0.3   # Slope of recruit production model.
)

# Step 2 in Figure 1.1. This is how ipmr initializes a model object.
# All functions prefixed with define_* generate proto_ipm objects. These
# are converted into IPMs using the make_ipm() function in step 5.

example_proto_ipm <- init_ipm(sim_gen   = "simple",
                              di_dd     = "di",
                              det_stoch = "det")

# Step 3 in Figure 1.1. Note the link between how the model was defined
# mathematically and how it is defined here.

example_proto_ipm <- define_kernel(
  example_proto_ipm,
  name          = "P",
```



Mathematical/graphical notation
ipmr/R representation
ln(Surface Area, t + 1)
ln(Surface Area, t)
Fit vital rate models (1)
Other packages (e.g. lme4, brms, mgcv, stats, nlme)
lm(size_2 ~ size_1)
Deterministic:
$n(z',t+1) = \int_L^U K(z',z)n(z,t)dz$
Parameter re-sampled stochastic:
$n(z',t+1) = \int_L^U K(z',z,\theta)n(z,t)dz$
Decide if IPM is simple/general, density-(in)dependent, deterministic/stochastic, parameter/kernel stochastic (2)
init_ipm("simple", "di", "det")
init_ipm("simple", "di", "stoch", "param")
Sub-kernel formula:
$P(z',z) = s(z) * G(z',z)$
Vital rate expressions:
$Logit(s)) = \beta_0 + \beta_1 * z$
$G(z',z) = f_G(z',\mu(z),\sigma)$
$\mu(z) = \beta_0 + \beta_1 * z$
Symbolically define kernels (3)
define_kernel(name = "P", formula = s*G, s = plogis(s_int + s_slope * z_1), G = dnorm(z_2, mu_G, sigma_G), mu_G = G_int + G_slope * z_1)
$L = 1.2, U = 7.8, N_{mesh} = 100$
$\theta \sim Norm(0, \sigma_{Temp})$
Numerically define kernels and initial conditions (4)
define_impl(), define_domains(), define_pop_state(), define_env_state()
Generate model object (5)
make_ipm()
size z
v(z)
t + 1
t
w(z)
Population level inference (6)
lambda(), right_ev(), left_ev(), mean_kernel(), plot(), other packages (e.g. popbio, popdemo)

Figure 1.1: There are generally 6 steps in defining an IPM with `ipmr`. (1) Vital rate models are fit to demographic data collected from field sites. This step requires the use of other packages, as `ipmr` does not contain facilities for regression modeling. The figure on the left shows the fitted relationship between size at t and $t+1$ for *Carpobrotus spp.* in Case Study 1. (2) The next step is deciding what type of IPM is needed. This is determined by both the research question and the data used to parameterize the regression models. This process is initiated with `init_ipm()`. In step (3), kernels are defined using `ipmr`'s syntax to represent kernels and vital rate functions. (4) Having defined symbolic representations of the model, the numerical definition is given. Here, the integration rule, domain bounds, and initial population conditions are defined. For some models, initial environmental conditions can also be defined. (5) `make_ipm()` numerically implements the `proto_ipm` object, (6) which can then be analyzed further. The figure at the bottom left shows a $K(z',z)$ kernel created by `make_ipm()` and `make_iter_kernel()`. The line plots above and to the right display the left and right eigenvectors, extracted with `left_ev()` and `right_ev()`, respectively.

```r
  formula      = surv * Grow,
  surv         = plogis(s_i + s_z * z_1),
  Grow         = dnorm(z_2, mu_G, sd_G),
  mu_G         = G_i + G_z * z_1,
  data_list    = data_list,
  states       = list(c("z"))
)

example_proto_ipm <- define_kernel(
  example_proto_ipm,
  name        = "F",
  formula     = recr_number * recr_size,
  recr_number = exp(r_n_i + r_n_z * z_1),
  recr_size   = dnorm(z_2, mu_r, sd_r),
  data_list   = data_list,
  states      = list(c("z"))
)

# Step 4 in Figure 1.1. These next 3 functions define:
# 1. The numerical integration rules and how to iterate the
#    model (define_impl).
# 2. The range of values the the trait "z" can take on, and the number of
#    meshpoints to use when dividing the interval (define_domains).
# 3. The initial population state (define_pop_state).

example_proto_ipm <- define_impl(
  example_proto_ipm,
  list(
    P = list(int_rule = "midpoint", state_start = "z", state_end = "z"),
    F = list(int_rule = "midpoint", state_start = "z", state_end = "z")
  )
)

example_proto_ipm <- define_domains(
  example_proto_ipm,
  z = c(-2.65, 4.5, 250) # format: c(L, U, m), m is number of meshpoints
)

example_proto_ipm <- define_pop_state(
  example_proto_ipm,
  n_z = rep(1/250, 250)
)

# Step 5 in Figure 1.1.

example_ipm <- make_ipm(example_proto_ipm)

# Step 6 in Figure 1.1.

lambda(example_ipm)
```

*Box 1.1: Code to implement a simple IPM from parameter estimates in **ipmr**. Because **ipmr** does not include functions to assist with regression modeling, this example skips the step of working with actual data and instead uses hypothetical parameter values. We see that given this set of conditions, if nothing were to change,* *the population would increase by ~2% each year. The case studies provide details on further use cases and analyses that are possible with* `ipmr`*.*

The code in Box 1.1 substitutes the actual probability density function (`dnorm()`) for f_G and f_{r_d}, and uses inverse link functions instead of link functions. Otherwise, the math and the code should look quite similar.

2.5 Case study 1 - A simple IPM

One use for IPMs is to evaluate potential performance and management of invasive species in their non-native range (*e.g.* Erickson et al., 2017). Calculating sensitivities and elasticities of λ to kernel perturbations can help identify conservation management strategies (de Kroon et al., 1986, Caswell, 2001, Baxter et al., 2006, Ellner et al., 2016). Bogdan et al. (2021) constructed a simple IPM for a *Carpobrotus* species growing north of Tel Aviv, Israel. The model includes four regressions, and an estimated recruit size distribution. Table 1.1 provides the mathematical formulae, the corresponding R model formulae, and the `ipmr` notation for each one. The case study materials also offer an alternative implementation that uses the generic `predict()` function to generate the same output. The final part of the case study provides examples of functions that compute kernel sensitivity and elasticity, the per-generation growth rate, and generation time for the model, as well as how to visualize these results.

2.6 Case study 2 - A general age $\times$ size IPM

We use an age- and size-structured IPM from Ellner et al. (2016) to illustrate how to create general IPMs with `ipmr`. This case study demonstrates the suffix syntax for vital rate and kernel expressions, which is a key feature of `ipmr` (highlighted in bold in the 'ipmr' column in Table 1.2). The suffixes appended to each variable name in the `ipmr` formulation correspond to the sub- and/or super-scripts used in the mathematical formulation. `ipmr` internally expands the model expressions and substitutes the range of ages and/or grouping variables in for the suffixes. This allows users to specify their model in a way that closely mirrors its mathematical notation, and saves users from the potentially error-prone process of re-typing model definitions many times or using `for` loops over the range of discrete states. The case study then demonstrates how to compute age-specific survival and fertility from the model outputs.

2.7 Discussion of additional applications

We have shown above how `ipmr` handles a variety of model implementations that go beyond the capabilities of existing scripts and packages. The underlying implementation based on metaprogramming should be able to readily incorporate future developments in parameterization methods. Regression modeling is a field that is constantly introducing new methods. As long as these new methods have functional forms for their expected value (or a function to compute them, such as `predict()`), `ipmr` should be able to implement IPMs using them.

Finally, one particularly useful aspect of the package is the `proto_ipm` data structure. The `proto_ipm` is the common data structure used to represent every model class in `ipmr` and provides a concise, standardized format for representing IPMs. Furthermore, the `proto_ipm` object is created without any raw data, only functional forms and parameters. The PADRINO IPM database uses `ipmr` and `proto_ipm`s as an "engine" to re-build published IPMs using only functional forms and parameter estimates. This database could act as an IPM equivalent of the popular COMPADRE and COMADRE matrix population model databases (Salguero-Gómez et al., 2015, Salguero-Gómez et al., 2016). Recent work has highlighted the power of syntheses that harness many structured population models (Adler et al., 2014, Salguero-Gómez et al., 2016, Compagnoni et al., 2021). Despite the wide variety of models that are currently published in the IPM literature, `ipmr`'s functional approach is able to reproduce nearly all of them without requiring any raw data at all. PADRINO and its usage are discussed in the next chapter of this dissertation.

Table 1.1: Translations between mathematical notation, R's formula notation, and ipmr's notation for the simplified version of Bogdan et al.'s Carpobrotus IPM. The ipmr column contains the expressions used in each kernel's definition. R expressions are not provided for sub-kernels and model iteration procedures because they typically require defining functions separately, and there are many ways to do this step (examples are in the R code for each case study in the appendix). The plogis() function computes the inverse logit transformation of an expression. s corresponds to survival, G corresponds to change in size conditional on survival, r_p is the probability of reproducing, r_n is the number of propagules produced by reproductive individuals, and p_r is the probability that a propagule becomes a new recruit at $t+1$.

Math Formula	R Formula	ipmr
$\mu_G = \alpha_G + \beta_G * z$	size_2 size_1, family = gaussian()	mu_G = G_int + G_slope * z
$G(z', z) = f_G(z', \mu_G, \sigma_G)$	G = dnorm(z_2, mu_G, sd_G)	G = dnorm(z_2, mu_G, sd_G)
$logit(s(z)) = \alpha_s + \beta_s * z$	surv size_1, family = binomial()	s = plogis(s_int + s_slope * z)
$log(r_n(z)) = \alpha_{r_n} + \beta_{r_n} * z$	fec size_1, family = poisson()	r_n = exp(r_n_int + r_n_slope * z)
$logit(r_p(z)) = \alpha_{r_p} + \beta_{r_p} * z$	repr size_1, family = binomial()	r_p = plogis(r_p_int + r_p_slope * z)
$r_d(z') = f_{r_d}(z', \mu_{r_d}, \sigma_{r_d})$	dnorm(z_2, mu_f_d, sigma_f_d)	r_d = dnorm(z_2, f_d_mu, f_d_sigma)
$p_r = \frac{\#Recruits(t+1)}{\#flowers(t)}$	p_r = n_new_recruits / n_flowers	p_r = n_new / n_flowers
$P = s(z) * G(z', z)$		P = s * G
$F(z', z) = r_p(z) * r_n(z) * r_d(z') * p_r$		F = r_p * r_n * r_d * p_r
$n(z', t+1) = \int_L^U [P(z', z) + F(z', z)] n(z, t) dz$		

Table 1.2: Translations between mathematical notation, R's formula notation, and ipmr's notation for Ellner et al. (2016) Ovis aries IPM. The ipmr column contains the expressions used in each kernel's definition. R expressions are not provided for sub-kernels and model iteration procedures because they typically require defining functions separately, and there are many ways to do this step (examples are in the R code for each case study in the appendix). ipmr supports a suffix based syntax to avoid repetitively typing out the levels of discrete grouping variables. These are represented as 'a' in the Math column, 'age' in the R formula column, and are highlighted with '*'s in the ipmr column. s corresponds to survival, G corresponds to change in size conditional on survival, m_p is the probability of mating, r_p is the probability that a mating produces a new recruit at $t+1$, and B is the size distribution of new recruits at $t+1$ whose mean depends on parent size at time t. F_a is divided by 2 because this IPM only tracks females.

Math Formula	R Formula	ipmr
$Logit(s(z,a)) = \alpha_s + \beta_{s,z} * z + \beta_{s,a} * a$	surv size_1 + age, family = binomial()	s_*age* = plogis(s_int + s_z * z_1 + s_a * *age*)
$G(z',z,a) = f_G(z', \mu_G(z,a), \sigma_G)$	G = dnorm(size_2, mu_G_age, sigma_G)	G_*age* = dnorm(z_2, mu_G_*age*, sigma_G)
$\mu_G(z,a) = \alpha_G + \beta_{G,z} * z + \beta_{G,a} * a$	size_2 size_1 + age, family = gaussian()	mu_G_*age* = G_int + G_z * z + G_a * *age*
$Logit(m_p(z,a)) = \alpha_{m_p} + \beta_{m_p,z} * z + \beta_{m_p,a} * a$	repr size_1 + age, family = binomial()	m_p_*age* = plogis(m_p_int + m_p_z * z + m_p_a * *age*)
$Logit(r_p(a)) = \alpha_{r_p} + \beta_{r_p,a} * a$	recr age, family = binomial()	r_p_*age* = plogis(r_p_int + r_p_a * *age*)
$B(z',z) = f_B(z', \mu_B(z), \sigma_B)$	b = dnorm(size_2, mu_rc_size, sigma_rc_size)	rc_size = dnorm(z_2, mu_rc_size, sigma_rc_size)
$\mu_B(z) = \alpha_B + \beta_{B,z} * z$	rc_size_2 size_1, family = gaussian()	mu_rc_size = rc_size_int + rc_size_z * z
$P_a(z',z) = s(z,a) * G(z',z,a)$		P_*age* = s_*age* * g_*age* * d_z
$F_a(z',z) = s(z,a) * m_p(z,a) * r_p(a) * B(z',z)/2$		F_*age* = s_*age* * f_p_*age* * r_p_*age* * rc_size / 2
$n_0(z',t+1) = \sum\limits_{a=0}^{M+1} \int_L^U F_a(z',z) n_a(z,t) dz$		
$n_a(z',t+1) = \int_L^U P_{a-1}(z',z) n_{a-1}(z,t) dz$		
$n_{M+1}(z',t+1) = \int_L^U [P_{M+1}(z',z) n_{M+1}(z,t) + P_M(z',z) n_M(z,t)] dz$		

3 Chapter 2: PADRINO and Rpadrino: a toolkit for rebuilding and analyzing published Integral Projection Models

Blah blah blah

4 How climate change will alter the threat of invasive species: a case study with the *Carpobrotus* genus

5 Discussion

6 Citations

1. Adler, P.B., Ellner, S.P. & Levine, J.M. (2010). Coexistance of perennial plants: an embarassment of niches. Ecology Letters 13: 1019-1029. https://doi.org/10.1111/j.1461-0248.2010.01496.x

2. Adler, P.B., Salguero-Gómez, R., Compagnoni, A., Hsu, J.S., Ray-Mukherjee, J., Mbeau-Ache, C. & Franco, M. (2014). Functional traits explain variation in plant life history strategies. Proceedings of the National Academy of Sciences 111(2): 740-745. https://doi.org/10.1073/pnas.1315179111

3. Baxter, P.W.J., McCarthy, M.A., Possingham, H.P., Menkhorst, P.W. & McLean, N. (2006). Accounting for management costs in sensitivity analyses of matrix population models. Conservation Biology 20(3): 893-905. https://doi.org/10.1111/j.1523-1739.2006.00378.x

4. Bache, S.M., & Wickham, H. (2020). magrittr: A Forward-Pipe Operator for R. R package version 2.0.1. https://CRAN.R-project.org/package=magrittr

5. Bogdan, A., Levin, S.C., Salguero-Gómez, R., Knight, T.M. (2021). Demographic analysis of Israeli Carpobrotus populations: management strategies and future directions. PLoS ONE 16(4): e0250879. https://doi.org/10.1101/2020.12.08.415174

6. Bruno, J.F., Ellner, S.P., Vu, I., Kim, K., & Harvell, C.D. (2011). Impacts of aspergillosis on sea fan coral demography: modeling a moving target. Ecological Monographs 81(1): 123-139. https://doi.org/19.1890/09-1178.1

7. Caswell, H. (2001) Matrix population models: construction, analysis, and interpretation, 2nd edn. Sunderland, MA: Sinauer Associates Inc

8. Caswell, H., & Salguero-Gómez R. (2013). Age, stage and senescence in plants. Journal of Ecology 101(3): 585-595. https://doi.org/10.1111/1365-2745.12088

9. Childs, D.Z., Rees, M., Rose, K.E., Grubb, P.J., & Ellner, S.P. (2004). Evolution of size-dependent flowering in a variable environment: construction and analysis of a stochastic integral projection model. Proceedings of the Royal Society B 271(1547): 425-434. https://doi.org/10.1098/rpsb.2003.2597

10. Childs DZ, Coulson T, Pemberton JM, Clutton-Brock TH, & Rees M (2011). Predicting trait values and measuring selection in complex life histories: reproductive allocation decisions in Soay sheep. Ecology Letters 14: 985-992.

11. Clutton-Brock TH & Pemberton JM (2004). Soay Sheep: Dynamics and Selection in an Island Population. Cambridge University Press, Cambridge.

12. Compagnoni, A., Levin, S.C., Childs, D.Z., Harpole, S., Paniw, M., Roemer, G., Burns, J.H., Che-Castaldo, J., Rueger, N., Kunstler, G., Bennett, J.M., Archer, C.R., Jones, O.R., Salguero-Gomez, R., & Knight, T.M. (2021). Herbaceous perennial plants with short generation time have stronger

responses to climate anomalies than those with longer generation time. Nature Communications 12: 1824. https://doi.org/10.1038/s41467-021-21977-9

13. Coulson, T.N. (2012). Integral projection models, their construction and use in posing hypotheses in ecology. Oikos 121: 1337-1350. https://doi.org/10.1111/j.1600-0706.2012.00035.x
14. Coulson, T., Tuljapurkar, S., & Childs, D.Z. (2010). Using evolutionary demography to link life history theory, quantitative genetics and population ecology. Journal of Animal Ecology 79: 1226-1240. https://doi.org/10.1111/j.1365-2656.2010.01734.x
15. Crandall, R.M. & Knight, T.M. (2017). Role of multiple invasion mechanisms and their interaction in regulating the population dynamics of an exotic tree. Journal of Applied Ecology 55(2):885-894. https://doi.org/10.1111/1365-2664.13020
16. de Kroon, H., Plaisier, A., van Goenendael, J., & Caswell, H. (1986). Elasticity: the relative contribution of demographic parameters to population growth rate. Ecology 67(5): 1427-1431.
17. Easterling, M.R., Ellner, S.P., & Dixon, P.M. (2000). Size specific sensitivity: applying a new structured population model. Ecology 81(3): 694-708.
18. Ellner, S.P., Childs, D.Z., Rees, M. (2016) Data-driven modelling of structured populations: a practical guide to the integral projection model. Basel, Switzerland: Springer International Publishing AG
19. Ellner, S.P. & Rees, M. (2006). Integral Projection Models for species with complex demography. The American Naturalist 167(3): 410-428.
20. Erickson, R.A., Eager, E.A., Brey, M.B., Hansen, M.J., & Kocovsky, P.M. (2017). An integral projection model with YY-males and application to evaluating grass carp control. Ecological Modelling 361: 14-25. https://doi.org/10.1016/j.ecolmodel.2017.07.030
21. Ferrer-Cervantes, M.E., Mendez-Gonzalez, M.E., Quintana-Ascencio, P-F., Dorantes, A., Dzib, G., & Duran, R. (2012). Population dynamics of the cactus *Mammillaria gaumeri*: an integral projection model approach. Population Ecology 54: 321-334. DOI: https://doi.org/10.1007/s10144-012-0308-7
22. Henry, L., & Wickham, H. (2020). rlang: Functions for Base Types and Core R and 'Tidyverse' Features. R package version 0.4.7. https://CRAN.R-project.org/package=rlang
23. Jongejans, E., Shea, K., Skarpaas, O., Kelly, D., & Ellner, S.P. (2011). Importance of individual and environmental variation for invasive species spread: a spatial integral projection model. Ecology 92(1): 86-97. https://doi.org/10.1890/09-2226.1
24. Merow, C., Dahlgren, J.P., Metcalf, C.J.E., Childs, D.Z., Evans, M.E.K., Jongejans, E., Record, S., Rees, M., Salguero-Gomez R., & McMahon, S.M. (2014). Advancing population ecology with integral projection models: a practical guide. Methods in Ecology and Evolution 5: 99-110. https://doi.org/10.1111/2041-210X.12146S
25. Metcalf, C.J.E., Ellner, S.P., Childs, D.Z., Salguero-Gómez, R., Merow, C., McMahon, S.M., Jongejans, E., & Rees, M. (2015). Statistical modelling of annual variation for inference on stochastic population dynamics using Integral Projection Models. Methods in Ecology and Evolution 6(9): 1007-1017. https://doi.org/10.1111/2041-210X.12405
26. Metcalf, C. J. E., McMahon, S. M., Salguero-Gómez, R. & Jongejans, E. (2013). IPMpack: an R package for integral projection models. Methods in Ecology and Evolution. 4(2): 195-200. https://doi.org/10.1111/2041-210x.12001
27. Ramula, S., Rees, M. & Buckley, Y. M. (2009). Integral projection models perform better for small demographic data sets than matrix population models: a case study of two perennial herbs. Journal of Applied Ecology 46(5): 1048-1053. https://doi.org/10.1111/j.1365-2664.2009.01706.x
28. Salguero-Gómez, R, Jones, O.R., Archer, C.A., Buckley, Y.M., Che-Castaldo, J., Caswell, C., Hodgson, D., Scheuerlein, A., Conde, D.A., Brinks, E., de Buhr, H., Farack, C., Gottschalk, F., Hartmann, A., Henning, A., Hoppe, G., Roemer, G., Runge, J., Ruoff, T., et al. (2014) The COMPADRE Plant

Matrix Database: an online repository for plant population dynamics. Journal of Ecology 103: 202-218. https://doi.org/10.1111/1365-2745.12334

29. Salguero-Gómez, R., Jones, O.R., Archer, C.R., Bein, C., de Buhr, H., Farack, C., Gottschalk, F., Hartmann, A., Henning, A., Hoppe, G., Roemer, G., Ruoff, T., Sommer, V., Wille, J. Voigt, J., Zeh, S., Vieregg, D., Buckley, Y.M., Che-Castaldo, J., Hodgson, D., et al. (2016) COMADRE: a global database of animal demography. Journal of Animal Ecology 85: 371-384. https://doi.org/10.1111/1365-2656.12482
30. Shefferson, R.P., Kurokawa, S., & Ehrlen, J. (2020). LEFKO3: analysing individual history through size-classified matrix population models. Methods in Ecology and Evolution. https://doi.org/10.1111/2041-210X.13526
31. Williams, J.L., Miller, T.E.X., & Ellner, S.P. (2012). Avoiding unintentional eviction from integral projection models. Ecology 93(9): 2008-2014. https://doi.org/10.1890/11-2147.1

7 Appendix 1: `ipmr` Case Study 1

7.0.1 Two versions of a simple model

The first case study in this dissertation creates a model for *Carpobrotus spp.* The dataset used in this case study was collected in Havatselet Ha'Sharon, a suburb of Tel Aviv, Israel. The data were collected by drones taking aerial imagery of the population in successive years. Images were combined into a single high-resolution orthomosaic and georeferenced so the map from year 2 laid on top of the map from year 1. Flowers on each plant were counted using a point layer, and polygons were drawn around each ramet to estimate sizes and survival from year to year. Plants that had 0 flowers were classified as non-reproductive, and any plant with 1 or more flowers was classified as reproductive. This led to four regression models - survival, growth conditional on survival, probability of flowering, and number of flowers produced conditional on flowering. Finally, plants present in year 2 that were not present in year 1 were considered new recruits. The mean and variance of their sizes were computed, and this was used to model the recruit size distribution.

The resulting IPM 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 name comes from the common name for *Carpobrotus* species, which is "iceplants").

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. $F(z', z) = p_f(z) * r_s(z) * p_r * r_d(z')$

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

5. $Logit(s(z)) = \alpha_s + \beta_s * z$
6. $G(z', z) = f_G(z', \mu_G(z), \sigma_G)$
7. $\mu_G(z) = \alpha_G + \beta_G * z$
8. $Logit(p_f(z)) = \alpha_{p_f} + \beta_{p_f} * z$
9. $Log(r_s(z)) = \alpha_{r_s} + \beta_{r_s} * z$
10. $r_d(z') = f_{r_d}(z', \mu_{r_d}, \sigma_{r_d})$

αs and βs correspond to intercepts and slopes from regression models, respectively. Here, f_G and f_{r_d} are used to denote normal probability density functions. The other parameters are constants derived directly from the data itself.

```r
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

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.

NB: `L` is multiplied by 1.2 because the log of the minimum observed size is negative, and we want to extend the size range to make it more negative. If `L` were positive, we'd multiply by 0.8.

```
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 start with the function `init_ipm()`. This function has five arguments: `sim_gen`, `di_dd`, `det_stoch`, `kern_param`, and `uses_age`. For now, we will ignore the last argument, as it is covered in case study 2. The first 4 arguments specify the type of IPM we are building:

1. `sim_gen`: `"simple"`/`"general"`
   - A. **simple**: This describes an IPM with a single continuous state variable and no discrete stages.
   - B. **general**: This describes and IPM with either more than one continuous state variable, one or more discrete stages, or both of the above. Basically, anything other than an IPM with a single continuous state variable.
2. `di_dd`: `"di"`/`"dd"`
   - A. **di**: This is used to denote a **d**ensity-**i**ndependent IPM.
   - B. **dd**: This is used to denote a **d**ensity-**d**ependent IPM.
3. `det_stoch`: `"det"`/`"stoch"`
   - A. **det**: This is used to denote a deterministic IPM. If this is the third argument of `init_ipm`, `kern_param` must be left as `NULL`.
   - B. **stoch**: This is used to denote a stochastic IPM. If this is the third argument of `init_ipm`, `kern_param` must be specified.

This particular model is deterministic, as there are no data on temporal or spatial changes in vital rates. An introduction to stochastic models is available here. This example does not make use of the final argument, `kern_param`, because it is not a stochastic model, so we'll ignore it for now.

Once we've decided on the type of model we want, we create the model class using one of the two options for each argument. Since there is no stochasticity, we can leave the fourth argument empty (its default is `NULL`).

This case study is a simple, density independent, deterministic IPM, so we use the following:

```
carpobrotus_ipm <- init_ipm(sim_gen = "simple", di_dd = "di", det_stoch = "det")
```

After we have initialized our IPM, we need to start adding sub-kernels using the `define_kernel()` function. These correspond to equations 3 and 4 above. We'll start with the `P` kernel. It contains functions that describe survival of individual ramets, and, if they survive, their new sizes. Note that in `ipmr`, the order in which we define kernels for an IPM makes no difference, so we could also start with the `F` if we wanted to.

1. Survival is modeled with a logistic regression to predict the probability of survival to $t + 1$ based on the size of the ramet at t (`surv_mod`). In order to use the coefficients from that model to generate a survival probability, we need to know the inverse logit transformation, or, a function that performs it for us based on the linear predictor.

2. Size at $t + 1$ is modeled with a Gaussian distribution with two parameters: the mean and standard deviation from the mean. The mean value of size at $t + 1$ (`mu_G`) is itself a linear function of size at t and is parameterized with coefficients from the linear model (`grow_mod`). The standard deviation is a constant derived from the residual variance from the linear model we fit.

We start providing information on the `P` kernel by giving it a `name`. The name is important because we can use it to reference this kernel in higher level expressions later on. It can have any name we want, but `P` is consistent with the literature in this field (e.g. Easterling, Ellner & Dixon 2000, Ellner & Rees 2006). Next, we write the `formula`. The `formula` is the form of the kernel, and should look like Equation 3, without the z and z' arguments.

```
carpobrotus_ipm <-  define_kernel(
  proto_ipm = carpobrotus_ipm,
  name      = "P",
  formula   = s * G,
  ...
)
```

The `family` comes after `formula`. It describes the type of transition the kernel is implementing. `family` can be one of 4 options:

1. `"CC"`: Continuous state -> continuous state.

2. `"DC"`: discrete state -> continuous state.

3. `"CD"`: continuous state -> discrete state.

4. `"DD"`: discrete state -> discrete state.

Since this is a simple IPM with only 1 continuous state variable and 0 discrete state variables, the `family` will always be `"CC"`. In general IPMs, this will not always be true.

```
carpobrotus_ipm <-  define_kernel(
  proto_ipm = carpobrotus_ipm,
  name      = "P",
  formula   = s * G,
  family    = "CC",
  ...
)
```

We've now reached the `...` section of `define_kernel()`. The `...` part takes a set of named expressions that represent the vital rate functions we described in equations 5-7 above. The names on the left hand side of the `=` should appear either in the `formula` argument, or in other parts of the `...`. The expressions on the right hand side should generate the values that we want to plug in. For example, Equation 5 ($Logit(s(z)) = \alpha_s + \beta_s * z$) makes use of the `plogis` function in the `stats` package to compute the survival probabilities from our linear model. The names of the coefficients match the names in the `all_params` object we generated above. Another thing to note is the use of `z_1` and `z_2`. These are place-holders for z, z' in the equations above. `ipmr` will generate values for these internally using information that we provide in some of the next steps.

```
carpobrotus_ipm <-  define_kernel(
  proto_ipm = carpobrotus_ipm,
  name      = "P",
  formula   = s * G,
  family    = "CC",
  G         = dnorm(z_2, mu_g, grow_sdv),
  mu_g      = grow_int + grow_slo * z_1,
  s         = plogis(surv_int + surv_slo * z_1),
  ...
)
```

After setting up our vital rate functions, the next step is to provide a couple more kernel-specific details:

1. `data_list`: this is the `all_params` object we created above. It contains the names and values of all the constants in our model.
2. `states`: A list that contains the names of the state variables in the kernel. In our case, we've just called them `"z"`. The `states` argument controls the names of the variables `z_1` and `z_2` that are generated internally. We could just as easily call them something else - we would just have to change the vital rate expressions to use those names instead. For example, in this model, z, z' is the log-transformed surface area of ramets. We could abbreviate that with `"log_sa"`. In that case, `z_1,z_2` would become `log_sa_1, log_sa_2` in the vital rate expressions.
3. `evict_cor`: Whether or not to correct for eviction (Williams et al. 2012).
4. `evict_fun`: If we decide to correct for eviction, then a function that will correct it. In this example, we use `ipmr`'s `truncated_distributions` function. It takes two arguments: `fun`, which is the abbreviated form of the probability function family (e.g. "norm" for Gaussian, "lnorm" for log-normal, etc.), and `target`, which is the name in `...` that it modifies.

```
carpobrotus_ipm <-  define_kernel(
  proto_ipm = carpobrotus_ipm,
  name      = "P",
  formula   = s * G,
  family    = "CC",
  G         = dnorm(z_2, mu_g, grow_sdv),
  mu_g      = grow_int + grow_slo * z_1,
  s         = plogis(surv_int + surv_slo * z_1),
  data_list = all_params,
  states    = list(c("z")),
  evict_cor = TRUE,
  evict_fun = truncated_distributions(fun    = "norm",
                                      target = "G")
)
```

We've now defined our first sub-kernel. The next step is to repeat this process for the `F` kernel, which is Equations 4 and 8-10.

```
carpobrotus_ipm <-  define_kernel(
  proto_ipm = carpobrotus_ipm,
  name      = "F",
  formula   = recr_pr * r_s * r_d * p_f,
  family    = "CC",
  r_s       = exp(flow_int + flow_slo * z_1),
  r_d       = dnorm(z_2, recr_mu, recr_sd),
  p_f       = plogis(repr_int + repr_slo * z_1),
  data_list = all_params,
  states    = list(c("z")),
  evict_cor = TRUE,
  evict_fun = truncated_distributions(fun    = "norm",
                                      target =  "r_d")
)
```

We've defined our sub-kernels. The next step is tell `ipmr` how to implement it numerically, and provide the information needed to generate the correct iteration kernel. To do this, we use `define_impl()`, `define_domains()`, and `define_pop_state()`.

The first function tells `ipmr` which integration rule to use, which state variable each kernel acts on (`state_start`), and which state variable each kernel produces (`state_end`). The format of the list it takes in the `kernel_impl_list` argument can be tricky to implement right, so the helper function `make_impl_args_list()` makes sure everything is formatted properly. The `kernel_names` argument can be in any order. The `int_rule`, `state_start`, and `state_end` arguments are then matched to kernels in the `proto_ipm` based on the order in the `kernel_names`. Note that, at the moment, the only integration rule that's implemented is `"midpoint"`. `"b2b"` (bin to bin) and `"cdf"` (cumulative density functions) are in the works, and others can be implemented by popular demand.

```
carpobrotus_ipm <-  define_impl(
  proto_ipm = carpobrotus_ipm,
  make_impl_args_list(
    kernel_names = c("P", "F"),
    int_rule     = rep('midpoint', 2),
    state_start    = rep('z', 2),
    state_end      = rep('z', 2)
  )
)
```

Next, we define the range of values that our state variable, z/`z` can take on. This is done using `define_domains`. The `...` argument should have named vectors. The name should match the name of the `state`/`domain`. The first value in the vector is lower boundary, the second entry is the upper boundary, and the third entry is the number of bins to divide that range into.

```
carpobrotus_ipm <-  define_domains(
  proto_ipm = carpobrotus_ipm,
  z         = c(L, U, n_mesh_p)
)
```

Finally, we define the initial population state. In this case, we just use a uniform vector, but we could also use custom functions we defined on our own, or pre-specified vectors. The name of the population vector should be the name of the `state`/`domain`, with an `"n_"` attached to the front.

```
carpobrotus_ipm <-  define_pop_state(
  proto_ipm = carpobrotus_ipm,
    n_z     = rep(1/100, n_mesh_p)
)
```

Up until this point, all we've done is add components to the `proto_ipm`. We now have enough information in `proto_ipm` object to build a model, iterate it, and compute some basic quantities. `make_ipm()` is the next function we need. It generates the vital rate functions from the parameters and integration details we provided, and then builds the sub-kernels. At this point, it checks to make sure that everything makes numerical sense (e.g. there are no negative values or `NA`s generated). If we set `iterate = TRUE`, `make_ipm()` also generates expressions for iterating the model internally, and then evaluates those for the number of iterations supplied by `iterations`. There are a number of other arguments to `make_ipm()` that can prove helpful for subsequent analyses. `return_main_env` is one of these. The `main_env` object contains, among other things, the integration mesh and bin width information specified in `define_domains()`. We'll need the meshpoints and bin width for the analyses we'll do in the Further Analyses section, so we'll set `return_main_env = TRUE`.

```
carpobrotus_ipm <-  make_ipm(
  proto_ipm       = carpobrotus_ipm,
  iterate         = TRUE,
  iterations      = 100,
  return_main_env = TRUE
)


asymp_grow_rate <- lambda(carpobrotus_ipm)
asymp_grow_rate
```

```
##    lambda
## 0.9759257
```

We see that the population is projected to shrink slightly. `ipmr` computes all values by iteration. Our measure of the asymptotic growth rate is the ratio $\frac{N_{t+1}}{N_t}$ for the final iteration of the model. If we are concerned about whether or not we've iterated our model enough to trust this value, we have two options: check for convergence using the helper `is_conv_to_asymptotic()`, or create the full iteration kernel, compute the dominant eigenvalue of that, and compare our estimate with the value obtained by iteration.

```
# Option 1: is_conv_to_asymptotic

is_conv_to_asymptotic(carpobrotus_ipm)
```

```
## lambda
##   TRUE
```

```
# Option 2: generate iteration kernel and compute eigenvalues

K <- make_iter_kernel(carpobrotus_ipm)

lam_eigen <- Re(eigen(K$mega_matrix)$values[1])

# If we've iterated our model enough, this should be approximately 0 (though
# maybe a little off due to floating point errors).

asymp_grow_rate - lam_eigen
```

```
##       lambda
## 3.352874e-14
```

We can also inspect our sub-kernels, the time series of the population trait distribution, and make alterations to our model using some helpers from `ipmr`.

```
# Sub-kernels have their own print method to display the range of values
# and some diagnotic information.

carpobrotus_ipm$sub_kernels
```

```
## $P
##
##  Minimum value: 0, maximum value: 0.08763
## All entries greater than or equal to 0: TRUE
```

```
##
## $F
##
##  Minimum value: 0, maximum value: 0.02512
## All entries greater than or equal to 0: TRUE
```

```
# Extract the time series of the population state (n_z),
# and the n_t+1/n_t values (lambda)

pop_time_series    <- carpobrotus_ipm$pop_state$n_z
lambda_time_series <- carpobrotus_ipm$pop_state$lambda

# Next, we'll tweak the intercept of the p_f function and re-fit the model.

new_proto_ipm      <- carpobrotus_ipm$proto_ipm

# The parameters setter function takes a list. It can replace single values,
# create new values, or replace the entire parameter list, depending on how you
# set up the right hand side of the expression.

parameters(new_proto_ipm) <- list(repr_int = -0.3)

new_carp_ipm <- make_ipm(new_proto_ipm,
                         iterations = 100)

lambda(new_carp_ipm)
```

```
##    lambda
## 0.9720439
```

Next, we'll go through an alternative implementation of the model using `predict(surv_mod)` instead of the mathematical form of the linear predictors. After that, we'll explore a couple additional analyses to see what is going on with this population of iceplants.

7.0.2 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.

Above, we added parts to the `carpobrotus_ipm` object in a stepwise fashion. However, every `define_*` function in `ipmr` takes a `proto_ipm` as the first argument and returns a `proto_ipm` object. Thus, we can also use the `%>%` operator from the `magrittr` package to chain together the model creation pipeline. The `%>%` is included in `ipmr`, so we don't need to load any additional packages to access it. This example will demonstrate that process as well.

```
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(sim_gen = "simple",
                                       di_dd = "di",
                                       det_stoch = "det") %>%
  define_kernel(
    name      = "P",
    formula   = s * G,
    family    = "CC",
    G         = dnorm(z_2, mu_g, grow_sdv),
    mu_g      = predict(grow_mod,
                        newdata = data.frame(log_size = z_1),
                        type = 'response'),
    s         = predict(surv_mod,
                        newdata = data.frame(log_size = z_1),
                        type = "response"),
    data_list = pred_par_list,
    states    = list(c('z')),
    evict_cor = TRUE,
    evict_fun = truncated_distributions("norm", "G")
  ) %>%
  define_kernel(
    name      = "F",
    formula   = recr_pr * r_s * r_d * p_f,
    family    = "CC",
    r_s       = predict(flow_mod,
                        newdata = data.frame(log_size = z_1),
                        type = "response"),
    r_d       = dnorm(z_2, recr_mu, recr_sd),
    p_f       = predict(repr_mod,
                        newdata = data.frame(log_size = z_1),
                        type = "response"),
    data_list = pred_par_list,
    states    = list(c("z")),
    evict_cor = TRUE,
    evict_fun = truncated_distributions("norm", "r_d")
  ) %>%
  define_impl(
    make_impl_args_list(
      kernel_names = c("P", "F"),
      int_rule     = rep('midpoint', 2),
      state_start    = rep('z', 2),
      state_end      = rep('z', 2)
    )
  ) %>%
  define_domains(
    z = c(L, U, n_mesh_p)
  )  %>%
  define_pop_state(
    n_z = rep(1/100, n_mesh_p)
  ) %>%
  make_ipm(iterate    = TRUE,
           iterations = 100)
```

7.0.3 Further analyses

Many research questions require a bit more than just computing asymptotic growth rate (λ). Below, we will compute the kernel sensitivity, elasticity, R_0, and generation time. First, we will define a couple of 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.

The first is sensitivity of λ to perturbations in the projection kernel. Here, we can use the `right_ev` and `left_ev` functions in `ipmr` to get the right and left eigenvectors, and then compute the sensitivity surface.

Technical note: `right_ev` and `left_ev` both compute eigenvectors via iteration. `left_ev` generates a transpose iteration using the `state_start` and `state_end` information contained in the `proto_ipm` object (defined in `define_impl`, for a full overview of transpose iteration, see Ellner & Rees, 2006, Appendix A). Because the form of for left iteration is different from the default of right iteration, `left_ev()` will always have to iterate a model. On the other hand, `right_ev` will always check to see if the model is already iterated. If so, and the population's trait distribution has converged to its asymptotic state, then it will just pull out the final distribution from the `ipm` object, scale it to sum to 1, and then return that without re-iterating anything. If not, it will use the final trait distribution from the `ipm` object as the starting point and iterate the model for 100 iterations (this can be adjusted as needed using the `iterations` argument to `right_ev`). If this fails to converge, it will return `NA` with a warning.

It is also important to note that we have a second argument here named `d_z`. This is the width of the integration bins. We'll see how to get that from our IPM below.

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

  w <- right_ev(ipm_obj)[[1]]
  v <- left_ev(ipm_obj)[[1]]

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

}
```

Next, we can define a function to compute the elasticity of λ to kernel perturbations. This uses the `sens` function from above, and the `lambda()` function from `ipmr`.

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

  K           <- make_iter_kernel(ipm_obj)$mega_matrix

  sensitivity <- sens(ipm_obj, d_z)

  lamb        <- lambda(ipm_obj)

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

  return(out)
```

```r
}
```

We may also want to compute the per-generation population growth rate. The function below uses the sub-kernels contained in the `carpobrotus_ipm` object to do that.

```r
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]
  )

}
```

Finally, generation time is a useful metric in many analyses. Below, we make use of our `R_nought` function to compute one version of this quantity (though other definitions exist. Covering those is beyond the scope of this case study).

```r
gen_time <- function(ipm_obj) {

  lamb     <- unname(lambda(ipm_obj))

  r_nought <- R_nought(ipm_obj)

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

We need to extract the `d_z` value and meshpoints from the IPM we built. We can extract this information in a list form using the `int_mesh()` function from `ipmr` on our IPM object. The `d_z` in this case will be called `d_z` because we named our domain `"z"` when we implemented the model. However, it will have a different name if the `states` argument in `define_kernel` has different values. Once we have that, we can begin computing all the values of interest. For example, if `states = list(c("dbh", "height"))`, then `int_mesh()` would a return a list with `d_dbh` and `d_height`.

```r
mesh_info <- int_mesh(carpobrotus_ipm)

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

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

R0
```

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

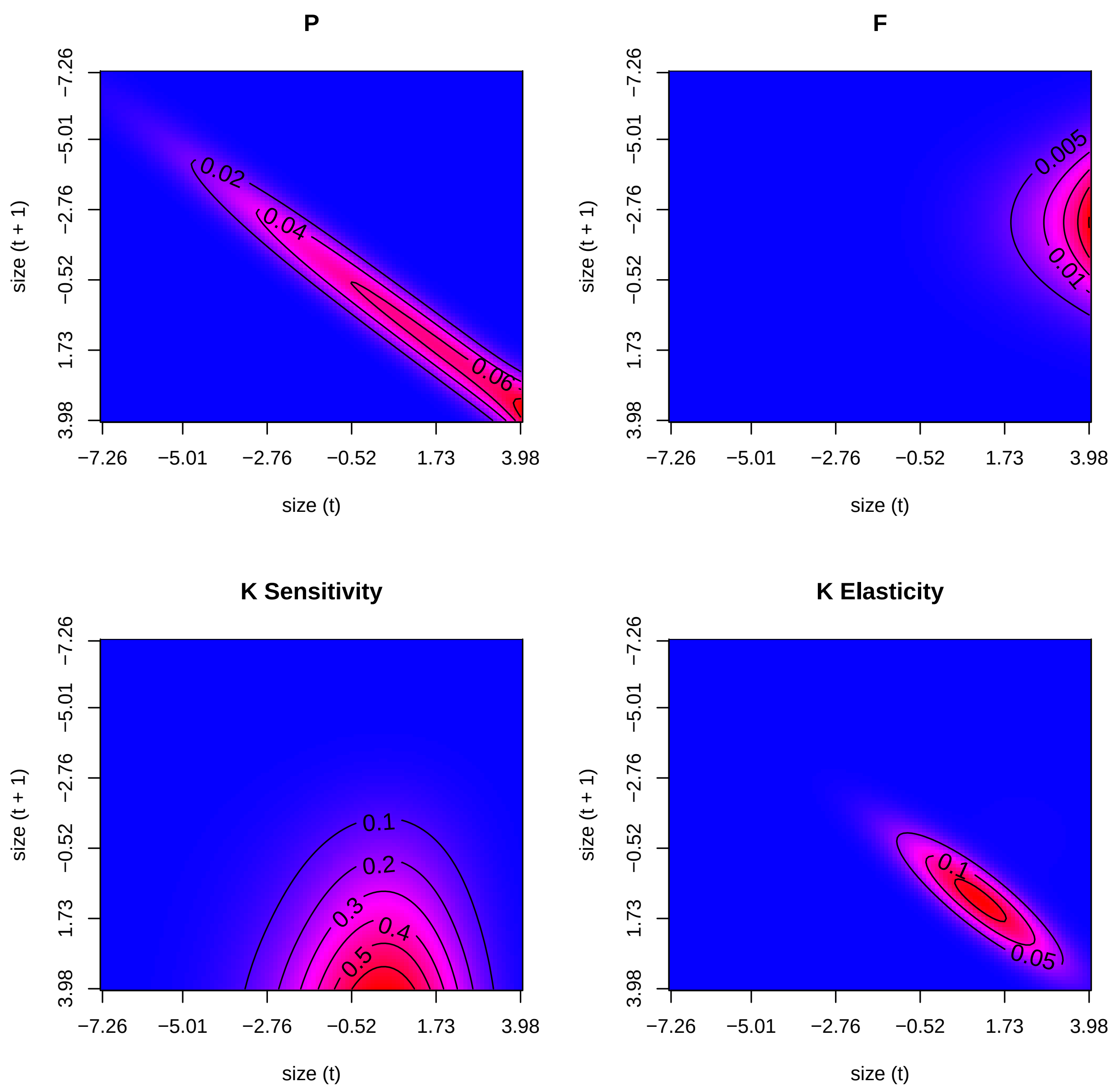
```r
gen_T
```

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

We may want to visualize our sub-kernels, iteration kernel, and the results of our sensitivity and elasticity analyses. We'll go through two options: one using the `graphics` package and one using the `ggplot2` package.

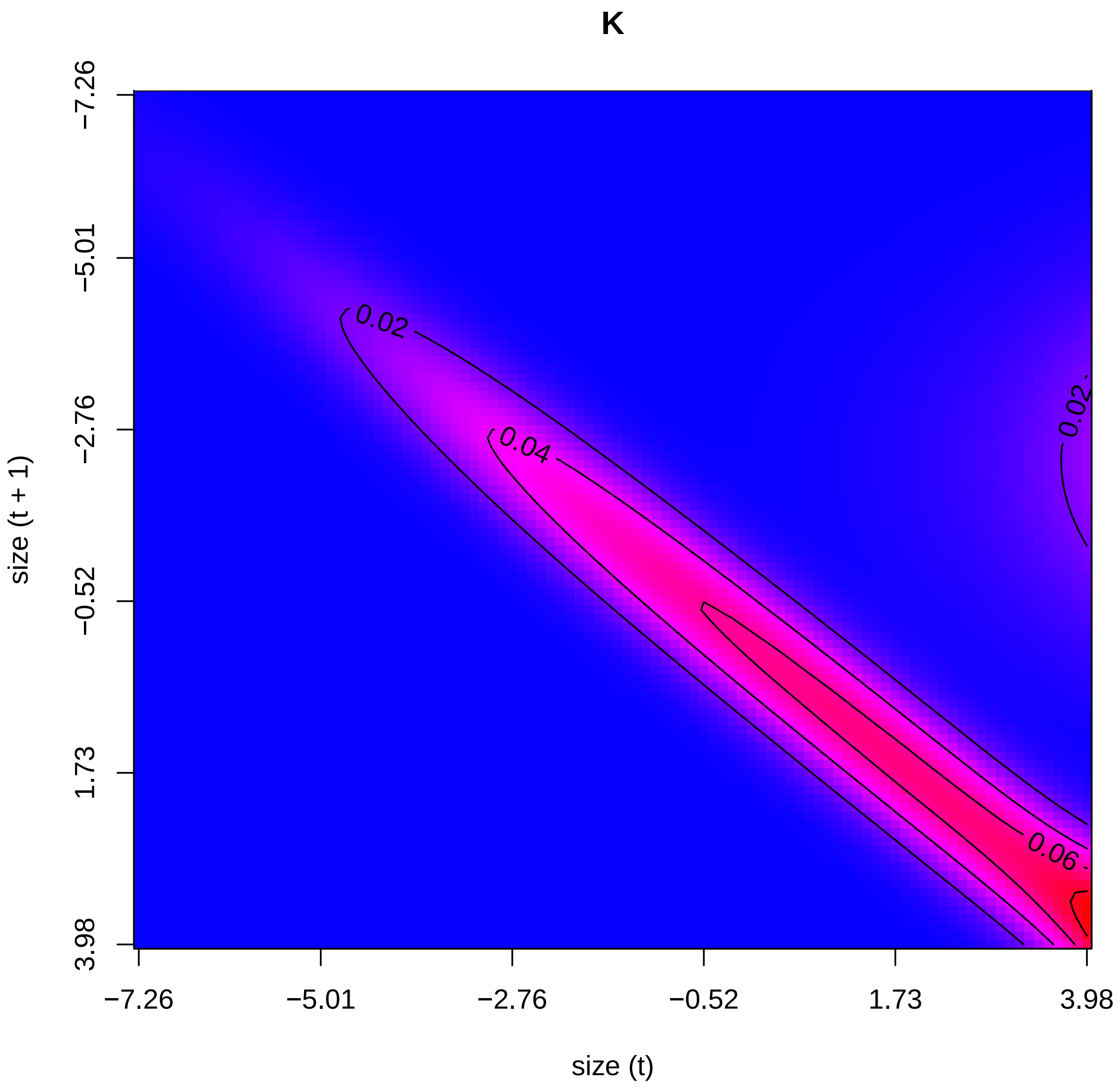
First, the `graphics` package.

```
lab_seq  <- round(seq(L, U, length.out = 6), 2)
tick_seq <- c(1, 20, 40, 60, 80, 100)

par(mfrow = c(2, 2))

# Sub-kernels - ipmr contains plot methods for sub-kernels

plot(carpobrotus_ipm$sub_kernels$P,
     do_contour = TRUE,
     main       = "P",
     xlab       = "size (t)",
     ylab       = "size (t + 1)",
     yaxt       = "none",
     xaxt       = "none")
axis(1, at = tick_seq, labels = as.character(lab_seq))
axis(2, at = tick_seq, labels = as.character(lab_seq))

plot(carpobrotus_ipm$sub_kernels$F,
     do_contour = TRUE,
     main       = "F",
     xlab       = "size (t)",
     ylab       = "size (t + 1)",
     yaxt       = "none",
     xaxt       = "none")
axis(1, at = tick_seq, labels = as.character(lab_seq))
axis(2, at = tick_seq, labels = as.character(lab_seq))



# Sensitivity and elasticity

class(sens_mat) <- c("ipmr_matrix", class(sens_mat))
class(elas_mat) <- c("ipmr_matrix", class(elas_mat))

plot(sens_mat,
     do_contour = TRUE,
     main       = "K Sensitivity",
     xlab       = "size (t)",
     ylab       = "size (t + 1)",
     yaxt       = "none",
     xaxt       = "none")
axis(1, at = tick_seq, labels = as.character(lab_seq))
axis(2, at = tick_seq, labels = as.character(lab_seq))


plot(elas_mat,
     do_contour = TRUE,
     main       = "K Elasticity",
     xlab       = "size (t)",
     ylab       = "size (t + 1)",
     yaxt        = "none",
     xaxt        = "none")
axis(1, at = tick_seq, labels = as.character(lab_seq))
axis(2, at = tick_seq, labels = as.character(lab_seq))
```



If we want to plot the iteration kernel, we can use `ipmr`'s `make_iter_kernel()` function to create one, and then the `plot()` method to plot that as well.

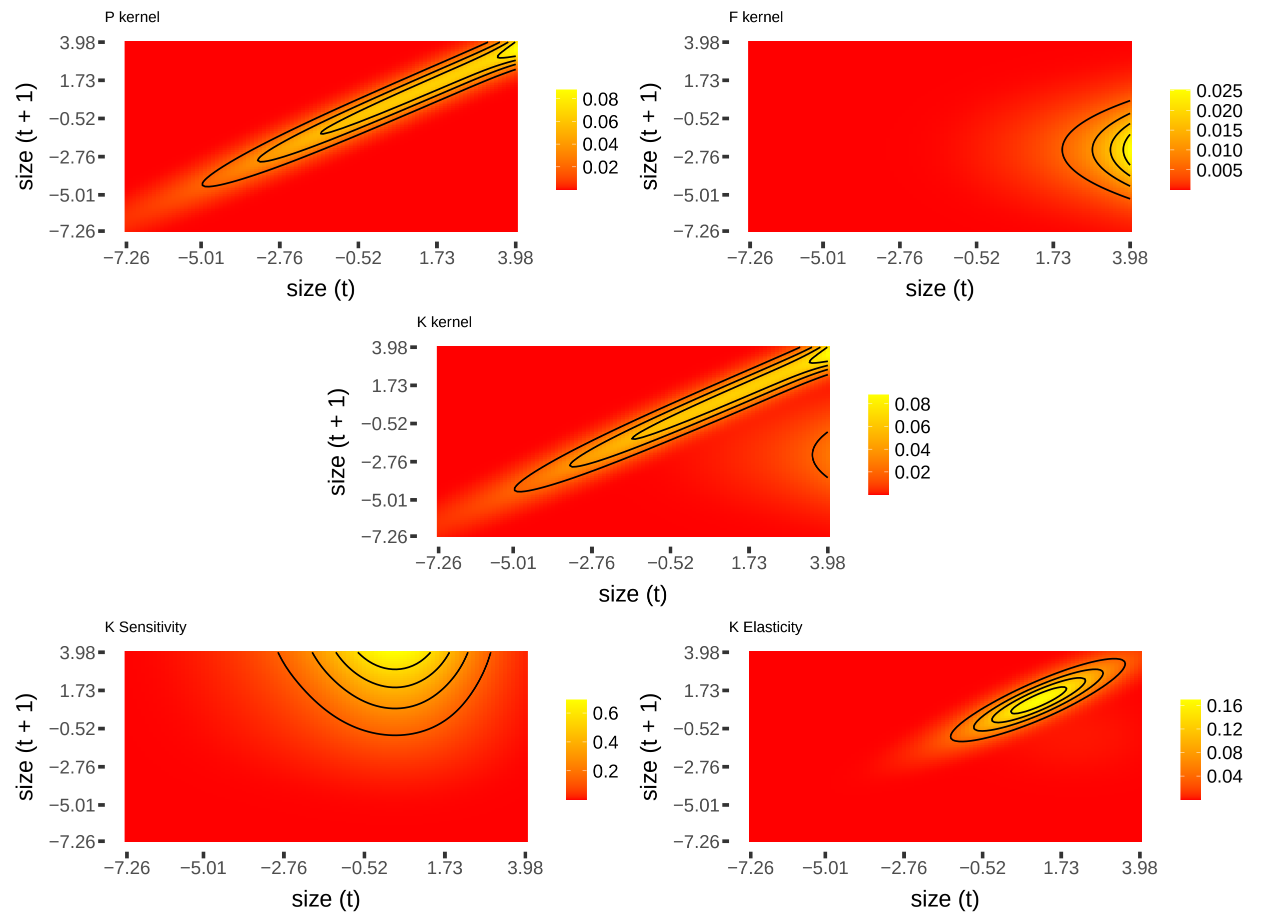
```
par(mfrow = c(1, 1))
K <- make_iter_kernel(carpobrotus_ipm)

plot(K$mega_matrix,
     do_contour = TRUE,
     main       = "K",
     xlab       = "size (t)",
     ylab       = "size (t + 1)",
     yaxt       = "none",
     xaxt       = "none")
axis(1, at = tick_seq, labels = as.character(lab_seq))
axis(2, at = tick_seq, labels = as.character(lab_seq))
```



Now, for the `ggplot2` version. First, we create a long format of the matrix using `ipmr`'s `ipm_to_df` function. `ipm_to_df` can handle either bare matrices, or objects produced by `make_ipm`. The latter case is useful for plotting kernels directly using `ggplot2`. Once we've generated the long format sensitivity and elasticity matrices, 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)

p_df    <- ipm_to_df(carpobrotus_ipm$sub_kernels$P)
f_df    <- ipm_to_df(carpobrotus_ipm$sub_kernels$F)
k_df    <- ipm_to_df(K$mega_matrix)

sens_df <- ipm_to_df(sens_mat)
elas_df <- ipm_to_df(elas_mat)

# Create a default theme for our plots

def_theme <- theme(
  panel.background  = element_blank(),
  axis.text         = element_text(size = 16),
  axis.ticks        = element_line(size = 1.5),
  axis.ticks.length = unit(0.08, "in"),
  axis.title.x      = element_text(
    size   = 20,
    margin = margin(
      t = 10,
      r = 0,
      l = 0,
      b = 2
    )
  ),
  axis.title.y = element_text(
    size   = 20,
    margin = margin(
      t = 0,
      r = 10,
      l = 2,
      b = 0
    )
  ),
  legend.text = element_text(size = 16)
)

p_plt <- ggplot(p_df) +
  geom_tile(aes(x    = t,
                y    = t_1,
                fill = value)) +
  geom_contour(aes(x = t,
                   y = t_1,
                   z = value),
               color = "black",
               size = 0.7,
               bins = 5) +
  scale_fill_gradient("Value",
                      low = "red",
                      high = "yellow") +
  scale_x_continuous(name = "size (t)",
                     labels = lab_seq,
                       breaks = tick_seq) +
  scale_y_continuous(name = "size (t + 1)",
                       labels = lab_seq,
                       breaks = tick_seq) +
  def_theme +
  theme(legend.title = element_blank()) +
  ggtitle("P kernel")

f_plt <- ggplot(f_df) +
  geom_tile(aes(x    = t,
                y    = t_1,
                fill = value)) +
  geom_contour(aes(x = t,
                   y = t_1,
                   z = value),
               color = "black",
               size = 0.7,
               bins = 5) +
  scale_fill_gradient("Value",
                        low = "red",
                        high = "yellow") +
  scale_x_continuous(name = "size (t)",
                       labels = lab_seq,
                       breaks = tick_seq) +
  scale_y_continuous(name = "size (t + 1)",
                       labels = lab_seq,
                       breaks = tick_seq) +
  def_theme +
  theme(legend.title = element_blank()) +
  ggtitle("F kernel")

k_plt <- ggplot(k_df) +
  geom_tile(aes(x    = t,
                y    = t_1,
                fill = value)) +
  geom_contour(aes(x = t,
                   y = t_1,
                   z = value),
               color = "black",
               size = 0.7,
               bins = 5) +
  scale_fill_gradient("Value",
                        low = "red",
                        high = "yellow") +
  scale_x_continuous(name = "size (t)",
                       labels = lab_seq,
                       breaks = tick_seq) +
  scale_y_continuous(name = "size (t + 1)",
                       labels = lab_seq,
                       breaks = tick_seq) +
  def_theme +
  theme(legend.title = element_blank()) +
  ggtitle("K kernel")
```

```r
sens_plt <- ggplot(sens_df) +
  geom_tile(aes(x    = t,
                y    = t_1,
                fill = value)) +
  geom_contour(aes(x = t,
                   y = t_1,
                   z = value),
               color = "black",
               size = 0.7,
               bins = 5) +
  scale_fill_gradient("Value",
                      low = "red",
                      high = "yellow") +
  scale_x_continuous(name = "size (t)",
                     labels = lab_seq,
                     breaks = tick_seq) +
  scale_y_continuous(name = "size (t + 1)",
                     labels = lab_seq,
                     breaks = tick_seq) +
  def_theme +
  theme(legend.title = element_blank()) +
  ggtitle("K Sensitivity")

elas_plt <- ggplot(elas_df) +
  geom_tile(aes(x    = t,
                y    = t_1,
                fill = value)) +
  geom_contour(aes(x = t,
                   y = t_1,
                   z = value),
               color = "black",
               size = 0.7,
               bins = 5) +
  scale_fill_gradient("Value",
                      low = "red",
                      high = "yellow") +
  scale_x_continuous(name = "size (t)",
                     labels = lab_seq,
                     breaks = tick_seq) +
  scale_y_continuous(name = "size (t + 1)",
                     labels = lab_seq,
                     breaks = tick_seq) +
  def_theme +
  theme(legend.title = element_blank()) +
  ggtitle("K Elasticity")


grid.arrange(
  p_plt,    f_plt, k_plt,
  sens_plt, elas_plt,
             layout_matrix = matrix(c(1, 1, 2, 2,
                                      NA, 3, 3, NA,
                                      4, 4, 5, 5),
                                    nrow = 3,
                                    byrow = TRUE))
```



8 Appendix 2: ipmr Case Study 2

8.0.1 A more complicated, age and size structured model

Many life cycles cannot be described by a single, continuous state variable. For example, some plants are best modeled using height or diameter at breast height (DBH) as a state variable, and may also form a seed bank. Seeds in the seed bank can't have a value for height or DBH, but may lose their viability as they age. Thus, we require a discrete state to capture the dynamics of the seed bank. Models that include discrete states and/or multiple continuous state variables are *general IPMs*.

Many species exhibit age-dependent demography. Age may interact with a measure of size, for example body mass, resulting in neither single variable reliably predicting demography on its own. Data sets for which individuals are cross-classified by age and size represent an interesting opportunity for demographic research.

This case study will use an age and size structured population to illustrate how to implement general IPMs in `ipmr`. We will use an age and size structured model from Ellner, Childs, & Rees (2016) to explore how to work with general IPMs in `ipmr`. The data are from a long term study of Soay sheep (*Ovis aries*) on St. Kilda. The population has been studied in detail since 1985 (Clutton-Brock & Pemberton 2004). Individuals are caught, weighed, and tagged shortly after birth, and re-weighed each subsequent year. Maternity can be inferred from field observations, so we can also model the link between parental state and offspring state.

More detailed methods are provided in Clutton-Brock & Pemberton (2004) and Childs et al. 2011.

In addition to computing the per-capita growth rate (λ) and right and left eigenvectors ($w_a(z)$ and $v_a(z)$, respectively), we will also show how to compute age specific survival and fertility for these models.

With size denoted z, z', age denoted a, and the maximum age an individual can have denoted M, the model can be written as:

1. $n_0(z', t+1) = \sum\limits_{a=0}^{M} \int_L^U F_a(z', z) n_a(z, t) dz$
2. $n_a(z', t+1) = \int_L^U P_{a-1}(z', z) n_{a-1}(z, t) dz$ for $a = 1, 2, ..., M$

In this case, there is also an "greybeard" age class $M+1$ defined as $a \geq M+1$. We need to define one more equation to indicate the number of individuals in that age group.

3. $n_{M+1}(z', t+1) = \int_L^U [P_M(z', z) n_M(z, t) + P_{M+1}(z', z) n_{M+1}(z, t)] dz$

Below, f_G and f_B denote normal probability density functions. The sub-kernel $P_a(z', z)$ is comprised of the following functions:

4. $P_a(z', z) = s(z, a) * G(z', z, a)$
5. Survival: $Logit(s(z, a)) = \alpha_s + \beta_{s,z} * z + \beta_{s,a} * a$
6. Growth: $G(z', z, a) = f_G(z', \mu_G(z, a), \sigma_G)$
7. Mean Growth: $\mu_G(z, a) = \alpha_G + \beta_{G,z} * z + \beta_{G,a} * a$

and the sub-kernel $F_a(z', z)$ is comprised of the following functions:

8. $F_0 = 0$
9. $F_a = s(z, a) * p_b(z, a) * p_r(a) * B(z', z) * 0.5$

This model only follows females, and we assume that the population is half female. Thus, we multiply the F_a kernel by 0.5. If needed, we could adjust the sex ratio based on observed data, and update this multiplication accordingly.

10. Probability of reproducing: $Logit(p_b(z, a)) = \alpha_{p_b} + \beta_{p_b,z} * z + \beta_{p_b,a} * a$
11. Probability of recruiting: $Logit(p_r(a)) = \alpha_{p_r} + \beta_{p_r,a} * a$
12. Recruit size distribution: $B(z', z) = f_B(z', \mu_B(z), \sigma_B)$
13. Mean recruit size: $\mu_B(z) = \alpha_B + \beta_{B,z} * z$

Equations 5-7 and 10-13 are parameterized from regression models. The parameter values are taken from Ellner, Childs & Rees, Chapter 6 (2016). These can be found here. In the code from the book, these parameters were used to simulate an individual based model (IBM) to generate a data set. The data were then used to fit regression models and an age$\times$size IPM. We are going to skip the simulation and regression model fitting steps steps and just use the "true" parameter estimates to generate the IPM.

8.0.2 Model Code

First, we will define all the model parameters and a function for the F_a kernels.

```
library(ipmr)

# Set parameter values and names

param_list <- list(
  ## Survival
  surv_int  = -1.70e+1,
  surv_z    =  6.68e+0,
  surv_a    = -3.34e-1,
  ## growth
  grow_int  =  1.27e+0,
  grow_z    =  6.12e-1,
  grow_a    = -7.24e-3,
  grow_sd   =  7.87e-2,
  ## reproduce or not
  repr_int  = -7.88e+0,
  repr_z    =  3.11e+0,
  repr_a    = -7.80e-2,
  ## recruit or not
  recr_int  =  1.11e+0,
  recr_a    =  1.84e-1,
  ## recruit size
  rcsz_int  =  3.62e-1,
  rcsz_z    =  7.09e-1,
  rcsz_sd   =  1.59e-1
)

# define a custom function to handle the F kernels. We could write a rather
# verbose if(age == 0) {0} else {other_math} in the define_kernel(), but that
# might look ugly. Note that we CANNOT use ifelse(), as its output is the same
# same length as its input (in this case, it would return 1 number, not 10000
# numbers).

r_fun <- function(age, s_age, pb_age, pr_age, recr) {

  if(age == 0) return(0)

  s_age * pb_age * pr_age * recr * 0.5

}
```

Next, we set up the P_a kernels (Equations 4-7 above). Because this is a general, deterministic IPM, we use `init_ipm(sim_gen = "general", di_dd = "di", det_stoch = "det", uses_age = TRUE)`. We set `uses_age = TRUE` to indicate that our model has age structure as well as size structure. There are 3 key things to note:

1. the use of the suffix `_age` appended to the names of the `"P_age"` kernel and the `mu_g_age` variable.
2. the value `age` used in the vital rate expressions.
3. the list in the `age_indices` argument.

The values in the `age_indices` list will automatically get substituted in for `"age"` each time it appears in the vital rate expressions and kernels. We add a second variable to this list, `"max_age"`, to indicate that we have a "greybeard" class. If we wanted our model to kill all individuals above age 20, we would simply omit the `"max_age"` slot in the `age_indices` list.

This single call to `define_kernel()` will result in 22 actual kernels, one for each value of `age` from 0-21. For general IPMs that are not age-structured, we would use `uses_par_sets` and `par_set_indices` in the same way we're using `age` below.

The `plogis` function is part of the `stats` package in *R*, and performs the inverse logit transformation.

```
age_size_ipm <- init_ipm(sim_gen = "general",
                         di_dd = "di",
                         det_stoch = "det",
                         uses_age = TRUE) %>%
  define_kernel(
    name           = "P_age",
    family         = "CC",
    formula        = s_age * g_age * d_z,
    s_age          = plogis(surv_int + surv_z * z_1 + surv_a * age),
    g_age          = dnorm(z_2, mu_g_age, grow_sd),
    mu_g_age       = grow_int + grow_z * z_1 + grow_a * age,
    data_list      = param_list,
    states         = list(c("z")),
    uses_par_sets = FALSE,
    age_indices    = list(age = c(0:20), max_age = 21),
    evict_cor      = FALSE
  )
```

The F_a kernel (equations 8-13) will follow a similar pattern - we append a suffix to the `name` paramter, and then make sure that our functions also include `_age` suffixes and `age` values where they need to appear.

```
age_size_ipm <-   define_kernel(
  proto_ipm      = age_size_ipm,
  name           = "F_age",
  family         = "CC",
  formula        = r_fun(age, s_age, pb_age, pr_age, recr) * d_z,
  s_age          = plogis(surv_int + surv_z * z_1 + surv_a * age),
  pb_age         = plogis(repr_int + repr_z * z_1 + repr_a * age),
  pr_age         = plogis(recr_int + recr_a * age),
  recr           = dnorm(z_2, rcsz_mu, rcsz_sd),
  rcsz_mu        = rcsz_int + rcsz_z * z_1,
  data_list      = param_list,
  states         = list(c("z")),
  uses_par_sets = FALSE,
  age_indices    = list(age = c(0:20), max_age = 21),
  evict_cor      = FALSE
)
```

Once we've defined the P_a and F_a kernels, we need to define starting and ending states for each kernel. Age-size structured populations will look a little different from other models, as we need to ensure that all fecundity kernels produce `age-0` individuals, and all survival-growth kernels produce `age` individuals. We define the implementation arguments using `define_impl()`, and set each kernel's `state_start` to `"z_age"`. Because the fecundity kernel produces age-0 individuals, regardless of the starting age, its `state_end` is `"z_0"`.

```
age_size_ipm <-  define_impl(
  proto_ipm = age_size_ipm,
  make_impl_args_list(
    kernel_names = c("P_age", "F_age"),
    int_rule     = rep("midpoint", 2),
    state_start     = c("z_age", "z_age"),
    state_end       = c("z_age", "z_0")
  )
)
```

We define the domains using `define_domains()` in the same way we did for Case Study 1.

```
age_size_ipm <- age_size_ipm %>%
  define_domains(
    z = c(1.6, 3.7, 100)
  )
```

Our definition of the initial population state will look a little different though. We want to create 22 copies of the initial population state, one for each age group in the model. We do this by appending `_age` to the `n_z` in the `...` part of `define_pop_state`. We'll also set `make_ipm(...,return_all_envs = TRUE)` so we can access the computed values for each vital rate function in the model.

```
age_size_ipm <-  define_pop_state(
  proto_ipm = age_size_ipm,
  n_z_age = rep(1/100, 100)
  ) %>%
  make_ipm(
    usr_funs = list(r_fun = r_fun),
    iterate   = TRUE,
    iterations = 100,
    return_all_envs = TRUE
  )
```

8.0.3 Basic analysis

We see that the population is projected to grow by about 1.5% each year. As in Case Study 1, we can check for convergence using the `is_conv_to_asymptotic()` function.

```
lamb <- lambda(age_size_ipm)
lamb
```

```
##   lambda
## 1.014833
```

```
is_conv_to_asymptotic(age_size_ipm)
```

```
## lambda
##   TRUE
```

For some analyses, we may want to get the actual vital rate function values, rather than the sub-kernels and/or iteration kernels. We can access those with `vital_rate_funs()`. Note that right now, this function always returns a full bivariate form of the vital rate function (i.e. for survival, it returns a n X n kernel, rather than a 1 x n vector, where n is the number of meshpoints). It is also important to note that these functions are **not yet discretized**, and so need to be treated as such (i.e. any vital rate with a probability density function will contain probability densities, not probabilities).

```
vr_funs <- vital_rate_funs(age_size_ipm)

# Age 0 survival and growth vital rate functions

vr_funs$P_0
```

```
## s_0 (not yet discretized): A 100 x 100 kernel with minimum value: 0.0019 and maximum value: 0.9995
## g_0 (not yet discretized): A 100 x 100 kernel with minimum value: 0 and maximum value: 5.0691
## mu_g_0 (not yet discretized): A 100 x 100 kernel with minimum value: 2.2556 and maximum value: 3.528
```

```
# Age 12 fecundity functions

vr_funs$F_12
```

```
## s_12 (not yet discretized): A 100 x 100 kernel with minimum value: 0 and maximum value: 0.9744
## pb_12 (not yet discretized): A 100 x 100 kernel with minimum value: 0.0217 and maximum value: 0.9345
## pr_12 (not yet discretized): A 100 x 100 kernel with minimum value: 0.965 and maximum value: 0.965
## recr (not yet discretized): A 100 x 100 kernel with minimum value: 0 and maximum value: 2.5091
## rcsz_mu (not yet discretized): A 100 x 100 kernel with minimum value: 1.5038 and maximum value: 2.97
```

We can also update the model to use a new functional form for a vital rate expression. For example, we could add parent size dependence for the probability of recruiting function. This requires 3 steps: extract the `proto_ipm` object, set the new functional form, and update the parameter list. We have to wrap the assignment in `new_fun_form()` to prevent parsing errors.

```
new_proto <- age_size_ipm$proto_ipm

vital_rate_exprs(new_proto,
                 kernel = "F_age",
                 vital_rate = "pr_age") <-
  new_fun_form(plogis(recr_int + recr_z * z_1 + recr_a * age))

parameters(new_proto) <- list(recr_z = 0.05)

new_ipm <- make_ipm(new_proto,
                    return_all_envs = TRUE)

lambda(new_ipm)
```

```
##   lambda
## 1.017868
```

Next, we'll extract and visualize eigenvectors and compute age specific fertility and survival.

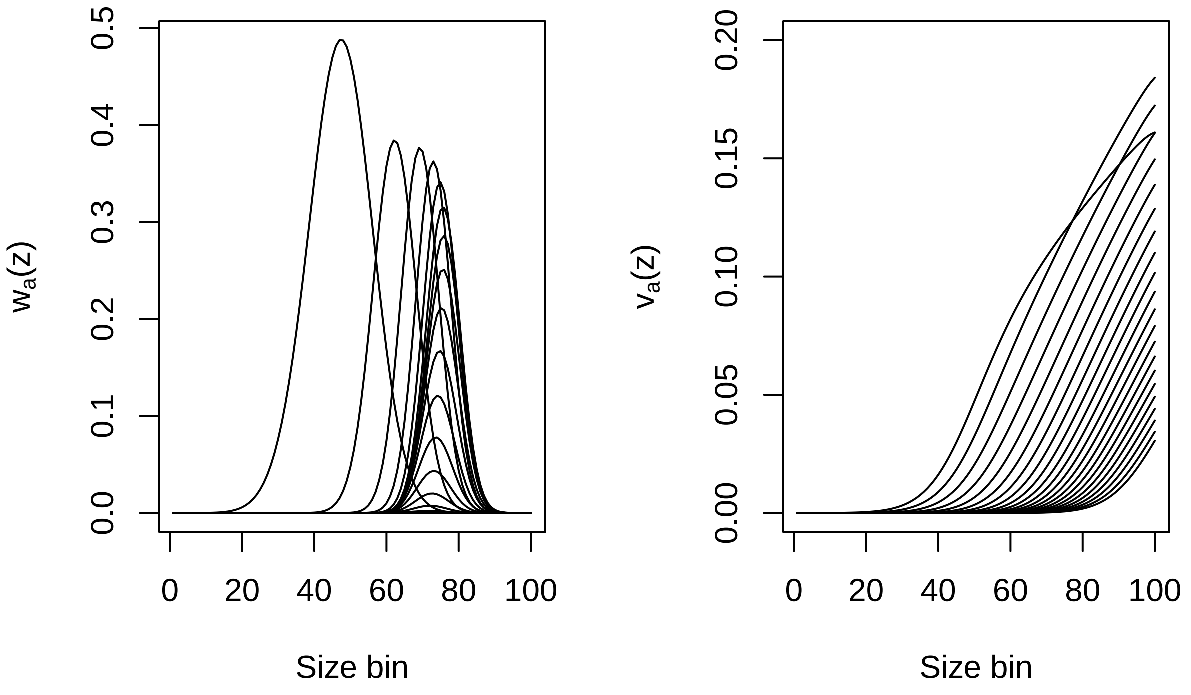
8.0.4 Further analyses

The `right_ev` and `left_ev` functions also work for age $\times$ size models. We can use extract these, and plot them using a call to `lapply`. We will use the notation from Ellner, Childs, & Rees (2016) to denote the left and right eigenvectors ($v_a(z)$ and $w_a(z)$, respectively).

NB: we assign the `lapply` call to a value here because `lines` returns `NULL` invisibly, and this clogs up the console. You probably don't need to do this for interactive use. We'll also divide the dz value back into each eigenvector so that they are continuous distributions, rather than discretized vectors.

```
d_z <- int_mesh(age_size_ipm)$d_z

stable_dists <- right_ev(age_size_ipm)

w_plot <- lapply(stable_dists, function(x, d_z) x / d_z,
         d_z = d_z)

repro_values <- left_ev(age_size_ipm)

v_plot <- lapply(repro_values, function(x, d_z) x / d_z,
                 d_z = d_z)
```

```r
par(mfrow = c(1, 2))
plot(w_plot[[1]], type = 'l',
     ylab = expression(paste("w"[a],"(z)")),
     xlab = "Size bin")

x <- lapply(w_plot[2:22], function(x) lines(x))

plot(v_plot[[1]], type = 'l',
     ylab = expression(paste("v"[a],"(z)")),
     xlab = "Size bin",
     ylim = c(0, 0.2))

x <- lapply(v_plot[2:22], function(x) lines(x))
```



Next, we'll compute age-specific survival ($\tilde{l}_a$/`l_a`) and fecundity ($\tilde{f}_a$/`f_a`) values. These are defined as follows:

$$\tilde{l}_a = eP^a c$$

$$\tilde{f}_a = (eFP^a c)/l_a$$

where c is some distribution of newborns.

We'll initialize a cohort using the stable size distribution for age-0 individuals that we obtained above. Next, we'll iterate them through our model for 100 years, and see who's left, and how much they reproduced.

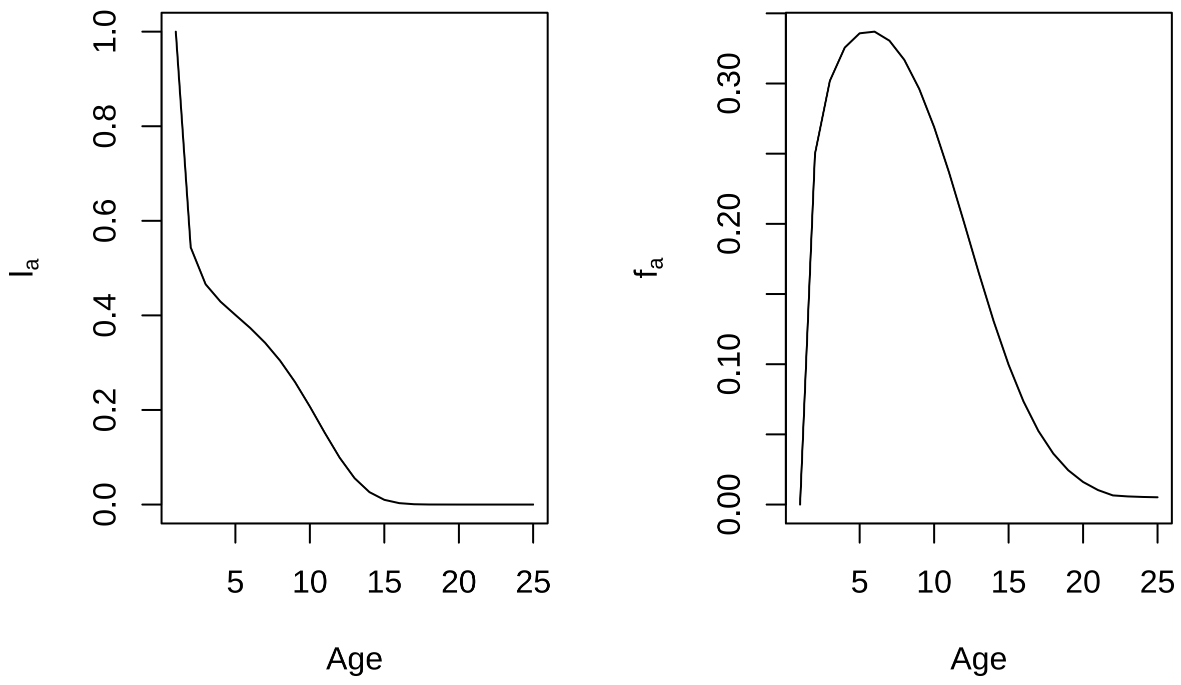
NB: do not try to use this method for computing R_0 - it will lead to incorrect results because in this particular model, parental state affects initial offspring state. For more details, see Ellner, Childs & Rees (2016), Chapters 3 and 6.

A couple technical notes:

1. We are going to split out our P and F sub-kernels into separate lists so that indexing them is easier during the iteration process.
2. We need to set our `max_age` variable to 22 now, so that we don't accidentally introduce an "off-by-1" error when we index the sub-kernels in our IPM object.
3. We use an identity matrix to compute the initial value of `l_a`, because by definition all age-0 individuals must survive to age-0.

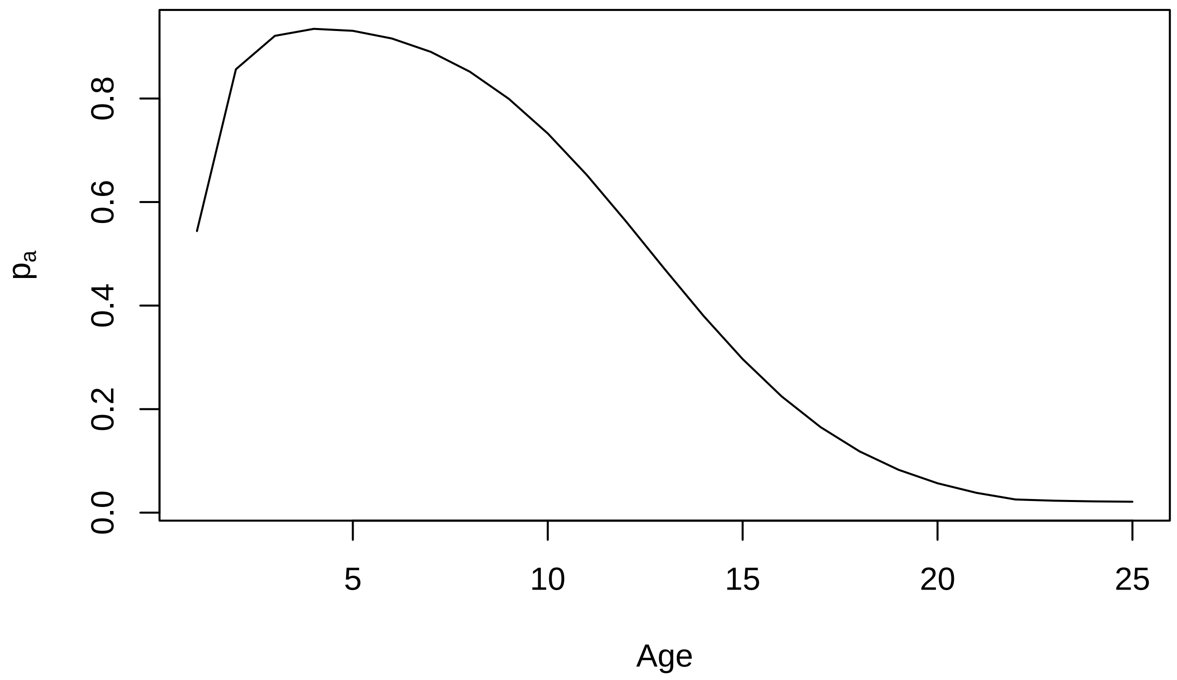
```
# Initialize a cohort and some vectors to hold our quantities of interest.

init_pop <- stable_dists[[1]] / sum(stable_dists[[1]])
n_yrs    <- 100L

l_a <- f_a <- numeric(n_yrs)

P_kerns <- age_size_ipm$sub_kernels[grepl("P", names(age_size_ipm$sub_kernels))]
F_kerns <- age_size_ipm$sub_kernels[grepl("F", names(age_size_ipm$sub_kernels))]

# We have to bump max_age because R indexes from 1, and our minimum age is 0.

max_age <- 22

P_a <- diag(length(init_pop))

for(yr in seq_len(n_yrs)) {

  # When we start, we want to use age-specific kernels until we reach max_age.
  # after that, all survivors have entered the "greybeard" class.

  if(yr < max_age) {

    P_now <- P_kerns[[yr]]
    F_now <- F_kerns[[yr]]

  } else {

    P_now <- P_kerns[[max_age]]
    F_now <- F_kerns[[max_age]]

  }

  l_a[yr] <- sum(colSums(P_a) * init_pop)
  f_a[yr] <- sum(colSums(F_now %*% P_a) * init_pop)

  P_a <- P_now %*% P_a
}

f_a <- f_a / l_a
```

```
# Looks like most are dead at after 25 years, so we'll restrict our
# plot range to that time span

par(mfrow = c(1, 2))

plot(l_a[1:25], type = 'l',
     ylab = expression(paste("l"[a])),
     xlab = "Age")
plot(f_a[1:25], type = 'l',
     ylab = expression(paste("f"[a])),
     xlab = "Age")
```



We can also calculate the age-specific survival probability p_a as $\frac{l_{a+1}}{l_a}$. We'll restrict our calculations to the first 25 years of life, as we'll see that almost no sheep live longer than that.

```
p_a <- l_a[2:26] / l_a[1:25]

plot(p_a, type = 'l',
     ylab = expression(paste("p"[a])),
     xlab = "Age")
```



9 Supplementary Information for Chapter 1

9.0.1 Figure S1

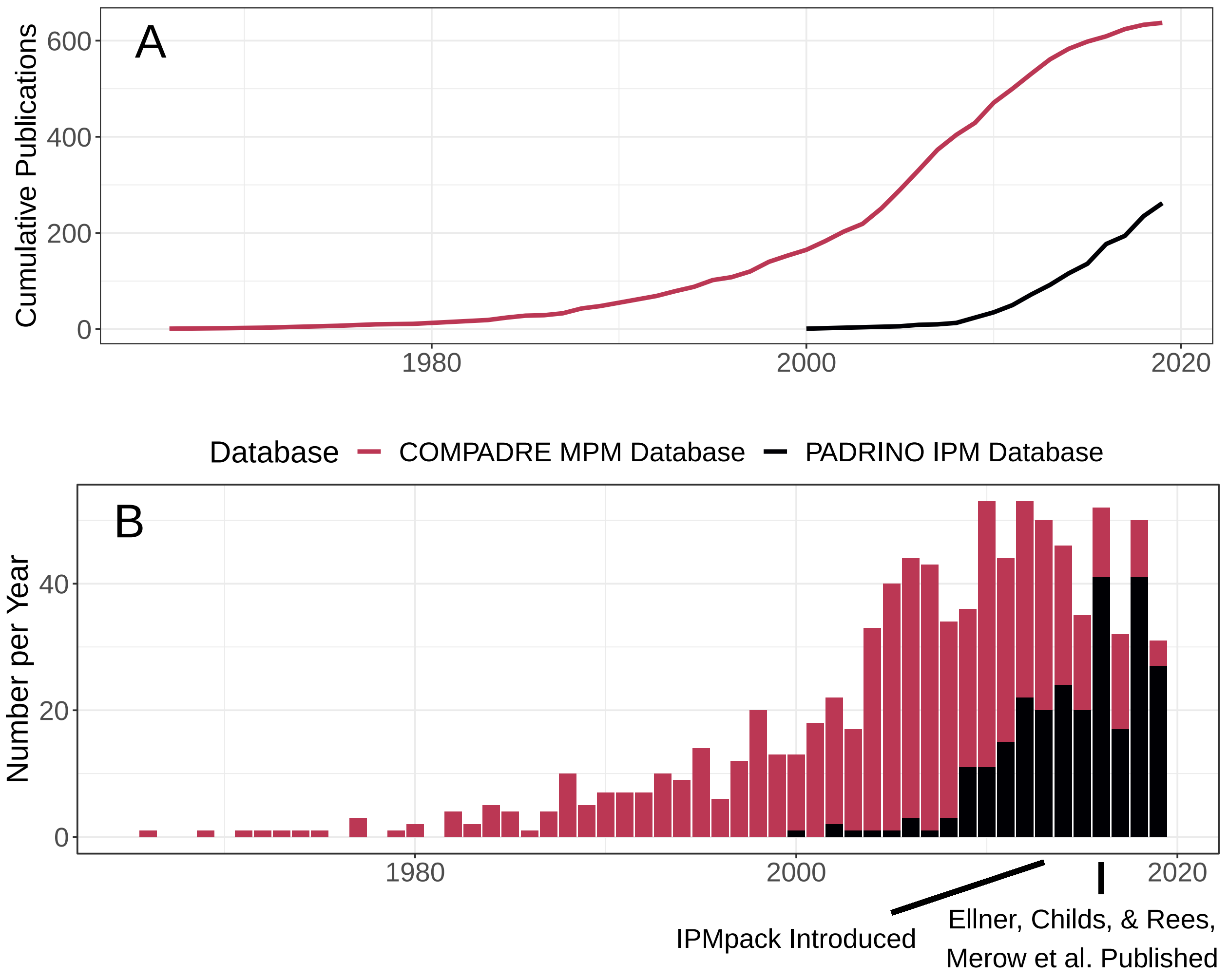


Figure S1: The usage of integral projection models (IPMs) has increased rapidly since their introduction. Cumulative number of publications using matrix projection models (MPMs, red) and IPMs (black) (A) and number of publications per year for each type of model (B). IPMs have been adopted rapidly since their introduction in 2000. Unfortunately, software packages to assist with their implementation have not kept pace with their theoretical advancements and applications to ever more complex demographic data.